User Guide for Aurora

Amazon Aurora

Amazon Aurora: User Guide for Aurora

Table of Contents

What is Amazon Aurora?

Amazon Aurora (Aurora) is a fully managed relational database engine that's compatible with MySQL and PostgreSQL. You already know how MySQL and PostgreSQL combine the speed and reliability of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. The code, tools, and applications you use today with your existing MySQL and PostgreSQL databases can be used with Aurora. With some workloads, Aurora can deliver up to five times the throughput of MySQL and up to three times the throughput of PostgreSQL without requiring changes to most of your existing applications.

Aurora includes a high-performance storage subsystem. Its MySQL- and PostgreSQL-compatible database engines are customized to take advantage of that fast distributed storage. The underlying storage grows automatically as needed. An Aurora cluster volume can grow to a maximum size of 128 tebibytes (TiB). Aurora also automates and standardizes database clustering and replication, which are typically among the most challenging aspects of database configuration and administration.

Aurora is part of the managed database service Amazon Relational Database Service (Amazon RDS). Amazon RDS makes it easier to set up, operate, and scale a relational database in the cloud. If you are not already familiar with Amazon RDS, see the *Amazon Relational Database Service User Guide*. To learn more about the variety of database options available on Amazon Web Services, see Choosing the right database for your organization on AWS.

Topics

- Replication with Amazon Aurora
- DB instance billing for Aurora

Amazon RDS shared responsibility model

Amazon RDS is responsible for hosting the software components and infrastructure of DB instances and DB clusters. You are responsible for query tuning, which is the process of adjusting SQL queries to improve performance. Query performance is highly dependent on database design, data size, data distribution, application workload, and query patterns, which can vary greatly. Monitoring and tuning are highly individualized processes that you own for your RDS databases. You can use Amazon RDS Performance Insights and other tools to identify problematic queries.

How Amazon Aurora works with Amazon RDS

The following points illustrate how Amazon Aurora relates to the standard MySQL and PostgreSQL engines available in Amazon RDS:

- You choose Aurora MySQL or Aurora PostgreSQL as the DB engine option when setting up new database servers through Amazon RDS.
- Aurora takes advantage of the familiar Amazon Relational Database Service (Amazon RDS) features for management and administration. Aurora uses the Amazon RDS AWS Management Console interface, AWS CLI commands, and API operations to handle routine database tasks such as provisioning, patching, backup, recovery, failure detection, and repair.
- Aurora management operations typically involve entire clusters of database servers that are synchronized through replication, instead of individual database instances. The automatic clustering, replication, and storage allocation make it simple and cost-effective to set up, operate, and scale your largest MySQL and PostgreSQL deployments.
- You can bring data from Amazon RDS for MySQL and Amazon RDS for PostgreSQL into Aurora by creating and restoring snapshots, or by setting up one-way replication. You can use push-button migration tools to convert your existing RDS for MySQL and RDS for PostgreSQL applications to Aurora.

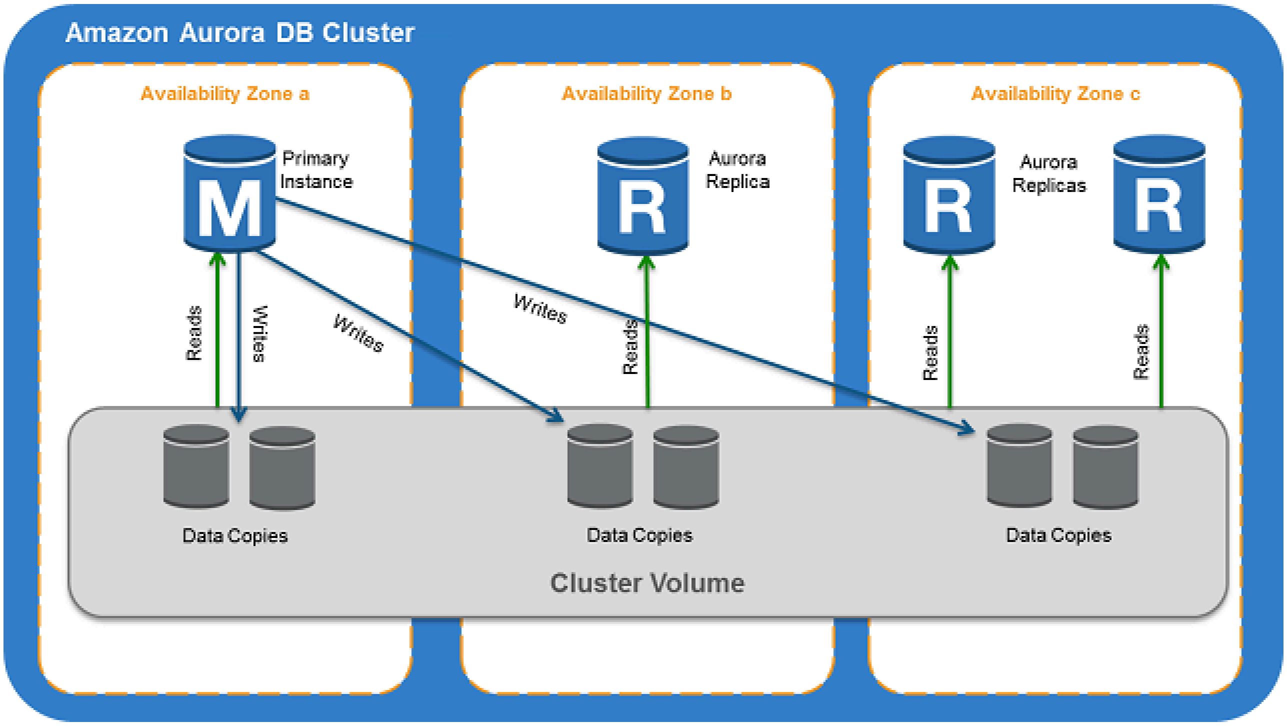
Before using Amazon Aurora, complete the steps in Setting up your environment for Amazon Aurora, and then review the concepts and features of Aurora in Amazon Aurora DB clusters.

Amazon Aurora DB clusters

An Amazon Aurora *DB cluster* consists of one or more DB instances and a cluster volume that manages the data for those DB instances. An Aurora *cluster volume* is a virtual database storage volume that spans multiple Availability Zones, with each Availability Zone having a copy of the DB cluster data. Two types of DB instances make up an Aurora DB cluster:

- **Primary DB instance** – Supports read and write operations, and performs all of the data modifications to the cluster volume. Each Aurora DB cluster has one primary DB instance.
- **Aurora Replica** – Connects to the same storage volume as the primary DB instance and supports only read operations. Each Aurora DB cluster can have up to 15 Aurora Replicas in addition to the primary DB instance. Maintain high availability by locating Aurora Replicas in separate Availability Zones. Aurora automatically fails over to an Aurora Replica in case the primary DB instance becomes unavailable. You can specify the failover priority for Aurora Replicas. Aurora Replicas can also offload read workloads from the primary DB instance.

The following diagram illustrates the relationship between the cluster volume, the primary DB instance, and Aurora Replicas in an Aurora DB cluster.



Note

The preceding information applies to provisioned clusters, parallel query clusters, global database clusters, Aurora Serverless clusters, and all MySQL 8.0-compatible, 5.7-compatible, and PostgreSQL-compatible clusters.

The Aurora cluster illustrates the separation of compute capacity and storage. For example, an Aurora configuration with only a single DB instance is still a cluster, because the underlying storage volume involves multiple storage nodes distributed across multiple Availability Zones (AZs).

Input/output (I/O) operations in Aurora DB clusters are counted the same way, regardless of whether they're on a writer or reader DB instance. For more information, see Storage configurations for Amazon Aurora DB clusters.

Amazon Aurora versions

Amazon Aurora reuses code and maintains compatibility with the underlying MySQL and PostgreSQL DB engines. However, Aurora has its own version numbers, release cycle, time line for version deprecation, and so on. The following section explains the common points and differences. This information can help you to decide such things as which version to choose and how to verify which features and fixes are available in each version. It can also help you to decide how often to upgrade and how to plan your upgrade process.

Topics

Relational databases that are available on Aurora

The following relational databases are available on Aurora:

- Amazon Aurora MySQL-Compatible Edition. For usage information, see Working with Amazon Aurora MySQL. For a detailed list of available versions, see Database engine updates for Amazon Aurora MySQL.
- Amazon Aurora PostgreSQL-Compatible Edition. For usage information, see Working with Amazon Aurora PostgreSQL. For a detailed list of available versions, see Amazon Aurora PostgreSQL updates.

Differences in version numbers between community databases and Aurora

Each Amazon Aurora version is compatible with a specific community database version of either MySQL or PostgreSQL. You can find the community version of your database using the `version` function and the Aurora version using the `aurora_version` function.

Examples for Aurora MySQL and Aurora PostgreSQL are shown following.

```
mysql> select version();
+------------------+
|   version()      |
+------------------+
|  5.7.12          |
+------------------+

mysql> select aurora_version(), @@aurora_version;
+------------------+------------------+
| aurora_version() | @@aurora_version |
+------------------+------------------+
| 2.08.1           | 2.08.1           |
+------------------+------------------+
```

```
postgres=> select version();
-----------------------------------------------------------------------------
PostgreSQL 11.7 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.9.3, 64-bit
(1 row)

postgres=> select aurora_version();
aurora_version
----------------
3.2.2
```

For more information, see Checking Aurora MySQL versions using SQL and Identifying versions of Amazon Aurora PostgreSQL.

Amazon Aurora major versions

Aurora versions use the *major*.*minor*.*patch* scheme. An *Aurora major version* refers to the MySQL or PostgreSQL community major version that Aurora is compatible with. Aurora MySQL and Aurora PostgreSQL major versions are available under standard support at least until community end of life for the corresponding community version. You can continue running a major version past its Aurora end of standard support date for a fee. For more information, see Using Amazon RDS Extended Support and Amazon Aurora pricing.

For more information on Aurora MySQL major versions and the release calendar, see *Release calendar for Aurora MySQL major versions*.

For more information on Aurora PostgreSQL major versions and the release calendar, see *Release calendar for Aurora PostgreSQL major versions*.

Note

Amazon RDS Extended Support for Aurora MySQL version 2 starts on November 1, 2024, but you won't be charged until December 1, 2024. Between November 1 and November 30, 2024, all Aurora MySQL version 2 DB clusters are covered under Amazon RDS Extended Support.
Amazon RDS Extended Support for PostgreSQL 11 starts on March 1, 2024, but you won't be charged until April 1, 2024. Between March 1 and March 31, 2024, all Aurora PostgreSQL version 11 DB clusters are covered under Amazon RDS Extended Support.

Amazon Aurora minor versions

Aurora versions use the *major*.*minor*.*patch* scheme. An *Aurora minor version* provides incremental community and Aurora-specific improvements to the service, for example new features and fixes.

For more information on Aurora MySQL minor versions and the release calendar, see *Release calendar for Aurora MySQL minor versions*.

For more information on Aurora PostgreSQL minor versions and the release calendar, see *Release calendar for Aurora PostgreSQL minor versions*.

Amazon Aurora patch versions

Aurora versions use the `major.minor.patch` scheme. An Aurora patch version includes important fixes added to a minor version after its initial release (for example, Aurora MySQL 2.10.0, 2.10.1, ..., 2.10.3). While each new minor version provides new Aurora features, new patch versions within a specific minor version are primarily used to resolve important issues.

For more information on patching, see Maintaining an Amazon Aurora DB cluster.

Learning what's new in each Amazon Aurora version

Each new Aurora version comes with release notes that list the new features, fixes, other enhancements, and so on that apply to each version.

For Aurora MySQL release notes, see *Release Notes for Aurora MySQL*. For Aurora PostgreSQL release notes, see *Release Notes for Aurora PostgreSQL*.

Specifying the Amazon Aurora database version for your database cluster

You can specify any currently available version (major and minor) when creating a new DB cluster using the **Create database** operation in the AWS Management Console, the AWS CLI, or the `CreateDBCluster` API operation. Not every Aurora database version is available in every AWS Region.

To learn how to create Aurora clusters, see Creating an Amazon Aurora DB cluster. To learn how to change the version of an existing Aurora cluster, see Modifying an Amazon Aurora DB cluster.

Default Amazon Aurora versions

When a new Aurora minor version contains significant improvements compared to a previous one, it's marked as the default version for new DB clusters. Typically, we release two default versions for each major version per year.

We recommend that you keep your DB cluster upgraded to the most current default minor version, because that version contains the latest security and functionality fixes.

Automatic minor version upgrades

You can stay up to date with Aurora minor versions by turning on **Auto minor version upgrade** for every DB instance in the Aurora cluster. Aurora only performs the automatic upgrade if all DB instances in your cluster have this setting turned on. Auto minor version upgrades are performed to the default minor version.

We typically schedule automatic upgrades twice a year for DB clusters that have the **Auto minor version upgrade** setting set to `Yes`. These upgrades are started during the maintenance window that you specify for your cluster. For more information, see Automatic minor version upgrades for Aurora DB clusters.

Automatic minor version upgrades are communicated in advance through an Amazon RDS DB cluster event with a category of `maintenance` and ID of `RDS-EVENT-0156`. For more information, see Amazon RDS event categories and event messages for Aurora.

How long Amazon Aurora major versions remain available

Amazon Aurora major versions remain available at least until community end of life for the corresponding community version. You can use Aurora end of standard support dates to plan your testing and upgrade cycles. These dates represent the earliest date that an upgrade to a newer version might be required. For more information on the dates, see Amazon Aurora major versions.

Before we ask that you upgrade to a newer major version and to help you plan, we generally provide a reminder at least 12 months in advance. We do so to communicate the detailed upgrade process. Details include the timing of certain milestones, the impact on your DB clusters, and the actions that we recommend that you take. We always recommend that you thoroughly test your applications with new database versions before performing a major version upgrade.

After the major version reaches the Aurora end of standard support, any DB cluster still running the older version will be automatically upgraded to an Extended Support version during a scheduled maintenance window. Extended Support charges may apply. For more information on Amazon RDS Extended Support, see Using Amazon RDS Extended support.

How often Amazon Aurora minor versions are released

In general, Amazon Aurora minor versions are released quarterly. The release schedule might vary to pick up additional features or fixes.

How long Amazon Aurora minor versions remain available

We intend to make each Amazon Aurora minor version of a particular major version available for at least 12 months. At the end of this period, Aurora might apply an auto minor version upgrade to the subsequent default minor version. Such an upgrade is started during the scheduled maintenance window for any cluster that is still running the older minor version.

We might replace a minor version of a particular major version sooner than the usual 12-month period if there are critical matters such as security issues, or if the major version has reached end of life.

Before beginning automatic upgrades of minor versions that are approaching end of life, we generally provide a reminder three months in advance. We do so to communicate the detailed upgrade process. Details include the timing of certain milestones, the impact on your DB clusters, and the actions that we recommend that you take. Notifications with less than three months notice are used when there are critical matters, such as security issues, that necessitate quicker action.

If you don't have the **Auto minor version upgrade** setting enabled, you get a reminder but no RDS event notification. Upgrades occur within a maintenance window after the mandatory upgrade deadline has passed.

If you do have the **Auto minor version upgrade** setting enabled, you get a reminder and an Amazon RDS DB cluster event with a category of `maintenance` and ID of `RDS-EVENT-0156`. Upgrades occur during the next maintenance window.

For more information on auto minor version upgrades, see Automatic minor version upgrades for Aurora DB clusters.

Long-term support for selected Amazon Aurora minor versions

For each Aurora major version, certain minor versions are designated as long-term-support (LTS) versions and made available for at least three years. That is, at least one minor version per major version is made available for longer than the typical 12 months. We generally provide a reminder six months before the end of this period. We do so to communicate the detailed upgrade process. Details include the timing of certain milestones, the impact on your DB clusters, and the actions that we recommend that you take. Notifications with less than six months notice are used when there are critical matters, such as security issues, that necessitate quicker action.

LTS minor versions include only critical fixes (through patch versions). An LTS version doesn't include new features released after its introduction. Once a year, DB clusters running on an LTS minor version are patched to the latest patch version of the LTS release. We do this patching to help ensure that you benefit from cumulative security and stability fixes. We might patch an LTS minor version more frequently if there are critical fixes, such as for security, that need to be applied.

> **Note**
>
> If you want to remain on an LTS minor version for the duration of its lifecycle, make sure to turn off **Auto minor version upgrade** for your DB instances. To avoid automatically upgrading your DB cluster from the LTS minor version, set **Auto minor version upgrade** to No on any DB instance in your Aurora cluster.

For the version numbers of all Aurora LTS versions, see Aurora MySQL long-term support (LTS) releases and Aurora PostgreSQL long-term support (LTS) releases.

Amazon RDS Extended Support for selected Aurora versions

With Amazon RDS Extended Support, you can continue running your database on a major engine version past the Aurora end of standard support date for an additional cost. During RDS Extended Support, Amazon RDS will supply patches for Critical and High CVEs as defined by the National Vulnerability Database (NVD) CVSS severity ratings. For more information, see Using Amazon RDS Extended Support.

RDS Extended Support is only available on certain Aurora versions. For more information, see Amazon Aurora major versions.

Manually controlling if and when your database cluster is upgraded to new versions

Auto minor version upgrades are performed to the default minor version. We typically schedule automatic upgrades twice a year for DB clusters that have the **Auto minor version upgrade** setting enabled. These upgrades are started during customer-specified maintenance windows. If you want to turn off automatic minor version upgrades, disable **Auto minor version upgrade** on any DB instance within an Aurora cluster. Aurora performs an automatic minor version upgrade only if all DB instances in your cluster have the setting enabled.

> **Note**
>
> However, for mandatory upgrades such as minor-version end of life, the DB cluster will be upgraded even if the **Auto minor version upgrade** setting is disabled. You get a reminder but no RDS event notification. Upgrades occur within a maintenance window after the mandatory upgrade deadline has passed.

Because major version upgrades involve some compatibility risk, they don't occur automatically. You must initiate these, except in the case of a major version deprecation, as explained earlier. We always recommend that you thoroughly test your applications with new database versions before performing a major version upgrade.

For more information about upgrading a DB cluster to a new Aurora major version, see Upgrading Amazon Aurora MySQL DB clusters and Upgrading Amazon Aurora PostgreSQL DB clusters.

Required Amazon Aurora upgrades

For certain critical fixes, we might perform a managed upgrade to a newer patch level within the same minor version. These required upgrades happen even if **Auto minor version upgrade** is turned off. Before doing so, we communicate the detailed upgrade process. Details include the timing of certain milestones, the impact on your DB clusters, and the actions that we recommend that you take. Such managed upgrades are performed automatically. Each such upgrade is started within the cluster maintenance window.

Testing your DB cluster with a new Aurora version before upgrading

You can test the upgrade process and how the new version works with your application and workload. Use one of the following methods:

- Clone your cluster using the Amazon Aurora fast database clone feature. Perform the upgrade and any post-upgrade testing on the new cluster.
- Restore from a cluster snapshot to create a new Aurora cluster. You can create a cluster snapshot yourself from an existing Aurora cluster. Aurora also automatically creates periodic snapshots for you for each of your clusters. You can then initiate a version upgrade for the new cluster. You can experiment on the upgraded copy of your cluster before deciding whether to upgrade your original cluster.

For more information on these ways to create new clusters for testing, see Cloning a volume for an Amazon Aurora DB cluster and Creating a DB cluster snapshot.

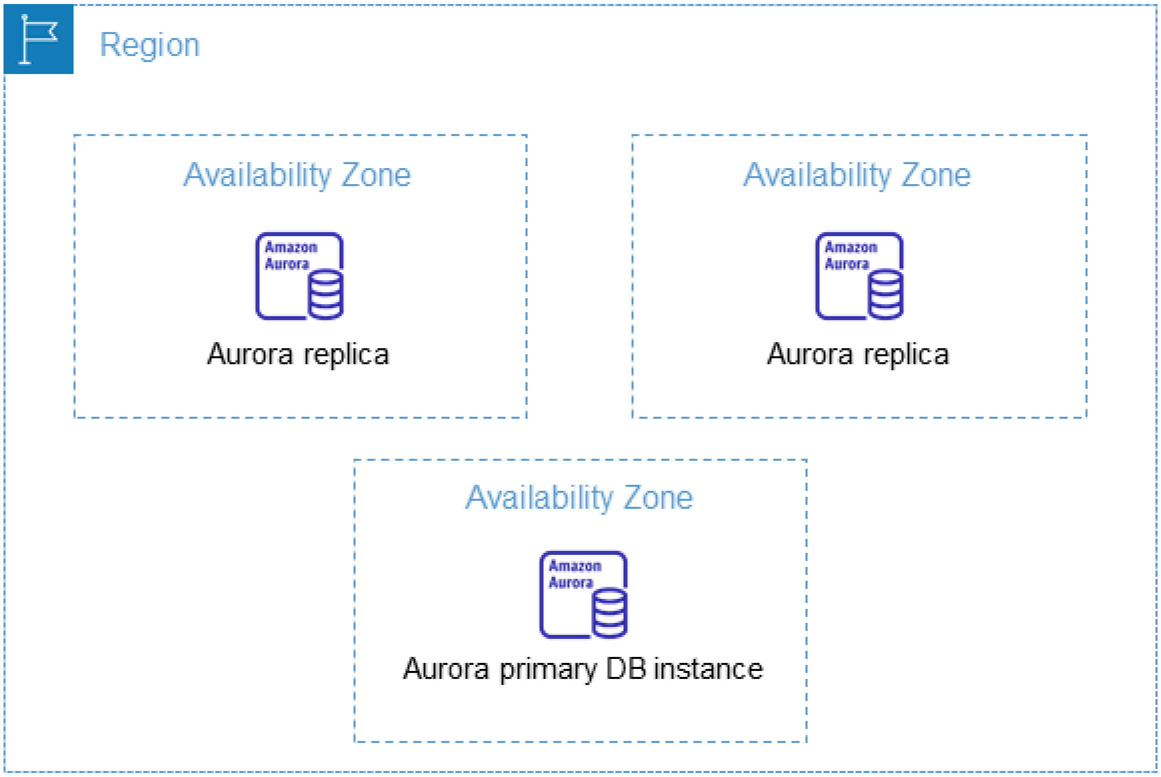
Regions and Availability Zones

Amazon cloud computing resources are hosted in multiple locations world-wide. These locations are composed of AWS Regions and Availability Zones. Each *AWS Region* is a separate geographic area. Each AWS Region has multiple, isolated locations known as *Availability Zones*.

Note

For information about finding the Availability Zones for an AWS Region, see Describe your Availability Zones in the Amazon EC2 documentation.

Amazon operates state-of-the-art, highly-available data centers. Although rare, failures can occur that affect the availability of DB instances that are in the same location. If you host all your DB instances in one location that is affected by such a failure, none of your DB instances will be available.



It is important to remember that each AWS Region is completely independent. Any Amazon RDS activity you initiate (for example, creating database instances or listing available database instances) runs only in your current default AWS Region. The default AWS Region can be changed in the console, or by setting the `AWS_DEFAULT_REGION` environment variable. Or it can be overridden by using the `--region` parameter with the AWS Command Line Interface (AWS CLI). For more information, see Configuring the AWS Command Line Interface, specifically the sections about environment variables and command line options.

Amazon RDS supports special AWS Regions called AWS GovCloud (US). These are designed to allow US government agencies and customers to move more sensitive workloads into the cloud. The AWS GovCloud (US) Regions address the US government's specific regulatory and compliance requirements. For more information, see What is AWS GovCloud (US)?

To create or work with an Amazon RDS DB instance in a specific AWS Region, use the corresponding regional service endpoint.

> **Note**
>
> Aurora doesn't support Local Zones.

AWS Regions

Each AWS Region is designed to be isolated from the other AWS Regions. This design achieves the greatest possible fault tolerance and stability.

When you view your resources, you see only the resources that are tied to the AWS Region that you specified. This is because AWS Regions are isolated from each other, and we don't automatically replicate resources across AWS Regions.

Region availability

When you work with an Aurora DB cluster using the command line interface or API operations, make sure that you specify its regional endpoint.

Topics

Aurora MySQL Region availability

The following table shows the AWS Regions where Aurora MySQL is currently available and the endpoint for each Region.

Region Name	Region	Endpoint	Protocol	
US East (Ohio)	us-east-2	rds.us-east-2.amazonaws.com	HTTPS	
US East (N. Virginia)	us-east-1	rds.us-east-1.amazonaws.com	HTTPS	
US West (N. California)	us-west-1	rds.us-west-1.amazonaws.com	HTTPS	
US West (Oregon)	us-west-2	rds.us-west-2.amazonaws.com	HTTPS	
Africa (Cape Town)	af-south-1	rds.af-south-1.amazonaws.com	HTTPS	
Asia Pacific (Hong Kong)	ap-east-1	rds.ap-east-1.amazonaws.com	HTTPS	
Asia Pacific (Hyderabad)	ap-south-2	rds.ap-south-2.amazonaws.com	HTTPS	

Region Name	Region	Endpoint	Protocol
Asia Pacific (Jakarta)	ap-southeast-3	rds.ap-southeast-3.amazonaws.com	HTTPS
Asia Pacific (Melbourne)	ap-southeast-4	rds.ap-southeast-4.amazonaws.com	HTTPS
Asia Pacific (Mumbai)	ap-south-1	rds.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (Osaka)	ap-northeast-3	rds.ap-northeast-3.amazonaws.com	HTTPS
Asia Pacific (Seoul)	ap-northeast-2	rds.ap-northeast-2.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	rds.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	rds.ap-southeast-2.amazonaws.com	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	rds.ap-northeast-1.amazonaws.com	HTTPS
Canada (Central)	ca-central-1	rds.ca-central-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Canada West (Calgary)	ca-west-1	rds.ca-west-1.amazonaws.com	HTTPS
Europe (Frankfurt)	eu-central-1	rds.eu-central-1.amazonaws.com	HTTPS
Europe (Ireland)	eu-west-1	rds.eu-west-1.amazonaws.com	HTTPS
Europe (London)	eu-west-2	rds.eu-west-2.amazonaws.com	HTTPS
Europe (Milan)	eu-south-1	rds.eu-south-1.amazonaws.com	HTTPS
Europe (Paris)	eu-west-3	rds.eu-west-3.amazonaws.com	HTTPS
Europe (Spain)	eu-south-2	rds.eu-south-2.amazonaws.com	HTTPS
Europe (Stockholm)	eu-north-1	rds.eu-north-1.amazonaws.com	HTTPS
Europe (Zurich)	eu-central-2	rds.eu-central-2.amazonaws.com	HTTPS
Israel (Tel Aviv)	il-central-1	rds.il-central-1.amazonaws.com	HTTPS
Middle East (Bahrain)	me-south-1	rds.me-south-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Middle East (UAE)	me-central-1	rds.me-central-1.amazonaws.com	HTTPS
South America (São Paulo)	sa-east-1	rds.sa-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-East)	us-gov-east-1	rds.us-gov-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-West)	us-gov-west-1	rds.us-gov-west-1.amazonaws.com	HTTPS

Aurora PostgreSQL Region availability

The following table shows the AWS Regions where Aurora PostgreSQL is currently available and the endpoint for each Region.

Region Name	Region	Endpoint	Protocol
US East (Ohio)	us-east-2	rds.us-east-2.amazonaws.com	HTTPS
US East (N. Virginia)	us-east-1	rds.us-east-1.amazonaws.com	HTTPS
US West (N.	us-west-1	rds.us-west-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
California)			
US West (Oregon)	us-west-2	rds.us-west-2.amazonaws.com	HTTPS
Africa (Cape Town)	af-south-1	rds.af-south-1.amazonaws.com	HTTPS
Asia Pacific (Hong Kong)	ap-east-1	rds.ap-east-1.amazonaws.com	HTTPS
Asia Pacific (Hyderabad)	ap-south-2	rds.ap-south-2.amazonaws.com	HTTPS
Asia Pacific (Jakarta)	ap-southeast-3	rds.ap-southeast-3.amazonaws.com	HTTPS
Asia Pacific (Melbourne)	ap-southeast-4	rds.ap-southeast-4.amazonaws.com	HTTPS
Asia Pacific (Mumbai)	ap-south-1	rds.ap-south-1.amazonaws.com	HTTPS
Asia Pacific (Osaka)	ap-northeast-3	rds.ap-northeast-3.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Seoul)	ap-northeast-2	rds.ap-northeast-2.amazonaws.com	HTTPS
Asia Pacific (Singapore)	ap-southeast-1	rds.ap-southeast-1.amazonaws.com	HTTPS
Asia Pacific (Sydney)	ap-southeast-2	rds.ap-southeast-2.amazonaws.com	HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	rds.ap-northeast-1.amazonaws.com	HTTPS
Canada (Central)	ca-central-1	rds.ca-central-1.amazonaws.com	HTTPS
Canada West (Calgary)	ca-west-1	rds.ca-west-1.amazonaws.com	HTTPS
Europe (Frankfurt)	eu-central-1	rds.eu-central-1.amazonaws.com	HTTPS
Europe (Ireland)	eu-west-1	rds.eu-west-1.amazonaws.com	HTTPS
Europe (London)	eu-west-2	rds.eu-west-2.amazonaws.com	HTTPS
Europe (Milan)	eu-south-1	rds.eu-south-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
Europe (Paris)	eu-west-3	rds.eu-west-3.amazonaws.com	HTTPS
Europe (Spain)	eu-south-2	rds.eu-south-2.amazonaws.com	HTTPS
Europe (Stockholm)	eu-north-1	rds.eu-north-1.amazonaws.com	HTTPS
Europe (Zurich)	eu-central-2	rds.eu-central-2.amazonaws.com	HTTPS
Israel (Tel Aviv)	il-central-1	rds.il-central-1.amazonaws.com	HTTPS
Middle East (Bahrain)	me-south-1	rds.me-south-1.amazonaws.com	HTTPS
Middle East (UAE)	me-central-1	rds.me-central-1.amazonaws.com	HTTPS
South America (São Paulo)	sa-east-1	rds.sa-east-1.amazonaws.com	HTTPS
AWS GovCloud (US-East)	us-gov-east-1	rds.us-gov-east-1.amazonaws.com	HTTPS

Region Name	Region	Endpoint	Protocol
AWS GovCloud (US-West)	us-gov-west-1	rds.us-gov-west-1.amazonaws.com	HTTPS

Availability Zones

An Availability Zone is an isolated location in a given AWS Region. Each Region has multiple Availability Zones (AZ) designed to provide high availability for the Region. An AZ is identified by the AWS Region code followed by a letter identifier (for example, `us-east-1a`). If you create your VPC and subnets rather than using the default VPC, you define each subnet in a specific AZ. When you create an Aurora DB cluster, Aurora creates the primary instance in one of the subnets in the VPC's DB subnet group. It thus associates that instance with a specific AZ chosen by Aurora.

Each Aurora DB cluster hosts copies of its storage in three separate AZs selected automatically by Aurora from the AZs in your DB subnet group. Every DB instance in the cluster must be in one of these three AZs.

When you create a DB instance in your cluster, Aurora automatically chooses an appropriate AZ for that instance if you don't specify an AZ.

Use the describe-availability-zones Amazon EC2 command as follows to describe the Availability Zones within the specified Region that are enabled for your account.

```
aws ec2 describe-availability-zones --region region-name
```

For example, to describe the Availability Zones within the US East (N. Virginia) Region (us-east-1) that are enabled for your account, run the following command:

```
aws ec2 describe-availability-zones --region us-east-1
```

To learn how to specify the AZ when you create a cluster or add instances to it, see Configure the network for the DB cluster.

Local time zone for Amazon Aurora DB clusters

By default, the time zone for an Amazon Aurora DB cluster is Universal Time Coordinated (UTC). You can set the time zone for instances in your DB cluster to the local time zone for your application instead.

To set the local time zone for a DB cluster, set the time zone parameter to one of the supported values. You set this parameter in the cluster parameter group for your DB cluster.

- For Aurora MySQL, the name of this parameter is `time_zone`. For information on best practices for setting the `time_zone` parameter, see Optimizing timestamp operations.
- For Aurora PostgreSQL, the name of this parameter is `timezone`.

When you set the time zone parameter for a DB cluster, all instances in the DB cluster change to use the new local time zone. In some cases, other Aurora DB clusters might be using the same cluster parameter group. If so, all instances in those DB clusters change to use the new local time zone also. For information on cluster-level parameters, see Amazon Aurora DB cluster and DB instance parameters.

After you set the local time zone, all new connections to the database reflect the change. In some cases, you might have open connections to your database when you change the local time zone. If so, you don't see the local time zone update until after you close the connection and open a new connection.

If you are replicating across AWS Regions, the replication source DB cluster and the replica use different parameter groups. Parameter groups are unique to an AWS Region. To use the same local time zone for each instance, make sure to set the time zone parameter in the parameter groups for both the replication source and the replica.

When you restore a DB cluster from a DB cluster snapshot, the local time zone is set to UTC. You can update the time zone to your local time zone after the restore is complete. In some cases, you might restore a DB cluster to a point in time. If so, the local time zone for the restored DB cluster is the time zone setting from the parameter group of the restored DB cluster.

The following table lists some of the values to which you can set your local time zone. To list all of the available time zones, you can use the following SQL queries:

- Aurora MySQL: `select * from mysql.time_zone_name;`
- Aurora PostgreSQL: `select * from pg_timezone_names;`

Note

For some time zones, time values for certain date ranges can be reported incorrectly as noted in the table. For Australia time zones, the time zone abbreviation returned is an outdated value as noted in the table.

Time zone	Notes
`Africa/Harare`	This time zone setting can return incorrect values from 28 Feb 1903 21:49:40 GMT to 28 Feb 1903 21:55:48 GMT.
`Africa/Monrovia`	
`Africa/Nairobi`	This time zone setting can return incorrect values from 31 Dec 1939 21:30:00 GMT to 31 Dec 1959 21:15:15 GMT.
`Africa/Windhoek`	
`America/Bogota`	This time zone setting can return incorrect values from 23 Nov 1914 04:56:16 GMT to 23 Nov 1914 04:56:20 GMT.
`America/Caracas`	
`America/Chihuahua`	
`America/Cuiaba`	
`America/Denver`	
`America/Fortaleza`	In some cases, for a DB cluster in the South America (Sao Paulo) Region, time doesn't show correctly for a recently changed Brazil time zone. If so, reset the DB cluster's time zone parameter to `America/Fortaleza` .
`America/Guatemala`	

Time zone	Notes
`America/Halifax`	This time zone setting can return incorrect values from 27 Oct 1918 05:00:00 GMT to 31 Oct 1918 05:00:00 GMT.
`America/Manaus`	If your DB cluster is in the South America (Cuiaba) time zone and the expected time doesn't show correctly for the recently changed Brazil time zone, reset the DB cluster's time zone parameter to `America/Manaus` .
`America/Matamoros`	
`America/Monterrey`	
`America/Montevideo`	
`America/Phoenix`	
`America/Tijuana`	
`Asia/Ashgabat`	
`Asia/Baghdad`	
`Asia/Baku`	
`Asia/Bangkok`	
`Asia/Beirut`	
`Asia/Calcutta`	
`Asia/Kabul`	
`Asia/Karachi`	
`Asia/Kathmandu`	

Time zone	Notes
`Asia/Muscat`	This time zone setting can return incorrect values from 31 Dec 1919 20:05:36 GMT to 31 Dec 1919 20:05:40 GMT.
`Asia/Riyadh`	This time zone setting can return incorrect values from 13 Mar 1947 20:53:08 GMT to 31 Dec 1949 20:53:08 GMT.
`Asia/Seoul`	This time zone setting can return incorrect values from 30 Nov 1904 15:30:00 GMT to 07 Sep 1945 15:00:00 GMT.
`Asia/Shanghai`	This time zone setting can return incorrect values from 31 Dec 1927 15:54:08 GMT to 02 Jun 1940 16:00:00 GMT.
`Asia/Singapore`	
`Asia/Taipei`	This time zone setting can return incorrect values from 30 Sep 1937 16:00:00 GMT to 29 Sep 1979 15:00:00 GMT.
`Asia/Tehran`	
`Asia/Tokyo`	This time zone setting can return incorrect values from 30 Sep 1937 15:00:00 GMT to 31 Dec 1937 15:00:00 GMT.
`Asia/Ulaanbaatar`	
`Atlantic/Azores`	This time zone setting can return incorrect values from 24 May 1911 01:54:32 GMT to 01 Jan 1912 01:54:32 GMT.
`Australia/Adelaide`	The abbreviation for this time zone is returned as CST instead of ACDT/ACST.
`Australia/Brisbane`	The abbreviation for this time zone is returned as EST instead of AEDT/AEST.
`Australia/Darwin`	The abbreviation for this time zone is returned as CST instead of ACDT/ACST.

Time zone	Notes
`Australia/Hobart`	The abbreviation for this time zone is returned as EST instead of AEDT/AEST.
`Australia/Perth`	The abbreviation for this time zone is returned as WST instead of AWDT/AWST.
`Australia/Sydney`	The abbreviation for this time zone is returned as EST instead of AEDT/AEST.
`Brazil/East`	
`Canada/Saskatchewan`	This time zone setting can return incorrect values from 27 Oct 1918 08:00:00 GMT to 31 Oct 1918 08:00:00 GMT.
`Europe/Amsterdam`	
`Europe/Athens`	
`Europe/Dublin`	
`Europe/Helsinki`	This time zone setting can return incorrect values from 30 Apr 1921 22:20:08 GMT to 30 Apr 1921 22:20:11 GMT.
`Europe/Paris`	
`Europe/Prague`	
`Europe/Sarajevo`	
`Pacific/Auckland`	
`Pacific/Guam`	
`Pacific/Honolulu`	This time zone setting can return incorrect values from 21 May 1933 11:30:00 GMT to 30 Sep 1945 11:30:00 GMT.

Time zone	Notes
`Pacific/Samoa`	This time zone setting can return incorrect values from 01 Jan 1911 11:22:48 GMT to 01 Jan 1950 11:30:00 GMT.
`US/Alaska`	
`US/Central`	
`US/Eastern`	
`US/East-Indiana`	
`US/Pacific`	
`UTC`	

Supported features in Amazon Aurora by AWS Region and Aurora DB engine

Aurora MySQL- and PostgreSQL-compatible database engines support several Amazon Aurora and Amazon RDS features and options. The support varies across specific versions of each database engine, and across AWS Regions. To identify Aurora database engine version support and availability for a feature in a given AWS Region, you can use the following sections.

Some of these features are Aurora-only capabilities. For example, Aurora Serverless, Aurora global databases, and support for integration with AWS machine learning services aren't supported by Amazon RDS. Other features, such as Amazon RDS Proxy, are supported by both Amazon Aurora and Amazon RDS.

Supported Regions and DB engines

Table conventions

The tables in the feature sections use the following patterns to specify version numbers and level of support:

- **Version x.y** – The specific version alone is supported.
- **Version x.y and higher** – The specified version and all higher minor versions of its major version are supported. For example, "version 10.11 and higher" means that versions 10.11, 10.11.1, and 10.12 are supported.
- **-** – The feature is not currently available for that particular Aurora feature for the given Aurora database engine, or in that specific AWS Region.

Supported Regions and Aurora DB engines for Blue/Green Deployments

A blue/green deployment copies a production database environment in a separate, synchronized staging environment. By using Amazon RDS Blue/Green Deployments, you can make changes to the database in the staging environment without affecting the production environment. For example, you can upgrade the major or minor DB engine version, change database parameters, or make schema changes in the staging environment. When you are ready, you can promote the staging environment to be the new production database environment. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.

Blue/Green Deployments with Aurora MySQL

The Blue/Green Deployments feature is available for all versions of Aurora MySQL in all AWS Regions.

Blue/Green Deployments with Aurora PostgreSQL

The following Regions and engine versions are available for Blue/Green Deployments with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
All AWS Regions	Version 16.1 and higher	Version 15.4 and higher	Version 14.9 and higher	Version 13.12 and higher	Version 12.16 and higher	Version 11.21 and higher

Supported Regions and Aurora DB engines for cluster storage configurations

Amazon Aurora has two DB cluster storage configurations, Aurora I/O-Optimized and Aurora Standard. For more information, see Storage configurations for Amazon Aurora DB clusters.

Aurora I/O-Optimized

Aurora I/O-Optimized is available in all AWS Regions for the following Amazon Aurora versions:

- Aurora MySQL version 3.03.1 and higher
- Aurora PostgreSQL versions 16.1 and higher, 15.2 and higher, 14.7 and higher, and 13.10 and higher

Aurora Standard

Aurora Standard is available in all AWS Regions for all Aurora MySQL and Aurora PostgreSQL versions.

Supported Regions and Aurora DB engines for database activity streams

By using database activity streams in Aurora, you can monitor and set alarms for auditing activity in your Aurora database. For more information, see Monitoring Amazon Aurora with Database Activity Streams.

Database activity streams aren't supported for the following features:

- Aurora Serverless v1

- Aurora Serverless v2
- Babelfish for Aurora PostgreSQL

Topics

Database activity streams with Aurora MySQL

The following Regions and engine versions are available for database activity streams with Aurora MySQL.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	All available versions	Aurora version 2.11 and higher
US East (N. Virginia)	All available versions	Aurora version 2.11 and higher
US West (N. California)	All available versions	Aurora version 2.11 and higher
US West (Oregon)	All available versions	Aurora version 2.11 and higher
Africa (Cape Town)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Hong Kong)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Hyderabad)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Jakarta)	All available versions	Aurora version 2.11 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Mumbai)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Osaka)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Seoul)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Singapore)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Sydney)	All available versions	Aurora version 2.11 and higher
Asia Pacific (Tokyo)	All available versions	Aurora version 2.11 and higher
Canada (Central)	All available versions	Aurora version 2.11 and higher
Canada West (Calgary)	–	–
China (Beijing)	–	–
China (Ningxia)	–	–
Europe (Frankfurt)	All available versions	Aurora version 2.11 and higher
Europe (Ireland)	All available versions	Aurora version 2.11 and higher
Europe (London)	All available versions	Aurora version 2.11 and higher
Europe (Milan)	All available versions	Aurora version 2.11 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Europe (Paris)	All available versions	Aurora version 2.11 and higher
Europe (Spain)	All available versions	Aurora version 2.11 and higher
Europe (Stockholm)	All available versions	Aurora version 2.11 and higher
Europe (Zurich)	–	–
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	All available versions	Aurora version 2.11 and higher
Middle East (UAE)	All available versions	Aurora version 2.11 and higher
South America (São Paulo)	All available versions	Aurora version 2.11 and higher

Database activity streams with Aurora PostgreSQL

The following Regions and engine versions are available for database activity streams with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (Ohio)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (N. Virginia)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
US West (N. California)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
US West (Oregon)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Africa (Cape Town)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Hong Kong)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Hyderabad)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Jakarta)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Melbourne)	–	–	–	–	–	–
Asia Pacific (Mumbai)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Osaka)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Seoul)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Singapore)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Sydney)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Canada (Central)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Canada West (Calgary)	–	–	–	–	–	–
China (Beijing)	–	–	–	–	–	–
China (Ningxia)	–	–	–	–	–	–

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Frankfurt)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (Ireland)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (London)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (Milan)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (Paris)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Spain)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (Stockholm)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Europe (Zurich)	–	–	–	–	–	–
Israel (Tel Aviv)	–	–	–	–	–	–
Middle East (Bahrain)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher
Middle East (UAE)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
South America (São Paulo)	Version 16.1 and higher	Version 15.2 and higher	All available versions	All available versions	All available versions	Version 11.9 and version 11.13 and higher

Supported Regions and Aurora DB engines for exporting cluster data to Amazon S3

You can export Aurora DB cluster data to an Amazon S3 bucket. After the data is exported, you can analyze the exported data directly through tools like Amazon Athena or Amazon Redshift Spectrum. For more information, see Exporting DB cluster data to Amazon S3.

Exporting cluster data to S3 is available in the following AWS Regions:

- Asia Pacific (Hong Kong)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Canada West (Calgary)
- China (Ningxia)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Europe (Stockholm)

- South America (São Paulo)
- US East (N. Virginia)
- US East (Ohio)
- US West (N. California)
- US West (Oregon)

Topics

- Exporting cluster data to S3 with Aurora MySQL
- Exporting cluster data to S3 with Aurora PostgreSQL

Exporting cluster data to S3 with Aurora MySQL

All currently available Aurora MySQL engine versions support exporting DB cluster data to Amazon S3. For more information about versions, see *Release Notes for Aurora MySQL*.

Exporting cluster data to S3 with Aurora PostgreSQL

All currently available Aurora PostgreSQL engine versions support exporting DB cluster data to Amazon S3. For more information about versions, see the *Release Notes for Aurora PostgreSQL*.

Supported Regions and Aurora DB engines for exporting snapshot data to Amazon S3

You can export Aurora DB cluster snapshot data to an Amazon S3 bucket. You can export manual snapshots and automated system snapshots. After the data is exported, you can analyze the exported data directly through tools like Amazon Athena or Amazon Redshift Spectrum. For more information, see Exporting DB cluster snapshot data to Amazon S3.

Exporting snapshots to S3 is available in all AWS Regions except the following:

- Asia Pacific (Hyderabad)
- Asia Pacific (Jakarta)
- Asia Pacific (Melbourne)
- Canada West (Calgary)
- Europe (Spain)
- Europe (Zurich)

- Israel (Tel Aviv)
- Middle East (UAE)
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)

Topics

- Exporting snapshot data to S3 with Aurora MySQL
- Exporting snapshot data to S3 with Aurora PostgreSQL

Exporting snapshot data to S3 with Aurora MySQL

All currently available Aurora MySQL engine versions support exporting DB cluster snapshot data to Amazon S3. For more information about versions, see *Release Notes for Aurora MySQL*.

Exporting snapshot data to S3 with Aurora PostgreSQL

All currently available Aurora PostgreSQL engine versions support exporting DB cluster snapshot data to Amazon S3. For more information about versions, see the *Release Notes for Aurora PostgreSQL*.

Supported Regions and DB engines for Aurora global databases

An *Aurora global database* is a single database that spans multiple AWS Regions, enabling low-latency global reads and disaster recovery from any Region-wide outage. It provides built-in fault tolerance for your deployment because the DB instance relies not on a single AWS Region, but upon multiple Regions and different Availability Zones. For more information, see Using Amazon Aurora global databases.

Topics

- Aurora global databases with Aurora MySQL
- Aurora global databases with Aurora PostgreSQL

Aurora global databases with Aurora MySQL

The following Regions and engine versions are available for Aurora global databases with Aurora MySQL.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	Version 3.01.0 and higher	Version 2.07.0 and higher
US East (N. Virginia)	Version 3.01.0 and higher	Version 2.07.0 and higher
US West (N. California)	Version 3.01.0 and higher	Version 2.07.0 and higher
US West (Oregon)	Version 3.01.0 and higher	Version 2.07.0 and higher
Africa (Cape Town)	Version 3.01.0 and higher	Version 2.07.1 and higher
Asia Pacific (Hong Kong)	Version 3.01.0 and higher	Version 2.07.1 and higher
Asia Pacific (Hyderabad)	Version 3.02.0 and higher	Version 2.11.2 and higher
Asia Pacific (Jakarta)	Version 3.01.0 and higher	Version 2.07.6 and higher
Asia Pacific (Melbourne)	Version 3.03.0 and higher	–
Asia Pacific (Mumbai)	Version 3.01.0 and higher	Version 2.07.0 and higher
Asia Pacific (Osaka)	Version 3.01.0 and higher	Version 2.07.3 and higher
Asia Pacific (Seoul)	Version 3.01.0 and higher	Version 2.07.0 and higher
Asia Pacific (Singapore)	Version 3.01.0 and higher	Version 2.07.0 and higher
Asia Pacific (Sydney)	Version 3.01.0 and higher	Version 2.07.0 and higher
Asia Pacific (Tokyo)	Version 3.01.0 and higher	Version 2.07.0 and higher
Canada (Central)	Version 3.01.0 and higher	Version 2.07.0 and higher
Canada West (Calgary)	Version 3.01.0 and higher	Version 2.07.0 and higher
China (Beijing)	Version 3.01.0 and higher	Version 2.07.2 and higher
China (Ningxia)	Version 3.01.0 and higher	Version 2.07.2 and higher
Europe (Frankfurt)	Version 3.01.0 and higher	Version 2.07.0 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Europe (Ireland)	Version 3.01.0 and higher	Version 2.07.0 and higher
Europe (London)	Version 3.01.0 and higher	Version 2.07.0 and higher
Europe (Milan)	Version 3.01.0 and higher	Version 2.07.1 and higher
Europe (Paris)	Version 3.01.0 and higher	Version 2.07.0 and higher
Europe (Spain)	Version 3.02.0 and higher	–
Europe (Stockholm)	Version 3.01.0 and higher	Version 2.07.0 and higher
Europe (Zurich)	Version 3.02.0 and higher	–
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	Version 3.01.0 and higher	Version 2.07.1 and higher
Middle East (UAE)	Version 3.02.0 and higher	–
South America (São Paulo)	Version 3.01.0 and higher	Version 2.07.1 and higher
AWS GovCloud (US-East)	Version 3.01.0 and higher	Version 2.07.0 and higher
AWS GovCloud (US-West)	Version 3.01.0 and higher	Version 2.07.0 and higher

Aurora global databases with Aurora PostgreSQL

The following Regions and engine versions are available for Aurora global databases with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (Ohio)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
						11.13 and higher
US East (N. Virginia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
US West (N. California)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
US West (Oregon)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Africa (Cape Town)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Hong Kong)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Hyderabad)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Jakarta)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Melbourne)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Mumbai)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Osaka)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Seoul)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Singapore)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Sydney)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Canada (Central)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Canada West (Calgary)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
China (Beijing)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
China (Ningxia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Frankfurt)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Ireland)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (London)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Milan)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Paris)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Spain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Stockholm)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Zurich)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Israel (Tel Aviv)	–	–	–	–	–	–
Middle East (Bahrain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Middle East (UAE)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
South America (São Paulo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
AWS GovCloud (US-East)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
AWS GovCloud (US-West)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Supported Regions and Aurora DB engines for IAM database authentication

With IAM database authentication in Aurora, you can authenticate to your DB cluster using AWS Identity and Access Management (IAM) database authentication. With this authentication method, you don't need to use a password when you connect to a DB cluster. Instead, you use an authentication token. For more information, see IAM database authentication.

Topics

- IAM database authentication with Aurora MySQL
- IAM database authentication with Aurora PostgreSQL

IAM database authentication with Aurora MySQL

IAM database authentication with Aurora MySQL is available in all Regions for the following versions:

- Aurora MySQL 3 – All available versions
- Aurora MySQL 2 – All available versions

IAM database authentication with Aurora PostgreSQL

IAM database authentication with Aurora PostgreSQL is available in all Regions for the following engine versions:

- Aurora PostgreSQL 16 – All available versions

- Aurora PostgreSQL 15 – All available versions
- Aurora PostgreSQL 14 – All available versions
- Aurora PostgreSQL 13 – All available versions
- Aurora PostgreSQL 12 – All available versions
- Aurora PostgreSQL 11 – All available versions

Supported Regions and Aurora DB engines for Kerberos authentication

By using Kerberos authentication with Aurora, you can support external authentication of database users using Kerberos and Microsoft Active Directory. Using Kerberos and Active Directory provides the benefits of single sign-on and centralized authentication of database users. Kerberos and Active Directory are available with AWS Directory Service for Microsoft Active Directory, a feature of AWS Directory Service. For more information, see Kerberos authentication.

Topics

- Kerberos authentication with Aurora MySQL
- Kerberos authentication with Aurora PostgreSQL

Kerberos authentication with Aurora MySQL

The following Regions and engine versions are available for Kerberos Authentication with Aurora MySQL.

Region	Aurora MySQL version 3
US East (Ohio)	Version 3.03.0 and higher
US East (N. Virginia)	Version 3.03.0 and higher
US West (N. California)	Version 3.03.0 and higher
US West (Oregon)	Version 3.03.0 and higher
Africa (Cape Town)	–
Asia Pacific (Hong Kong)	–

Region	Aurora MySQL version 3
Asia Pacific (Jakarta)	–
Asia Pacific (Mumbai)	Version 3.03.0 and higher
Asia Pacific (Osaka)	–
Asia Pacific (Seoul)	Version 3.03.0 and higher
Asia Pacific (Singapore)	Version 3.03.0 and higher
Asia Pacific (Sydney)	Version 3.03.0 and higher
Asia Pacific (Tokyo)	Version 3.03.0 and higher
Canada (Central)	Version 3.03.0 and higher
Canada West (Calgary)	–
China (Beijing)	Version 3.03.0 and higher
China (Ningxia)	Version 3.03.0 and higher
Europe (Frankfurt)	Version 3.03.0 and higher
Europe (Ireland)	Version 3.03.0 and higher
Europe (London)	Version 3.03.0 and higher
Europe (Milan)	–
Europe (Paris)	Version 3.03.0 and higher
Europe (Spain)	–
Europe (Stockholm)	Version 3.03.0 and higher
Europe (Zurich)	–
Israel (Tel Aviv)	–

Region	Aurora MySQL version 3
Middle East (Bahrain)	–
Middle East (UAE)	–
South America (São Paulo)	Version 3.03.0 and higher
AWS GovCloud (US-East)	Version 3.03.0 and higher
AWS GovCloud (US-West)	Version 3.03.0 and higher

Kerberos authentication with Aurora PostgreSQL

The following Regions and engine versions are available for Kerberos Authentication with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (Ohio)	All versions	All versions	All versions	All versions	All versions	All versions
US East (N. Virginia)	All versions	All versions	All versions	All versions	All versions	All versions
US West (N. California)	All versions	All versions	All versions	All versions	All versions	All versions
US West (Oregon)	All versions	All versions	All versions	All versions	All versions	All versions
Africa (Cape Town)	–	–	–	–	–	–

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Hong Kong)	–	–	–	–	–	–
Asia Pacific (Hyderabad)	–	–	–	–	–	–
Asia Pacific (Jakarta)	–	–	–	–	–	–
Asia Pacific (Melbourne)	–	–	–	–	–	–
Asia Pacific (Mumbai)	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Osaka)	–	–	–	–	–	–
Asia Pacific (Seoul)	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Singapore)	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Sydney)	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Tokyo)	All versions	All versions	All versions	All versions	All versions	All versions

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Canada (Central)	All versions	All versions	All versions	All versions	All versions	All versions
Canada West (Calgary)	–	–	–	–	–	–
China (Beijing)	All versions	All versions	All versions	All versions	All versions	All versions
China (Ningxia)	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Frankfurt)	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Ireland)	All versions	All versions	All versions	All versions	All versions	All versions
Europe (London)	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Milan)	–	–	–	–	–	–
Europe (Paris)	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Spain)	–	–	–	–	–	–
Europe (Stockholm)	All versions	All versions	All versions	All versions	All versions	All versions

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Zurich)	–	–	–	–	–	–
Israel (Tel Aviv)	–	–	–	–	–	–
Middle East (Bahrain)	–	–	–	–	–	–
Middle East (UAE)	–	–	–	–	–	–
South America (São Paulo)	All versions	All versions	All versions	All versions	All versions	All versions
AWS GovCloud (US-East)	All versions	All versions	All versions	All versions	All versions	All versions
AWS GovCloud (US-West)	All versions	All versions	All versions	All versions	All versions	All versions

Supported Regions and DB engines for Aurora machine learning

By using Amazon Aurora machine learning, you can integrate your Aurora DB cluster with Amazon Comprehend or Amazon SageMaker, depending on your needs. Amazon Comprehend and SageMaker each support different machine learning use cases. Amazon Comprehend is a *natural language processing* (NLP) service that's used to extract insights from documents. By using Aurora machine learning with Amazon Comprehend, you can determine the sentiment of text in your database tables. SageMaker is a full-featured *machine learning* service. Data scientists use Amazon SageMaker to build, train, and test machine learning models for a variety of inference tasks, such as fraud detection. By using Aurora machine learning with SageMaker, database developers can invoke the SageMaker functionality in SQL code.

Not all AWS Regions support both Amazon Comprehend and SageMaker, and only certain AWS Regions support Aurora machine learning and thus provide access to these services from an Aurora DB cluster. The integration process for Aurora machine learning also differs by database engine. For more information, see Using Amazon Aurora machine learning.

Topics

- Aurora machine learning with Aurora MySQL
- Aurora machine learning with Aurora PostgreSQL

Aurora machine learning with Aurora MySQL

Aurora machine learning is supported for Aurora MySQL in the AWS Regions listed in the table. In addition to having your version of Aurora MySQL available, the AWS Region must also support the service that you want to use. For a list of AWS Regions where Amazon SageMaker is available, see Amazon SageMaker endpoints and quotas in the *Amazon Web Services General Reference*. For a list of AWS Regions where Amazon Comprehend is available, see Amazon Comprehend endpoints and quotas in the *Amazon Web Services General Reference*.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	Version 3.01.0 and higher	Version 2.07 and higher
US East (N. Virginia)	Version 3.01.0 and higher	Version 2.07 and higher
US West (N. California)	Version 3.01.0 and higher	Version 2.07 and higher
US West (Oregon)	Version 3.01.0 and higher	Version 2.07 and higher
Africa (Cape Town)	–	–
Asia Pacific (Hong Kong)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Hyderabad)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Jakarta)	Version 3.01.0 and higher	Version 2.07 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Melbourne)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Mumbai)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Osaka)	Version 3.01.0 and higher	Version 2.07.3 and higher
Asia Pacific (Seoul)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Singapore)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Sydney)	Version 3.01.0 and higher	Version 2.07 and higher
Asia Pacific (Tokyo)	Version 3.01.0 and higher	Version 2.07 and higher
Canada (Central)	Version 3.01.0 and higher	Version 2.07 and higher
Canada West (Calgary)	Version 3.01.0 and higher	Version 2.07 and higher
China (Beijing)	Version 3.01.0 and higher	Version 2.07 and higher
China (Ningxia)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Frankfurt)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Ireland)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (London)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Milan)	–	–
Europe (Paris)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Spain)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Stockholm)	Version 3.01.0 and higher	Version 2.07 and higher
Europe (Zurich)	Version 3.01.0 and higher	Version 2.07 and higher
Israel (Tel Aviv)	Version 3.01.0 and higher	Version 2.07 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Middle East (Bahrain)	Version 3.01.0 and higher	Version 2.07 and higher
Middle East (UAE)	Version 3.01.0 and higher	Version 2.07 and higher
South America (São Paulo)	Version 3.01.0 and higher	Version 2.07 and higher
AWS GovCloud (US-East)	Version 3.01.0 and higher	Version 2.07 and higher
AWS GovCloud (US-West)	Version 3.01.0 and higher	Version 2.07 and higher

Aurora machine learning with Aurora PostgreSQL

Aurora machine learning is supported for Aurora PostgreSQL in the AWS Regions listed in the table. In addition to having your version of Aurora PostgreSQL available, the AWS Region must also support the service that you want to use. For a list of AWS Regions where Amazon SageMaker is available, see Amazon SageMaker endpoints and quotas in the *Amazon Web Services General Reference*. For a list of AWS Regions where Amazon Comprehend is available, see Amazon Comprehend endpoints and quotas in the *Amazon Web Services General Reference*.

The following Regions and engine versions are available for Aurora machine learning with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (Ohio)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
US East (N. Virginia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US West (N. California)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
US West (Oregon)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Africa (Cape Town)	–	–	–	–	–	–
Asia Pacific (Hong Kong)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Hyderabad)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Jakarta)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Melbourne)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Mumbai)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Osaka)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Seoul)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Singapore)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Sydney)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Canada (Central)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Canada West (Calgary)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
China (Beijing)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
China (Ningxia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (Frankfurt)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Ireland)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (London)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (Milan)	–	–	–	–	–	–
Europe (Paris)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (Spain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (Stockholm)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Europe (Zurich)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Israel (Tel Aviv)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
Middle East (Bahrain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Middle East (UAE)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
South America (São Paulo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
AWS GovCloud (US-East)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher
AWS GovCloud (US-West)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3	Version 13.3 and higher	Version 12.4 and higher	Version 11.9, 11.12 and higher

Supported Regions and Aurora DB engines for Performance Insights

Performance Insights expands on existing Amazon RDS monitoring features to illustrate and help you analyze your database performance. With the Performance Insights dashboard, you can visualize the database load on your Amazon RDS DB instance load and filter the load by waits, SQL statements, hosts, or users. For more information, see Overview of Performance Insights on Amazon Aurora.

For the region, DB engine, and instance class support information for Performance Insights features, see Amazon Aurora DB engine, Region, and instance class support for Performance Insights features.

Topics

- Performance Insights with Aurora MySQL
- Performance Insights with Aurora PostgreSQL
- Performance Insights with Aurora Serverless

Performance Insights with Aurora MySQL

Note

Engine version support is different for Performance Insights with Aurora MySQL if you have parallel query turned on. For more information on parallel query, see Working with parallel query for Amazon Aurora MySQL.

Topics

- Performance Insights with Aurora MySQL and parallel query turned off
- Performance Insights with Aurora MySQL and parallel query turned on

Performance Insights with Aurora MySQL and parallel query turned off

The following Regions and engine versions are available for Performance Insights with Aurora MySQL and parallel query turned off.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	All versions	All versions
US East (N. Virginia)	All versions	All versions
US West (N. California)	All versions	All versions
US West (Oregon)	All versions	All versions
Africa (Cape Town)	All versions	All versions
Asia Pacific (Hong Kong)	All versions	All versions
Asia Pacific (Hyderabad)	All versions	All versions
Asia Pacific (Jakarta)	All versions	All versions
Asia Pacific (Melbourne)	All versions	All versions
Asia Pacific (Mumbai)	All versions	All versions

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Osaka)	All versions	All versions
Asia Pacific (Seoul)	All versions	All versions
Asia Pacific (Singapore)	All versions	All versions
Asia Pacific (Sydney)	All versions	All versions
Asia Pacific (Tokyo)	All versions	All versions
Canada (Central)	All versions	All versions
Canada West (Calgary)	All versions	All versions
China (Beijing)	All versions	All versions
China (Ningxia)	All versions	All versions
Europe (Frankfurt)	All versions	All versions
Europe (Ireland)	All versions	All versions
Europe (London)	All versions	All versions
Europe (Milan)	All versions	All versions
Europe (Paris)	All versions	All versions
Europe (Spain)	All versions	All versions
Europe (Stockholm)	All versions	All versions
Europe (Zurich)	All versions	All versions
Israel (Tel Aviv)	All versions	All versions
Middle East (Bahrain)	All versions	All versions
Middle East (UAE)	All versions	All versions

Region	Aurora MySQL version 3	Aurora MySQL version 2
South America (São Paulo)	All versions	All versions
AWS GovCloud (US-East)	All versions	All versions
AWS GovCloud (US-West)	All versions	All versions

Performance Insights with Aurora MySQL and parallel query turned on

The following Regions and engine versions are available for Performance Insights with Aurora MySQL and parallel query turned on.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	–	Version 2.09.0 and higher
US East (N. Virginia)	–	Version 2.09.0 and higher
US West (N. California)	–	Version 2.09.0 and higher
US West (Oregon)	–	Version 2.09.0 and higher
Africa (Cape Town)	–	Version 2.09.0 and higher
Asia Pacific (Hong Kong)	–	Version 2.09.0 and higher
Asia Pacific (Hyderabad)	–	All versions
Asia Pacific (Jakarta)	–	Version 2.09.0 and higher
Asia Pacific (Melbourne)	–	Version 2.09.0 and higher
Asia Pacific (Mumbai)	–	Version 2.09.0 and higher
Asia Pacific (Osaka)	–	Version 2.09.0 and higher
Asia Pacific (Seoul)	–	Version 2.09.0 and higher
Asia Pacific (Singapore)	–	Version 2.09.0 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Sydney)	–	Version 2.09.0 and higher
Asia Pacific (Tokyo)	–	Version 2.09.0 and higher
Canada (Central)	–	Version 2.09.0 and higher
Canada West (Calgary)	–	Version 2.09.0 and higher
China (Beijing)	–	Version 2.09.0 and higher
China (Ningxia)	–	Version 2.09.0 and higher
Europe (Frankfurt)	–	Version 2.09.0 and higher
Europe (Ireland)	–	Version 2.09.0 and higher
Europe (London)	–	Version 2.09.0 and higher
Europe (Milan)	–	Version 2.09.0 and higher
Europe (Paris)	–	Version 2.09.0 and higher
Europe (Spain)	–	Version 2.09.0 and higher
Europe (Stockholm)	–	Version 2.09.0 and higher
Europe (Zurich)	–	Version 2.09.0 and higher
Israel (Tel Aviv)	–	Version 2.09.0 and higher
Middle East (Bahrain)	–	Version 2.09.0 and higher
Middle East (UAE)	–	Version 2.09.0 and higher
South America (São Paulo)	–	Version 2.09.0 and higher
AWS GovCloud (US-East)	–	Version 2.09.0 and higher
AWS GovCloud (US-West)	–	Version 2.09.0 and higher

Performance Insights with Aurora PostgreSQL

The following Regions and engine versions are available for Performance Insights with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11	Aurora PostgreSQL 10
US East (Ohio)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
US East (N. Virginia)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
US West (N. California)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
US West (Oregon)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Africa (Cape Town)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Hong Kong)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Hyderabad)	All versions	All versions	All versions	All versions	All versions	All versions	All versions

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11	Aurora PostgreSQL 10
Asia Pacific (Jakarta)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Melbourne)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Mumbai)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Osaka)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Seoul)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Singapore)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Sydney)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Asia Pacific (Tokyo)	All versions	All versions	All versions	All versions	All versions	All versions	All versions

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11	Aurora PostgreSQL 10
Canada (Central)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Canada West (Calgary)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
China (Beijing)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
China (Ningxia)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Frankfurt)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Ireland)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (London)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Milan)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Paris)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Spain)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Europe (Stockholm)	All versions	All versions	All versions	All versions	All versions	All versions	All versions

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11	Aurora PostgreSQL 10
Europe (Zurich)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Israel (Tel Aviv)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Middle East (Bahrain)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
Middle East (UAE)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
South America (São Paulo)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
AWS GovCloud (US-East)	All versions	All versions	All versions	All versions	All versions	All versions	All versions
AWS GovCloud (US-West)	All versions	All versions	All versions	All versions	All versions	All versions	All versions

Performance Insights with Aurora Serverless

Aurora Serverless v2 supports Performance Insights for all MySQL-compatible and PostgreSQL-compatible versions. We recommend that you set the minimum capacity to at least 2 Aurora capacity units (ACUs).

Aurora Serverless v1 doesn't support Performance Insights.

Supported Regions and Aurora DB engines for zero-ETL integrations with Amazon Redshift

Amazon Aurora zero-ETL integrations with Amazon Redshift is a fully managed solution for making transactional data available in Amazon Redshift after it's written to an Aurora cluster. For more information, see *Working with zero-ETL integrations* .

The following Regions and engine versions are available for zero-ETL integrations with Amazon Redshift.

Topics

- Aurora MySQL Zero-ETL integrations
- Aurora PostgreSQL Zero-ETL integrations

Aurora MySQL Zero-ETL integrations

Region	Aurora MySQL version 3
US East (N. Virginia)	Version 3.05.2 and higher
US East (Ohio)	Version 3.05.2 and higher
US West (Oregon)	Version 3.05.2 and higher
US West (N. California)	Version 3.05.2 and higher
Asia Pacific (Tokyo)	Version 3.05.2 and higher
Asia Pacific (Singapore)	Version 3.05.2 and higher
Asia Pacific (Seoul)	Version 3.05.2 and higher
Asia Pacific (Mumbai)	Version 3.05.2 and higher
Asia Pacific (Hong Kong)	Version 3.05.2 and higher
Asia Pacific (Osaka)	Version 3.05.2 and higher

Region	Aurora MySQL version 3
Asia Pacific (Sydney)	Version 3.05.2 and higher
Europe (Frankfurt)	Version 3.05.2 and higher
Europe (Stockholm)	Version 3.05.2 and higher
Europe (Ireland)	Version 3.05.2 and higher
Europe (Paris)	Version 3.05.2 and higher
Europe (London)	Version 3.05.2 and higher
Europe (Milan)	Version 3.05.2 and higher
South America (São Paulo)	Version 3.05.2 and higher
Canada (Central)	Version 3.05.2 and higher
Middle East (Bahrain)	Version 3.05.2 and higher
Africa (Cape Town)	Version 3.05.2 and higher
China (Beijing)	Version 3.05.2 and higher
China (Ningxia)	Version 3.05.2 and higher

Aurora PostgreSQL Zero-ETL integrations

For the preview release of Aurora PostgreSQL zero-ETL integrations with Amazon Redshift, you must create integrations within the Amazon RDS Database Preview Environment, in the US East (Ohio) (us-east-2) AWS Region. The preview environment allows you to test beta, release candidate, and early production versions of PostgreSQL database engine software.

Your source DB cluster must be running **Aurora PostgreSQL (compatible with PostgreSQL 15.4 and Zero-ETL Support)**.

Supported Regions and Aurora DB engines for Amazon RDS Proxy

Amazon RDS Proxy is a fully managed, highly available database proxy that makes applications more scalable by pooling and sharing established database connections. For more information about RDS Proxy, see Using Amazon RDS Proxy for Aurora.

Topics

- Amazon RDS Proxy with Aurora MySQL
- Amazon RDS Proxy with Aurora PostgreSQL

Amazon RDS Proxy with Aurora MySQL

The following Regions and engine versions are available for RDS Proxy with Aurora MySQL.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
US East (N. Virginia)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
US West (N. California)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
US West (Oregon)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Africa (Cape Town)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Hong Kong)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Hyderabad)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Jakarta)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Melbourne)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Mumbai)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Osaka)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Seoul)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Singapore)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Sydney)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Asia Pacific (Tokyo)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Canada (Central)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Canada West (Calgary)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
China (Beijing)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
China (Ningxia)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Frankfurt)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher

Region	Aurora MySQL version 3	Aurora MySQL version 2
Europe (Ireland)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (London)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Milan)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Paris)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Spain)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Stockholm)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Europe (Zurich)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Israel (Tel Aviv)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Middle East (Bahrain)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
Middle East (UAE)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
South America (São Paulo)	Version 3.01.0 and higher	Version 2.07 and version 2.11 and higher
AWS GovCloud (US-East)	–	–
AWS GovCloud (US-West)	–	–

Amazon RDS Proxy with Aurora PostgreSQL

The following Regions and engine versions are available for RDS Proxy with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
US East (Ohio)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
US East (N. Virginia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
US West (N. California)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
US West (Oregon)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Africa (Cape Town)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Hong Kong)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Hyderabad)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Jakarta)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Melbourne)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Mumbai)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Asia Pacific (Osaka)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Seoul)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Singapore)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Sydney)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Canada (Central)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Canada West (Calgary)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
China (Beijing)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
China (Ningxia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Frankfurt)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Ireland)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (London)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Milan)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Paris)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Spain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
Europe (Stockholm)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Europe (Zurich)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Israel (Tel Aviv)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Middle East (Bahrain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
Middle East (UAE)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13	Aurora PostgreSQL 12	Aurora PostgreSQL 11
South America (São Paulo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.4 and higher	Version 12.8 and higher	Version 11.9 and version 11.13 and higher
AWS GovCloud (US-East)	–	–	–	–	–	–
AWS GovCloud (US-West)	–	–	–	–	–	–

Supported Regions and Aurora DB engines for Secrets Manager integration

With AWS Secrets Manager, you can replace hard-coded credentials in your code, including database passwords, with an API call to Secrets Manager to retrieve the secret programmatically. For more information about Secrets Manager, see AWS Secrets Manager User Guide.

You can specify that Amazon Aurora manages the master user password in Secrets Manager for an Aurora DB cluster. Aurora generates the password, stores it in Secrets Manager, and rotates it regularly. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.

Secrets Manager integration is available in all AWS Regions.

Supported Regions and Aurora DB engines for Aurora Serverless v2

Aurora Serverless v2 is an on-demand, auto-scaling feature designed to be a cost-effective approach to running intermittent or unpredictable workloads on Amazon Aurora. It automatically scales capacity up or down as needed by your applications. The scaling is faster and more granular than with Aurora Serverless v1. With Aurora Serverless v2, each cluster can contain a writer DB instance and multiple reader DB instances. You can combine Aurora Serverless v2 and traditional provisioned DB instances within the same cluster. For more information, see Using Aurora Serverless v2.

Topics

- Aurora Serverless v2 with Aurora MySQL
- Aurora Serverless v2 with Aurora PostgreSQL

Aurora Serverless v2 with Aurora MySQL

The following Regions and engine versions are available for Aurora Serverless v2 with Aurora MySQL.

Region	Aurora MySQL version 3
US East (Ohio)	Version 3.02.0 and higher
US East (N. Virginia)	Version 3.02.0 and higher
US West (N. California)	Version 3.02.0 and higher
US West (Oregon)	Version 3.02.0 and higher
Africa (Cape Town)	Version 3.02.0 and higher
Asia Pacific (Hong Kong)	Version 3.02.0 and higher
Asia Pacific (Hyderabad)	Version 3.02.3 and higher
Asia Pacific (Jakarta)	Version 3.02.0 and higher
Asia Pacific (Melbourne)	Version 3.02.3 and higher
Asia Pacific (Mumbai)	Version 3.02.0 and higher
Asia Pacific (Osaka)	Version 3.02.0 and higher
Asia Pacific (Seoul)	Version 3.02.0 and higher
Asia Pacific (Singapore)	Version 3.02.0 and higher

Region	Aurora MySQL version 3
Asia Pacific (Sydney)	Version 3.02.0 and higher
Asia Pacific (Tokyo)	Version 3.02.0 and higher
Canada (Central)	Version 3.02.0 and higher
Canada West (Calgary)	Versions 3.04.0, 3.04.1, 3.05.0, 3.05.1 and higher
China (Beijing)	Version 3.02.2 and higher
China (Ningxia)	Version 3.02.2 and higher
Europe (Frankfurt)	Version 3.02.0 and higher
Europe (Ireland)	Version 3.02.0 and higher
Europe (London)	Version 3.02.0 and higher
Europe (Milan)	Version 3.02.0 and higher
Europe (Paris)	Version 3.02.0 and higher
Europe (Spain)	Version 3.02.3 and higher
Europe (Stockholm)	Version 3.02.0 and higher
Europe (Zurich)	Version 3.02.3 and higher
Israel (Tel Aviv)	Versions 3.02.3 and higher, 3.03.1 and higher
Middle East (Bahrain)	Version 3.02.0 and higher
Middle East (UAE)	Version 3.02.3 and higher
South America (São Paulo)	Version 3.02.0 and higher
AWS GovCloud (US-East)	Version 3.02.2 and higher
AWS GovCloud (US-West)	Version 3.02.2 and higher

Aurora Serverless v2 with Aurora PostgreSQL

The following Regions and engine versions are available for Aurora Serverless v2 with Aurora PostgreSQL.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
US East (Ohio)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
US East (N. Virginia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
US West (N. California)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
US West (Oregon)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Africa (Cape Town)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Hong Kong)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Hyderabad)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
Asia Pacific (Jakarta)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Melbourne)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
Asia Pacific (Mumbai)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Osaka)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
Asia Pacific (Seoul)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Singapore)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Sydney)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Canada (Central)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Canada West (Calgary)	Version 16.1 and higher	Version 15.3 and higher	Version 14.6, 14.8 and higher	Version 13.9, 13.11 and higher
China (Beijing)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
China (Ningxia)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (Frankfurt)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (Ireland)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (London)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (Milan)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
Europe (Paris)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (Spain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
Europe (Stockholm)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Europe (Zurich)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
Israel (Tel Aviv)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
Middle East (Bahrain)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
Middle East (UAE)	Version 16.1 and higher	Version 15.2 and higher	Version 14.6 and higher	Version 13.9 and higher
South America (São Paulo)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
AWS GovCloud (US-East)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher
AWS GovCloud (US-West)	Version 16.1 and higher	Version 15.2 and higher	Version 14.3 and higher	Version 13.6 and higher

Supported Regions and Aurora DB engines for Aurora Serverless v1

Aurora Serverless v1 is an on-demand, auto-scaling feature designed to be a cost-effective approach to running intermittent or unpredictable workloads on Amazon Aurora. It automatically starts up, shuts down, and scales capacity up or down, as needed by your applications, using a single DB instance in each cluster. For more information, see Using Amazon Aurora Serverless v1.

Topics

- Aurora Serverless v1 with Aurora MySQL
- Aurora Serverless v1 with Aurora PostgreSQL

Aurora Serverless v1 with Aurora MySQL

The following Regions and engine versions are available for Aurora Serverless v1 with Aurora MySQL.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	–	Version 2.11.4
US East (N. Virginia)	–	Version 2.11.4
US West (N. California)	–	Version 2.11.4
US West (Oregon)	–	Version 2.11.4
Africa (Cape Town)	–	–
Asia Pacific (Hong Kong)	–	–
Asia Pacific (Hyderabad)	–	–
Asia Pacific (Jakarta)	–	–
Asia Pacific (Melbourne)	–	–
Asia Pacific (Mumbai)	–	Version 2.11.4
Asia Pacific (Osaka)	–	–
Asia Pacific (Seoul)	–	Version 2.11.4
Asia Pacific (Singapore)	–	Version 2.11.4
Asia Pacific (Sydney)	–	Version 2.11.4
Asia Pacific (Tokyo)	–	Version 2.11.4

Region	Aurora MySQL version 3	Aurora MySQL version 2
Canada (Central)	–	Version 2.11.4
Canada West (Calgary)	–	–
China (Beijing)	–	–
China (Ningxia)	–	Version 2.11.4
Europe (Frankfurt)	–	Version 2.11.4
Europe (Ireland)	–	Version 2.11.4
Europe (London)	–	Version 2.11.4
Europe (Milan)	–	–
Europe (Paris)	–	Version 2.11.4
Europe (Spain)	–	–
Europe (Stockholm)	–	–
Europe (Zurich)	–	–
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	–	–
Middle East (UAE)	–	–
South America (São Paulo)	–	–
AWS GovCloud (US-East)	–	–
AWS GovCloud (US-West)	–	–

Aurora Serverless v1 with Aurora PostgreSQL

The following Regions and engine versions are available for Aurora Serverless v1 with Aurora PostgreSQL.

Region	Aurora PostgreSQL 13
US East (Ohio)	Version 13.12
US East (N. Virginia)	Version 13.12
US West (N. California)	Version 13.12
US West (Oregon)	Version 13.12
Africa (Cape Town)	–
Asia Pacific (Hong Kong)	–
Asia Pacific (Hyderabad)	–
Asia Pacific (Jakarta)	–
Asia Pacific (Melbourne)	–
Asia Pacific (Mumbai)	Version 13.12
Asia Pacific (Osaka)	–
Asia Pacific (Seoul)	Version 13.12
Asia Pacific (Singapore)	Version 13.12
Asia Pacific (Sydney)	Version 13.12
Asia Pacific (Tokyo)	Version 13.12
Canada (Central)	Version 13.12
Canada West (Calgary)	–
China (Beijing)	–

Region	Aurora PostgreSQL 13
China (Ningxia)	–
Europe (Frankfurt)	Version 13.12
Europe (Ireland)	Version 13.12
Europe (London)	Version 13.12
Europe (Milan)	–
Europe (Paris)	Version 13.12
Europe (Spain)	–
Europe (Stockholm)	–
Europe (Zurich)	–
Israel (Tel Aviv)	–
Middle East (Bahrain)	–
Middle East (UAE)	–
South America (São Paulo)	–
AWS GovCloud (US-East)	–
AWS GovCloud (US-West)	–

Supported Regions and Aurora DB engines for RDS Data API

RDS Data API (Data API) provides a web-services interface to an Amazon Aurora DB cluster. Instead of managing database connections from client applications, you can run SQL commands against an HTTPS endpoint. For more information, see Using RDS Data API.

For Aurora MySQL, Data API isn't supported for Aurora Serverless v2 or for provisioned DB clusters.

Topics

- Data API with Aurora MySQL Serverless v1
- Data API with Aurora PostgreSQL Serverless v2 and provisioned
- Data API with Aurora PostgreSQL Serverless v1

Data API with Aurora MySQL Serverless v1

The following Regions and engine versions are available for Data API with Aurora MySQL Serverless v1.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	–	Version 2.11.3
US East (N. Virginia)	–	Version 2.11.3
US West (N. California)	–	Version 2.11.3
US West (Oregon)	–	Version 2.11.3
Africa (Cape Town)	–	–
Asia Pacific (Hong Kong)	–	–
Asia Pacific (Hyderabad)	–	–
Asia Pacific (Jakarta)	–	–
Asia Pacific (Melbourne)	–	–
Asia Pacific (Mumbai)	–	Version 2.11.3
Asia Pacific (Osaka)	–	–
Asia Pacific (Seoul)	–	Version 2.11.3
Asia Pacific (Singapore)	–	Version 2.11.3
Asia Pacific (Sydney)	–	Version 2.11.3
Asia Pacific (Tokyo)	–	Version 2.11.3

Region	Aurora MySQL version 3	Aurora MySQL version 2
Canada (Central)	–	Version 2.11.3
Canada West (Calgary)	–	–
China (Beijing)	–	–
China (Ningxia)	–	Version 2.11.3
Europe (Frankfurt)	–	Version 2.11.3
Europe (Ireland)	–	Version 2.11.3
Europe (London)	–	Version 2.11.3
Europe (Milan)	–	–
Europe (Paris)	–	Version 2.11.3
Europe (Spain)	–	–
Europe (Stockholm)	–	–
Europe (Zurich)	–	–
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	–	–
Middle East (UAE)	–	–
South America (São Paulo)	–	–
AWS GovCloud (US-East)	–	–
AWS GovCloud (US-West)	–	–

Data API with Aurora PostgreSQL Serverless v2 and provisioned

The following Regions and engine versions are available for Data API with Aurora PostgreSQL Serverless v2 and provisioned DB clusters.

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
US East (Ohio)	–	–	–	–
US East (N. Virginia)	Version 16.1 and higher	Version 15.3 and higher	Version 14.8 and higher	Version 13.11 and higher
US West (N. California)	–	–	–	–
US West (Oregon)	Version 16.1 and higher	Version 15.3 and higher	Version 14.8 and higher	Version 13.11 and higher
Africa (Cape Town)	–	–	–	–
Asia Pacific (Hong Kong)	–	–	–	–
Asia Pacific (Hyderabad)	–	–	–	–
Asia Pacific (Jakarta)	–	–	–	–
Asia Pacific (Melbourne)	–	–	–	–
Asia Pacific (Mumbai)	–	–	–	–
Asia Pacific (Osaka)	–	–	–	–

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
Asia Pacific (Seoul)	–	–	–	–
Asia Pacific (Singapore)	–	–	–	–
Asia Pacific (Sydney)	–	–	–	–
Asia Pacific (Tokyo)	Version 16.1 and higher	Version 15.3 and higher	Version 14.8 and higher	Version 13.11 and higher
Canada (Central)	–	–	–	–
Canada West (Calgary)	–	–	–	–
China (Beijing)	–	–	–	–
China (Ningxia)	–	–	–	–
Europe (Frankfurt)	Version 16.1 and higher	Version 15.3 and higher	Version 14.8 and higher	Version 13.11 and higher
Europe (Ireland)	–	–	–	–
Europe (London)	–	–	–	–
Europe (Milan)	–	–	–	–
Europe (Paris)	–	–	–	–
Europe (Spain)	–	–	–	–
Europe (Stockholm)	–	–	–	–
Europe (Zurich)	–	–	–	–

Region	Aurora PostgreSQL 16	Aurora PostgreSQL 15	Aurora PostgreSQL 14	Aurora PostgreSQL 13
Israel (Tel Aviv)	–	–	–	–
Middle East (Bahrain)	–	–	–	–
Middle East (UAE)	–	–	–	–
South America (São Paulo)	–	–	–	–
AWS GovCloud (US-East)	–	–	–	–
AWS GovCloud (US-West)	–	–	–	–

Data API with Aurora PostgreSQL Serverless v1

The following Regions and engine versions are available for Data API with Aurora PostgreSQL Serverless v1.

Region	Aurora PostgreSQL 13	Aurora PostgreSQL 11
US East (Ohio)	Version 13.9	Version 11.18
US East (N. Virginia)	Version 13.9	Version 11.18
US West (N. California)	Version 13.9	Version 11.18
US West (Oregon)	Version 13.9	Version 11.18
Africa (Cape Town)	–	–
Asia Pacific (Hong Kong)	–	–

Region	Aurora PostgreSQL 13	Aurora PostgreSQL 11
Asia Pacific (Hyderabad)	–	–
Asia Pacific (Jakarta)	–	–
Asia Pacific (Melbourne)	–	–
Asia Pacific (Mumbai)	Version 13.9	Version 11.18
Asia Pacific (Osaka)	–	–
Asia Pacific (Seoul)	Version 13.9	Version 11.18
Asia Pacific (Singapore)	Version 13.9	Version 11.18
Asia Pacific (Sydney)	Version 13.9	Version 11.18
Asia Pacific (Tokyo)	Version 13.9	Version 11.18
Canada (Central)	Version 13.9	Version 11.18
China (Beijing)	–	–
China (Ningxia)	–	–
Europe (Frankfurt)	Version 13.9	Version 11.18
Europe (Ireland)	Version 13.9	Version 11.18
Europe (London)	Version 13.9	Version 11.18
Europe (Milan)	–	–
Europe (Paris)	Version 13.9	Version 11.18
Europe (Spain)	–	–
Europe (Stockholm)	–	–
Europe (Zurich)	–	–

Region	Aurora PostgreSQL 13	Aurora PostgreSQL 11
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	–	–
Middle East (UAE)	–	–
South America (São Paulo)	–	–
AWS GovCloud (US-East)	–	–
AWS GovCloud (US-West)	–	–

Supported Regions and Aurora DB engines for zero-downtime patching (ZDP)

Performing upgrades for Aurora DB clusters involves the possibility of an outage when the database is shut down and while it's being upgraded. By default, if you start the upgrade while the database is busy, you lose all the connections and transactions that the DB cluster is processing. If you wait until the database is idle to perform the upgrade, you might have to wait a long time.

The zero-downtime patching (ZDP) feature attempts, on a best-effort basis, to preserve client connections through an Aurora upgrade. If ZDP completes successfully, application sessions are preserved and the database engine restarts while the upgrade is in progress. The database engine restart can cause a drop in throughput lasting for a few seconds to approximately one minute.

For detailed information on the conditions and engine versions where ZDP is available for Aurora MySQL upgrades, see Using zero-downtime patching.

For detailed information on the conditions and engine versions where ZDP is available for Aurora PostgreSQL upgrades, see Minor release upgrades and zero-downtime patching.

Supported Regions and DB engines for Aurora engine-native features

Aurora database engines also support additional features and functionality specifically for Aurora. Some engine-native features might have limited support or restricted privileges for a particular Aurora DB engine, version, or Region.

Topics

- Engine-native features for Aurora MySQL
- Engine-native features for Aurora PostgreSQL

Engine-native features for Aurora MySQL

Following are the engine-native features for Aurora MySQL.

- Advanced Auditing
- Backtrack
- Fault injection queries
- In-cluster write forwarding
- Parallel query

Engine-native features for Aurora PostgreSQL

Following are the engine-native features for Aurora PostgreSQL.

- Babelfish
- Fault injection queries
- Query plan management

Amazon Aurora connection management

Amazon Aurora typically involves a cluster of DB instances instead of a single instance. Each connection is handled by a specific DB instance. When you connect to an Aurora cluster, the host name and port that you specify point to an intermediate handler called an *endpoint*. Aurora uses the endpoint mechanism to abstract these connections. Thus, you don't have to hardcode all the hostnames or write your own logic for balancing and rerouting connections when some DB instances aren't available.

For certain Aurora tasks, different instances or groups of instances perform different roles. For example, the primary instance handles all data definition language (DDL) and data manipulation language (DML) statements. Up to 15 Aurora Replicas handle read-only query traffic.

Using endpoints, you can map each connection to the appropriate instance or group of instances based on your use case. For example, to perform DDL statements you can connect to whichever instance is the primary instance. To perform queries, you can connect to the reader endpoint, with Aurora automatically performing connection-balancing among all the Aurora Replicas. For clusters with DB instances of different capacities or configurations, you can connect to custom endpoints associated with different subsets of DB instances. For diagnosis or tuning, you can connect to a specific instance endpoint to examine details about a specific DB instance.

Topics

Types of Aurora endpoints

An endpoint is represented as an Aurora-specific URL that contains a host address and a port. The following types of endpoints are available from an Aurora DB cluster.

Cluster endpoint

> A *cluster endpoint* (or *writer endpoint*) for an Aurora DB cluster connects to the current primary DB instance for that DB cluster. This endpoint is the only one that can perform write operations such as DDL statements. Because of this, the cluster endpoint is the one that you connect to when you first set up a cluster or when your cluster only contains a single DB instance.
>
> Each Aurora DB cluster has one cluster endpoint and one primary DB instance.

You use the cluster endpoint for all write operations on the DB cluster, including inserts, updates, deletes, and DDL changes. You can also use the cluster endpoint for read operations, such as queries.

The cluster endpoint provides failover support for read/write connections to the DB cluster. If the current primary DB instance of a DB cluster fails, Aurora automatically fails over to a new primary DB instance. During a failover, the DB cluster continues to serve connection requests to the cluster endpoint from the new primary DB instance, with minimal interruption of service.

The following example illustrates a cluster endpoint for an Aurora MySQL DB cluster.

```
mydbcluster.cluster-c7tj4example.us-east-1.rds.amazonaws.com:3306
```

Reader endpoint

A *reader endpoint* for an Aurora DB cluster provides connection-balancing support for read-only connections to the DB cluster. Use the reader endpoint for read operations, such as queries. By processing those statements on the read-only Aurora Replicas, this endpoint reduces the overhead on the primary instance. It also helps the cluster to scale the capacity to handle simultaneous `SELECT` queries, proportional to the number of Aurora Replicas in the cluster. Each Aurora DB cluster has one reader endpoint.

If the cluster contains one or more Aurora Replicas, the reader endpoint balances each connection request among the Aurora Replicas. In that case, you can only perform read-only statements such as `SELECT` in that session. If the cluster only contains a primary instance and no Aurora Replicas, the reader endpoint connects to the primary instance. In that case, you can perform write operations through the endpoint.

The following example illustrates a reader endpoint for an Aurora MySQL DB cluster.

```
mydbcluster.cluster-ro-c7tj4example.us-east-1.rds.amazonaws.com:3306
```

Custom endpoint

A *custom endpoint* for an Aurora cluster represents a set of DB instances that you choose. When you connect to the endpoint, Aurora performs connection balancing and chooses one of the instances in the group to handle the connection. You define which instances this endpoint refers to, and you decide what purpose the endpoint serves.

An Aurora DB cluster has no custom endpoints until you create one. You can create up to five custom endpoints for each provisioned Aurora cluster or Aurora Serverless v2 cluster. You can't use custom endpoints for Aurora Serverless v1 clusters.

The custom endpoint provides balanced database connections based on criteria other than the read-only or read/write capability of the DB instances. For example, you might define a custom endpoint to connect to instances that use a particular AWS instance class or a particular DB parameter group. Then you might tell particular groups of users about this custom endpoint. For example, you might direct internal users to low-capacity instances for report generation or ad hoc (one-time) querying, and direct production traffic to high-capacity instances.

Because the connection can go to any DB instance that is associated with the custom endpoint, we recommend that you make sure that all the DB instances within that group share some similar characteristic. Doing so ensures that the performance, memory capacity, and so on, are consistent for everyone who connects to that endpoint.

This feature is intended for advanced users with specialized kinds of workloads where it isn't practical to keep all the Aurora Replicas in the cluster identical. With custom endpoints, you can predict the capacity of the DB instance used for each connection. When you use custom endpoints, you typically don't use the reader endpoint for that cluster.

The following example illustrates a custom endpoint for a DB instance in an Aurora MySQL DB cluster.

```
myendpoint.cluster-custom-c7tj4example.us-east-1.rds.amazonaws.com:3306
```

Instance endpoint

An *instance endpoint* connects to a specific DB instance within an Aurora cluster. Each DB instance in a DB cluster has its own unique instance endpoint. So there is one instance endpoint for the current primary DB instance of the DB cluster, and there is one instance endpoint for each of the Aurora Replicas in the DB cluster.

The instance endpoint provides direct control over connections to the DB cluster, for scenarios where using the cluster endpoint or reader endpoint might not be appropriate. For example, your client application might require more fine-grained connection balancing based on workload type. In this case, you can configure multiple clients to connect to different Aurora Replicas in a DB cluster to distribute read workloads. For an example that uses instance endpoints to improve connection speed after a failover for Aurora PostgreSQL, see Fast failover with Amazon Aurora PostgreSQL. For an example that uses instance endpoints to improve connection speed after a failover for Aurora MySQL, see MariaDB Connector/J failover support - case Amazon Aurora.

The following example illustrates an instance endpoint for a DB instance in an Aurora MySQL DB cluster.

```
mydbinstance.c7tj4example.us-east-1.rds.amazonaws.com:3306
```

Viewing the endpoints for an Aurora cluster

In the AWS Management Console, you see the cluster endpoint, the reader endpoint, and any custom endpoints in the detail page for each cluster. You see the instance endpoint in the detail page for each instance. When you connect, you must append the associated port number, following a colon, to the endpoint name shown on this detail page.

With the AWS CLI, you see the writer, reader, and any custom endpoints in the output of the describe-db-clusters command. For example, the following command shows the endpoint attributes for all clusters in your current AWS Region.

```
aws rds describe-db-clusters --query '*[].
{Endpoint:Endpoint,ReaderEndpoint:ReaderEndpoint,CustomEndpoints:CustomEndpoints}'
```

With the Amazon RDS API, you retrieve the endpoints by calling the DescribeDBClusterEndpoints function.

Using the cluster endpoint

Because each Aurora cluster has a single built-in cluster endpoint, whose name and other attributes are managed by Aurora, you can't create, delete, or modify this kind of endpoint.

You use the cluster endpoint when you administer your cluster, perform extract, transform, load (ETL) operations, or develop and test applications. The cluster endpoint connects to the primary instance of the cluster. The primary instance is the only DB instance where you can create tables and indexes, run `INSERT` statements, and perform other DDL and DML operations.

The physical IP address pointed to by the cluster endpoint changes when the failover mechanism promotes a new DB instance to be the read/write primary instance for the cluster. If you use any form of connection pooling or other multiplexing, be prepared to flush or reduce the time-to-live for any cached DNS information. Doing so ensures that you don't try to establish a read/write connection to a DB instance that became unavailable or is now read-only after a failover.

Using the reader endpoint

You use the reader endpoint for read-only connections for your Aurora cluster. This endpoint uses a connection-balancing mechanism to help your cluster handle a query-intensive workload. The reader endpoint is the endpoint that you supply to applications that do reporting or other read-only operations on the cluster.

The reader endpoint balances connections to available Aurora Replicas in an Aurora DB cluster. It doesn't balance individual queries. If you want to balance each query to distribute the read workload for a DB cluster, open a new connection to the reader endpoint for each query.

Each Aurora cluster has a single built-in reader endpoint, whose name and other attributes are managed by Aurora. You can't create, delete, or modify this kind of endpoint.

If your cluster contains only a primary instance and no Aurora Replicas, the reader endpoint connects to the primary instance. In that case, you can perform write operations through this endpoint.

> **Tip**
>
> Through RDS Proxy, you can create additional read-only endpoints for an Aurora cluster. These endpoints perform the same kind of connection-balancing as the Aurora reader endpoint. Applications can reconnect more quickly to the proxy endpoints than the Aurora reader endpoint if reader instances become unavailable. The proxy endpoints can also take advantage of other proxy features such as multiplexing. For more information, see Using reader endpoints with Aurora clusters.

Using custom endpoints

You use custom endpoints to simplify connection management when your cluster contains DB instances with different capacities and configuration settings.

Previously, you might have used the CNAMES mechanism to set up Domain Name Service (DNS) aliases from your own domain to achieve similar results. By using custom endpoints, you can avoid updating CNAME records when your cluster grows or shrinks. Custom endpoints also mean that you can use encrypted Transport Layer Security/Secure Sockets Layer (TLS/SSL) connections.

Instead of using one DB instance for each specialized purpose and connecting to its instance endpoint, you can have multiple groups of specialized DB instances. In this case, each group has its own custom endpoint. This way, Aurora can perform connection balancing among all the instances dedicated to tasks such as reporting or handling production or internal queries. The custom endpoints distribute connections across instances passively, using DNS to return the IP address of one of the instances randomly. If one of the DB instances within a group becomes unavailable, Aurora directs subsequent custom endpoint connections to one of the other DB instances associated with the same endpoint.

Topics

Specifying properties for custom endpoints

The maximum length for a custom endpoint name is 63 characters. The name format is the following:

```
endpoint_name.cluster-custom-customer_DNS_identifier.AWS_Region.rds.amazonaws.com
```

You can't reuse the same custom endpoint name for more than one cluster in the same AWS Region. The customer DNS identifier is a unique identifier associated with your AWS account in a particular AWS Region.

Each custom endpoint has an associated type that determines which DB instances are eligible to be associated with that endpoint. Currently, the type can be `READER`, `WRITER`, or `ANY`. The following considerations apply to the custom endpoint types:

- You can't select the custom endpoint type in the AWS Management Console. All the custom endpoints you create through the AWS Management Console have a type of `ANY`.

  You can set and modify the custom endpoint type using the AWS CLI or Amazon RDS API.
- Only reader DB instances can be part of a `READER` custom endpoint.
- Both reader and writer DB instances can be part of an `ANY` custom endpoint. Aurora directs connections to cluster endpoints with type `ANY` to any associated DB instance with equal probability. The `ANY` type applies to clusters using any replication topology.

- If you try to create a custom endpoint with a type that isn't appropriate based on the replication configuration for a cluster, Aurora returns an error.

Membership rules for custom endpoints

When you add a DB instance to a custom endpoint or remove it from a custom endpoint, any existing connections to that DB instance remain active.

You can define a list of DB instances to include in, or exclude from, a custom endpoint. We refer to these lists as *static* and *exclusion* lists, respectively. You can use the inclusion/exclusion mechanism to further subdivide the groups of DB instances, and to make sure that the set of custom endpoints covers all the DB instances in the cluster. Each custom endpoint can contain only one of these list types.

In the AWS Management Console:

- The choice is represented by the check box **Attach future instances added to this cluster**. When you keep the check box clear, the custom endpoint uses a static list containing only the DB instances specified on the page. When you choose the check box, the custom endpoint uses an exclusion list. In this case, the custom endpoint represents all DB instances in the cluster (including any that you add in the future) except the ones not selected on the page.
- The console doesn't allow you to specify the endpoint type. Any custom endpoint created using the console is of type `ANY`.

  Therefore, Aurora doesn't change the membership of the custom endpoint when DB instances change roles between writer and reader due to failover or promotion.

In the AWS CLI and Amazon RDS API:

- You can specify the endpoint type. Therefore, when the endpoint type is set to `READER` or `WRITER`, endpoint membership is automatically adjusted during failovers and promotions.

  For example, a custom endpoint with type `READER` includes an Aurora Replica that is then promoted to be a writer instance. The new writer instance is no longer part of the custom endpoint.
- You can add individual members to and remove them from the lists after they change their roles. Use the modify-db-cluster-endpoint AWS CLI command or ModifyDBClusterEndpoint API operation.

You can associate a DB instance with more than one custom endpoint. For example, suppose that you add a new DB instance to a cluster, or that Aurora adds a DB instance automatically through the autoscaling mechanism. In these cases, the DB instance is added to all custom endpoints for which it is eligible. Which endpoints the DB instance is added to is based on the custom endpoint type of `READER`, `WRITER`, or `ANY`, plus any static or exclusion lists defined for each endpoint. For example, if the endpoint includes a static list of DB instances, newly added Aurora Replicas aren't added to that endpoint. Conversely, if the endpoint has an exclusion list, newly added Aurora Replicas are added to the endpoint, if they aren't named in the exclusion list and their roles match the type of the custom endpoint.

If an Aurora Replica becomes unavailable, it remains associated with any custom endpoints. For example, it remains part of the custom endpoint when it is unhealthy, stopped, rebooting, and so on. However, you can't connect to it through those endpoints until it becomes available again.

Managing custom endpoints

Because newly created Aurora clusters have no custom endpoints, you must create and manage these objects yourself. You do so using the AWS Management Console, AWS CLI, or Amazon RDS API.

> **Note**
>
> You must also create and manage custom endpoints for Aurora clusters restored from snapshots. Custom endpoints are not included in the snapshot. You create them again after restoring, and choose new endpoint names if the restored cluster is in the same region as the original one.

To work with custom endpoints from the AWS Management Console, you navigate to the details page for your Aurora cluster and use the controls under the **Custom Endpoints** section.

To work with custom endpoints from the AWS CLI, you can use these operations:

- create-db-cluster-endpoint
- describe-db-cluster-endpoints
- modify-db-cluster-endpoint
- delete-db-cluster-endpoint

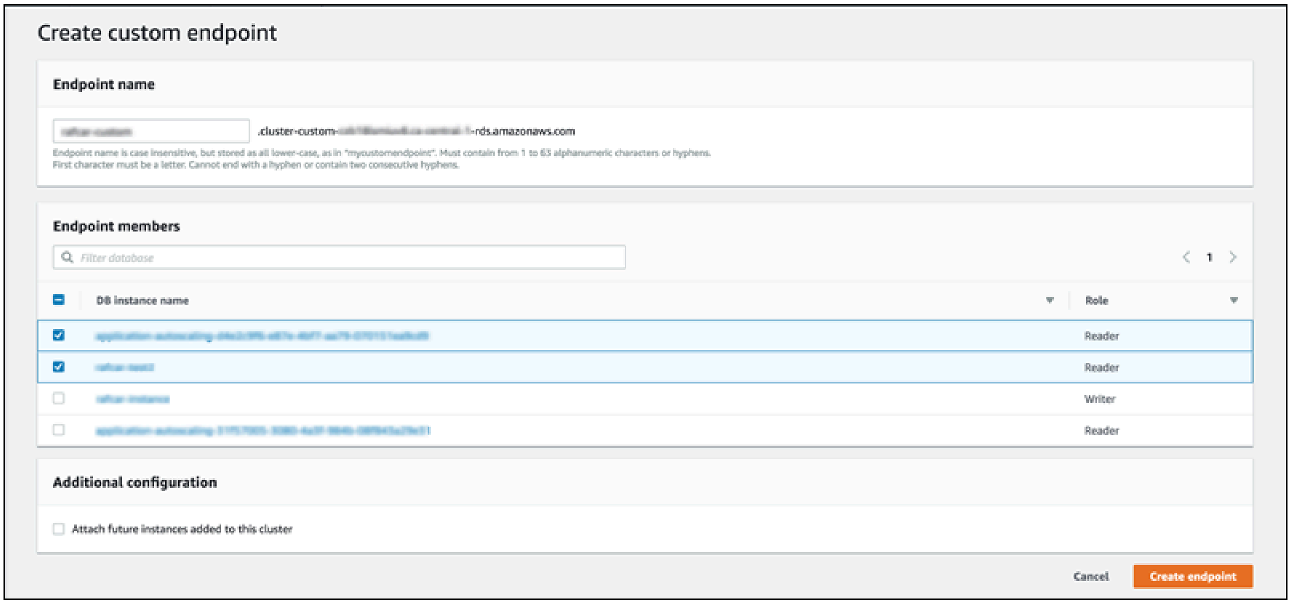
To work with custom endpoints through the Amazon RDS API, you can use the following functions:

- CreateDBClusterEndpoint
- DescribeDBClusterEndpoints
- ModifyDBClusterEndpoint
- DeleteDBClusterEndpoint

Creating a custom endpoint

Console

To create a custom endpoint with the AWS Management Console, go to the cluster detail page and choose the `Create custom endpoint` action in the **Endpoints** section. Choose a name for the custom endpoint, unique for your user ID and region. To choose a list of DB instances that remains the same even as the cluster expands, keep the check box **Attach future instances added to this cluster** clear. When you choose that check box, the custom endpoint dynamically adds any new instances as you add them to the cluster.



Create custom endpoint

Endpoint name

[redacted] .cluster-custom-[redacted]-rds.amazonaws.com

Endpoint name is case insensitive, but stored as all lower-case, as in "mycustomendpoint". Must contain from 1 to 63 alphanumeric characters or hyphens. First character must be a letter. Cannot end with a hyphen or contain two consecutive hyphens.

Endpoint members

Filter database

☐	DB instance name	Role
☑	[redacted]	Reader
☑	[redacted]	Reader
☐	[redacted]	Writer
☐	[redacted]	Reader

Additional configuration

☐ Attach future instances added to this cluster

You can't select the custom endpoint type of ANY or READER in the AWS Management Console. All the custom endpoints you create through the AWS Management Console have a type of ANY.

AWS CLI

To create a custom endpoint with the AWS CLI, run the create-db-cluster-endpoint command.

The following command creates a custom endpoint attached to a specific cluster. Initially, the endpoint is associated with all the Aurora Replica instances in the cluster. A subsequent command associates it with a specific set of DB instances in the cluster.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster-endpoint --db-cluster-endpoint-identifier custom-endpoint-
doc-sample \
  --endpoint-type reader \
  --db-cluster-identifier cluster_id

aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier custom-endpoint-
doc-sample \
  --static-members instance_name_1 instance_name_2
```

For Windows:

```
aws rds create-db-cluster-endpoint --db-cluster-endpoint-identifier custom-endpoint-
doc-sample ^
  --endpoint-type reader ^
  --db-cluster-identifier cluster_id

aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier custom-endpoint-
doc-sample ^
  --static-members instance_name_1 instance_name_2
```

RDS API

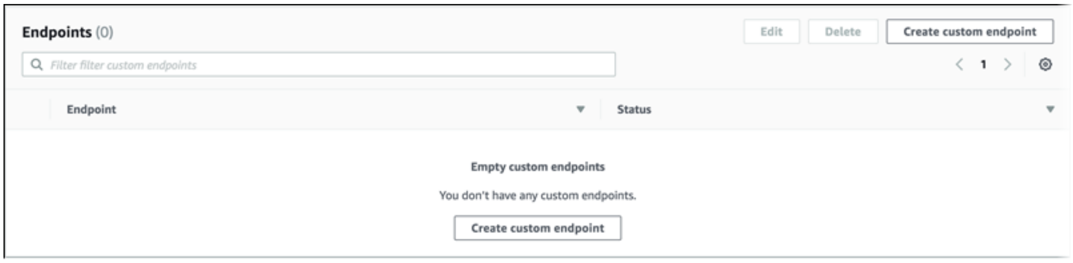
To create a custom endpoint with the RDS API, run the CreateDBClusterEndpoint operation.

Viewing custom endpoints

Console

To view custom endpoints with the AWS Management Console, go to the cluster detail page for the cluster and look under the **Endpoints** section. This section contains information only about custom endpoints. The details for the built-in endpoints are listed in the main **Details** section. To see the details for a specific custom endpoint, select its name to bring up the detail page for that endpoint.

The following screenshot shows how the list of custom endpoints for an Aurora cluster is initially empty.



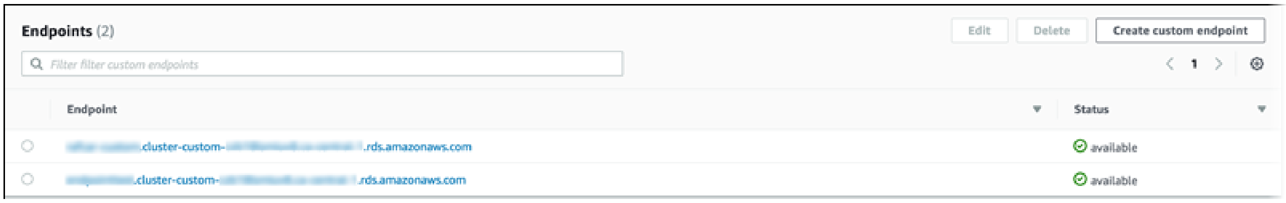
Endpoints (0)

Endpoint	Status

Empty custom endpoints

You don't have any custom endpoints.

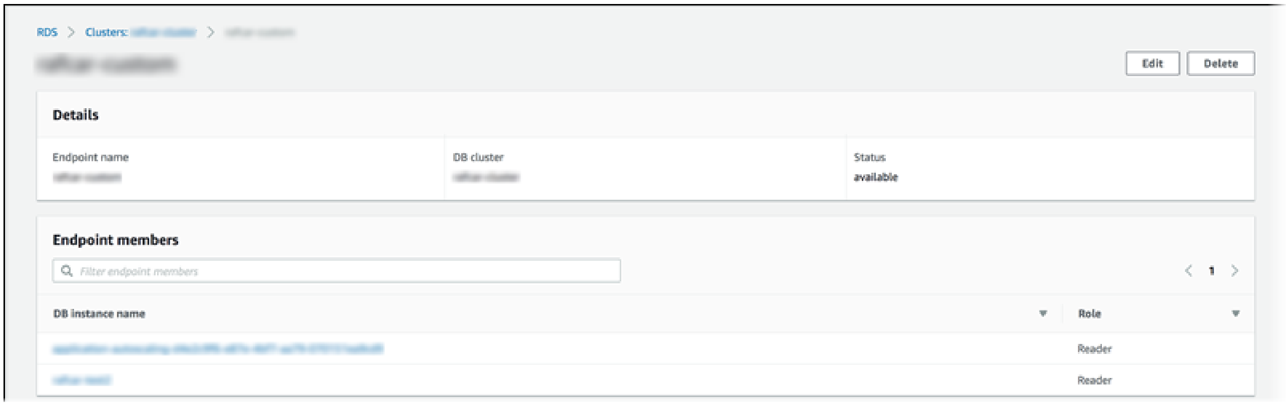
After you create some custom endpoints for that cluster, they are shown under the **Endpoints** section.



Endpoints (2)

Endpoint	Status
○ [redacted].cluster-custom-[redacted].rds.amazonaws.com	available
○ [redacted].cluster-custom-[redacted].rds.amazonaws.com	available

Clicking through to the detail page shows which DB instances the endpoint is currently associated with.



[redacted]

Details

Endpoint name	DB cluster	Status
[redacted]	[redacted]	available

Endpoint members

DB instance name	Role
[redacted]	Reader
[redacted]	Reader

To see the additional detail of whether new DB instances added to the cluster are automatically added to the endpoint also, open the **Edit** page for the endpoint.

AWS CLI

To view custom endpoints with the AWS CLI, run the describe-db-cluster-endpoints command.

The following command shows the custom endpoints associated with a specified cluster in a specified region. The output includes both the built-in endpoints and any custom endpoints.

For Linux, macOS, or Unix:

```
aws rds describe-db-cluster-endpoints --region region_name \
  --db-cluster-identifier cluster_id
```

For Windows:

```
aws rds describe-db-cluster-endpoints --region region_name ^
  --db-cluster-identifier cluster_id
```

The following shows some sample output from a `describe-db-cluster-endpoints` command. The `EndpointType` of `WRITER` or `READER` denotes the built-in read/write and read-only endpoints for the cluster. The `EndpointType` of `CUSTOM` denotes endpoints that you create and choose the associated DB instances. One of the endpoints has a non-empty `StaticMembers` field, denoting that it is associated with a precise set of DB instances. The other endpoint has a non-empty `ExcludedMembers` field, denoting that the endpoint is associated with all DB instances *other than* the ones listed under `ExcludedMembers`. This second kind of custom endpoint grows to accommodate new instances as you add them to the cluster.

```
{
 "DBClusterEndpoints": [
  {
   "Endpoint": "custom-endpoint-demo.cluster-c7tj4example.ca-
central-1.rds.amazonaws.com",
   "Status": "available",
   "DBClusterIdentifier": "custom-endpoint-demo",
   "EndpointType": "WRITER"
  },
  {
   "Endpoint": "custom-endpoint-demo.cluster-ro-c7tj4example.ca-
central-1.rds.amazonaws.com",
   "Status": "available",
   "DBClusterIdentifier": "custom-endpoint-demo",
   "EndpointType": "READER"
  },
  {
   "CustomEndpointType": "ANY",
   "DBClusterEndpointIdentifier": "powers-of-2",
   "ExcludedMembers": [],
   "DBClusterIdentifier": "custom-endpoint-demo",
   "Status": "available",
   "EndpointType": "CUSTOM",
   "Endpoint": "powers-of-2.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
   "StaticMembers": [
     "custom-endpoint-demo-04",
     "custom-endpoint-demo-08",
     "custom-endpoint-demo-01",
     "custom-endpoint-demo-02"
   ],
   "DBClusterEndpointResourceIdentifier": "cluster-endpoint-
W7PE3TLLFNSHXQKFU6J6NV5FHU",
   "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:powers-of-2"
  },
  {
   "CustomEndpointType": "ANY",
   "DBClusterEndpointIdentifier": "eight-and-higher",
   "ExcludedMembers": [
     "custom-endpoint-demo-04",
     "custom-endpoint-demo-02",
     "custom-endpoint-demo-07",
     "custom-endpoint-demo-05",
     "custom-endpoint-demo-03",
     "custom-endpoint-demo-06",
     "custom-endpoint-demo-01"
   ],
   "DBClusterIdentifier": "custom-endpoint-demo",
   "Status": "available",
   "EndpointType": "CUSTOM",
   "Endpoint": "eight-and-higher.cluster-custom-123456789012.ca-
central-1.rds.amazonaws.com",
   "StaticMembers": [],
   "DBClusterEndpointResourceIdentifier": "cluster-endpoint-
W7PE3TLLFNSHYQKFU6J6NV5FHU",
   "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:eight-and-higher"
  }
 ]
}
```

RDS API

To view custom endpoints with the RDS API, run the DescribeDBClusterEndpoints.html operation.

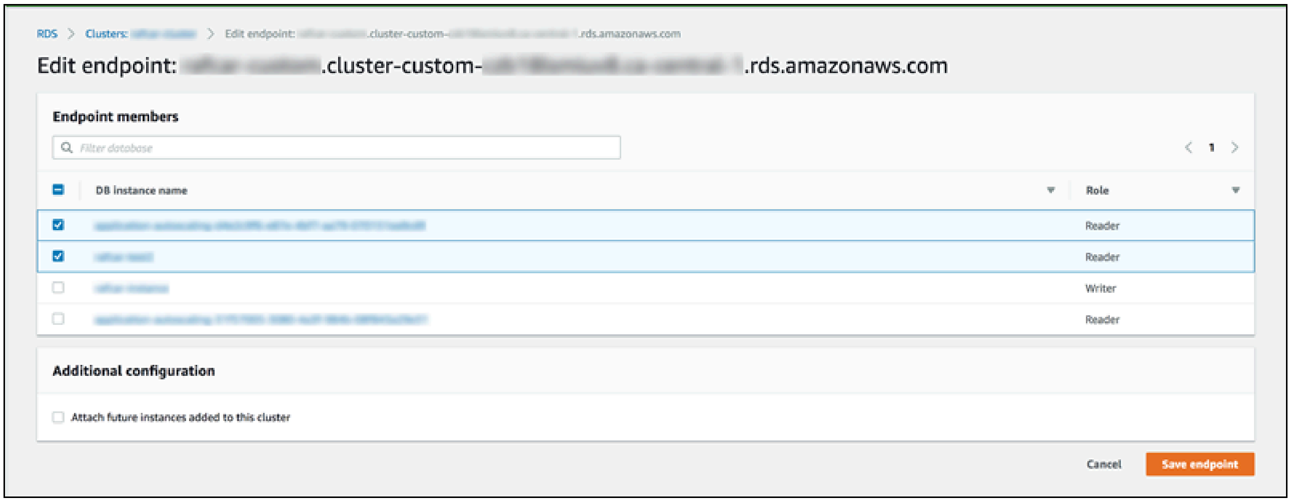
Editing a custom endpoint

You can edit the properties of a custom endpoint to change which DB instances are associated with the endpoint. You can also change an endpoint between a static list and an exclusion list. If you need more details about these endpoint properties, see Membership rules for custom endpoints.

You can continue connecting to and using a custom endpoint while the changes from an edit action are in progress.

Console

To edit a custom endpoint with the AWS Management Console, you can select the endpoint on the cluster detail page, or bring up the detail page for the endpoint, and choose the **Edit** action.



RDS > Clusters: [redacted] > Edit endpoint: [redacted].cluster-custom-[redacted].rds.amazonaws.com

Edit endpoint: [redacted].cluster-custom-[redacted].rds.amazonaws.com

Endpoint members

Filter database

	DB instance name	Role
☑	[redacted]	Reader
☑	[redacted]	Reader
□	[redacted]	Writer
□	[redacted]	Reader

Additional configuration

□ Attach future instances added to this cluster

AWS CLI

To edit a custom endpoint with the AWS CLI, run the modify-db-cluster-endpoint command.

The following commands change the set of DB instances that apply to a custom endpoint and optionally switches between the behavior of a static or exclusion list. The `--static-members` and `--excluded-members` parameters take a space-separated list of DB instance identifiers.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier my-custom-endpoint
 \
  --static-members db-instance-id-1 db-instance-id-2 db-instance-id-3 \
  --region region_name
```

```
aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier my-custom-endpoint
 \
  --excluded-members db-instance-id-4 db-instance-id-5 \
  --region region_name
```

For Windows:

```
aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier my-custom-endpoint
 ^
  --static-members db-instance-id-1 db-instance-id-2 db-instance-id-3 ^
  --region region_name

aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier my-custom-endpoint
 ^
  --excluded-members db-instance-id-4 db-instance-id-5 ^
  --region region_name
```

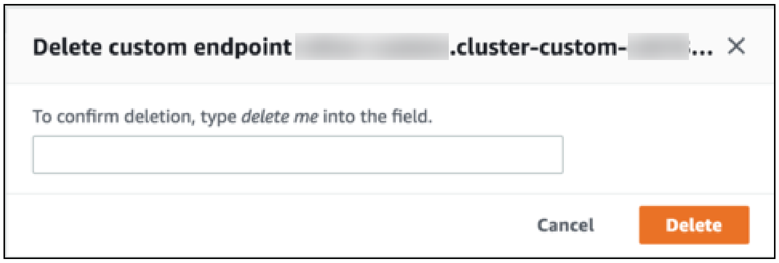
RDS API

To edit a custom endpoint with the RDS API, run the ModifyDBClusterEndpoint.html operation.

Deleting a custom endpoint

Console

To delete a custom endpoint with the AWS Management Console, go to the cluster detail page, select the appropriate custom endpoint, and select the **Delete** action.



Delete custom endpoint [redacted].cluster-custom-[redacted]...

To confirm deletion, type *delete me* into the field.

AWS CLI

To delete a custom endpoint with the AWS CLI, run the delete-db-cluster-endpoint command.

The following command deletes a custom endpoint. You don't need to specify the associated cluster, but you must specify the region.

For Linux, macOS, or Unix:

```
aws rds delete-db-cluster-endpoint --db-cluster-endpoint-identifier custom-end-point-id
 \
  --region region_name
```

For Windows:

```
aws rds delete-db-cluster-endpoint --db-cluster-endpoint-identifier custom-end-point-id
 ^
  --region region_name
```

RDS API

To delete a custom endpoint with the RDS API, run the DeleteDBClusterEndpoint operation.

End-to-end AWS CLI example for custom endpoints

The following tutorial uses AWS CLI examples with Unix shell syntax to show you might define a cluster with several "small" DB instances and a few "big" DB instances, and create custom endpoints to connect to each set of DB instances. To run similar commands on your own system, you should be familiar enough with the basics of working with Aurora clusters and AWS CLI usage to supply your own values for parameters such as region, subnet group, and VPC security group.

This example demonstrates the initial setup steps: creating an Aurora cluster and adding DB instances to it. This is a heterogeneous cluster, meaning not all the DB instances have the same capacity. Most instances use the AWS instance class `db.r4.4xlarge`, but the last two DB instances use `db.r4.16xlarge`. Each of these sample `create-db-instance` commands prints its output to the screen and saves a copy of the JSON in a file for later inspection.

```
aws rds create-db-cluster --db-cluster-identifier custom-endpoint-demo --engine aurora-
mysql \
     --engine-version 8.0.mysql_aurora.3.02.0 --master-username $MASTER_USER --manage-
master-user-password \
     --db-subnet-group-name $SUBNET_GROUP  --vpc-security-group-ids $VPC_SECURITY_GROUP
 \
     --region $REGION

for i in 01 02 03 04 05 06 07 08
do
```

```
  aws rds create-db-instance --db-instance-identifier custom-endpoint-demo-${i} \
     --engine aurora --db-cluster-identifier custom-endpoint-demo --db-instance-class
 db.r4.4xlarge \
     --region $REGION \
     | tee custom-endpoint-demo-${i}.json
done

for i in 09 10
do
  aws rds create-db-instance --db-instance-identifier custom-endpoint-demo-${i} \
     --engine aurora --db-cluster-identifier custom-endpoint-demo --db-instance-class
 db.r4.16xlarge \
     --region $REGION \
     | tee custom-endpoint-demo-${i}.json
done
```

The larger instances are reserved for specialized kinds of reporting queries. To make it unlikely for them to be promoted to the primary instance, the following example changes their promotion tier to the lowest priority. This example specifies the `--manage-master-user-password` option to generate the master user password and manage it in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

```
for i in 09 10
do
  aws rds modify-db-instance --db-instance-identifier custom-endpoint-demo-${i} \
    --region $REGION --promotion-tier 15
done
```

Suppose that you want to use the two "bigger" instances only for the most resource-intensive queries. To do this, you can first create a custom read-only endpoint. Then you can add a static list of members so that the endpoint connects only to those DB instances. Those instances are already in the lowest promotion tier, making it unlikely either of them will be promoted to the primary instance. If one of them is promoted to the primary instance, it becomes unreachable through this endpoint because we specified the `READER` type instead of the `ANY` type.

The following example demonstrates the create and modify endpoint commands, and sample JSON output showing the initial and modified state of the custom endpoint.

```
$ aws rds create-db-cluster-endpoint --region $REGION \
    --db-cluster-identifier custom-endpoint-demo \
    --db-cluster-endpoint-identifier big-instances --endpoint-type reader
{
    "EndpointType": "CUSTOM",
    "Endpoint": "big-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
    "DBClusterEndpointIdentifier": "big-instances",
    "DBClusterIdentifier": "custom-endpoint-demo",
    "StaticMembers": [],
    "DBClusterEndpointResourceIdentifier": "cluster-endpoint-
W7PE3TLLFNSHXQKFU6J6NV5FHU",
    "ExcludedMembers": [],
    "CustomEndpointType": "READER",
    "Status": "creating",
    "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:big-instances"
}

$ aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier big-instances \
    --static-members custom-endpoint-demo-09 custom-endpoint-demo-10 --region $REGION
{
    "EndpointType": "CUSTOM",
    "ExcludedMembers": [],
    "DBClusterEndpointIdentifier": "big-instances",
    "DBClusterEndpointResourceIdentifier": "cluster-endpoint-
W7PE3TLLFNSHXQKFU6J6NV5FHU",
    "CustomEndpointType": "READER",
    "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:big-instances",
    "StaticMembers": [
        "custom-endpoint-demo-10",
        "custom-endpoint-demo-09"
    ],
    "Status": "modifying",
    "Endpoint": "big-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
    "DBClusterIdentifier": "custom-endpoint-demo"
}
```

The default READER endpoint for the cluster can connect to either the "small" or "big" DB instances, making it impractical to predict query performance and scalability when the cluster becomes busy. To divide the workload cleanly between the sets of DB instances, you can ignore the default READER endpoint and create a second custom endpoint that connects to all other DB instances. The following example does so by creating a custom endpoint and then adding an exclusion list.

Any other DB instances you add to the cluster later will be added to this endpoint automatically. The ANY type means that this endpoint is associated with eight instances in total: the primary instance and another seven Aurora Replicas. If the example used the READER type, the custom endpoint would only be associated with the seven Aurora Replicas.

```
$ aws rds create-db-cluster-endpoint --region $REGION --db-cluster-identifier custom-
endpoint-demo \
    --db-cluster-endpoint-identifier small-instances --endpoint-type any
{
    "Status": "creating",
    "DBClusterEndpointIdentifier": "small-instances",
    "CustomEndpointType": "ANY",
    "EndpointType": "CUSTOM",
    "Endpoint": "small-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
    "StaticMembers": [],
    "ExcludedMembers": [],
    "DBClusterIdentifier": "custom-endpoint-demo",
    "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:small-instances",
    "DBClusterEndpointResourceIdentifier": "cluster-
endpoint-6RDDXQOC3AKKZT2PRD7ST37BMY"
}

$ aws rds modify-db-cluster-endpoint --db-cluster-endpoint-identifier small-instances \
    --excluded-members custom-endpoint-demo-09 custom-endpoint-demo-10 --region $REGION
{
    "DBClusterEndpointIdentifier": "small-instances",
    "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:c7tj4example:cluster-
endpoint:small-instances",
    "DBClusterEndpointResourceIdentifier": "cluster-
endpoint-6RDDXQOC3AKKZT2PRD7ST37BMY",
    "CustomEndpointType": "ANY",
    "Endpoint": "small-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
    "EndpointType": "CUSTOM",
    "ExcludedMembers": [
        "custom-endpoint-demo-09",
        "custom-endpoint-demo-10"
    ],
    "StaticMembers": [],
    "DBClusterIdentifier": "custom-endpoint-demo",
    "Status": "modifying"
}
```

The following example checks the state of the endpoints for this cluster. The cluster still has its original cluster endpoint, with `EndPointType` of `WRITER`, which you would still use for administration, ETL, and other write operations. It still has its original `READER` endpoint, which you wouldn't use because each connection to it might be directed to a "small" or "big" DB instance. The custom endpoints make this behavior predictable, with connections guaranteed to use one of the "small" or "big" DB instances based on the endpoint you specify.

```
$ aws rds describe-db-cluster-endpoints --region $REGION
{
    "DBClusterEndpoints": [
        {
            "EndpointType": "WRITER",
            "Endpoint": "custom-endpoint-demo.cluster-c7tj4example.ca-
central-1.rds.amazonaws.com",
            "Status": "available",
            "DBClusterIdentifier": "custom-endpoint-demo"
        },
        {
            "EndpointType": "READER",
            "Endpoint": "custom-endpoint-demo.cluster-ro-c7tj4example.ca-
central-1.rds.amazonaws.com",
            "Status": "available",
            "DBClusterIdentifier": "custom-endpoint-demo"
        },
        {
            "Endpoint": "small-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
            "CustomEndpointType": "ANY",
            "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:small-instances",
            "ExcludedMembers": [
                "custom-endpoint-demo-09",
                "custom-endpoint-demo-10"
            ],
            "DBClusterEndpointResourceIdentifier": "cluster-
endpoint-6RDDXQOC3AKKZT2PRD7ST37BMY",
            "DBClusterIdentifier": "custom-endpoint-demo",
            "StaticMembers": [],
            "EndpointType": "CUSTOM",
            "DBClusterEndpointIdentifier": "small-instances",
            "Status": "modifying"
        },
        {
            "Endpoint": "big-instances.cluster-custom-c7tj4example.ca-
central-1.rds.amazonaws.com",
            "CustomEndpointType": "READER",
            "DBClusterEndpointArn": "arn:aws:rds:ca-central-1:111122223333:cluster-
endpoint:big-instances",
            "ExcludedMembers": [],
            "DBClusterEndpointResourceIdentifier": "cluster-endpoint-
W7PE3TLLFNSHXQKFU6J6NV5FHU",
            "DBClusterIdentifier": "custom-endpoint-demo",
            "StaticMembers": [
                "custom-endpoint-demo-10",
                "custom-endpoint-demo-09"
            ],
            "EndpointType": "CUSTOM",
            "DBClusterEndpointIdentifier": "big-instances",
            "Status": "available"
        }
    ]
}
```

The final examples demonstrate how successive database connections to the custom endpoints connect to the various DB instances in the Aurora cluster. The `small-instances` endpoint always connects to the `db.r4.4xlarge` DB instances, which are the lower-numbered hosts in this cluster.

```
$ mysql -h small-instances.cluster-custom-c7tj4example.ca-central-1.rds.amazonaws.com -
u $MYUSER -p
mysql> select @@aurora_server_id;
+-------------------------+
| @@aurora_server_id      |
+-------------------------+
| custom-endpoint-demo-02 |
+-------------------------+

$ mysql -h small-instances.cluster-custom-c7tj4example.ca-central-1.rds.amazonaws.com -
u $MYUSER -p
mysql> select @@aurora_server_id;
+-------------------------+
| @@aurora_server_id      |
+-------------------------+
| custom-endpoint-demo-07 |
+-------------------------+

$ mysql -h small-instances.cluster-custom-c7tj4example.ca-central-1.rds.amazonaws.com -
u $MYUSER -p
mysql> select @@aurora_server_id;
+-------------------------+
| @@aurora_server_id      |
+-------------------------+
| custom-endpoint-demo-01 |
+-------------------------+
```

The `big-instances` endpoint always connects to the `db.r4.16xlarge` DB instances, which are the two highest-numbered hosts in this cluster.

```
$ mysql -h big-instances.cluster-custom-c7tj4example.ca-central-1.rds.amazonaws.com -u
 $MYUSER -p
mysql> select @@aurora_server_id;
+-------------------------+
| @@aurora_server_id      |
+-------------------------+
| custom-endpoint-demo-10 |
+-------------------------+

$ mysql -h big-instances.cluster-custom-c7tj4example.ca-central-1.rds.amazonaws.com -u
 $MYUSER -p
mysql> select @@aurora_server_id;
+-------------------------+
| @@aurora_server_id      |
+-------------------------+
| custom-endpoint-demo-09 |
+-------------------------+
```

Using the instance endpoints

Each DB instance in an Aurora cluster has its own built-in instance endpoint, whose name and other attributes are managed by Aurora. You can't create, delete, or modify this kind of endpoint. You might be familiar with instance endpoints if you use Amazon RDS. However, with Aurora you typically use the writer and reader endpoints more often than the instance endpoints.

In day-to-day Aurora operations, the main way that you use instance endpoints is to diagnose capacity or performance issues that affect one specific instance in an Aurora cluster. While connected to a specific instance, you can examine its status variables, metrics, and so on. Doing this can help you determine what's happening for that instance that's different from what's happening for other instances in the cluster.

In advanced use cases, you might configure some DB instances differently than others. In this case, use the instance endpoint to connect directly to an instance that is smaller, larger, or otherwise has different characteristics than the others. Also, set up failover priority so that this special DB instance is the last choice to take over as the primary instance. We recommend that you use custom endpoints instead of the instance endpoint in such cases. Doing so simplifies connection management and high availability as you add more DB instances to your cluster.

How Aurora endpoints work with high availability

For clusters where high availability is important, use the writer endpoint for read/write or general-purpose connections and the reader endpoint for read-only connections. The writer and reader endpoints manage DB instance failover better than instance endpoints do. Unlike the instance endpoints, the writer and reader endpoints automatically change which DB instance they connect to if a DB instance in your cluster becomes unavailable.

If the primary DB instance of a DB cluster fails, Aurora automatically fails over to a new primary DB instance. It does so by either promoting an existing Aurora Replica to a new primary DB instance or creating a new primary DB instance. If a failover occurs, you can use the writer endpoint to reconnect to the newly promoted or created primary DB instance, or use the reader endpoint to reconnect to one of the Aurora Replicas in the DB cluster. During a failover, the reader endpoint might direct connections to the new primary DB instance of a DB cluster for a short time after an Aurora Replica is promoted to the new primary DB instance.

If you design your own application logic to manage connections to instance endpoints, you can manually or programmatically discover the resulting set of available DB instances in the DB cluster. Use the describe-db-clusters AWS CLI command or DescribeDBClusters RDS API operation to find the DB cluster and reader endpoints, DB instances, whether DB instances are readers, and their promotion tiers. You can then confirm their instance classes after failover and connect to an appropriate instance endpoint.

For more information about failovers, see Fault tolerance for an Aurora DB cluster.

Aurora DB instance classes

The DB instance class determines the computation and memory capacity of an Amazon Aurora DB instance. The DB instance class that you need depends on your processing power and memory requirements.

A DB instance class consists of both the DB instance class type and the size. For example, db.r6g is a memory-optimized DB instance class type powered by AWS Graviton2 processors. Within the db.r6g instance class type, db.r6g.2xlarge is a DB instance class. The size of this class is 2xlarge.

For more information about instance class pricing, see Amazon RDS pricing.

Topics

DB instance class types

Amazon Aurora supports DB instance classes for the following use cases:

- Aurora Serverless v2
- Memory-optimized
- Burstable-performance
- Optimized Reads

For more information about Amazon EC2 instance types, see Instance types in the Amazon EC2 documentation.

Aurora Serverless v2 instance class type

The following Aurora Serverless v2 type is available:

- **db.serverless** – A special DB instance class type used by Aurora Serverless v2. Aurora adjusts the compute, memory, and network resources dynamically as the workload changes. For usage details, see Using Aurora Serverless v2.

Memory-optimized instance class type

The memory-optimized X family supports the following instance classes:

- **db.x2g –** Instance classes optimized for memory-intensive applications and powered by AWS Graviton2 processors. These instance classes offer low cost per GiB of memory.

  You can modify a DB instance to use one of the DB instance classes powered by AWS Graviton2 processors. To do so, complete the same steps as with any other DB instance modification.

The memory-optimized R family supports the following instance class types:

- **db.r7g –** Instance classes powered by AWS Graviton3 processors. These instance classes are ideal for running memory-intensive workloads.

  You can modify a DB instance to use one of the DB instance classes powered by AWS Graviton3 processors. To do so, complete the same steps as with any other DB instance modification.
- **db.r6g –** Instance classes powered by AWS Graviton2 processors. These instance classes are ideal for running memory-intensive workloads.

  You can modify a DB instance to use one of the DB instance classes powered by AWS Graviton2 processors. To do so, complete the same steps as with any other DB instance modification.
- **db.r6i –** Instance classes powered by 3rd Generation Intel Xeon Scalable processors. These instance classes are SAP-Certified and are an ideal fit for memory-intensive workloads in open-source databases such as MySQL and PostgreSQL.
- **db.r4 –** These instance classes are supported only for Aurora PostgreSQL 11 and 12 versions. For all Aurora PostgreSQL DB clusters that use db.r4 DB instance classes, we recommend that you upgrade to a higher generation instance class as soon as possible.

  The db.r4 instance classes aren't available for the Aurora I/O-Optimized cluster storage configuration.
- **db.r3 –** Instance classes that provide memory optimization.

  Amazon Aurora has started the end-of-life process for db.r3 DB instance classes using the following schedule, which includes upgrade recommendations. For all Aurora MySQL DB clusters that use db.r3 DB instance classes, we recommend that you upgrade to a db.r5 or higher DB instance class as soon as possible.

Action or recommendation	Dates
You can no longer create Aurora MySQL DB clusters that use db.r3 DB instance classes.	Now
Amazon Aurora started automatic upgrades of Aurora MySQL DB clusters that use db.r3 DB instance classes to equivalent db.r5 or higher DB instance classes.	January 31, 2023

Burstable-performance instance class types

The following burstable-performance DB instance class types are available:

- **db.t4g** – General-purpose instance classes powered by Arm-based AWS Graviton2 processors. These instance classes deliver better price performance than previous burstable-performance DB instance classes for a broad set of burstable general-purpose workloads. Amazon RDS db.t4g instances are configured for Unlimited mode. This means that they can burst beyond the baseline over a 24-hour window for an additional charge.

  You can modify a DB instance to use one of the DB instance classes powered by AWS Graviton2 processors. To do so, complete the same steps as with any other DB instance modification.
- **db.t3** – Instance classes that provide a baseline performance level, with the ability to burst to full CPU usage. The db.t3 instances are configured for Unlimited mode. These instance classes provide more computing capacity than the previous db.t2 instance classes. They are powered by the AWS Nitro System, a combination of dedicated hardware and lightweight hypervisor. We recommend using these instance classes only for development and test servers, or other non-production servers.
- **db.t2** – Instance classes that provide a baseline performance level, with the ability to burst to full CPU usage. The db.t2 instances are configured for Unlimited mode. We recommend using these instance classes only for development and test servers, or other non-production servers.

  The db.t2 instance classes aren't available for the Aurora I/O-Optimized cluster storage configuration.

> **Note**
>
> We recommend using the T DB instance classes only for development, test, or other nonproduction servers. For more detailed recommendations for the T instance classes, see Using T instance classes for development and testing.

For DB instance class hardware specifications, see Hardware specifications for DB instance classes for Aurora.

Optimized Reads instance class type

The following Optimized Reads instance class types are available:

- **db.r6gd** – Instance classes powered by AWS Graviton2 processors. These instance classes are ideal for running memory-intensive workloads and offer local NVMe-based SSD block-level storage for applications that need high-speed, low latency local storage.
- **db.r6id** – Instance classes powered by 3rd Generation Intel Xeon Scalable processors. These instance classes are SAP-Certified and are an ideal fit for memory-intensive workloads. They offer a maximum memory of 1 TiB and up to 7.6 TB of direct-attached NVMe-based SSD storage.

Supported DB engines for DB instance classes

In the following table, you can find details about supported Amazon Aurora DB instance classes for the Aurora DB engines.

Instance class	Aurora MySQL	Aurora PostgreSQL
db.serverless – Aurora Serverless v2 instance class with automatic capacity scaling		
db.serverless	See Supported Regions and Aurora DB engines for Aurora Serverless v2	See Supported Regions and Aurora DB engines for Aurora Serverless v2
db.x2g – memory-optimized instance classes powered by AWS Graviton2 processors		

Instance class	Aurora MySQL	Aurora PostgreSQL
db.x2g.16xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.12xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.8xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.4xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.2xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.x2g.large	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6gd – Optimized Reads instance classes powered by AWS Graviton2 processors		
db.r6gd.16xlarge	No	15.4 and higher, 14.9 and higher
db.r6gd.12xlarge	No	15.4 and higher, 14.9 and higher
db.r6gd.8xlarge	No	15.4 and higher, 14.9 and higher
db.r6gd.4xlarge	No	15.4 and higher, 14.9 and higher

Instance class	Aurora MySQL	Aurora PostgreSQL
db.r6gd.2xlarge	No	15.4 and higher, 14.9 and higher
db.r6gd.xlarge	No	15.4 and higher, 14.9 and higher
db.r6id – Optimized Reads instance classes		
db.r6id.32xlarge	No	15.4 and higher, 14.9 and higher
db.r6id.24xlarge	No	15.4 and higher, 14.9 and higher
db.r7g – memory-optimized instance classes powered by AWS Graviton3 processors		
db.r7g.16xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.12xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.8xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.4xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.2xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.xlarge	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r7g.large	2.12.0 and higher, 3.03.1 and higher	15.2 and higher, 14.7 and higher, 13.10 and higher
db.r6g – memory-optimized instance classes powered by AWS Graviton2 processors		
db.r6g.16xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher

Instance class	Aurora MySQL	Aurora PostgreSQL
db.r6g.12xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6g.8xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6g.4xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6g.2xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6g.xlarge	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6g.large	2.09.2 and higher, 2.10.0 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.8 and higher, 11.9, 11.12 and higher
db.r6i – memory-optimized instance classes		
db.r6i.32xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.24xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.16xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.12xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher

Instance class	Aurora MySQL	Aurora PostgreSQL
db.r6i.8xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.4xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.2xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.xlarge	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r6i.large	2.11.0 and higher, 3.02.1 and higher	15.2 and higher, 14.3 and higher, 13.5 and higher, 12.9 and higher
db.r5 – memory-optimized instance classes		
db.r5.24xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.16xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.12xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.8xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.4xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.2xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.xlarge	All version 2.x; 3.01.0 and higher	All currently available versions
db.r5.large	All version 2.x; 3.01.0 and higher	All currently available versions
db.r4 – memory-optimized instance classes		
db.r4.16xlarge	All version 2.x; not supported in 3.01.0 and higher	No

Instance class	Aurora MySQL	Aurora PostgreSQL
db.r4.8xlarge	All version 2.x; not supported in 3.01.0 and higher	No
db.r4.4xlarge	All version 2.x; not supported in 3.01.0 and higher	No
db.r4.2xlarge	All version 2.x; not supported in 3.01.0 and higher	No
db.r4.xlarge	All version 2.x; not supported in 3.01.0 and higher	No
db.r4.large	All version 2.x; not supported in 3.01.0 and higher	No
db.t4g – burstable-performance instance classes powered by AWS Graviton2 processors		
db.t4g.2xlarge	No	No
db.t4g.xlarge	No	No
db.t4g.large	2.11.1 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.7 and higher, 11.12 and higher
db.t4g.medium	2.11.1 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.7 and higher, 11.12 and higher
db.t4g.small	No	No
db.t3 – burstable-performance instance classes		
db.t3.2xlarge	No	No
db.t3.xlarge	No	No

Instance class	Aurora MySQL	Aurora PostgreSQL
db.t3.large	2.11.1 and higher, 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.7 and higher, 11.12 and higher
db.t3.medium	All 2.x versions; 3.01.0 and higher	15.2 and higher, 14.3 and higher, 13.3 and higher, 12.7 and higher, 11.12 and higher
db.t3.small	All 2.x versions; not supported in 3.01.0 and higher	No
db.t3.micro	No	No
db.t2 – burstable-performance instance classes		
db.t2.medium	All 2.x versions; not supported in 3.01.0 and higher	No
db.t2.small	All 2.x versions; not supported in 3.01.0 and higher	No

Determining DB instance class support in AWS Regions

To determine the DB instance classes supported by each DB engine in a specific AWS Region, you can take one of several approaches. You can use the AWS Management Console, the Amazon RDS Pricing page, or the describe-orderable-db-instance-options AWS CLI command.

Note

When you perform operations with the AWS Management Console, it automatically shows the supported DB instance classes for a specific DB engine, DB engine version, and AWS Region. Examples of the operations that you can perform include creating and modifying a DB instance.

Contents

- Using the Amazon RDS pricing page to determine DB instance class support in AWS Regions
- Using the AWS CLI to determine DB instance class support in AWS Regions
  - Listing the DB instance classes that are supported by a specific DB engine version in an AWS Region
  - Listing the DB engine versions that support a specific DB instance class in an AWS Region

Using the Amazon RDS pricing page to determine DB instance class support in AWS Regions

You can use the Amazon Aurora Pricing page to determine the DB instance classes supported by each DB engine in a specific AWS Region.

To use the pricing page to determine the DB instance classes supported by each engine in a Region

1. Go to Amazon Aurora Pricing.
2. Choose an Amazon Aurora engine in the **AWS Pricing Calculator** section.
3. In **Choose a Region**, choose an AWS Region.
4. In **Cluster Configuration Option**, choose a configuration option.
5. Use the section for compatible instances to view the supported DB instance classes.
6. (Optional) Choose other options in the calculator, and then choose **Save and view summary** or **Save and add service**.

Using the AWS CLI to determine DB instance class support in AWS Regions

You can use the AWS CLI to determine which DB instance classes are supported for specific DB engines and DB engine versions in an AWS Region.

To use the AWS CLI examples following, enter valid values for the DB engine, DB engine version, DB instance class, and AWS Region. The following table shows the valid DB engine values.

Engine name	Engine value in CLI commands	More information about versions
MySQL 5.7-compatible and 8.0-compatible Aurora	`aurora-mysql`	Database engine updates for Amazon Aurora MySQL version 2 and Database engine updates for Amazon Aurora MySQL version 3 in the *Release Notes for Aurora MySQL*
Aurora PostgreSQL	`aurora-postgresql`	*Release Notes for Aurora PostgreSQL*

For information about AWS Region names, see AWS Regions.

The following examples demonstrate how to determine DB instance class support in an AWS Region using the describe-orderable-db-instance-options AWS CLI command.

Topics

- Listing the DB instance classes that are supported by a specific DB engine version in an AWS Region
- Listing the DB engine versions that support a specific DB instance class in an AWS Region

Listing the DB instance classes that are supported by a specific DB engine version in an AWS Region

To list the DB instance classes that are supported by a specific DB engine version in an AWS Region, run the following command.

For Linux, macOS, or Unix:

```
aws rds describe-orderable-db-instance-options --engine engine --engine-version version
 \
    --query "OrderableDBInstanceOptions[].
{DBInstanceClass:DBInstanceClass,SupportedEngineModes:SupportedEngineModes[0]}" \
    --output table \
    --region region
```

For Windows:

```
aws rds describe-orderable-db-instance-options --engine engine --engine-version version
 ^
    --query "OrderableDBInstanceOptions[].
{DBInstanceClass:DBInstanceClass,SupportedEngineModes:SupportedEngineModes[0]}" ^
    --output table ^
    --region region
```

The output also shows the engine modes that are supported for each DB instance class.

For example, the following command lists the supported DB instance classes for version 13.6 of the Aurora PostgreSQL DB engine in US East (N. Virginia).

For Linux, macOS, or Unix:

```
aws rds describe-orderable-db-instance-options --engine aurora-postgresql --engine-
version 15.3 \
    --query "OrderableDBInstanceOptions[].
{DBInstanceClass:DBInstanceClass,SupportedEngineModes:SupportedEngineModes[0]}" \
    --output table \
    --region us-east-1
```

For Windows:

```
aws rds describe-orderable-db-instance-options --engine aurora-postgresql --engine-
version 15.3 ^
    --query "OrderableDBInstanceOptions[].
{DBInstanceClass:DBInstanceClass,SupportedEngineModes:SupportedEngineModes[0]}"  ^
    --output table ^
    --region us-east-1
```

Listing the DB engine versions that support a specific DB instance class in an AWS Region

To list the DB engine versions that support a specific DB instance class in an AWS Region, run the following command.

For Linux, macOS, or Unix:

```
aws rds describe-orderable-db-instance-options --engine engine --db-instance-
class DB_instance_class \
    --query "OrderableDBInstanceOptions[].
{EngineVersion:EngineVersion,SupportedEngineModes:SupportedEngineModes[0]}" \
    --output table \
    --region region
```

For Windows:

```
aws rds describe-orderable-db-instance-options --engine engine --db-instance-
class DB_instance_class ^
    --query "OrderableDBInstanceOptions[].
{EngineVersion:EngineVersion,SupportedEngineModes:SupportedEngineModes[0]}" ^
    --output table ^
    --region region
```

The output also shows the engine modes that are supported for each DB engine version.

For example, the following command lists the DB engine versions of the Aurora PostgreSQL DB engine that support the db.r5.large DB instance class in US East (N. Virginia).

For Linux, macOS, or Unix:

```
aws rds describe-orderable-db-instance-options --engine aurora-postgresql --db-
instance-class db.r7g.large \
    --query "OrderableDBInstanceOptions[].
{EngineVersion:EngineVersion,SupportedEngineModes:SupportedEngineModes[0]}" \
    --output table \
    --region us-east-1
```

For Windows:

```
aws rds describe-orderable-db-instance-options --engine aurora-postgresql --db-
instance-class db.r7g.large ^
    --query "OrderableDBInstanceOptions[].
{EngineVersion:EngineVersion,SupportedEngineModes:SupportedEngineModes[0]}" ^
    --output table ^
    --region us-east-1
```

Hardware specifications for DB instance classes for Aurora

The following terminology is used to describe hardware specifications for DB instance classes:

vCPU

The number of virtual central processing units (CPUs). A *virtual CPU* is a unit of capacity that you can use to compare DB instance classes. Instead of purchasing or leasing a particular processor to use for several months or years, you are renting capacity by the hour. Our goal is to make a consistent and specific amount of CPU capacity available, within the limits of the actual underlying hardware.

ECU

The relative measure of the integer processing power of an Amazon EC2 instance. To make it easy for developers to compare CPU capacity between different instance classes, we have defined an Amazon EC2 Compute Unit. The amount of CPU that is allocated to a particular instance is expressed in terms of these EC2 Compute Units. One ECU currently provides CPU capacity equivalent to a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor.

Memory (GiB)

The RAM, in gibibytes, allocated to the DB instance. There is often a consistent ratio between memory and vCPU. As an example, take the db.r4 instance class, which has a memory to vCPU ratio similar to the db.r5 instance class. However, for most use cases the db.r5 instance class provides better, more consistent performance than the db.r4 instance class.

Max. EBS bandwidth (Mbps)

The maximum EBS bandwidth in megabits per second. Divide by 8 to get the expected throughput in megabytes per second.

> **Note**
>
> This figure refers to I/O bandwidth for local storage within the DB instance. It doesn't apply to communication with the Aurora cluster volume.

Network bandwidth

The network speed relative to other DB instance classes.

In the following table, you can find hardware details about the Amazon RDS DB instance classes for Aurora.

For information about Aurora DB engine support for each DB instance class, see Supported DB engines for DB instance classes.

Instance class	vCPU	ECU	Memory (GiB)	Max. bandwidth (Mbps) of local storage	Network performance (Gbps)
db.x2g – memory-optimized instance classes					
db.x2g.16xlarge	64	—	1024	19,000	25
db.x2g.12xlarge	48	—	768	14,250	20
db.x2g.8xlarge	32	—	512	9,500	12
db.x2g.4xlarge	16	—	256	4,750	Up to 10
db.x2g.2xlarge	8	—	128	Up to 4,750	Up to 10
db.x2g.xlarge	4	—	64	Up to 4,750	Up to 10
db.x2g.large	2	—	32	Up to 4,750	Up to 10
db.r7g – memory-optimized instance classes powered by AWS Graviton3 processors					
db.r7g.16xlarge	64	—	512	20,000	30
db.r7g.12xlarge	48	—	384	15,000	22.5
db.r7g.8xlarge	32	—	256	10,000	15
db.r7g.4xlarge	16	—	128	Up to 10,000	Up to 15
db.r7g.2xlarge	8	—	64	Up to 10,000	Up to 15
db.r7g.xlarge	4	—	32	Up to 10,000	Up to 12.5
db.r7g.large	2	—	16	Up to 10,000	Up to 12.5
db.r6g – memory-optimized instance classes powered by AWS Graviton2 processors					
db.r6g.16xlarge	64	—	512	19,000	25

Instance class	vCPU	ECU	Memory (GiB)	Max. bandwidth (Mbps) of local storage	Network performance (Gbps)
db.r6g.12xlarge	48	—	384	13,500	20
db.r6g.8xlarge	32	—	256	9,000	12
db.r6g.4xlarge	16	—	128	4,750	Up to 10
db.r6g.2xlarge	8	—	64	Up to 4,750	Up to 10
db.r6g.xlarge	4	—	32	Up to 4,750	Up to 10
db.r6g.large	2	—	16	Up to 4,750	Up to 10
db.r6i – memory-optimized instance classes					
db.r6i.32xlarge	128	—	1,024	40,000	50
db.r6i.24xlarge	96	—	768	30,000	37.5
db.r6i.16xlarge	64	—	512	20,000	25
db.r6i.12xlarge	48	—	384	15,000	18.75
db.r6i.8xlarge	32	—	256	10,000	12.5
db.r6i.4xlarge	16	—	128	Up to 10,000	Up to 12.5
db.r6i.2xlarge	8	—	64	Up to 10,000	Up to 12.5
db.r6i.xlarge	4	—	32	Up to 10,000	Up to 12.5
db.r6i.large	2	—	16	Up to 10,000	Up to 12.5
db.r5 – memory-optimized instance classes					
db.r5.24xlarge	96	347	768	19,000	25
db.r5.16xlarge	64	264	512	13,600	20

Instance class	vCPU	ECU	Memory (GiB)	Max. bandwidth (Mbps) of local storage	Network performance (Gbps)
db.r5.12xlarge	48	173	384	9,500	12
db.r5.8xlarge	32	132	256	6,800	10
db.r5.4xlarge	16	71	128	4,750	Up to 10
db.r5.2xlarge	8	38	64	Up to 4,750	Up to 10
db.r5.xlarge	4	19	32	Up to 4,750	Up to 10
db.r5.large	2	10	16	Up to 4,750	Up to 10
db.r4 – memory-optimized instance classes					
db.r4.16xlarge	64	195	488	14,000	25
db.r4.8xlarge	32	99	244	7,000	10
db.r4.4xlarge	16	53	122	3,500	Up to 10
db.r4.2xlarge	8	27	61	1,700	Up to 10
db.r4.xlarge	4	13.5	30.5	850	Up to 10
db.r4.large	2	7	15.25	425	Up to 10
db.t4g – burstable-performance instance classes					
db.t4g.large	2	—	8	Up to 2,780	Up to 5
db.t4g.medium	2	—	4	Up to 2,085	Up to 5
db.t3 – burstable-performance instance classes					
db.t3.large	2	Variable	8	Up to 2,048	Up to 5
db.t3.medium	2	Variable	4	Up to 1,536	Up to 5

Instance class	vCPU	ECU	Memory (GiB)	Max. bandwidth (Mbps) of local storage	Network performance (Gbps)
db.t3.small	2	Variable	2	Up to 1,536	Up to 5
db.t2 – burstable-performance instance classes					
db.t2.medium	2	Variable	4	—	Moderate
db.t2.small	1	Variable	2	—	Low

Amazon Aurora storage and reliability

Following, you can learn about the Aurora storage subsystem. Aurora uses a distributed and shared storage architecture that is an important factor in performance, scalability, and reliability for Aurora clusters.

Topics

Overview of Amazon Aurora storage

Aurora data is stored in the *cluster volume*, which is a single, virtual volume that uses solid state drives (SSDs). A cluster volume consists of copies of the data across three Availability Zones in a single AWS Region. Because the data is automatically replicated across Availability Zones, your data is highly durable with less possibility of data loss. This replication also ensures that your database is more available during a failover. It does so because the data copies already exist in the other Availability Zones and continue to serve data requests to the DB instances in your DB cluster. The amount of replication is independent of the number of DB instances in your cluster.

Aurora uses separate local storage for nonpersistent, temporary files. This includes files that are used for such purposes as sorting large data sets during query processing, and building indexes. For more information, see Temporary storage limits for Aurora MySQL and Temporary storage limits for Aurora PostgreSQL.

What the cluster volume contains

The Aurora cluster volume contains all your user data, schema objects, and internal metadata such as the system tables and the binary log. For example, Aurora stores all the tables, indexes, binary large objects (BLOBs), stored procedures, and so on for an Aurora cluster in the cluster volume.

The Aurora shared storage architecture makes your data independent from the DB instances in the cluster. For example, you can add a DB instance quickly because Aurora doesn't make a new copy of the table data. Instead, the DB instance connects to the shared volume that already contains all your data. You can remove a DB instance from a cluster without removing any of the underlying data from the cluster. Only when you delete the entire cluster does Aurora remove the data.

Storage configurations for Amazon Aurora DB clusters

Amazon Aurora has two DB cluster storage configurations:

- **Aurora I/O-Optimized** – Improved price performance and predictability for I/O-intensive applications. You pay only for the usage and storage of your DB clusters, with no additional charges for read and write I/O operations.

  Aurora I/O-Optimized is the best choice when your I/O spending is 25% or more of your total Aurora database spending.

  You can choose Aurora I/O-Optimized when you create or modify a DB cluster with a DB engine version that supports the Aurora I/O-Optimized cluster configuration. You can switch from Aurora I/O-Optimized to Aurora Standard at any time.
- **Aurora Standard** – Cost-effective pricing for many applications with moderate I/O usage. In addition to the usage and storage of your DB clusters, you also pay a standard rate per 1 million requests for I/O operations.

  Aurora Standard is the best choice when your I/O spending is less than 25% of your total Aurora database spending.

You can switch from Aurora Standard to Aurora I/O-Optimized once every 30 days. There's no downtime when you switch from Aurora Standard to Aurora I/O-Optimized, or from Aurora I/O-Optimized to Aurora Standard.

For information on AWS Region and version support, see Supported Regions and Aurora DB engines for cluster storage configurations.

For more information on pricing for Amazon Aurora storage configurations, see Amazon Aurora pricing.

For information on choosing the storage configuration when creating a DB cluster, see Creating a DB cluster. For information on modifying the storage configuration for a DB cluster, see Settings for Amazon Aurora.

How Aurora storage automatically resizes

Aurora cluster volumes automatically grow as the amount of data in your database increases. The maximum size for an Aurora cluster volume is 128 tebibytes (TiB) or 64 TiB, depending on the DB engine version. For details about the maximum size for a specific version, see Amazon Aurora size limits. This automatic storage scaling is combined with a high-performance and highly distributed storage subsystem. These make Aurora a good choice for your important enterprise data when your main objectives are reliability and high availability.

To display the volume status, see Displaying volume status for an Aurora MySQL DB cluster or Displaying volume status for an Aurora PostgreSQL DB cluster. For ways to balance storage costs against other priorities, Storage scaling describes how to monitor the Amazon Aurora metrics `AuroraVolumeBytesLeftTotal` and `VolumeBytesUsed` in CloudWatch.

When Aurora data is removed, the space allocated for that data is freed. Examples of removing data include dropping or truncating a table. This automatic reduction in storage usage helps you to minimize storage charges.

> **Note**
>
> The storage limits and dynamic resizing behavior discussed here apply to persistent tables and other data stored in the cluster volume.
> For Aurora PostgreSQL, temporary table data is stored in the local DB instance.

For Aurora MySQL version 2, temporary table data is stored by default in the cluster volume for writer instances and in local storage for reader instances. For more information, see Storage engine for on-disk temporary tables.

For Aurora MySQL version 3, temporary table data is stored in the local DB instance or in the cluster volume. For more information, see New temporary table behavior in Aurora MySQL version 3.

The maximum size of temporary tables that reside in local storage is limited by the maximum local storage size of the DB instance. The local storage size depends on the instance class that you use. For more information, see Temporary storage limits for Aurora MySQL and Temporary storage limits for Aurora PostgreSQL.

Some storage features, such as the maximum size of a cluster volume and automatic resizing when data is removed, depend on the Aurora version of your cluster. For more information, see Storage scaling. You can also learn how to avoid storage issues and how to monitor the allocated storage and free space in your cluster.

How Aurora data storage is billed

Even though an Aurora cluster volume can grow up to 128 tebibytes (TiB), you are only charged for the space that you use in an Aurora cluster volume. In earlier Aurora versions, the cluster volume could reuse space that was freed up when you removed data, but the allocated storage space would never decrease. Now when Aurora data is removed, such as by dropping a table or database, the overall allocated space decreases by a comparable amount. Thus, you can reduce storage charges by dropping tables, indexes, databases, and so on that you no longer need.

Tip

For earlier versions without the dynamic resizing feature, resetting the storage usage for a cluster involved doing a logical dump and restoring to a new cluster. That operation can take a long time for a substantial volume of data. If you encounter this situation, consider upgrading your cluster to a version that supports dynamic volume resizing.

For information about which Aurora versions support dynamic resizing, and how to minimize storage charges by monitoring storage usage for your cluster, see Storage scaling. For information about Aurora backup storage billing, see Understanding Amazon Aurora backup storage usage. For pricing information about Aurora data storage, see Amazon RDS for Aurora pricing.

Amazon Aurora reliability

Aurora is designed to be reliable, durable, and fault tolerant. You can architect your Aurora DB cluster to improve availability by doing things such as adding Aurora Replicas and placing them in different Availability Zones, and also Aurora includes several automatic features that make it a reliable database solution.

Topics

Storage auto-repair

Because Aurora maintains multiple copies of your data in three Availability Zones, the chance of losing data as a result of a disk failure is greatly minimized. Aurora automatically detects failures in the disk volumes that make up the cluster volume. When a segment of a disk volume fails, Aurora immediately repairs the segment. When Aurora repairs the disk segment, it uses the data in the other volumes that make up the cluster volume to ensure that the data in the repaired segment is current. As a result, Aurora avoids data loss and reduces the need to perform a point-in-time restore to recover from a disk failure.

Survivable page cache

In Aurora, each DB instance's page cache is managed in a separate process from the database, which allows the page cache to survive independently of the database. (The page cache is also called the InnoDB *buffer pool* on Aurora MySQL and the *buffer cache* on Aurora PostgreSQL.)

In the unlikely event of a database failure, the page cache remains in memory, which keeps current data pages "warm" in the page cache when the database restarts. This provides a performance gain by bypassing the need for the initial queries to execute read I/O operations to "warm up" the page cache.

For Aurora MySQL, page cache behavior when rebooting and failing over is the following:

- Versions earlier than 2.10 – When the writer DB instance reboots, the page cache on the writer instance survives, but reader DB instances lose their page caches.

- Version 2.10 and higher – You can reboot the writer instance without rebooting the reader instances.
  - If the reader instances don't reboot when the writer instance reboots, they don't lose their page caches.
  - If the reader instances reboot when the writer instance reboots, they do lose their page caches.
- When a reader instance reboots, the page caches on the writer and reader instances both survive.
- When the DB cluster fails over, the effect is similar to when a writer instance reboots. On the new writer instance (previously the reader instance) the page cache survives, but on the reader instance (previously the writer instance), the page cache doesn't survive.

For Aurora PostgreSQL, you can use cluster cache management to preserve the page cache of a designated reader instance that becomes the writer instance after failover. For more information, see Fast recovery after failover with cluster cache management for Aurora PostgreSQL.

Recovery from unplanned restarts

Aurora is designed to recover from an unplanned restart almost instantaneously and continue to serve your application data without the binary log. Aurora recovers asynchronously on parallel threads, so that your database is open and available immediately after an unplanned restart.

For more information, see Fault tolerance for an Aurora DB cluster and Optimizations to reduce database restart time.

The following are considerations for binary logging and unplanned restart recovery on Aurora MySQL:

- Enabling binary logging on Aurora directly affects the recovery time after an unplanned restart, because it forces the DB instance to perform binary log recovery.
- The type of binary logging used affects the size and efficiency of logging. For the same amount of database activity, some formats log more information than others in the binary logs. The following settings for the `binlog_format` parameter result in different amounts of log data:
  - `ROW` – The most log data
  - `STATEMENT` – The least log data
  - `MIXED` – A moderate amount of log data that usually provides the best combination of data integrity and performance

The amount of binary log data affects recovery time. If there is more data logged in the binary logs, the DB instance must process more data during recovery, which increases recovery time.

- To reduce computational overhead and improve recovery times with binary logging, you can use enhanced binlog. Enhanced binlog improves the database recovery time by up to 99%. For more information, see Setting up enhanced binlog.
- Aurora does not need the binary logs to replicate data within a DB cluster or to perform point-in-time restore (PITR).
- If you don't need the binary log for external replication (or an external binary log stream), we recommend that you set the `binlog_format` parameter to `OFF` to disable binary logging. Doing so reduces recovery time.

For more information about Aurora binary logging and replication, see Replication with Amazon Aurora. For more information about the implications of different MySQL replication types, see Advantages and disadvantages of statement-based and row-based replication in the MySQL documentation.

Amazon Aurora security

Security for Amazon Aurora is managed at three levels:

- To control who can perform Amazon RDS management actions on Aurora DB clusters and DB instances, you use AWS Identity and Access Management (IAM). When you connect to AWS using IAM credentials, your AWS account must have IAM policies that grant the permissions required to perform Amazon RDS management operations. For more information, see Identity and access management for Amazon Aurora.

  If you are using IAM to access the Amazon RDS console, you must first log on to the AWS Management Console with your user credentials, and then go to the Amazon RDS console at https://console.aws.amazon.com/rds.
- Aurora DB clusters must be created in a virtual private cloud (VPC) based on the Amazon VPC service. To control which devices and Amazon EC2 instances can open connections to the endpoint and port of the DB instance for Aurora DB clusters in a VPC, you use a VPC security group. You can make these endpoint and port connections using Transport Layer Security (TLS)/Secure Sockets Layer (SSL). In addition, firewall rules at your company can control whether

devices running at your company can open connections to a DB instance. For more information on VPCs, see Amazon VPC VPCs and Amazon Aurora.

- To authenticate logins and permissions for an Amazon Aurora DB cluster, you can take either of the following approaches, or a combination of them.
  - You can take the same approach as with a stand-alone DB instance of MySQL or PostgreSQL.

    Techniques for authenticating logins and permissions for stand-alone DB instances of MySQL or PostgreSQL, such as using SQL commands or modifying database schema tables, also work with Aurora. For more information, see Security with Amazon Aurora MySQL or Security with Amazon Aurora PostgreSQL.
  - You can use IAM database authentication.

    With IAM database authentication, you authenticate to your Aurora DB cluster by using a user or IAM role and an authentication token. An *authentication token* is a unique value that is generated using the Signature Version 4 signing process. By using IAM database authentication, you can use the same credentials to control access to your AWS resources and your databases. For more information, see IAM database authentication.
  - You can use Kerberos authentication for Aurora PostgreSQL and Aurora MySQL.

    You can use Kerberos to authenticate users when they connect to your Aurora PostgreSQL and Aurora MySQLDB cluster. In this case, your DB cluster works with AWS Directory Service for Microsoft Active Directory to enable Kerberos authentication. AWS Directory Service for Microsoft Active Directory is also called AWS Managed Microsoft AD. Keeping all of your credentials in the same directory can save you time and effort. You have a centralized place for storing and managing credentials for multiple DB clusters. Using a directory can also improve your overall security profile. For more information, see Using Kerberos authentication with Aurora PostgreSQL and Using Kerberos authentication for Aurora MySQL.

For information about configuring security, see Security in Amazon Aurora.

Using SSL with Aurora DB clusters

Amazon Aurora DB clusters support Secure Sockets Layer (SSL) connections from applications using the same process and public key as Amazon RDS DB instances. For more information, see Security with Amazon Aurora MySQL, Security with Amazon Aurora PostgreSQL, or Using TLS/SSL with Aurora Serverless v1.

High availability for Amazon Aurora

The Amazon Aurora architecture involves separation of storage and compute. Aurora includes some high availability features that apply to the data in your DB cluster. The data remains safe even if some or all of the DB instances in the cluster become unavailable. Other high availability features apply to the DB instances. These features help to make sure that one or more DB instances are ready to handle database requests from your application.

Topics

High availability for Aurora data

Aurora stores copies of the data in a DB cluster across multiple Availability Zones in a single AWS Region. Aurora stores these copies regardless of whether the instances in the DB cluster span multiple Availability Zones. For more information on Aurora, see Managing an Amazon Aurora DB cluster.

When data is written to the primary DB instance, Aurora synchronously replicates the data across Availability Zones to six storage nodes associated with your cluster volume. Doing so provides data redundancy, eliminates I/O freezes, and minimizes latency spikes during system backups. Running a DB instance with high availability can enhance availability during planned system maintenance, and help protect your databases against failure and Availability Zone disruption. For more information on Availability Zones, see Regions and Availability Zones.

High availability for Aurora DB instances

After you create the primary (writer) instance, you can create up to 15 read-only Aurora Replicas. The Aurora Replicas are also known as reader instances.

During day-to-day operations, you can offload some of the work for read-intensive applications by using the reader instances to process `SELECT` queries. When a problem affects the primary instance, one of these reader instances takes over as the primary instance. This mechanism is known as *failover*. Many Aurora features apply to the failover mechanism. For example, Aurora detects database problems and activates the failover mechanism automatically when necessary. Aurora also has features that reduce the time for failover to complete. Doing so minimizes the time that the database is unavailable for writing during a failover.

Aurora is designed to recover as quickly as possible, and the fastest path to recovery is often to restart or to fail over to the same DB instance. Restarting is faster and involves less overhead than failover.

To use a connection string that stays the same even when a failover promotes a new primary instance, you connect to the cluster endpoint. The *cluster endpoint* always represents the current primary instance in the cluster. For more information about the cluster endpoint, see Amazon Aurora connection management.

Tip

Within each AWS Region, Availability Zones (AZs) represent locations that are distinct from each other to provide isolation in case of outages. We recommend that you distribute the primary instance and reader instances in your DB cluster over multiple Availability Zones to improve the availability of your DB cluster. That way, an issue that affects an entire Availability Zone doesn't cause an outage for your cluster.
You can set up a Multi-AZ DB cluster by making a simple choice when you create the cluster. You can use the AWS Management Console, the AWS CLI, or the Amazon RDS API. You can also convert an existing Aurora DB cluster into a Multi-AZ DB cluster by adding a new reader DB instance and specifying a different Availability Zone.

High availability across AWS Regions with Aurora global databases

For high availability across multiple AWS Regions, you can set up Aurora global databases. Each Aurora global database spans multiple AWS Regions, enabling low latency global reads and disaster recovery from outages across an AWS Region. Aurora automatically handles replicating all data and updates from the primary AWS Region to each of the secondary Regions. For more information, see Using Amazon Aurora global databases.

Fault tolerance for an Aurora DB cluster

An Aurora DB cluster is fault tolerant by design. The cluster volume spans multiple Availability Zones (AZs) in a single AWS Region, and each Availability Zone contains a copy of the cluster volume data. This functionality means that your DB cluster can tolerate a failure of an Availability Zone without any loss of data and only a brief interruption of service.

If the primary instance in a DB cluster fails, Aurora automatically fails over to a new primary instance in one of two ways:

- By promoting an existing Aurora Replica to the new primary instance
- By creating a new primary instance

If the DB cluster has one or more Aurora Replicas, then an Aurora Replica is promoted to the primary instance during a failure event. A failure event results in a brief interruption, during which read and write operations fail with an exception. However, service is typically restored in less than 60 seconds, and often less than 30 seconds. To increase the availability of your DB cluster, we recommend that you create at least one or more Aurora Replicas in two or more different Availability Zones.

> **Tip**
>
> In Aurora MySQL 2.10 and higher, you can improve availability during a failover by having more than one reader DB instance in a cluster. In Aurora MySQL 2.10 and higher, Aurora restarts only the writer DB instance and the reader instance to which it fails over. Other reader instances in the cluster remain available during a failover to continue processing queries through connections to the reader endpoint.
> You can also improve availability during a failover by using RDS Proxy with your Aurora DB cluster. For more information, see High availability with Amazon RDS Proxy.

You can customize the order in which your Aurora Replicas are promoted to the primary instance after a failure by assigning each replica a priority. Priorities range from 0 for the highest priority to 15 for the lowest priority. If the primary instance fails, Amazon RDS promotes the Aurora Replica with the highest priority to the new primary instance. You can modify the priority of an Aurora Replica at any time. Modifying the priority doesn't trigger a failover.

More than one Aurora Replica can share the same priority, resulting in promotion tiers. If two or more Aurora Replicas share the same priority, then Amazon RDS promotes the replica that is largest in size. If two or more Aurora Replicas share the same priority and size, then Amazon RDS promotes an arbitrary replica in the same promotion tier.

If the DB cluster doesn't contain any Aurora Replicas, then the primary instance is recreated in the same AZ during a failure event. A failure event results in an interruption during which read and write operations fail with an exception. Service is restored when the new primary instance is created, which typically takes less than 10 minutes. Promoting an Aurora Replica to the primary instance is much faster than creating a new primary instance.

Suppose that the primary instance in your cluster is unavailable because of an outage that affects an entire AZ. In this case, the way to bring a new primary instance online depends on whether your cluster uses a Multi-AZ configuration:

- If your provisioned or Aurora Serverless v2 cluster contains any reader instances in other AZs, Aurora uses the failover mechanism to promote one of those reader instances to be the new primary instance.
- If your provisioned or Aurora Serverless v2 cluster only contains a single DB instance, or if the primary instance and all reader instances are in the same AZ, make sure to manually create one or more new DB instances in another AZ.
- If your cluster uses Aurora Serverless v1, Aurora automatically creates a new DB instance in another AZ. However, this process involves a host replacement and thus takes longer than a failover.

> **Note**
>
> Amazon Aurora also supports replication with an external MySQL database, or an RDS MySQL DB instance. For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

High availability with Amazon RDS Proxy

With RDS Proxy, you can build applications that can transparently tolerate database failures without needing to write complex failure handling code. The proxy automatically routes traffic to a new database instance while preserving application connections. It also bypasses Domain Name System (DNS) caches to reduce failover times by up to 66% for Aurora Multi-AZ databases. For more information, see Using Amazon RDS Proxy for Aurora.

Replication with Amazon Aurora

There are several replication options with Aurora. Each Aurora DB cluster has built-in replication between multiple DB instances in the same cluster. You can also set up replication with your Aurora cluster as the source or the target. When you replicate data into or out of an Aurora cluster, you can choose between built-in features such as Aurora global databases or the traditional replication mechanisms for the MySQL or PostgreSQL DB engines. You can choose the appropriate options based on which one provides the right combination of high availability, convenience, and performance for your needs. The following sections explain how and when to choose each technique.

Topics

Aurora Replicas

When you create a second, third, and so on DB instance in an Aurora provisioned DB cluster, Aurora automatically sets up replication from the writer DB instance to all the other DB instances. These other DB instances are read-only and are known as Aurora Replicas. We also refer to them as reader instances when discussing the ways that you can combine writer and reader DB instances within a cluster.

Aurora Replicas have two main purposes. You can issue queries to them to scale the read operations for your application. You typically do so by connecting to the reader endpoint of the cluster. That way, Aurora can spread the load for read-only connections across as many Aurora Replicas as you have in the cluster. Aurora Replicas also help to increase availability. If the writer instance in a cluster becomes unavailable, Aurora automatically promotes one of the reader instances to take its place as the new writer.

An Aurora DB cluster can contain up to 15 Aurora Replicas. The Aurora Replicas can be distributed across the Availability Zones that a DB cluster spans within an AWS Region.

The data in your DB cluster has its own high availability and reliability features, independent of the DB instances in the cluster. If you aren't familiar with Aurora storage features, see Overview of Amazon Aurora storage. The DB cluster volume is physically made up of multiple copies of the data for the DB cluster. The primary instance and the Aurora Replicas in the DB cluster all see the data in the cluster volume as a single logical volume.

As a result, all Aurora Replicas return the same data for query results with minimal replica lag. This lag is usually much less than 100 milliseconds after the primary instance has written an update. Replica lag varies depending on the rate of database change. That is, during periods where a large amount of write operations occur for the database, you might see an increase in replica lag.

Note

Aurora Replica restarts, when it loses communication with the writer DB instance for more than 60 seconds in the following Aurora PostgreSQL versions:

- 14.6 and older versions
- 13.9 and older versions
- 12.13 and older versions
- All Aurora PostgreSQL 11 versions

Aurora Replicas work well for read scaling because they are fully dedicated to read operations on your cluster volume. Write operations are managed by the primary instance. Because the cluster volume is shared among all DB instances in your DB cluster, minimal additional work is required to replicate a copy of the data for each Aurora Replica.

To increase availability, you can use Aurora Replicas as failover targets. That is, if the primary instance fails, an Aurora Replica is promoted to the primary instance. There is a brief interruption during which read and write requests made to the primary instance fail with an exception.

Promoting an Aurora Replica by failover is much faster than recreating the primary instance. If your Aurora DB cluster doesn't include any Aurora Replicas, then your DB cluster isn't available while your DB instance is recovering from the failure.

When failover happens, some of the Aurora Replicas might be rebooted, depending on the DB engine version. For example, in Aurora MySQL 2.10 and higher, Aurora restarts only the writer DB instance and the failover target during a failover. For more information about the rebooting behavior of different Aurora DB engine versions, see Rebooting an Amazon Aurora DB cluster or Amazon Aurora DB instance. For information about what happens to page caches when rebooting or failover, see Survivable page cache.

For high-availability scenarios, we recommend that you create one or more Aurora Replicas. These should be of the same DB instance class as the primary instance and in different Availability Zones for your Aurora DB cluster. For more information about Aurora Replicas as failover targets, see Fault tolerance for an Aurora DB cluster.

You can't create an encrypted Aurora Replica for an unencrypted Aurora DB cluster. You can't create an unencrypted Aurora Replica for an encrypted Aurora DB cluster.

Tip

You can use Aurora Replicas within an Aurora cluster as your only form of replication to keep your data highly available. You can also combine the built-in Aurora replication with the other types of replication. Doing so can help to provide an extra level of high availability and geographic distribution of your data.

For details on how to create an Aurora Replica, see Adding Aurora Replicas to a DB cluster.

Replication with Aurora MySQL

In addition to Aurora Replicas, you have the following options for replication with Aurora MySQL:

- Aurora MySQL DB clusters in different AWS Regions.
  - You can replicate data across multiple Regions by using an Aurora global database. For details, see High availability across AWS Regions with Aurora global databases
  - You can create an Aurora read replica of an Aurora MySQL DB cluster in a different AWS Region, by using MySQL binary log (binlog) replication. Each cluster can have up to five read replicas created this way, each in a different Region.
- Two Aurora MySQL DB clusters in the same Region, by using MySQL binary log (binlog) replication.
- An RDS for MySQL DB instance as the source of data and an Aurora MySQL DB cluster, by creating an Aurora read replica of an RDS for MySQL DB instance. Typically, you use this approach for migration to Aurora MySQL, rather than for ongoing replication.

For more information about replication with Aurora MySQL, see Replication with Amazon Aurora MySQL.

Replication with Aurora PostgreSQL

In addition to Aurora Replicas, you have the following options for replication with Aurora PostgreSQL:

- An Aurora primary DB cluster in one Region and up to five read-only secondary DB clusters in different Regions by using an Aurora global database. Aurora PostgreSQL doesn't support cross-Region Aurora Replicas. However, you can use Aurora global database to scale your Aurora PostgreSQL DB cluster's read capabilities to more than one AWS Region and to meet availability goals. For more information, see Using Amazon Aurora global databases.
- Two Aurora PostgreSQL DB clusters in the same Region, by using PostgreSQL's logical replication feature.
- An RDS for PostgreSQL DB instance as the source of data and an Aurora PostgreSQL DB cluster, by creating an Aurora read replica of an RDS for PostgreSQL DB instance. Typically, you use this approach for migration to Aurora PostgreSQL, rather than for ongoing replication.

For more information about replication with Aurora PostgreSQL, see Replication with Amazon Aurora PostgreSQL.

DB instance billing for Aurora

Amazon RDS provisioned instances in an Amazon Aurora cluster are billed based on the following components:

- DB instance hours (per hour) – Based on the DB instance class of the DB instance (for example, db.t2.small or db.m4.large). Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. RDS usage is billed in 1-second increments, with a minimum of 10 minutes. For more information, see Aurora DB instance classes.
- Storage (per GiB per month) – Storage capacity that you have provisioned to your DB instance. If you scale your provisioned storage capacity within the month, your bill is prorated. For more information, see Amazon Aurora storage and reliability.
- Input/output (I/O) requests (per 1 million requests) – Total number of storage I/O requests that you have made in a billing cycle, for the Aurora Standard DB cluster configuration only.

  For more information on Amazon Aurora I/O billing, see Storage configurations for Amazon Aurora DB clusters.

- Backup storage (per GiB per month) – *Backup storage* is the storage that is associated with automated database backups and any active database snapshots that you have taken. Increasing your backup retention period or taking additional database snapshots increases the backup storage consumed by your database. Per second billing doesn't apply to backup storage (metered in GB-month).

  For more information, see Backing up and restoring an Amazon Aurora DB cluster.
- Data transfer (per GB) – Data transfer in and out of your DB instance from or to the internet and other AWS Regions.

Amazon RDS provides the following purchasing options to enable you to optimize your costs based on your needs:

- **On-Demand instances** – Pay by the hour for the DB instance hours that you use. Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. RDS usage is now billed in 1-second increments, with a minimum of 10 minutes.
- **Reserved instances** – Reserve a DB instance for a one-year or three-year term and get a significant discount compared to the on-demand DB instance pricing. With Reserved Instance usage, you can launch, delete, start, or stop multiple instances within an hour and get the Reserved Instance benefit for all of the instances.
- **Aurora Serverless v2** – Aurora Serverless v2 provides on-demand capacity where the billing unit is Aurora capacity unit (ACU) hours instead of DB instance hours. Aurora Serverless v2 capacity increases and decreases, within a range that you specify, depending on the load on your database. You can configure a cluster where all the capacity is Aurora Serverless v2. Or you can configure a combination of Aurora Serverless v2 and on-demand or reserved provisioned instances. For information about how Aurora Serverless v2 ACUs work, see How Aurora Serverless v2 works.

For Aurora pricing information, see the Aurora pricing page.

Topics

On-Demand DB instances for Aurora

Amazon RDS on-demand DB instances are billed based on the class of the DB instance (for example, db.t3.small or db.m5.large). For Amazon RDS pricing information, see the Amazon RDS product page.

Billing starts for a DB instance as soon as the DB instance is available. Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. Amazon RDS usage is billed in one-second increments, with a minimum of 10 minutes. In the case of billable configuration change, such as scaling compute or storage capacity, you're charged a 10-minute minimum. Billing continues until the DB instance terminates, which occurs when you delete the DB instance or if the DB instance fails.

If you no longer want to be charged for your DB instance, you must stop or delete it to avoid being billed for additional DB instance hours. For more information about the DB instance states for which you are billed, see Viewing DB instance status in an Aurora cluster.

Stopped DB instances

While your DB instance is stopped, you're charged for provisioned storage, including Provisioned IOPS. You are also charged for backup storage, including storage for manual snapshots and automated backups within your specified retention window. You aren't charged for DB instance hours.

Multi-AZ DB instances

If you specify that your DB instance should be a Multi-AZ deployment, you're billed according to the Multi-AZ pricing posted on the Amazon RDS pricing page.

Reserved DB instances for Aurora

Using reserved DB instances, you can reserve a DB instance for a one- or three-year term. Reserved DB instances provide you with a significant discount compared to on-demand DB instance pricing. Reserved DB instances are not physical instances, but rather a billing discount applied to the use of certain on-demand DB instances in your account. Discounts for reserved DB instances are tied to instance type and AWS Region.

The general process for working with reserved DB instances is: First get information about available reserved DB instance offerings, then purchase a reserved DB instance offering, and finally get information about your existing reserved DB instances.

Overview of reserved DB instances

When you purchase a reserved DB instance in Amazon RDS, you purchase a commitment to getting a discounted rate, on a specific DB instance type, for the duration of the reserved DB instance. To use an Amazon RDS reserved DB instance, you create a new DB instance just like you do for an on-demand instance.

The new DB instance that you create must have the same specifications as the reserved DB instance for the following:

- AWS Region
- DB engine (The DB engine's version number doesn't need to match.)
- DB instance type

If the specifications of the new DB instance match an existing reserved DB instance for your account, you are billed at the discounted rate offered for the reserved DB instance. Otherwise, the DB instance is billed at an on-demand rate.

You can modify a DB instance that you're using as a reserved DB instance. If the modification is within the specifications of the reserved DB instance, part or all of the discount still applies to the modified DB instance. If the modification is outside the specifications, such as changing the instance class, the discount no longer applies. For more information, see Size-flexible reserved DB instances.

Topics

- Aurora DB cluster configuration flexibility
- Size-flexible reserved DB instances
- Aurora reserved DB instance billing examples
- Deleting a reserved DB instance

For more information about reserved DB instances, including pricing, see Amazon RDS reserved instances.

Offering types

Reserved DB instances are available in three varieties—No Upfront, Partial Upfront, and All Upfront—that let you optimize your Amazon RDS costs based on your expected usage.

No Upfront

> This option provides access to a reserved DB instance without requiring an upfront payment. Your No Upfront reserved DB instance bills a discounted hourly rate for every hour within the term, regardless of usage, and no upfront payment is required. This option is only available as a one-year reservation.

Partial Upfront

> This option requires a part of the reserved DB instance to be paid upfront. The remaining hours in the term are billed at a discounted hourly rate, regardless of usage. This option is the replacement for the previous Heavy Utilization option.

All Upfront

> Full payment is made at the start of the term, with no other costs incurred for the remainder of the term regardless of the number of hours used.

If you are using consolidated billing, all the accounts in the organization are treated as one account. This means that all accounts in the organization can receive the hourly cost benefit of reserved DB instances that are purchased by any other account. For more information about consolidated billing, see Amazon RDS reserved DB instances in the *AWS Billing and Cost Management User Guide*.

Aurora DB cluster configuration flexibility

You can use Aurora reserved DB instances with both DB cluster configurations:

- Aurora I/O-Optimized – You pay only for the usage and storage of your DB clusters, with no additional charges for read and write I/O operations.
- Aurora Standard – In addition to the usage and storage of your DB clusters, you also pay a standard rate per 1 million requests for I/O operations.

Aurora automatically accounts for the price difference between these configurations. Aurora I/O-Optimized consumes 30% more normalized units per hour than Aurora Standard.

For more information about Aurora cluster storage configurations, see Storage configurations for Amazon Aurora DB clusters. For more information about pricing for Aurora cluster storage configurations, see Amazon Aurora pricing.

Size-flexible reserved DB instances

When you purchase a reserved DB instance, one thing that you specify is the instance class, for example db.r5.large. For more information about DB instance classes, see Aurora DB instance classes.

If you have a DB instance, and you need to scale it to larger capacity, your reserved DB instance is automatically applied to your scaled DB instance. That is, your reserved DB instances are automatically applied across all DB instance class sizes. Size-flexible reserved DB instances are available for DB instances with the same AWS Region and database engine. Size-flexible reserved DB instances can only scale in their instance class type. For example, a reserved DB instance for a db.r5.large can apply to a db.r5.xlarge, but not to a db.r6g.large, because db.r5 and db.r6g are different instance class types.

Reserved DB instance benefits also apply for both Multi-AZ and Single-AZ configurations. Flexibility means that you can move freely between configurations within the same DB instance class type. For example, you can move from a Single-AZ deployment running on one large DB instance (four normalized units per hour) to a Multi-AZ deployment running on two medium DB instances (2+2 = 4 normalized units per hour).

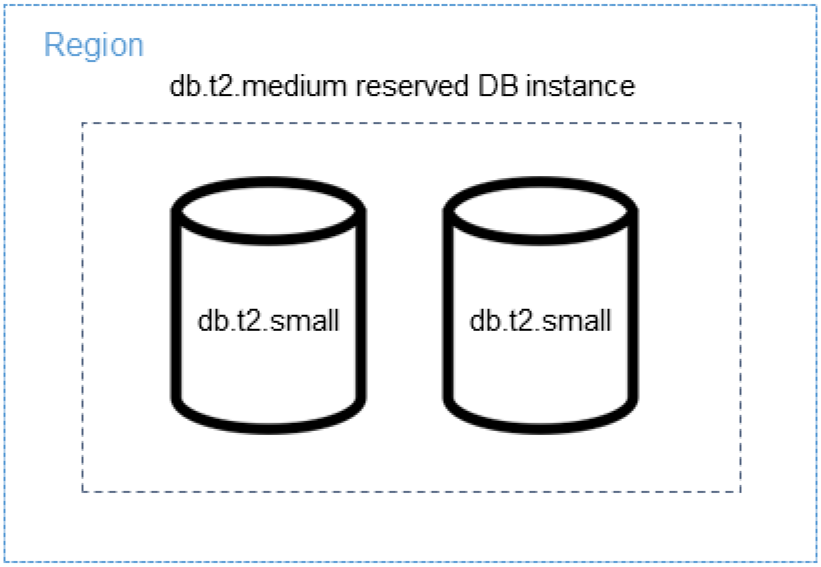
Size-flexible reserved DB instances are available for the following Aurora database engines:

- Aurora MySQL
- Aurora PostgreSQL

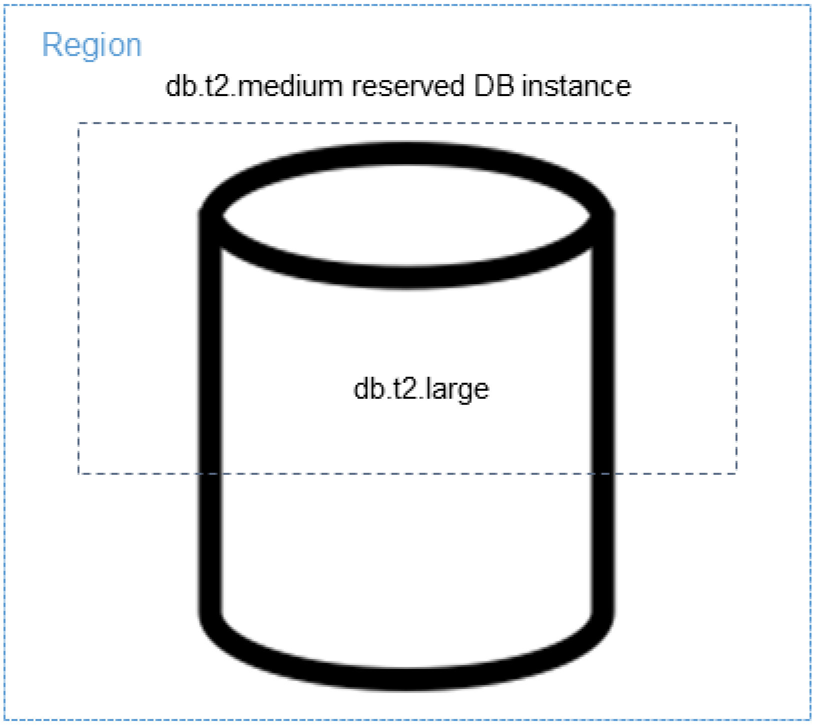
You can compare usage for different reserved DB instance sizes by using normalized units per hour. For example, one unit of usage on two db.r3.large DB instances is equivalent to eight normalized units per hour of usage on one db.r3.small. The following table shows the number of normalized units per hour for each DB instance size.

Instance size	Normalized units per hour for one DB instance, Aurora Standard	Normalized units per hour for one DB instance, Aurora I/O-Optimized	Normalized units per hour for three DB instances (writer and two readers), Aurora Standard	Normalized units per hour for three DB instances (writer and two readers), Aurora I/O-Optimized
small	1	1.3	3	3.9
medium	2	2.6	6	7.8
large	4	5.2	12	15.6
xlarge	8	10.4	24	31.2
2xlarge	16	20.8	48	62.4
4xlarge	32	41.6	96	124.8
8xlarge	64	83.2	192	249.6
12xlarge	96	124.8	288	374.4
16xlarge	128	166.4	384	499.2
24xlarge	192	249.6	576	748.8
32xlarge	256	332.8	768	998.4

For example, suppose that you purchase a `db.t2.medium` reserved DB instance, and you have two running `db.t2.small` DB instances in your account in the same AWS Region. In this case, the billing benefit is applied in full to both instances.



Alternatively, if you have one `db.t2.large` instance running in your account in the same AWS Region, the billing benefit is applied to 50 percent of the usage of the DB instance.



Note

We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.

Aurora reserved DB instance billing examples

The following examples illustrate the pricing for reserved DB instances for Aurora DB clusters using both the Aurora Standard and Aurora I/O-Optimized DB cluster configurations.

Example using Aurora Standard

The price for a reserved DB instance doesn't provide a discount for the costs associated with storage, backups, and I/O. The following example illustrates the total cost per month for a reserved DB instance:

- An Aurora MySQL reserved Single-AZ db.r5.large DB instance class in US East (N. Virginia) at a cost of $0.19 per hour, or $138.70 per month
- Aurora storage at a cost of $0.10 per GiB per month (assume $45.60 per month for this example)
- Aurora I/O at a cost of $0.20 per 1 million requests (assume $20 per month for this example)
- Aurora backup storage at $0.021 per GiB per month (assume $30 per month for this example)

Add all of these options ($138.70 + $45.60 + $20 + $30) with the reserved DB instance, and the total cost per month is $234.30.

If you choose to use an on-demand DB instance instead of a reserved DB instance, an Aurora MySQL Single-AZ db.r5.large DB instance class in US East (N. Virginia) costs $0.29 per hour, or $217.50 per month. So, for an on-demand DB instance, add all of these options ($217.50 + $45.60 + $20 + $30), and the total cost per month is $313.10. You save nearly $79 per month by using the reserved DB instance.

Example using an Aurora Standard DB cluster with two reader instances

To use reserved instances for Aurora DB clusters, simply purchase one reserved instance for each DB instance in the cluster.

Extending the first example, you have an Aurora MySQL DB cluster with one writer DB instance and two Aurora Replicas, for a total of three DB instances in the cluster. The two Aurora Replicas incur no extra storage or backups charges. If you purchase three db.r5.large Aurora MySQL reserved DB instances, your cost will be $234.30 (for the writer DB instance) + 2 * ($138.70 + $20 I/O per Aurora Replica), for a total of $551.70 per month.

The corresponding on-demand cost for an Aurora MySQL DB cluster with one writer DB instance and two Aurora Replicas is $313.10 + 2 * ($217.50 + $20 I/O per instance) for a total of $788.10 per month. You save $236.40 per month by using the reserved DB instances.

Example using Aurora I/O-Optimized

You can reuse your existing Aurora Standard reserved DB instances with Aurora I/O-Optimized. To fully use the benefits of your reserved instance discounts with Aurora I/O-Optimized, you can buy 30% additional reserved instances similar to your current reserved instances.

The following table shows examples of how to estimate the additional reserved instances when using Aurora I/O-Optimized. If the required reserved instances are a fraction, you can take advantage of the size flexibility available with reserved instances to get to a whole number. In these examples, "current" refers to the Aurora Standard reserved instances that you have now. Additional reserved instances are the number of Aurora Standard reserved instances that you must buy to maintain your current reserved instance discounts when using Aurora I/O-Optimized.

DB instance class	Current Aurora Standard reserved instances	Reserved instances required for Aurora I/O-Optimized	Additional reserved instances needed	Additional reserved instances needed, using size flexibility
db.r6g.large	10	10 * 1.3 = 13	3 * db.r6g.large	3 * db.r6g.large
db.r6g.4xlarge	20	20 * 1.3 = 26	6 * db.r6g.4xlarge	6 * db.r6g.4xlarge
db.r6g.12xlarge	5	5 * 1.3 = 6.5	1.5 * db.r6g.12xlarge	One each of db.r6g.12xlarge, r6g.4xlarge, and r6g.2xlarge

DB instance class	Current Aurora Standard reserved instances	Reserved instances required for Aurora I/O-Optimized	Additional reserved instances needed	Additional reserved instances needed, using size flexibility
				(0.5 * db.r6g.12xlarge = 1 * db.r6g.4xlarge + 1 * db.r6g.2xlarge)
db.r6i.24xlarge	15	15 * 1.3 = 19.5	4.5 * db.r6i.24xlarge	4 * db.r6i.24xlarge + 1 * db.r6i.12xlarge (0.5 * db.r6i.24xlarge = 1 * db.r6i.12xlarge)

Example using an Aurora I/O-Optimized DB cluster with two reader instances

You have an Aurora MySQL DB cluster with one writer DB instance and two Aurora Replicas, for a total of three DB instances in the cluster. They use the Aurora I/O-Optimized DB cluster configuration. To use reserved DB instances for this cluster, you would need to buy four reserved DB instances of the same DB instance class. Three DB instances using Aurora I/O-Optimized consume 3.9 normalized units per hour, compared to 3 normalized units per hour for three DB instances using Aurora Standard. However, you save the monthly I/O costs for each DB instance.

> **Note**
>
> The prices in these examples are sample prices and might not match actual prices. For Aurora pricing information, see Amazon Aurora pricing.

Deleting a reserved DB instance

The terms for a reserved DB instance involve a one-year or three-year commitment. You can't cancel a reserved DB instance. However, you can delete a DB instance that is covered by a reserved DB instance discount. The process for deleting a DB instance that is covered by a reserved DB instance discount is the same as for any other DB instance.

You're billed for the upfront costs regardless of whether you use the resources.

If you delete a DB instance that is covered by a reserved DB instance discount, you can launch another DB instance with compatible specifications. In this case, you continue to get the discounted rate during the reservation term (one or three years).

Working with reserved DB instances

You can use the AWS Management Console, the AWS CLI, and the RDS API to work with reserved DB instances.

Console

You can use the AWS Management Console to work with reserved DB instances as shown in the following procedures.

To get pricing and information about available reserved DB instance offerings

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Reserved instances**.
3. Choose **Purchase Reserved DB Instance**.
4. For **Product description**, choose the DB engine and licensing type.
5. For **DB instance class**, choose the DB instance class.
6. For **Deployment Option**, choose whether you want a Single-AZ or Multi-AZ DB instance deployment.

   > **Note**
   >
   > Reserved Amazon Aurora *instances* always have the deployment option set to **Single-AZ DB instance**. However, when you create an Aurora DB *cluster*, the default deployment option is **Create an Aurora Replica or Reader in a different AZ** (Multi-AZ).

> You must purchase a reserved DB instance for each instance that you plan to use, including Aurora Replicas. Therefore, for Multi-AZ deployments on Aurora, you must purchase extra reserved DB instances.

7. For **Term**, choose the length of time to reserve the the DB instance.
8. For **Offering type**, choose the offering type.

   After you select the offering type, you can see the pricing information.

   > ⚠ **Important**
   >
   > Choose **Cancel** to avoid purchasing the reserved DB instance and incurring any charges.

After you have information about the available reserved DB instance offerings, you can use the information to purchase an offering as shown in the following procedure.

To purchase a reserved DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Reserved instances**.
3. Choose **Purchase reserved DB instance**.
4. For **Product description**, choose the DB engine and licensing type.
5. For **DB instance class**, choose the DB instance class.
6. For **Multi-AZ deployment**, choose whether you want a Single-AZ or Multi-AZ DB instance deployment.

   > ⓘ **Note**
   >
   > Reserved Amazon Aurora *instances* always have the deployment option set to **Single-AZ DB instance**. When you create an Amazon Aurora DB *cluster* from your reserved DB instance, the DB cluster is automatically created as Multi-AZ. Make sure to purchase a reserved DB instance for each DB instance that you plan to use, including Aurora Replicas.

7. For **Term**, choose the length of time you want the DB instance reserved.

8. For **Offering type**, choose the offering type.

   After you choose the offering type, you can see the pricing information.

9. (Optional) You can assign your own identifier to the reserved DB instances that you purchase to help you track them. For **Reserved Id**, type an identifier for your reserved DB instance.

10. Choose **Submit**.

    Your reserved DB instance is purchased, then displayed in the **Reserved instances** list.

After you have purchased reserved DB instances, you can get information about your reserved DB instances as shown in the following procedure.

To get information about reserved DB instances for your AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the **Navigation** pane, choose **Reserved instances**.

   The reserved DB instances for your account appear. To see detailed information about a particular reserved DB instance, choose that instance in the list. You can then see detailed information about that instance in the detail pane at the bottom of the console.

AWS CLI

You can use the AWS CLI to work with reserved DB instances as shown in the following examples.

Example of getting available reserved DB instance offerings

To get information about available reserved DB instance offerings, call the AWS CLI command `describe-reserved-db-instances-offerings`.

```
aws rds describe-reserved-db-instances-offerings
```

This call returns output similar to the following:

```

OFFERING  OfferingId                            Class         Multi-AZ  Duration  Fixed
 Price  Usage Price  Description  Offering Type
```

```
OFFERING  438012d3-4052-4cc7-b2e3-8d3372e0e706  db.r3.large   y         1y
 1820.00 USD  0.368 USD    mysql        Partial  Upfront
OFFERING  649fd0c8-cf6d-47a0-bfa6-060f8e75e95f  db.r3.small   n         1y
 227.50 USD  0.046 USD    mysql        Partial  Upfront
OFFERING  123456cd-ab1c-47a0-bfa6-12345667232f  db.r3.small   n         1y
 162.00 USD   0.00 USD    mysql        All      Upfront
    Recurring Charges:   Amount  Currency  Frequency
    Recurring Charges:   0.123   USD       Hourly
OFFERING  123456cd-ab1c-37a0-bfa6-12345667232d  db.r3.large   y         1y
 700.00 USD   0.00 USD    mysql        All      Upfront
    Recurring Charges:   Amount  Currency  Frequency
    Recurring Charges:   1.25    USD       Hourly
OFFERING  123456cd-ab1c-17d0-bfa6-12345667234e  db.r3.xlarge  n         1y
 4242.00 USD   2.42 USD    mysql        No       Upfront
```

After you have information about the available reserved DB instance offerings, you can use the information to purchase an offering.

To purchase a reserved DB instance, use the AWS CLI command `purchase-reserved-db-instances-offering` with the following parameters:

- `--reserved-db-instances-offering-id` – The ID of the offering that you want to purchase. See the preceding example to get the offering ID.
- `--reserved-db-instance-id` – You can assign your own identifier to the reserved DB instances that you purchase to help track them.

Example of purchasing a reserved DB instance

The following example purchases the reserved DB instance offering with ID *649fd0c8-cf6d-47a0-bfa6-060f8e75e95f*, and assigns the identifier of *MyReservation*.

For Linux, macOS, or Unix:

```
aws rds purchase-reserved-db-instances-offering \
    --reserved-db-instances-offering-id 649fd0c8-cf6d-47a0-bfa6-060f8e75e95f \
    --reserved-db-instance-id MyReservation
```

For Windows:

```
aws rds purchase-reserved-db-instances-offering ^
```

```
    --reserved-db-instances-offering-id 649fd0c8-cf6d-47a0-bfa6-060f8e75e95f ^
    --reserved-db-instance-id MyReservation
```

The command returns output similar to the following:

```
RESERVATION  ReservationId      Class        Multi-AZ  Start Time
 Duration  Fixed Price  Usage Price  Count  State            Description  Offering Type
RESERVATION  MyReservation      db.r3.small  y         2011-12-19T00:30:23.247Z  1y
    455.00 USD   0.092 USD    1      payment-pending  mysql        Partial  Upfront
```

After you have purchased reserved DB instances, you can get information about your reserved DB instances.

To get information about reserved DB instances for your AWS account, call the AWS CLI command `describe-reserved-db-instances`, as shown in the following example.

Example of getting your reserved DB instances

```
aws rds describe-reserved-db-instances
```

The command returns output similar to the following:

```
RESERVATION  ReservationId     Class        Multi-AZ  Start Time
 Duration  Fixed Price  Usage Price  Count  State    Description  Offering Type
RESERVATION  MyReservation     db.r3.small  y         2011-12-09T23:37:44.720Z  1y
   455.00 USD   0.092 USD    1      retired  mysql        Partial  Upfront
```

RDS API

You can use the RDS API to work with reserved DB instances:

- To get information about available reserved DB instance offerings, call the Amazon RDS API operation `DescribeReservedDBInstancesOfferings`.
- After you have information about the available reserved DB instance offerings, you can use the information to purchase an offering. Call the `PurchaseReservedDBInstancesOffering` RDS API operation with the following parameters:
  - `--reserved-db-instances-offering-id` – The ID of the offering that you want to purchase.

  - `--reserved-db-instance-id` – You can assign your own identifier to the reserved DB instances that you purchase to help track them.
- After you have purchased reserved DB instances, you can get information about your reserved DB instances. Call the [`DescribeReservedDBInstances`](#) RDS API operation.

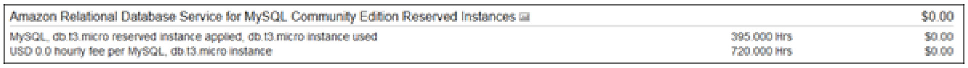
Viewing the billing for your reserved DB instances

You can view the billing for your reserved DB instances in the Billing Dashboard in the AWS Management Console.

To view reserved DB instance billing

1. Sign in to the AWS Management Console.
2. From the **account menu** at the upper right, choose **Billing Dashboard**.
3. Choose **Bill Details** at the upper right of the dashboard.
4. Under **AWS Service Charges**, expand **Relational Database Service**.
5. Expand the AWS Region where your reserved DB instances are, for example **US West (Oregon)**.

   Your reserved DB instances and their hourly charges for the current month are shown under **Amazon Relational Database Service for *`Database Engine`* Reserved Instances**.

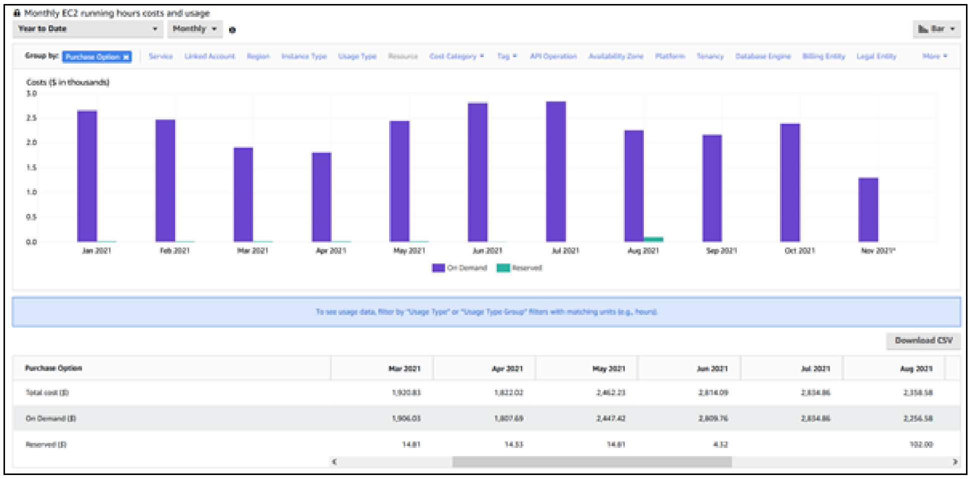
   

Amazon Relational Database Service for MySQL Community Edition Reserved Instances		$0.00
MySQL, db.t3.micro reserved instance applied, db.t3.micro instance used	395.000 Hrs	$0.00
USD 0.0 hourly fee per MySQL, db.t3.micro instance	720.000 Hrs	$0.00

   The reserved DB instance in this example was purchased All Upfront, so there are no hourly charges.
6. Choose the **Cost Explorer** (bar graph) icon next to the **Reserved Instances** heading.

   The Cost Explorer displays the **Monthly EC2 running hours costs and usage** graph.
7. Clear the **Usage Type Group** filter to the right of the graph.
8. Choose the time period and time unit for which you want to examine usage costs.

   The following example shows usage costs for on-demand and reserved DB instances for the year to date by month.



To see usage data, filter by "Usage Type" or "Usage Type Group" filters with matching units (e.g., hours).

Purchase Option	Mar 2021	Apr 2021	May 2021	Jun 2021	Jul 2021	Aug 2021
Total cost ($)	1,920.83	1,822.02	2,462.23	2,814.09	2,834.86	2,358.58
On Demand ($)	1,906.03	1,807.69	2,447.42	2,809.76	2,834.86	2,256.58
Reserved ($)	14.81	14.33	14.81	4.32		102.00

The reserved DB instance costs from January through June 2021 are monthly charges for a Partial Upfront instance, while the cost in August 2021 is a one-time charge for an All Upfront instance.

The reserved instance discount for the Partial Upfront instance expired in June 2021, but the DB instance wasn't deleted. After the expiration date, it was simply charged at the on-demand rate.

Setting up your environment for Amazon Aurora

Before you use Amazon Aurora for the first time, complete the following tasks.

Topics

If you already have an AWS account, know your Aurora requirements, and prefer to use the defaults for IAM and VPC security groups, skip ahead to Getting started with Amazon Aurora.

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Grant programmatic access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.

Which user needs programmatic access?	To	By
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

Determine requirements

The basic building block of Aurora is the DB cluster. One or more DB instances can belong to a DB cluster. A DB cluster provides a network address called the *cluster endpoint*. Your applications connect to the cluster endpoint exposed by the DB cluster whenever they need to access the databases created in that DB cluster. The information you specify when you create the DB cluster controls configuration elements such as memory, database engine and version, network configuration, security, and maintenance periods.

Before you create a DB cluster and a security group, you must know your DB cluster and network needs. Here are some important things to consider:

- **Resource requirements –** What are the memory and processor requirements for your application or service? You will use these settings when you determine what DB instance class you will use when you create your DB cluster. For specifications about DB instance classes, see Aurora DB instance classes.
- **VPC, subnet, and security group –** Your DB cluster will be in a virtual private cloud (VPC). Security group rules must be configured to connect to a DB cluster. The following list describes the rules for each VPC option:
  - **Default VPC** — If your AWS account has a default VPC in the AWS Region, that VPC is configured to support DB clusters. If you specify the default VPC when you create the DB cluster:
    - Make sure to create a *VPC security group* that authorizes connections from the application or service to the Aurora DB cluster. Use the **Security Group** option on the VPC console or the AWS CLI to create VPC security groups. For information, see Step 3: Create a VPC security group.
    - You must specify the default DB subnet group. If this is the first DB cluster you have created in the AWS Region, Amazon RDS will create the default DB subnet group when it creates the DB cluster.
  - **User-defined VPC** — If you want to specify a user-defined VPC when you create a DB cluster:
    - Make sure to create a *VPC security group* that authorizes connections from the application or service to the Aurora DB cluster. Use the **Security Group** option on the VPC console or the AWS CLI to create VPC security groups. For information, see Step 3: Create a VPC security group.
    - The VPC must meet certain requirements in order to host DB clusters, such as having at least two subnets, each in a separate availability zone. For information, see Amazon VPC VPCs and Amazon Aurora.
    - You must specify a DB subnet group that defines which subnets in that VPC can be used by the DB cluster. For information, see the DB Subnet Group section in Working with a DB cluster in a VPC.
- **High availability:** Do you need failover support? On Aurora, a Multi-AZ deployment creates a primary instance and Aurora Replicas. You can configure the primary instance and Aurora Replicas to be in different Availability Zones for failover support. We recommend Multi-AZ deployments for production workloads to maintain high availability. For development and test

purposes, you can use a non-Multi-AZ deployment. For more information, see High availability for Amazon Aurora.

- **IAM policies:** Does your AWS account have policies that grant the permissions needed to perform Amazon RDS operations? If you are connecting to AWS using IAM credentials, your IAM account must have IAM policies that grant the permissions required to perform Amazon RDS operations. For more information, see Identity and access management for Amazon Aurora.
- **Open ports:** What TCP/IP port will your database be listening on? The firewall at some companies might block connections to the default port for your database engine. If your company firewall blocks the default port, choose another port for the new DB cluster. Note that once you create a DB cluster that listens on a port you specify, you can change the port by modifying the DB cluster.
- **AWS Region:** What AWS Region do you want your database in? Having the database close in proximity to the application or web service could reduce network latency. For more information, see Regions and Availability Zones.

Once you have the information you need to create the security group and the DB cluster, continue to the next step.

Provide access to the DB cluster in the VPC by creating a security group

Your DB cluster will be created in a VPC. Security groups provide access to the DB cluster in the VPC. They act as a firewall for the associated DB cluster, controlling both inbound and outbound traffic at the cluster level. DB clusters are created by default with a firewall and a default security group that prevents access to the DB cluster. You must therefore add rules to a security group that enable you to connect to your DB cluster. Use the network and configuration information you determined in the previous step to create rules to allow access to your DB cluster.

For example, if you have an application that will access a database on your DB cluster in a VPC, you must add a custom TCP rule that specifies the port range and IP addresses that application will use to access the database. If you have an application on an Amazon EC2 instance, you can use the VPC security group you set up for the Amazon EC2 instance.

You can configure connectivity between an Amazon EC2 instance a DB cluster when you create the DB cluster. For more information, see Configure automatic network connectivity with an EC2 instance.

> **Tip**
>
> You can set up network connectivity between an Amazon EC2 instance and a DB cluster automatically when you create the DB cluster. For more information, see Configure automatic network connectivity with an EC2 instance.

For more information about creating a VPC for use with Aurora, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only). For information about common scenarios for accessing a DB instance, see Scenarios for accessing a DB cluster in a VPC.

To create a VPC security group

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   > **Note**
   >
   > Make sure you are in the VPC console, not the RDS console.

2. In the top right corner of the AWS Management Console, choose the AWS Region where you want to create your VPC security group and DB cluster. In the list of Amazon VPC resources for that AWS Region, you should see at least one VPC and several subnets. If you don't, you don't have a default VPC in that AWS Region.
3. In the navigation pane, choose **Security Groups**.
4. Choose **Create security group**.

   The **Create security group** page appears.
5. In **Basic details**, enter the **Security group name** and **Description**. For **VPC**, choose the VPC that you want to create your DB cluster in.
6. In **Inbound rules**, choose **Add rule**.

   a. For **Type**, choose **Custom TCP**.

   b. For **Port range**, enter the port value to use for your DB cluster.

   c. For **Source**, choose a security group name or type the IP address range (CIDR value) from where you access the DB cluster. If you choose **My IP**, this allows access to the DB cluster from the IP address detected in your browser.

7. If you need to add more IP addresses or different port ranges, choose **Add rule** and enter the information for the rule.
8. (Optional) In **Outbound rules**, add rules for outbound traffic. By default, all outbound traffic is allowed.
9. Choose **Create security group**.

You can use the VPC security group you just created as the security group for your DB cluster when you create it.

> **Note**
>
> If you use a default VPC, a default subnet group spanning all of the VPC's subnets is created for you. When you create a DB cluster, you can select the default VPC and use **default** for **DB Subnet Group**.

Once you have completed the setup requirements, you can create a DB cluster using your requirements and security group by following the instructions in Creating an Amazon Aurora DB cluster. For information about getting started by creating a DB cluster that uses a specific DB engine, see Getting started with Amazon Aurora.

Getting started with Amazon Aurora

In this section, you can find out how to create and connect to an Aurora DB cluster using Amazon RDS.

The following procedures are tutorials that demonstrate the basics of getting started with Aurora. Later sections introduce more advanced Aurora concepts and procedures, such as the different kinds of endpoints and how to scale Aurora clusters up and down.

> ⚠ **Important**
>
> Before you can create or connect to a DB cluster, make sure to complete the tasks in Setting up your environment for Amazon Aurora.

Topics

Creating and connecting to an Aurora MySQL DB cluster

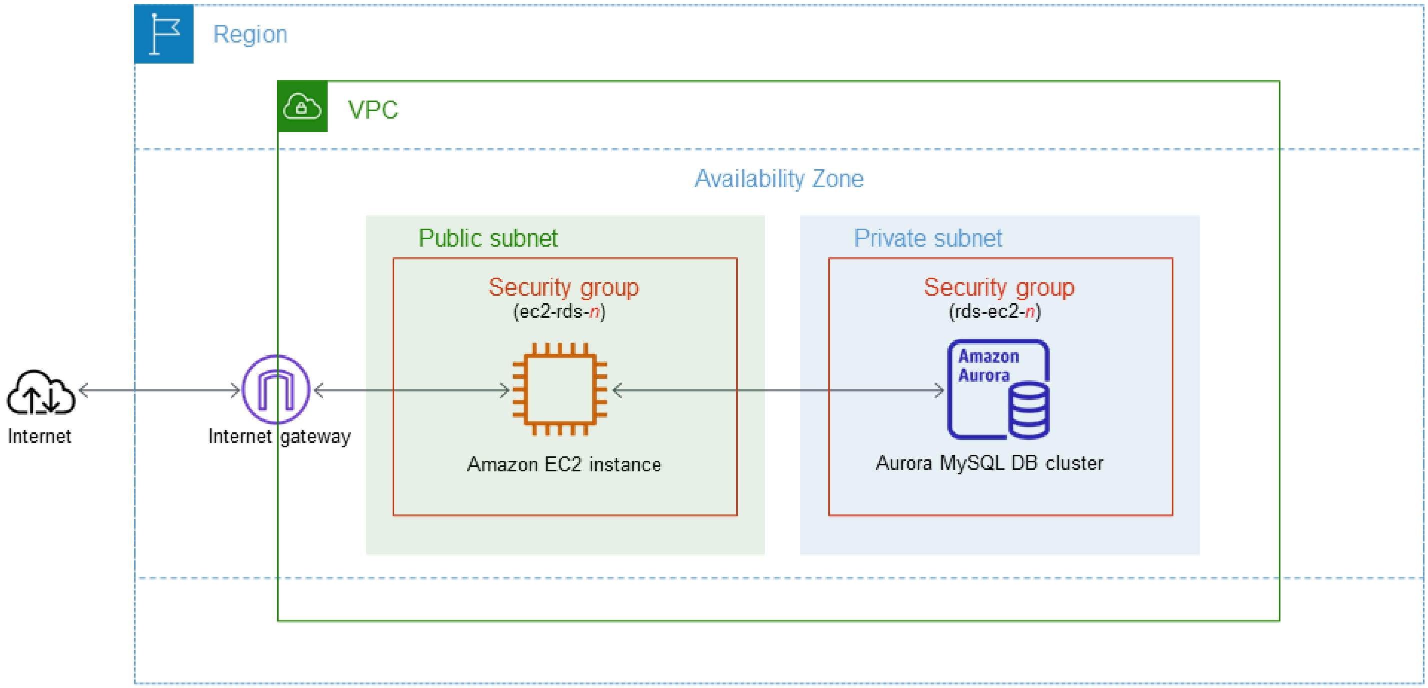
This tutorial creates an EC2 instance and an Aurora MySQL DB cluster. The tutorial shows you how to access the DB cluster from the EC2 instance using a standard MySQL client. As a best practice, this tutorial creates a private DB cluster in a virtual private cloud (VPC). In most cases, other resources in the same VPC, such as EC2 instances, can access the DB cluster, but resources outside of the VPC can't access it.

After you complete the tutorial, there is a public and private subnet in each Availability Zone in your VPC. In one Availability Zone, the EC2 instance is in the public subnet, and the DB instance is in the private subnet.

Important

There's no charge for creating an AWS account. However, by completing this tutorial, you might incur costs for the AWS resources that you use. You can delete these resources after you complete the tutorial if they are no longer needed.

The following diagram shows the configuration when the tutorial is complete.



This tutorial allows you to create your resources by using one of the following methods:

1. Use the AWS Management Console - Step 1: Create an EC2 instance and Step 2: Create an Aurora MySQL DB cluster
2. Use AWS CloudFormation to create the database instance and EC2 instance - (Optional) Create VPC, EC2 instance, and Aurora MySQL cluster using AWS CloudFormation

The first method uses **Easy create** to create a private Aurora MySQL DB cluster with the AWS Management Console. Here, you specify only the DB engine type, DB instance size, and DB cluster identifier. **Easy create** uses the default settings for the other configuration options.

When you use **Standard create** instead, you can specify more configuration options when you create a DB cluster. These options include settings for availability, security, backups, and maintenance. To create a public DB cluster, you must use **Standard create**. For information, see the section called "Creating a DB cluster".

Topics

- Prerequisites
- Step 1: Create an EC2 instance
- Step 2: Create an Aurora MySQL DB cluster
- (Optional) Create VPC, EC2 instance, and Aurora MySQL cluster using AWS CloudFormation
- Step 3: Connect to an Aurora MySQL DB cluster
- Step 4: Delete the EC2 instance and DB cluster
- (Optional) Delete the EC2 instance and DB cluster created with CloudFormation
- (Optional) Connect your DB cluster to a Lambda function

Prerequisites

Before you begin, complete the steps in the following sections:

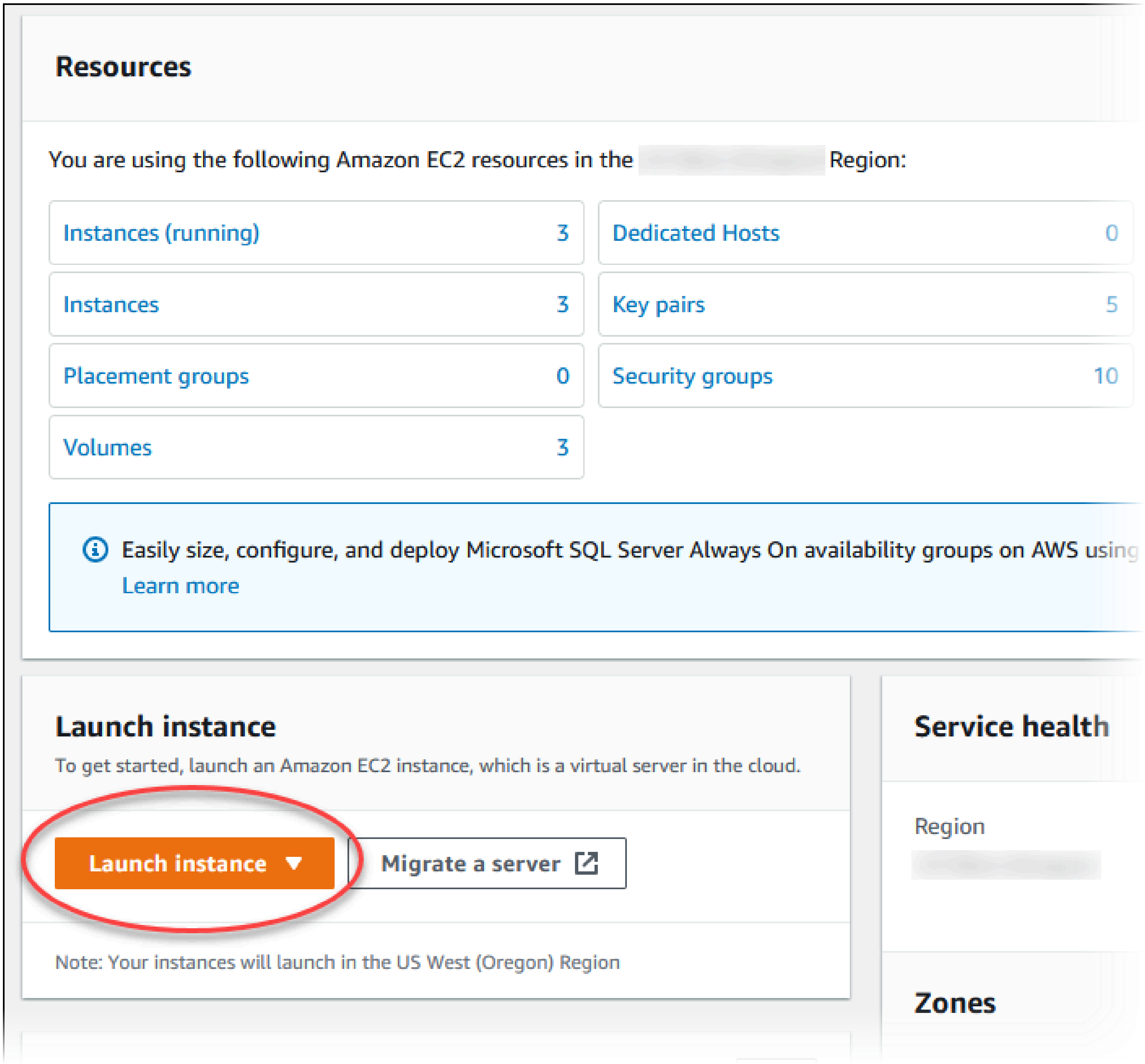
- Sign up for an AWS account
- Create a user with administrative access

Step 1: Create an EC2 instance

Create an Amazon EC2 instance that you will use to connect to your database.

To create an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the EC2 instance.
3. Choose **EC2 Dashboard**, and then choose **Launch instance**, as shown in the following image.



Resources

You are using the following Amazon EC2 resources in the [redacted] Region:

Instances (running)	3	Dedicated Hosts	0
Instances	3	Key pairs	5
Placement groups	0	Security groups	10
Volumes	3		

ⓘ Easily size, configure, and deploy Microsoft SQL Server Always On availability groups on AWS using
Learn more

Launch instance

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

Note: Your instances will launch in the US West (Oregon) Region

Service health

Region

[redacted]

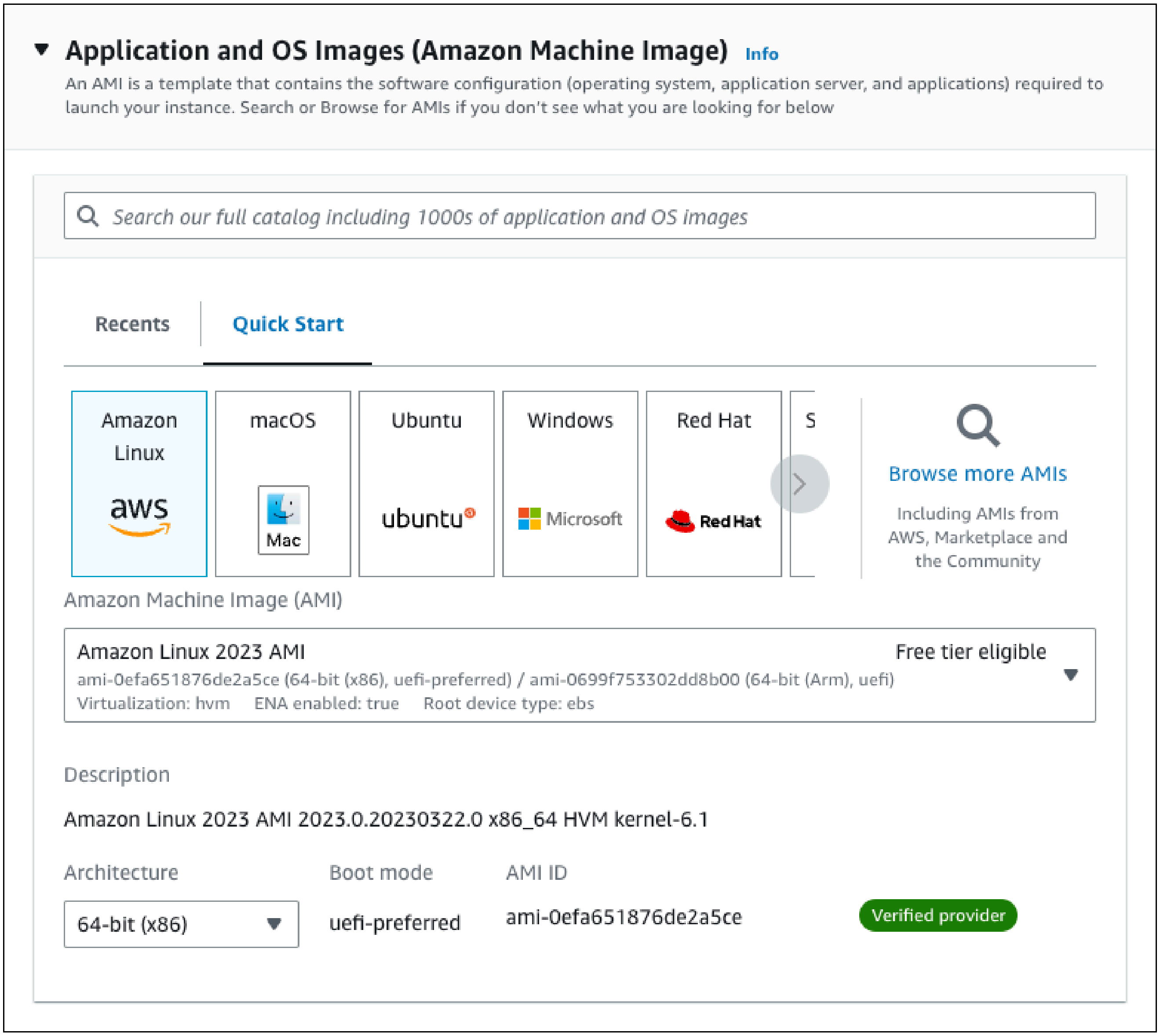
Zones

The **Launch an instance** page opens.

4. Choose the following settings on the **Launch an instance** page.

   a. Under **Name and tags**, for **Name**, enter **`ec2-database-connect`**.

   b. Under **Application and OS Images (Amazon Machine Image)**, choose **Amazon Linux**, and then choose the **Amazon Linux 2023 AMI**. Keep the default selections for the other choices.



Application and OS Images (Amazon Machine Image) Info

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images
Recents
Quick Start
Amazon Linux
aws
macOS
Mac
Ubuntu
ubuntu
Windows
Microsoft
Red Hat
Red Hat
S
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI — Free tier eligible
ami-0efa651876de2a5ce (64-bit (x86), uefi-preferred) / ami-0699f753302dd8b00 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 AMI 2023.0.20230322.0 x86_64 HVM kernel-6.1

Architecture: 64-bit (x86)

Boot mode: uefi-preferred

AMI ID: ami-0efa651876de2a5ce

c. Under **Instance type**, choose **t2.micro**.

d. Under **Key pair (login)**, choose a **Key pair name** to use an existing key pair. To create a new key pair for the Amazon EC2 instance, choose **Create new key pair** and then use the **Create key pair** window to create it.

For more information about creating a new key pair, see Create a key pair in the *Amazon EC2 User Guide*.

e. For **Allow SSH traffic** in **Network settings**, choose the source of SSH connections to the EC2 instance.

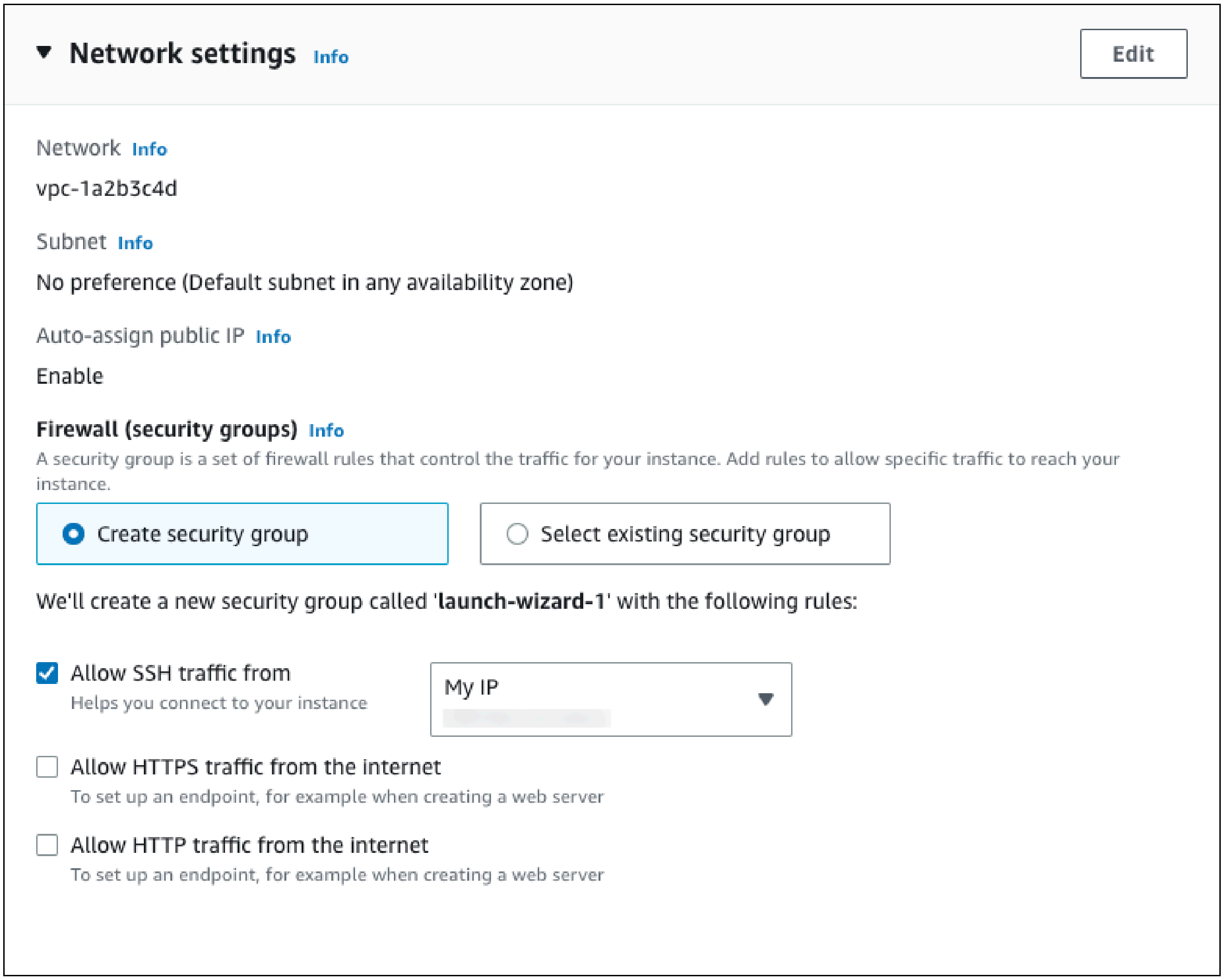
You can choose **My IP** if the displayed IP address is correct for SSH connections. Otherwise, you can determine the IP address to use to connect to EC2 instances in your

VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com. An example of an IP address is 192.0.2.1/32.

In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, make sure to determine the range of IP addresses used by client computers.

> ⚠ **Warning**
>
> If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public EC2 instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your EC2 instances using SSH.

The following image shows an example of the **Network settings** section.



▼ **Network settings** Info

Edit

Network Info

vpc-1a2b3c4d

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

● Create security group ○ Select existing security group

We'll create a new security group called '**launch-wizard-1**' with the following rules:

☑ Allow SSH traffic from
Helps you connect to your instance

My IP ▼
[redacted]

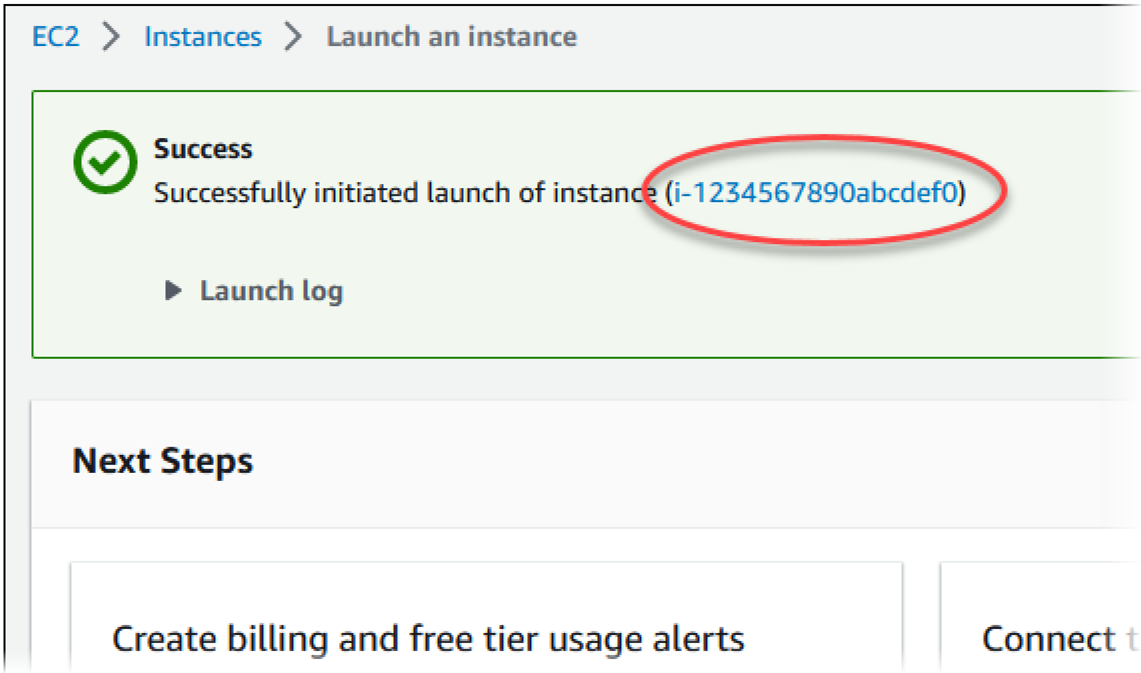
□ Allow HTTPS traffic from the internet
To set up an endpoint, for example when creating a web server

□ Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server

f. Leave the default values for the remaining sections.

g. Review a summary of your EC2 instance configuration in the **Summary** panel, and when you're ready, choose **Launch instance**.

5. On the **Launch Status** page, note the identifier for your new EC2 instance, for example: `i-1234567890abcdef0`.



EC2 > Instances > Launch an instance

Success

Successfully initiated launch of instance (i-1234567890abcdef0)

▶ Launch log

Next Steps

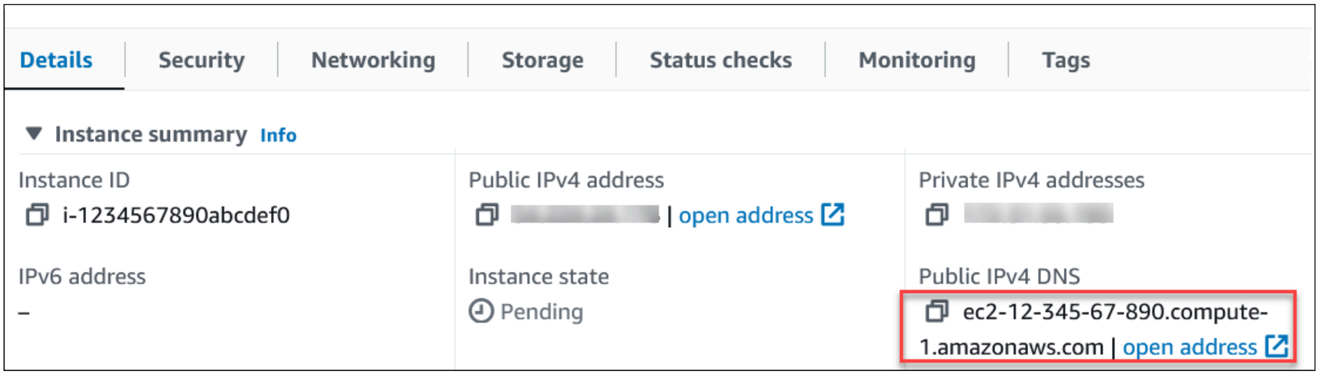
Create billing and free tier usage alerts

Connect t

6. Choose the EC2 instance identifier to open the list of EC2 instances, and then select your EC2 instance.

7. In the **Details** tab, note the following values, which you need when you connect using SSH:

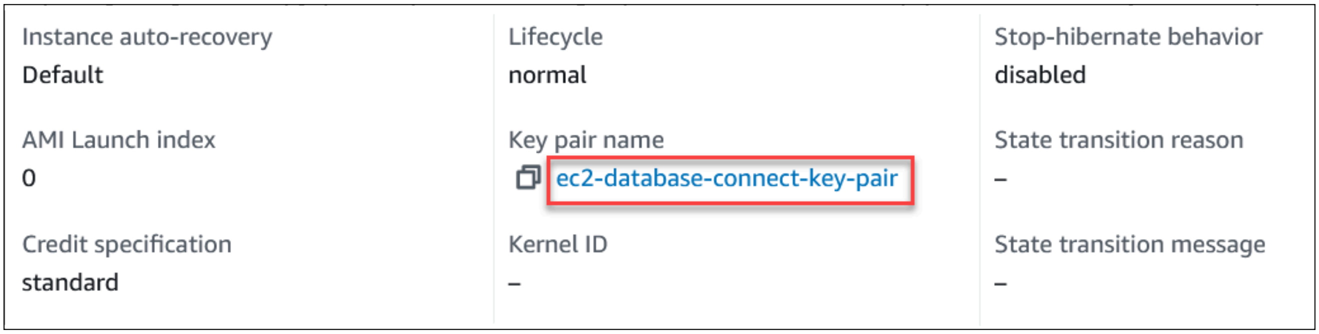
   a. In **Instance summary**, note the value for **Public IPv4 DNS**.



▼ Instance summary Info

Instance ID i-1234567890abcdef0	Public IPv4 address [redacted] | open address	Private IPv4 addresses [redacted]
IPv6 address –	Instance state Pending	Public IPv4 DNS ec2-12-345-67-890.compute-1.amazonaws.com | open address

   b. In **Instance details**, note the value for **Key pair name**.



Instance auto-recovery Default	Lifecycle normal	Stop-hibernate behavior disabled
AMI Launch index 0	Key pair name ec2-database-connect-key-pair	State transition reason –
Credit specification standard	Kernel ID –	State transition message –

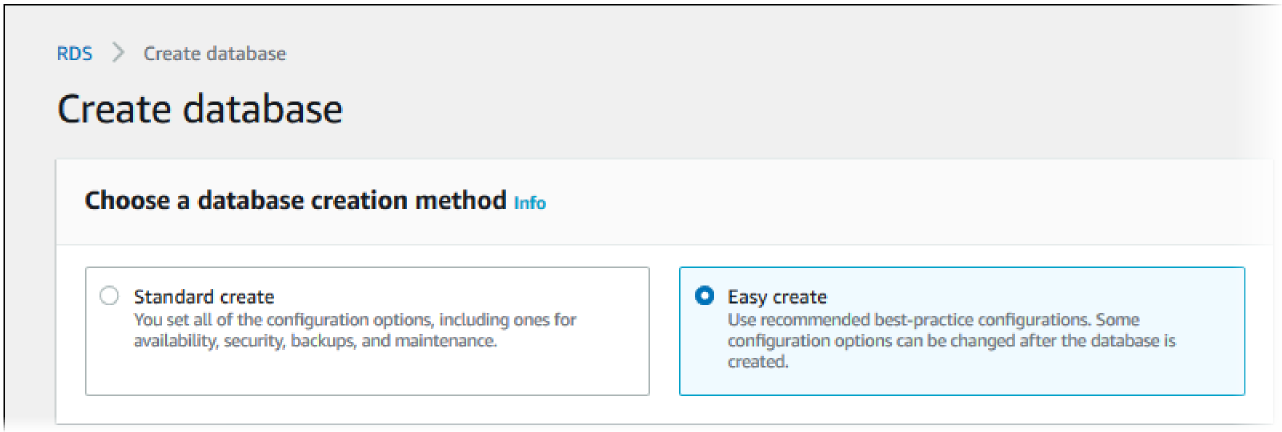
8. Wait until the **Instance state** for your EC2 instance has a status of **Running** before continuing.

Step 2: Create an Aurora MySQL DB cluster

In this example, you use **Easy create** to create an Aurora MySQL DB cluster with a db.r6g.large DB instance class.

To create an Aurora MySQL DB cluster with Easy create

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the Amazon RDS console, choose the AWS Region in which you want to create the DB cluster.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database** and make sure that **Easy create** is chosen.



RDS > Create database

Create database

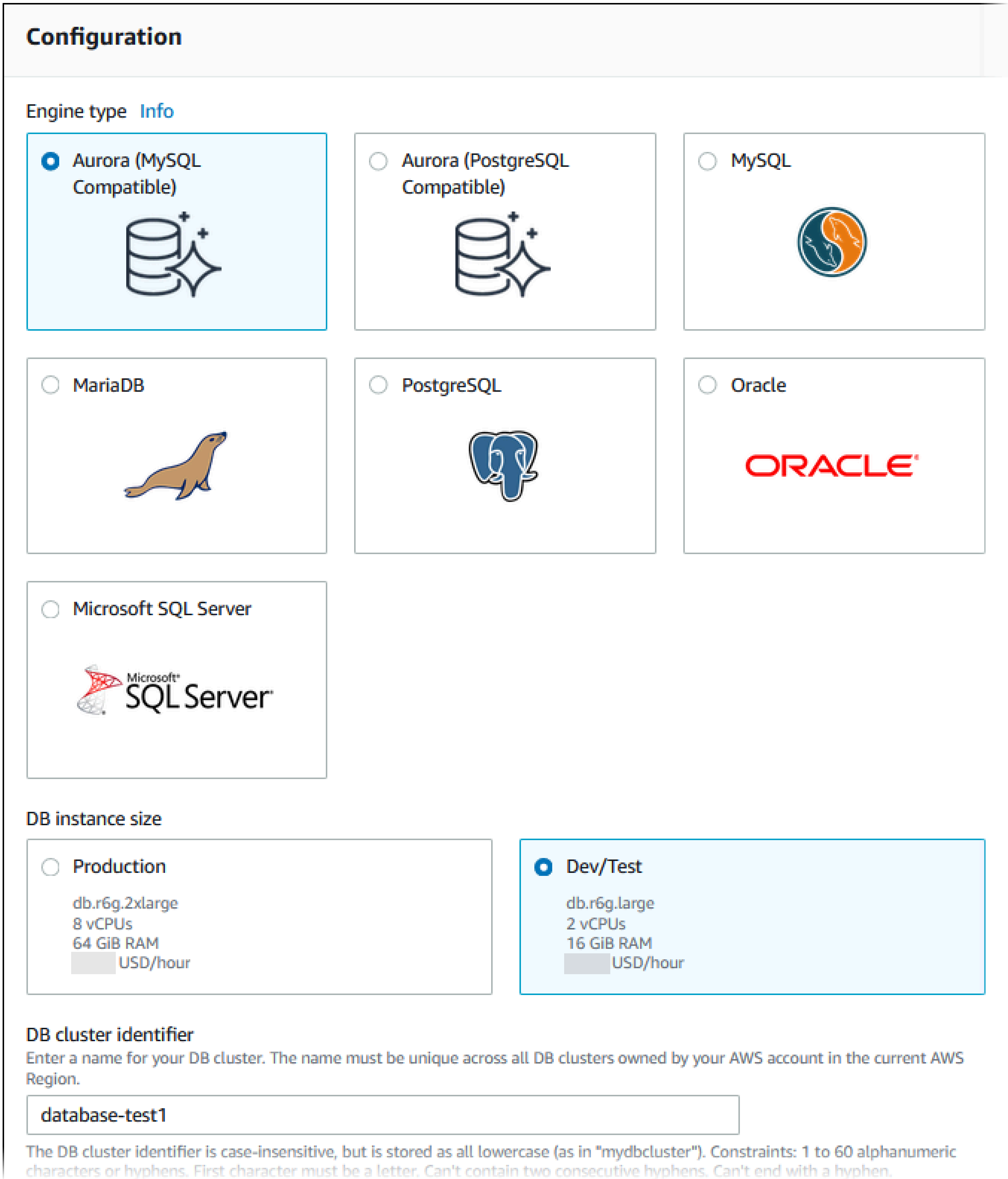
Choose a database creation method Info

○ Standard create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

● Easy create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

5. In **Configuration**, choose **Aurora (MySQL Compatible)** for **Engine type**.
6. For **DB instance size**, choose **Dev/Test**.
7. For **DB cluster identifier**, enter **`database-test1`**.

   The **Create database** page should look similar to the following image.



Configuration

Engine type Info

● Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

DB instance size

○ Production
db.r6g.2xlarge
8 vCPUs
64 GiB RAM
[redacted] USD/hour

● Dev/Test
db.r6g.large
2 vCPUs
16 GiB RAM
[redacted] USD/hour

DB cluster identifier
Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

database-test1

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

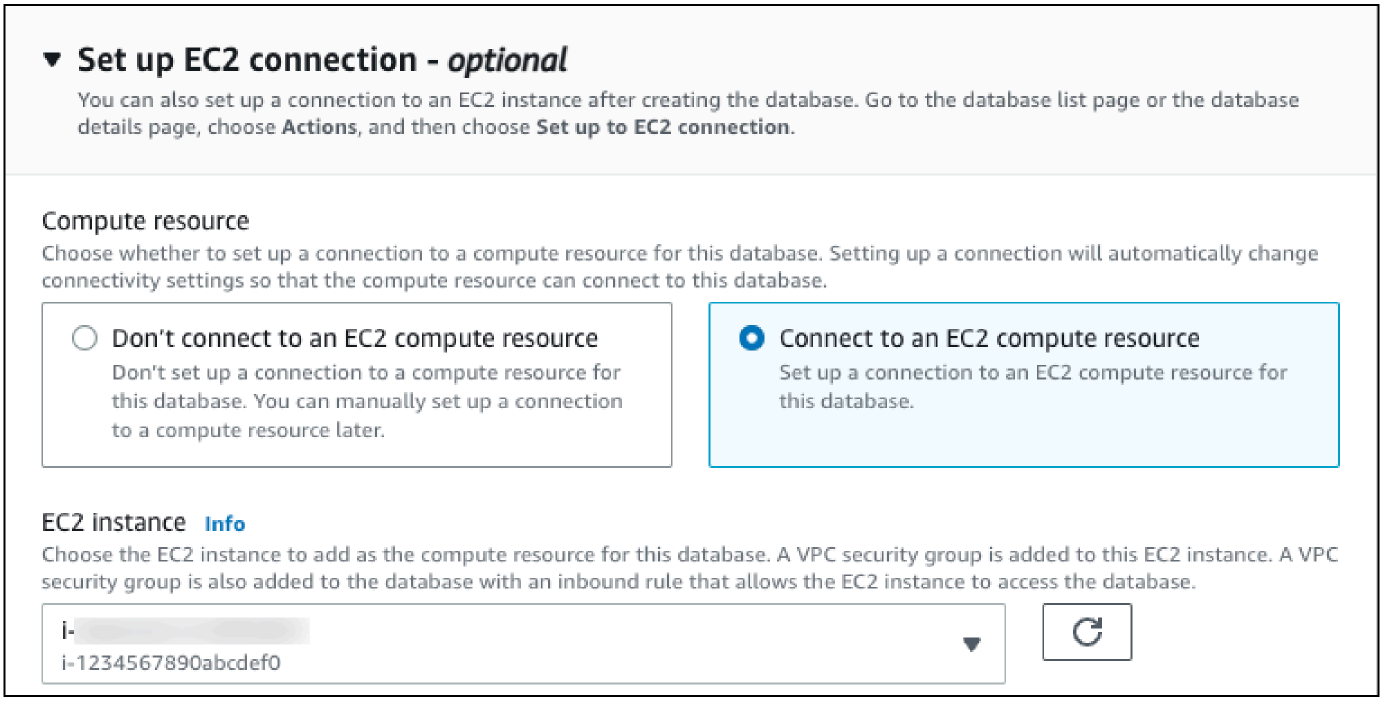
8. For **Master username**, enter a name for the master user, or keep the default name.

9. To use an automatically generated master password for the DB cluster, select **Auto generate a password**.

To enter your master password, make sure that **Auto generate a password** is cleared, and then enter the same password in **Master password** and **Confirm password**.

10. To set up a connection with the EC2 instance you created previously, open **Set up EC2 connection - *optional***.

    Select **Connect to an EC2 compute resource**. Choose the EC2 instance you created previously.



▼ Set up EC2 connection - *optional*

You can also set up a connection to an EC2 instance after creating the database. Go to the database list page or the database details page, choose **Actions**, and then choose **Set up to EC2 connection**.

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

○ Don't connect to an EC2 compute resource

Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

● Connect to an EC2 compute resource

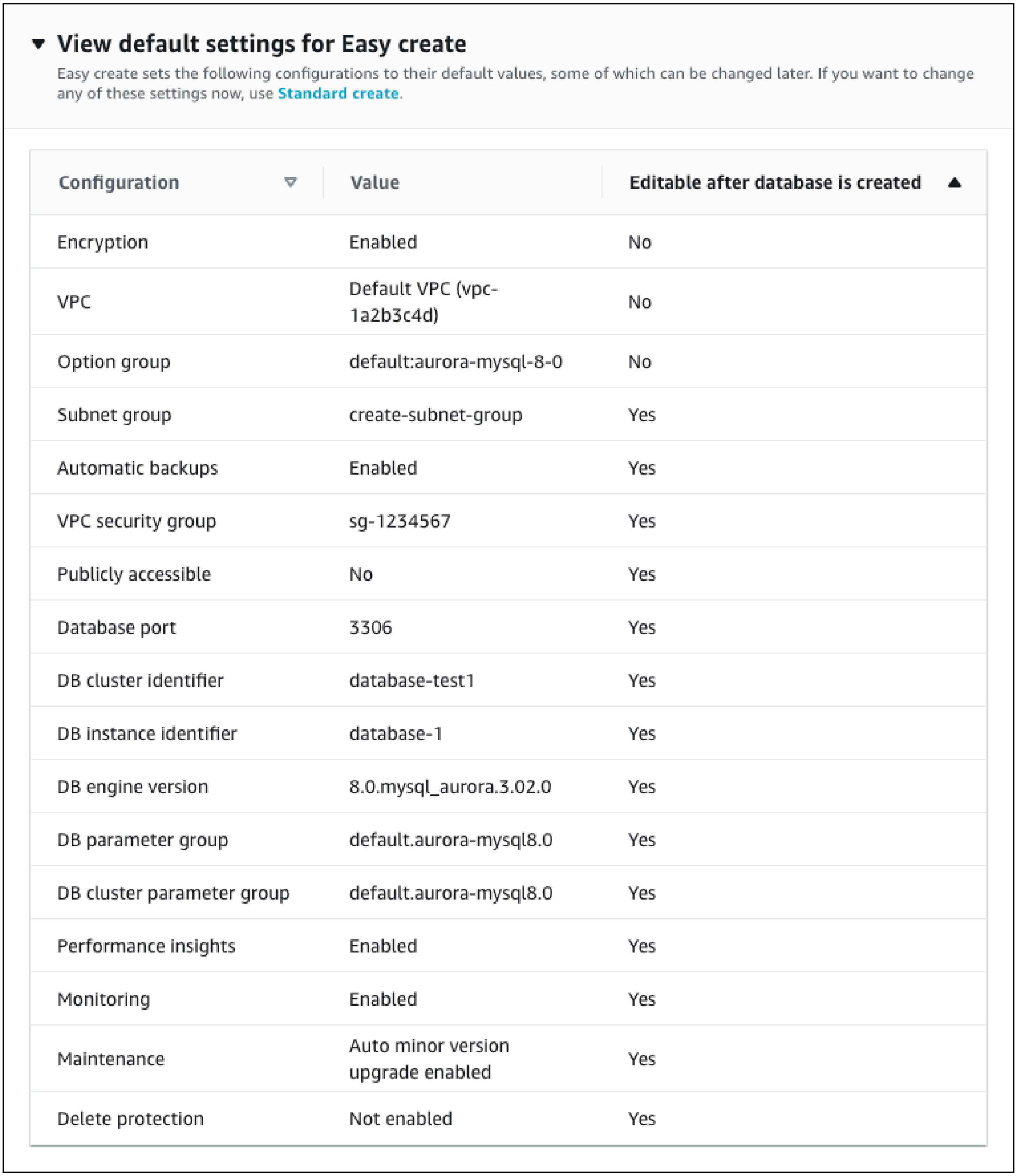
Set up a connection to an EC2 compute resource for this database.

EC2 instance Info

Choose the EC2 instance to add as the compute resource for this database. A VPC security group is added to this EC2 instance. A VPC security group is also added to the database with an inbound rule that allows the EC2 instance to access the database.

i-[redacted]
i-1234567890abcdef0

11. Open **View default settings for Easy create**.



▼ View default settings for Easy create

Easy create sets the following configurations to their default values, some of which can be changed later. If you want to change any of these settings now, use **Standard create**.

Configuration ▽	Value	Editable after database is created ▲
Encryption	Enabled	No
VPC	Default VPC (vpc-1a2b3c4d)	No
Option group	default:aurora-mysql-8-0	No
Subnet group	create-subnet-group	Yes
Automatic backups	Enabled	Yes
VPC security group	sg-1234567	Yes
Publicly accessible	No	Yes
Database port	3306	Yes
DB cluster identifier	database-test1	Yes
DB instance identifier	database-1	Yes
DB engine version	8.0.mysql_aurora.3.02.0	Yes
DB parameter group	default.aurora-mysql8.0	Yes
DB cluster parameter group	default.aurora-mysql8.0	Yes
Performance insights	Enabled	Yes
Monitoring	Enabled	Yes
Maintenance	Auto minor version upgrade enabled	Yes
Delete protection	Not enabled	Yes

You can examine the default settings used with **Easy create**. The **Editable after database is created** column shows which options you can change after you create the database.

- If a setting has **No** in that column, and you want a different setting, you can use **Standard create** to create the DB cluster.
- If a setting has **Yes** in that column, and you want a different setting, you can either use **Standard create** to create the DB cluster, or modify the DB cluster after you create it to change the setting.

12. Choose **Create database**.

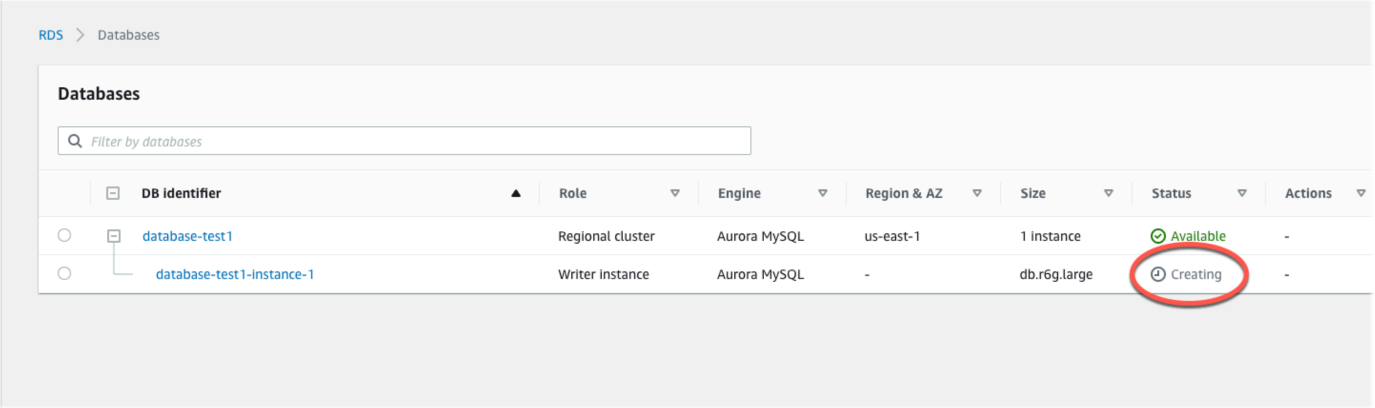
    To view the master username and password for the DB cluster, choose **View credential details**.

    You can use the username and password that appears to connect to the DB cluster as the master user.

    > **Important**
    >
    > You can't view the master user password again. If you don't record it, you might have to change it.
    > If you need to change the master user password after the DB cluster is available, you can modify the DB cluster to do so. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

13. In the **Databases** list, choose the name of the new Aurora MySQL DB cluster to show its details.

    The writer instance has a status of **Creating** until the DB cluster is ready to use.



		DB identifier	Role	Engine	Region & AZ	Size	Status	Actions
○	⊟	database-test1	Regional cluster	Aurora MySQL	us-east-1	1 instance	Available	-
○		database-test1-instance-1	Writer instance	Aurora MySQL	-	db.r6g.large	Creating	-

When the status of the writer instance changes to **Available**, you can connect to the DB cluster. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new DB cluster is available.

(Optional) Create VPC, EC2 instance, and Aurora MySQL cluster using AWS CloudFormation

Instead of using the console to create your VPC, EC2 instance, and Aurora MySQL DB cluster, you can use AWS CloudFormation to provision AWS resources by treating infrastructure as code. To help you organize your AWS resources into smaller and more manageable units, you can use the AWS CloudFormation nested stack functionality. For more information, see Creating a stack on the AWS CloudFormation console and Working with nested stacks.

Important

AWS CloudFormation is free, but the resources that CloudFormation creates are live. You incur the standard usage fees for these resources until you terminate them. The total charges will be minimal. For information about how you might minimize any charges, go to AWS Free Tier.

To create your resources using the AWS CloudFormation console, complete the following steps:

- Step 1: Download the CloudFormation template
- Step 2: Configure your resources using CloudFormation

Download the CloudFormation template

A CloudFormation template is a JSON or YAML text file that contains the configuration information about the resources you want to create in the stack. This template also creates a VPC and a bastion host for you along with the Aurora cluster.

To download the template file, open the following link, Aurora MySQL CloudFormation template.

In the Github page, click the *Download raw file* button to save the template YAML file.

Configure your resources using CloudFormation

Note

Before starting this process, make sure you have a Key pair for an EC2 instance in your AWS account. For more information, see Amazon EC2 key pairs and Linux instances.

When you use the AWS CloudFormation template, you must select the correct parameters to make sure your resources are created properly. Follow the steps below:

1. Sign in to the AWS Management Console and open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create Stack**.
3. In the Specify template section, select **Upload a template file from your computer**, and then choose **Next**.
4. In the **Specify stack details** page, set the following parameters:
   a. Set **Stack name** to **AurMySQLTestStack**.
   b. Under **Parameters**, set **Availability Zones** by selecting two availability zones.
   c. Under **Linux Bastion Host configuration**, for **Key Name**, select a key pair to login to your EC2 instance.
   d. In **Linux Bastion Host configuration** settings, set the **Permitted IP range** to your IP address. To connect to EC2 instances in your VPC using Secure Shell (SSH), determine your public IP address using the service at https://checkip.amazonaws.com. An example of an IP address is 192.0.2.1/32.

   > ⚠ **Warning**
   >
   > If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public EC2 instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your EC2 instances using SSH.

   e. Under **Database General configuration**, set **Database instance class** to **db.r6g.large**.
   f. Set **Database name** to **`database-test1`**.
   g. For **Database master username**, enter a name for the master user.
   h. Set **Manage DB master user password with Secrets Manager** to `false` for this tutorial.
   i. For **Database password**, set a password of your choice. Remember this password for further steps in the tutorial.
   j. Set **Multi-AZ deployment** to `false`.
   k. Leave all other settings as the default values. Click **Next** to continue.

5. In the **Configure stack options** page, leave all the default options. Click **Next** to continue.
6. In the **Review stack** page, select **Submit** after checking the database and Linux bastion host options.

After the stack creation process completes, view the stacks with names *BastionStack* and *AMSNS* to note the information you need to connect to the database. For more information, see Viewing AWS CloudFormation stack data and resources on the AWS Management Console.

Step 3: Connect to an Aurora MySQL DB cluster

You can use any standard SQL client application to connect to the DB cluster. In this example, you connect to the Aurora MySQL DB cluster using the mysql command line client.

To connect to the Aurora MySQL DB cluster

1. Find the endpoint (DNS name) and port number of the writer instance for your DB cluster.

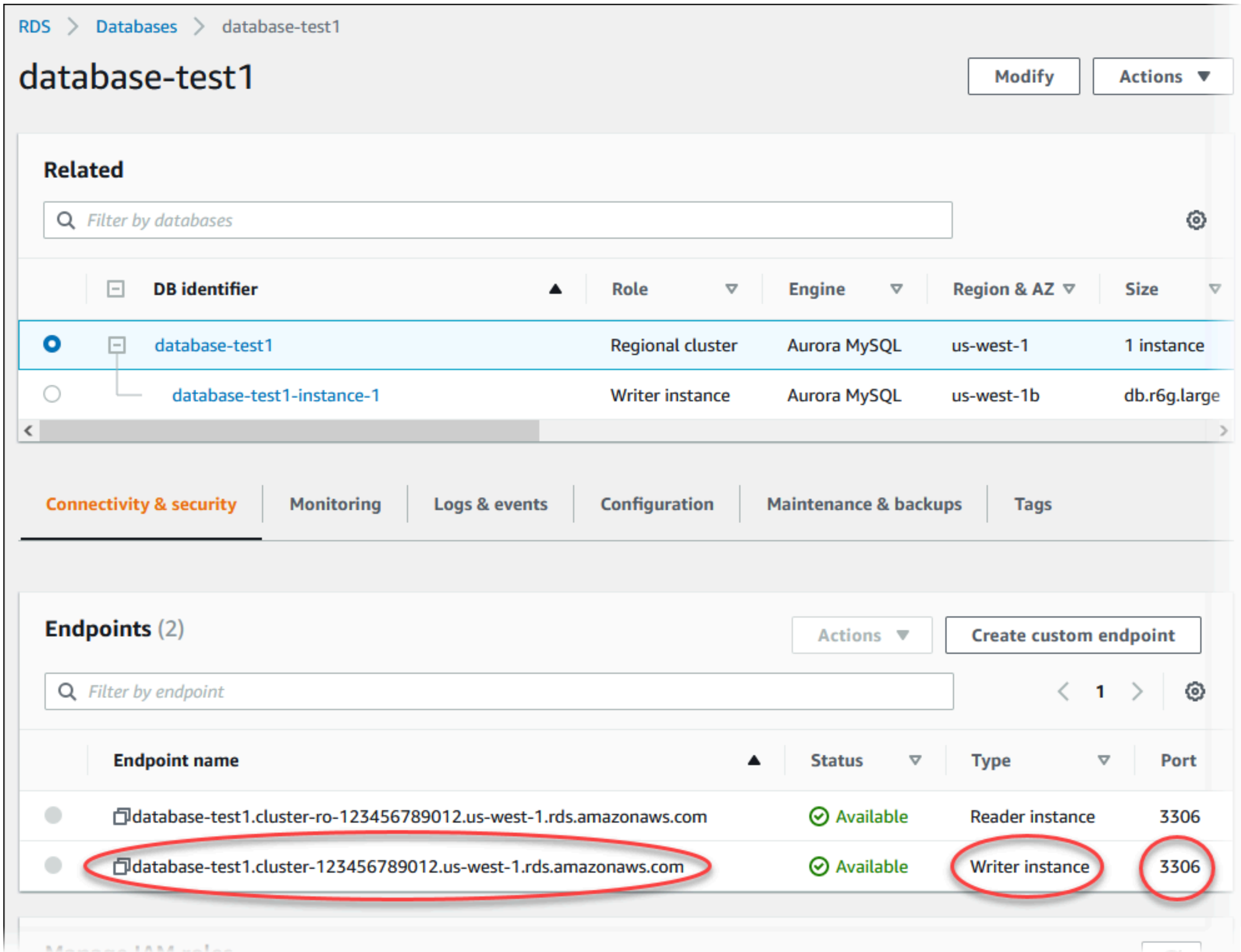
   a. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   b. In the upper-right corner of the Amazon RDS console, choose the AWS Region for the DB cluster.

   c. In the navigation pane, choose **Databases**.

   d. Choose the Aurora MySQL DB cluster name to display its details.

   e. On the **Connectivity & security** tab, copy the endpoint of the writer instance. Also, note the port number. You need both the endpoint and the port number to connect to the DB cluster.



	DB identifier	Role	Engine	Region & AZ	Size
●	database-test1	Regional cluster	Aurora MySQL	us-west-1	1 instance
○	database-test1-instance-1	Writer instance	Aurora MySQL	us-west-1b	db.r6g.large

	Endpoint name	Status	Type	Port
○	database-test1.cluster-ro-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Reader instance	3306
○	database-test1.cluster-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Writer instance	3306

2. Connect to the EC2 instance that you created earlier by following the steps in Connect to your Linux instance in the *Amazon EC2 User Guide*.

   We recommend that you connect to your EC2 instance using SSH. If the SSH client utility is installed on Windows, Linux, or Mac, you can connect to the instance using the following command format:

   ```
   ssh -i location_of_pem_file ec2-user@ec2-instance-public-dns-name
   ```

   For example, suppose that `ec2-database-connect-key-pair.pem` is stored in `/dir1` on Linux, and the public IPv4 DNS for your EC2 instance is `ec2-12-345-678-90.compute-1.amazonaws.com`. Then, your SSH command would look as follows:

```
ssh -i /dir1/ec2-database-connect-key-pair.pem ec2-
user@ec2-12-345-678-90.compute-1.amazonaws.com
```

3. Get the latest bug fixes and security updates by updating the software on your EC2 instance. To do so, use the following command.

   Note

   The `-y` option installs the updates without asking for confirmation. To examine updates before installing, omit this option.

   ```
   sudo dnf update -y
   ```

4. To install the mysql command line client from MariaDB on Amazon Linux 2023, run the following command:

   ```
   sudo dnf install mariadb105
   ```

5. Connect to the Aurora MySQL DB cluster. For example, enter the following command. This action lets you connect to the Aurora MySQL DB cluster using the MySQL client.

   Substitute the endpoint of the writer instance for *`endpoint`*, and substitute the master username that you used for *`admin`*. Provide the master password that you used when prompted for a password.

   ```
   mysql -h endpoint -P 3306 -u admin -p
   ```

   After you enter the password for the user, you should see output similar to the following.

   ```
   Welcome to the MariaDB monitor.  Commands end with ; or \g.
   Your MySQL connection id is 217
   Server version: 8.0.23 Source distribution

   Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

   Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
   ```

```
MySQL [(none)]>
```

For more information about connecting to an Aurora MySQL DB cluster, see Connecting to an Amazon Aurora MySQL DB cluster. If you can't connect to your DB cluster, see Can't connect to Amazon RDS DB instance.

For security, it is a best practice to use encrypted connections. Only use an unencrypted MySQL connection when the client and server are in the same VPC and the network is trusted. For information about using encrypted connections, see Connecting to Aurora MySQL using SSL.

6. Run SQL commands.

   For example, the following SQL command shows the current date and time:

   ```
   SELECT CURRENT_TIMESTAMP;
   ```

Step 4: Delete the EC2 instance and DB cluster

After you connect to and explore the sample EC2 instance and DB cluster that you created, delete them so you're no longer charged for them.

If you used AWS CloudFormation to create resources, skip this step and go to the next step.

To delete the EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Instances**.
3. Select the EC2 instance, and choose **Instance state, Terminate instance**.
4. Choose **Terminate** when prompted for confirmation.

For more information about deleting an EC2 instance, see Terminate your instance in the *Amazon EC2 User Guide*.

To delete the DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** and then choose the DB instance associated with the DB cluster.
3. For **Actions**, choose **Delete**.
4. Clear **Create final snapshot?**.
5. Complete the acknowledgement and choose **Delete**.

After all of the DB instances associated with a DB cluster are deleted, the DB cluster is deleted automatically.

(Optional) Delete the EC2 instance and DB cluster created with CloudFormation

If you used AWS CloudFormation to create resources, delete the CloudFormation stack after you connect to and explore the sample EC2 instance and DB cluster, so you're no longer charged for them.

To delete the CloudFormation resources

1. Open the AWS CloudFormation console.
2. On the **Stacks** page in the CloudFormation console, select the root stack (the stack without the name VPCStack, BastionStack or AMSNS).
3. Choose **Delete**.
4. Select **Delete stack** when prompted for confirmation.

For more information about deleting a stack in CloudFormation, see Deleting a stack on the AWS CloudFormation console in the *AWS CloudFormation User Guide*.

(Optional) Connect your DB cluster to a Lambda function

You can also connect your Aurora MySQL DB cluster to a Lambda serverless compute resource. Lambda functions allow you to run code without provisioning or managing infrastructure. A Lambda function also allows you to automatically respond to code execution requests at any scale, from a dozen events a day to hundreds of per second. For more information, see Automatically connecting a Lambda function and an Aurora DB cluster.

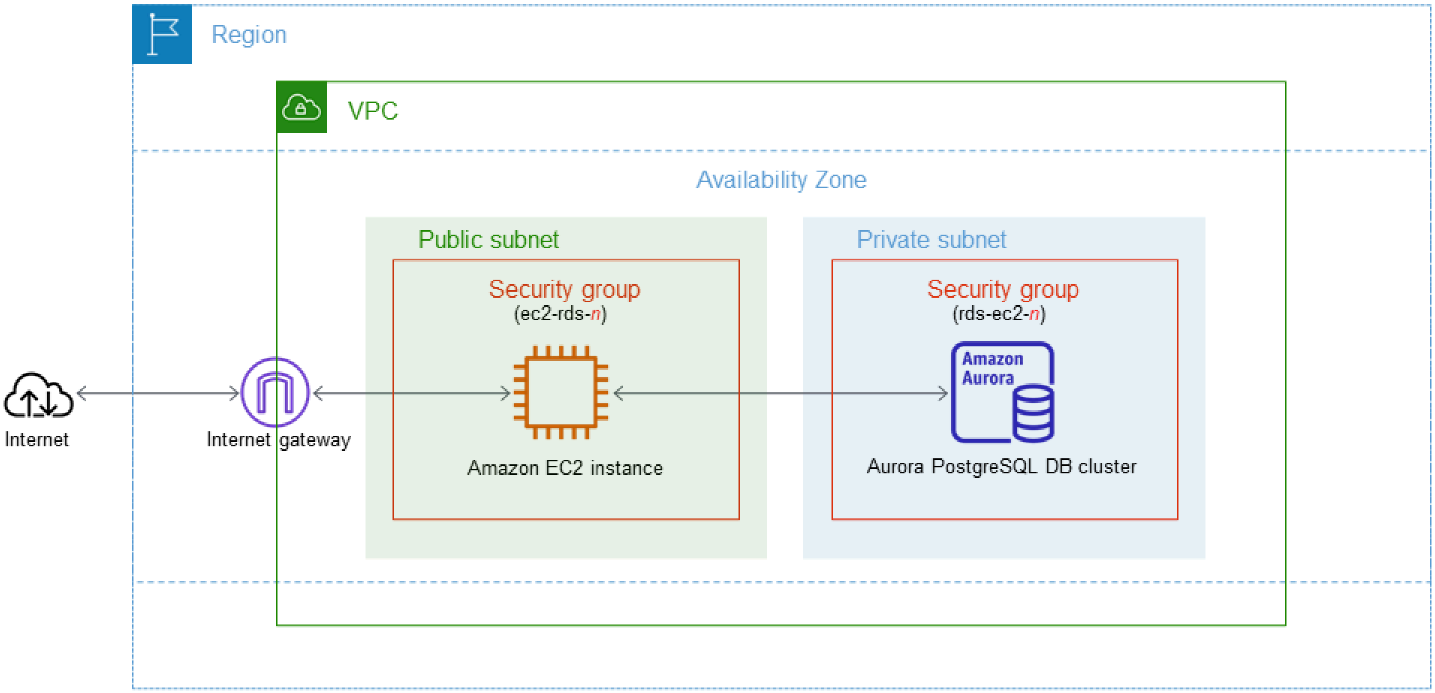
Creating and connecting to an Aurora PostgreSQL DB cluster

This tutorial creates an EC2 instance and an Aurora PostgreSQL DB cluster. The tutorial shows you how to access the DB cluster from the EC2 instance using a standard PostgreSQL client. As a best practice, this tutorial creates a private DB cluster in a virtual private cloud (VPC). In most cases, other resources in the same VPC, such as EC2 instances, can access the DB cluster, but resources outside of the VPC can't access it.

After you complete the tutorial, there is a public and private subnet in each Availability Zone in your VPC. In one Availability Zone, the EC2 instance is in the public subnet, and the DB instance is in the private subnet.

> ⚠ **Important**
>
> There's no charge for creating an AWS account. However, by completing this tutorial, you might incur costs for the AWS resources that you use. You can delete these resources after you complete the tutorial if they are no longer needed.

The following diagram shows the configuration when the tutorial is complete.



This tutorial allows you to create your resources by using one of the following methods:

1. Use the AWS Management Console - Step 1: Create an EC2 instanceand Step 2: Create an Aurora PostgreSQL DB cluster
2. Use AWS CloudFormation to create the database instance and EC2 instance - (Optional) Create VPC, EC2 instance, and Aurora PostgreSQL cluster using AWS CloudFormation

The first method uses **Easy create** to create a private Aurora PostgreSQL DB cluster with the AWS Management Console. Here, you specify only the DB engine type, DB instance size, and DB cluster identifier. **Easy create** uses the default settings for the other configuration options.

When you use **Standard create** instead, you can specify more configuration options when you create a DB cluster. These options include settings for availability, security, backups, and maintenance. To create a public DB cluster, you must use **Standard create**. For information, see the section called “Creating a DB cluster”.

Topics

Prerequisites

Before you begin, complete the steps in the following sections:

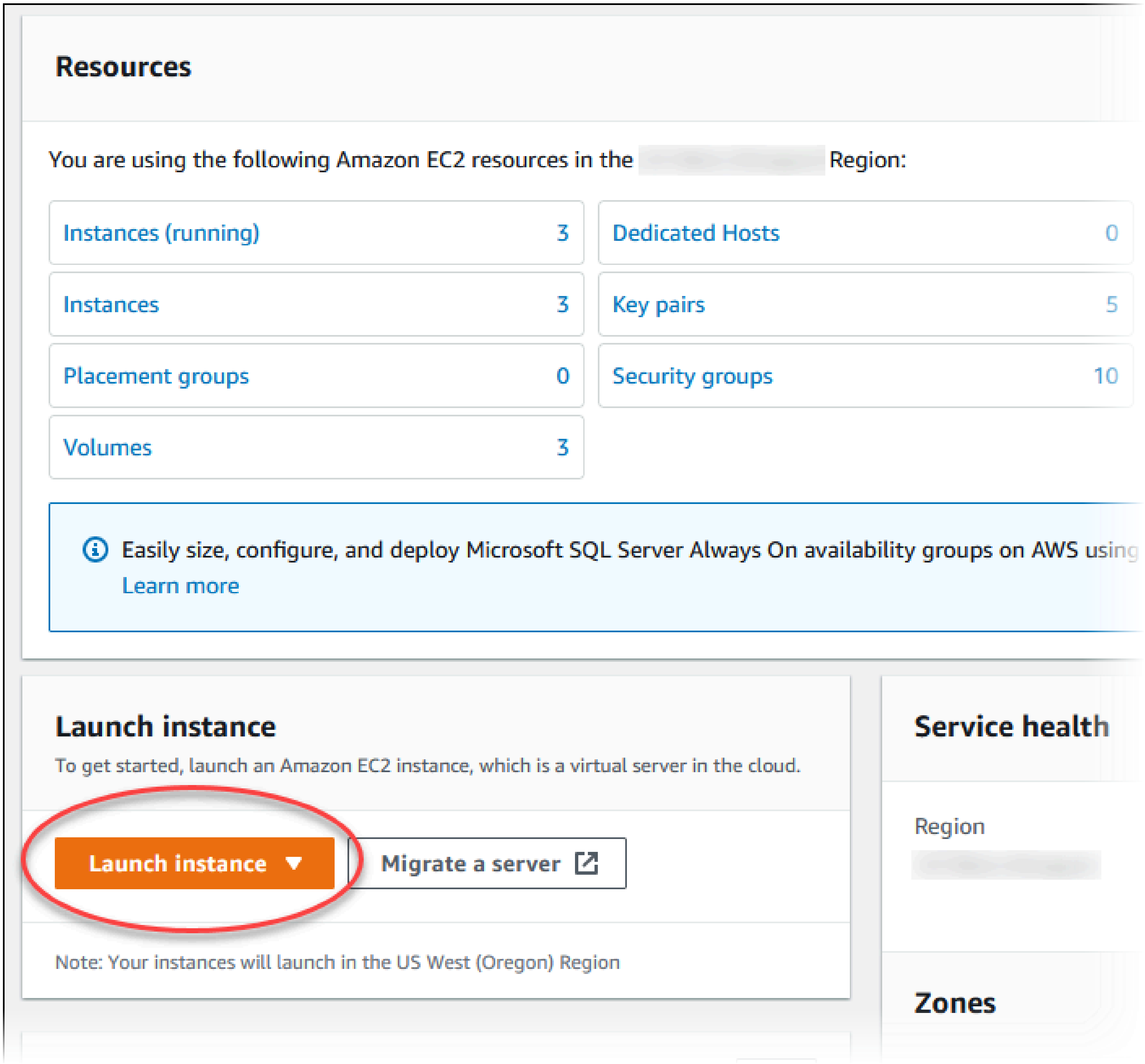
- Sign up for an AWS account
- Create a user with administrative access

Step 1: Create an EC2 instance

Create an Amazon EC2 instance that you will use to connect to your database.

To create an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the EC2 instance.
3. Choose **EC2 Dashboard**, and then choose **Launch instance**, as shown in the following image.



Resources

You are using the following Amazon EC2 resources in the [redacted] Region:

Instances (running)	3	Dedicated Hosts	0
Instances	3	Key pairs	5
Placement groups	0	Security groups	10
Volumes	3		

ⓘ Easily size, configure, and deploy Microsoft SQL Server Always On availability groups on AWS using
Learn more

Launch instance

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

Launch instance ▼ Migrate a server

Note: Your instances will launch in the US West (Oregon) Region

Service health

Region

[redacted]

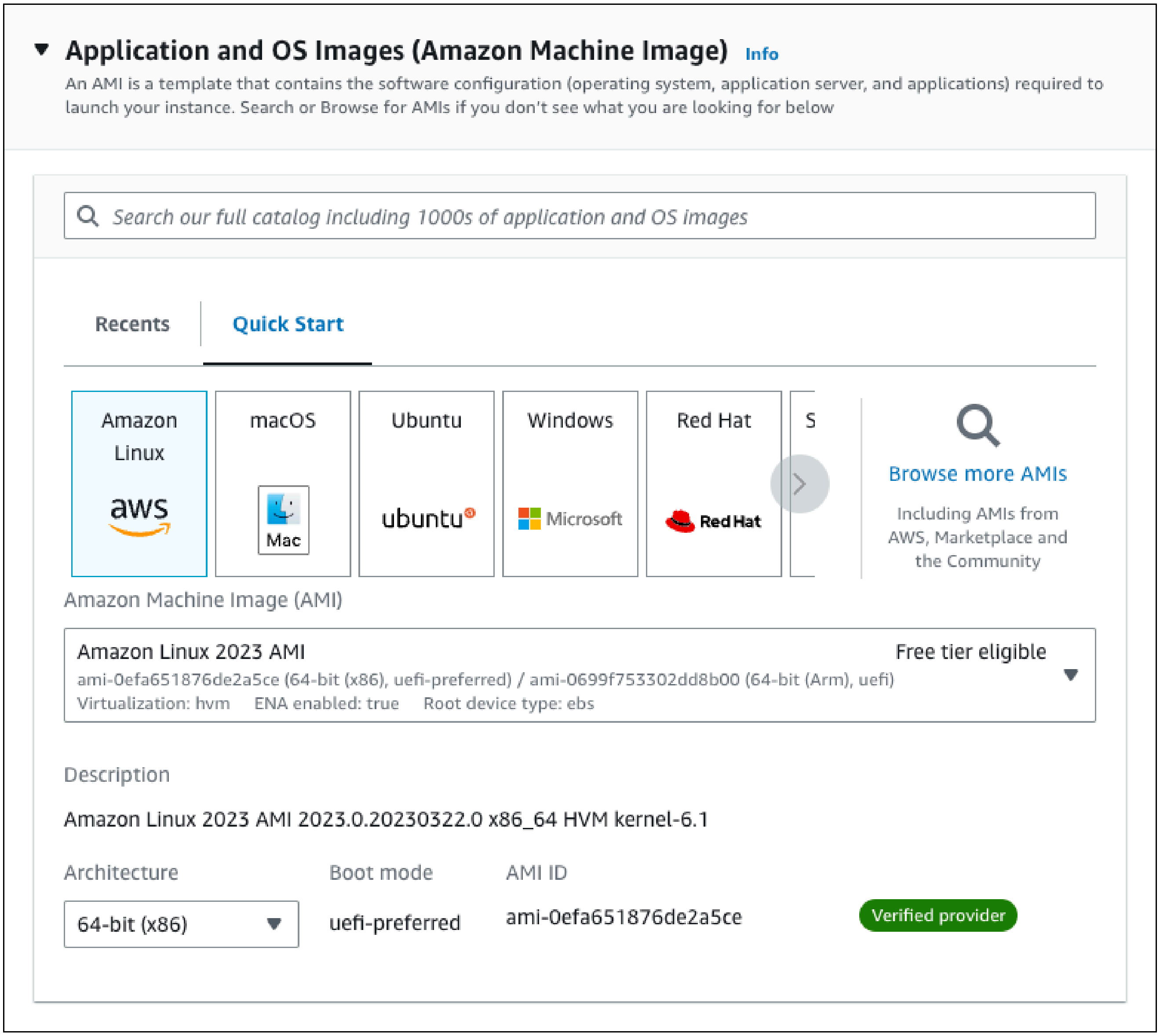
Zones

The **Launch an instance** page opens.

4. Choose the following settings on the **Launch an instance** page.

   a. Under **Name and tags**, for **Name**, enter **`ec2-database-connect`**.

   b. Under **Application and OS Images (Amazon Machine Image)**, choose **Amazon Linux**, and then choose the **Amazon Linux 2023 AMI**. Keep the default selections for the other choices.



Info
Search our full catalog including 1000s of application and OS images
Recents
Quick Start
Amazon Linux
aws
macOS
Mac
Ubuntu
ubuntu
Windows
Microsoft
Red Hat
Red Hat
S
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community
Verified provider

Application and OS Images (Amazon Machine Image)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI Free tier eligible
ami-0efa651876de2a5ce (64-bit (x86), uefi-preferred) / ami-0699f753302dd8b00 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 AMI 2023.0.20230322.0 x86_64 HVM kernel-6.1

Architecture: 64-bit (x86)

Boot mode: uefi-preferred

AMI ID: ami-0efa651876de2a5ce

c. Under **Instance type**, choose **t2.micro**.

d. Under **Key pair (login)**, choose a **Key pair name** to use an existing key pair. To create a new key pair for the Amazon EC2 instance, choose **Create new key pair** and then use the **Create key pair** window to create it.

For more information about creating a new key pair, see Create a key pair in the *Amazon EC2 User Guide*.

e. For **Allow SSH traffic** in **Network settings**, choose the source of SSH connections to the EC2 instance.

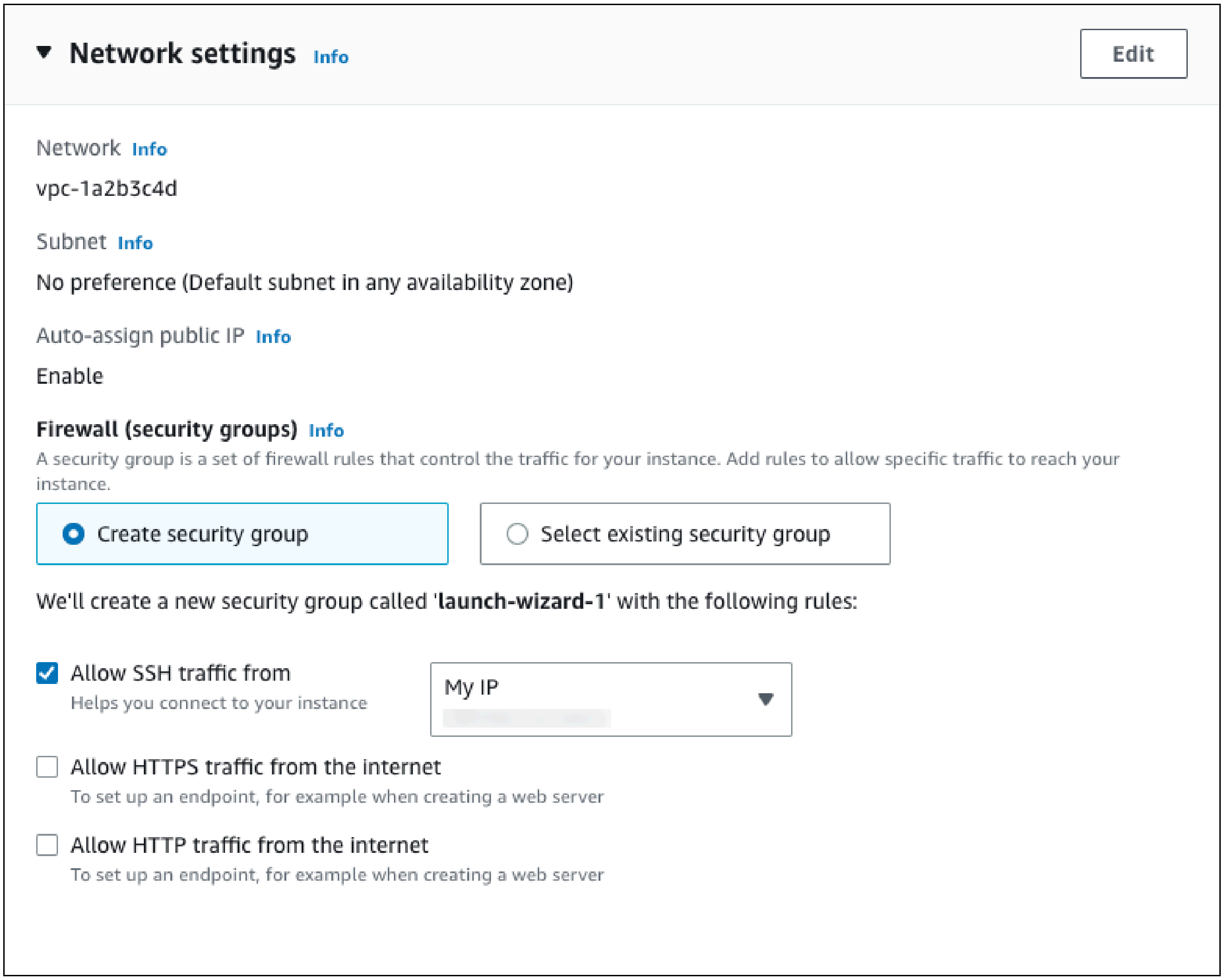
You can choose **My IP** if the displayed IP address is correct for SSH connections. Otherwise, you can determine the IP address to use to connect to EC2 instances in your

VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com. An example of an IP address is 192.0.2.1/32.

In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, make sure to determine the range of IP addresses used by client computers.

> ⚠ **Warning**
>
> If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public EC2 instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your EC2 instances using SSH.

The following image shows an example of the **Network settings** section.



▼ Network settings Info

Network Info

vpc-1a2b3c4d

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

● Create security group ○ Select existing security group

We'll create a new security group called '**launch-wizard-1**' with the following rules:

☑ Allow SSH traffic from — My IP ▼
Helps you connect to your instance

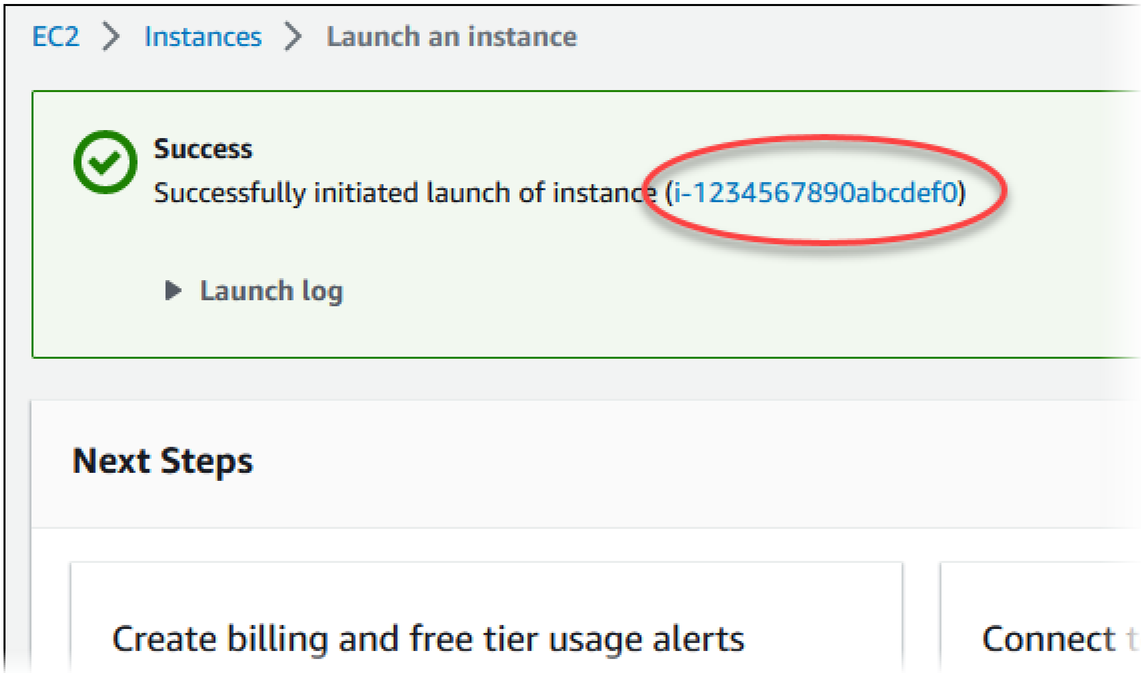
☐ Allow HTTPS traffic from the internet
To set up an endpoint, for example when creating a web server

☐ Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server

f. Leave the default values for the remaining sections.

g. Review a summary of your EC2 instance configuration in the **Summary** panel, and when you're ready, choose **Launch instance**.

5. On the **Launch Status** page, note the identifier for your new EC2 instance, for example: `i-1234567890abcdef0`.


EC2 > Instances > Launch an instance

Success

Successfully initiated launch of instance (i-1234567890abcdef0)

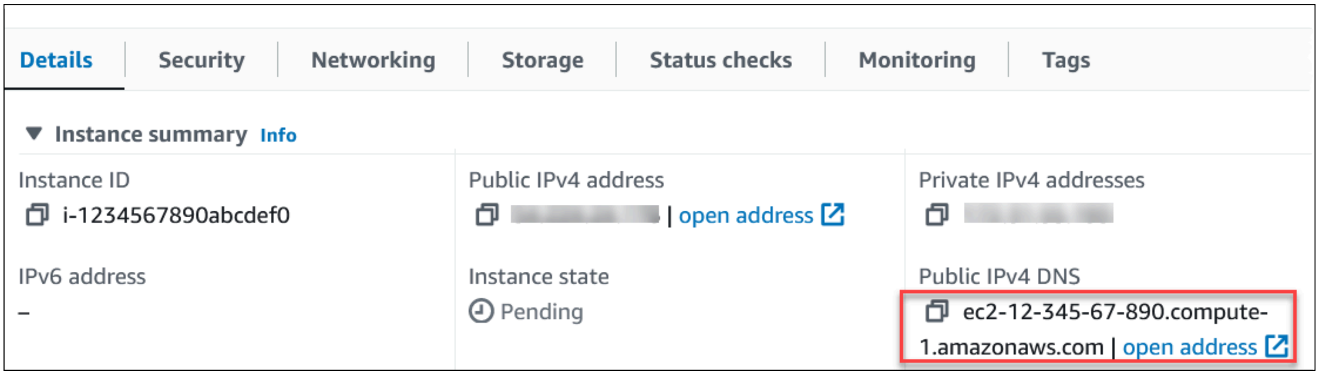
Next Steps

Create billing and free tier usage alerts
Connect t

6. Choose the EC2 instance identifier to open the list of EC2 instances, and then select your EC2 instance.

7. In the **Details** tab, note the following values, which you need when you connect using SSH:

   a. In **Instance summary**, note the value for **Public IPv4 DNS**.


Details | Security | Networking | Storage | Status checks | Monitoring | Tags
▼ Instance summary Info

Instance ID
i-1234567890abcdef0

Public IPv4 address
[redacted] | open address

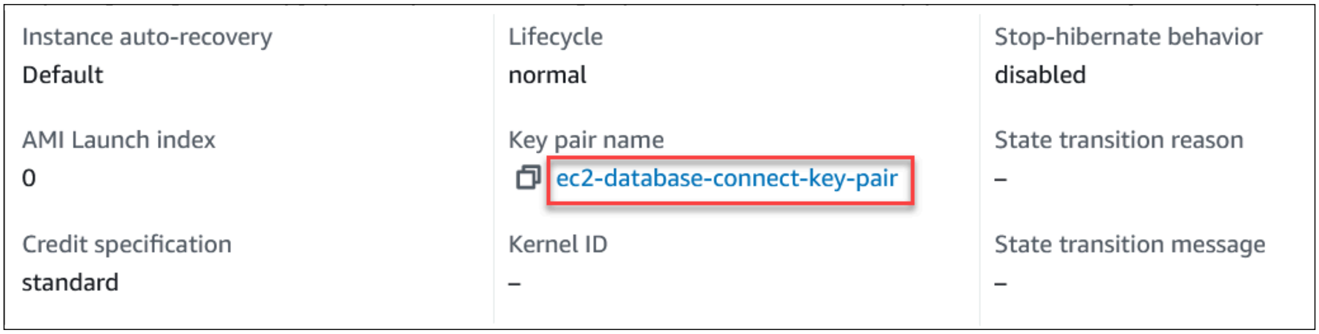
Private IPv4 addresses
[redacted]

IPv6 address
–

Instance state
Pending

Public IPv4 DNS
ec2-12-345-67-890.compute-1.amazonaws.com | open address

   b. In **Instance details**, note the value for **Key pair name**.



Instance auto-recovery
Default

Lifecycle
normal

Stop-hibernate behavior
disabled

AMI Launch index
0

Key pair name
ec2-database-connect-key-pair

State transition reason
–

Credit specification
standard

Kernel ID
–

State transition message
–

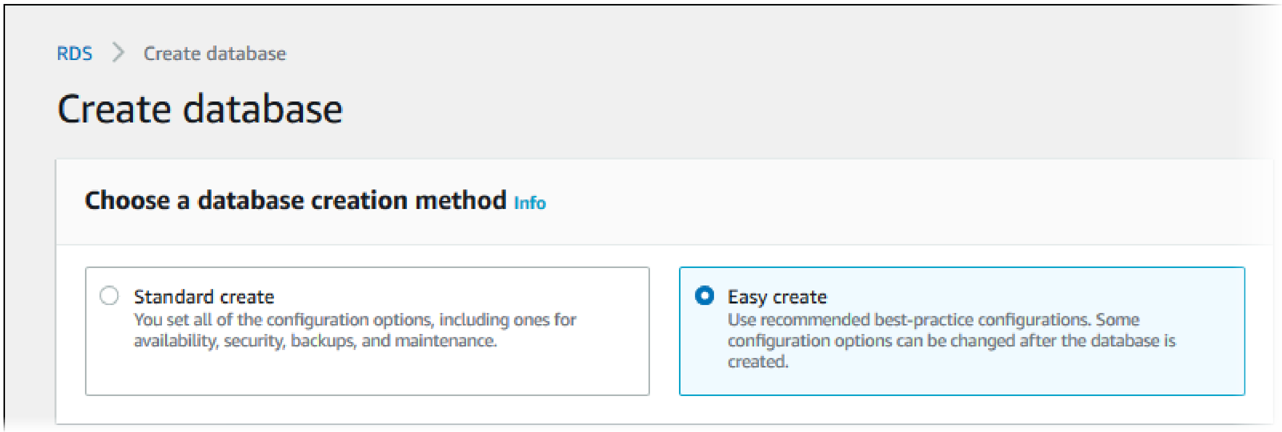
8. Wait until the **Instance state** for your EC2 instance has a status of **Running** before continuing.

Step 2: Create an Aurora PostgreSQL DB cluster

In this example, you use **Easy create** to create an Aurora PostgreSQL DB cluster with a db.t4g.large DB instance class.

To create an Aurora PostgreSQL DB cluster with Easy create

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the Amazon RDS console, choose the AWS Region in which you want to create the DB cluster.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database**, and make sure that **Easy create** is chosen.



RDS > Create database

Create database

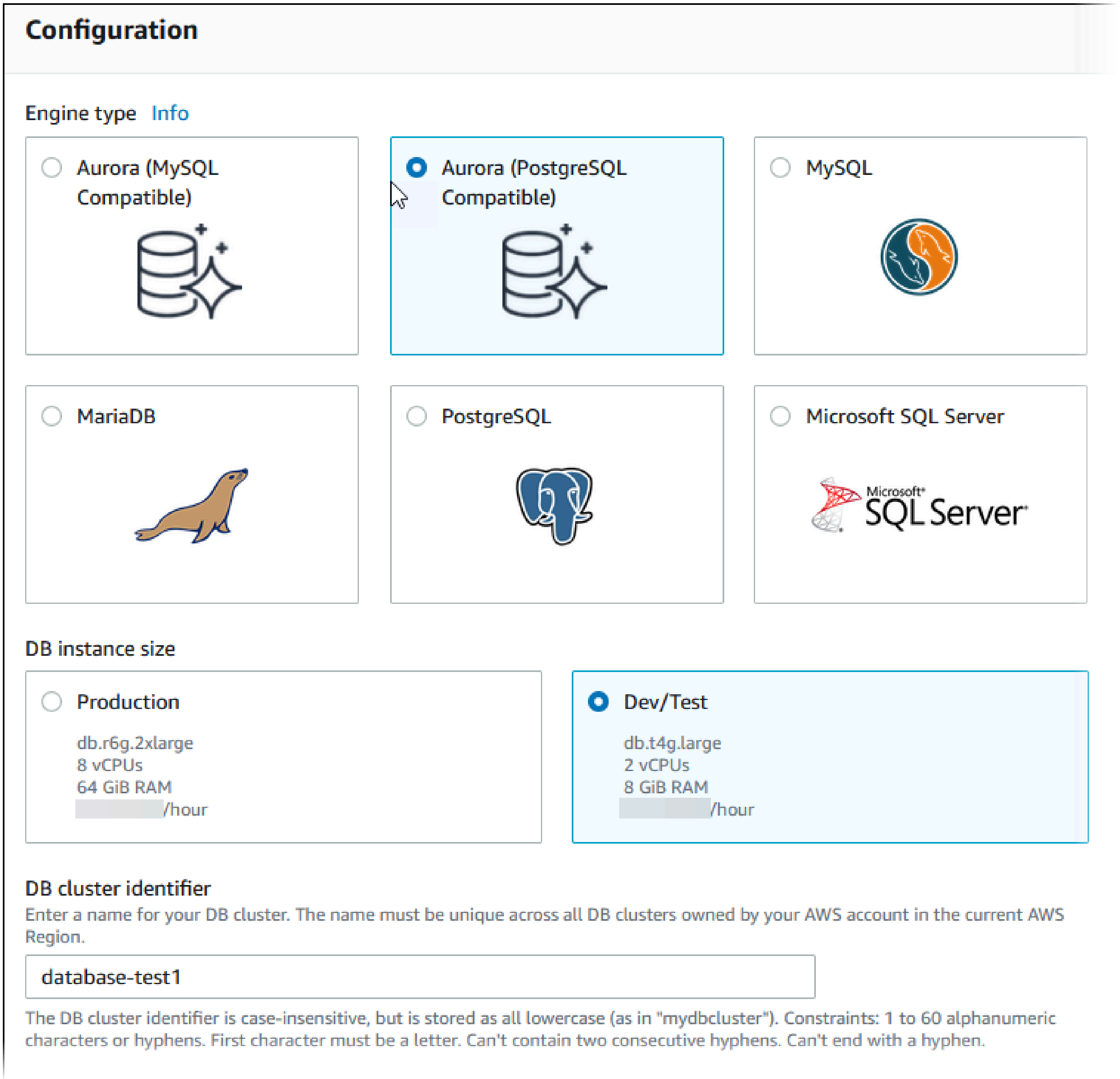
Choose a database creation method Info

○ Standard create
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

● Easy create
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

5. In **Configuration**, choose **Aurora (PostgreSQL Compatible)** for **Engine type**.
6. For **DB instance size**, choose **Dev/Test**.
7. For **DB cluster identifier**, enter **`database-test1`**.

   The **Create database** page should look similar to the following image.



Configuration

Engine type Info

○ Aurora (MySQL Compatible)

● Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Microsoft SQL Server

DB instance size

○ Production
db.r6g.2xlarge
8 vCPUs
64 GiB RAM
[redacted]/hour

● Dev/Test
db.t4g.large
2 vCPUs
8 GiB RAM
[redacted]/hour

DB cluster identifier

Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

database-test1

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

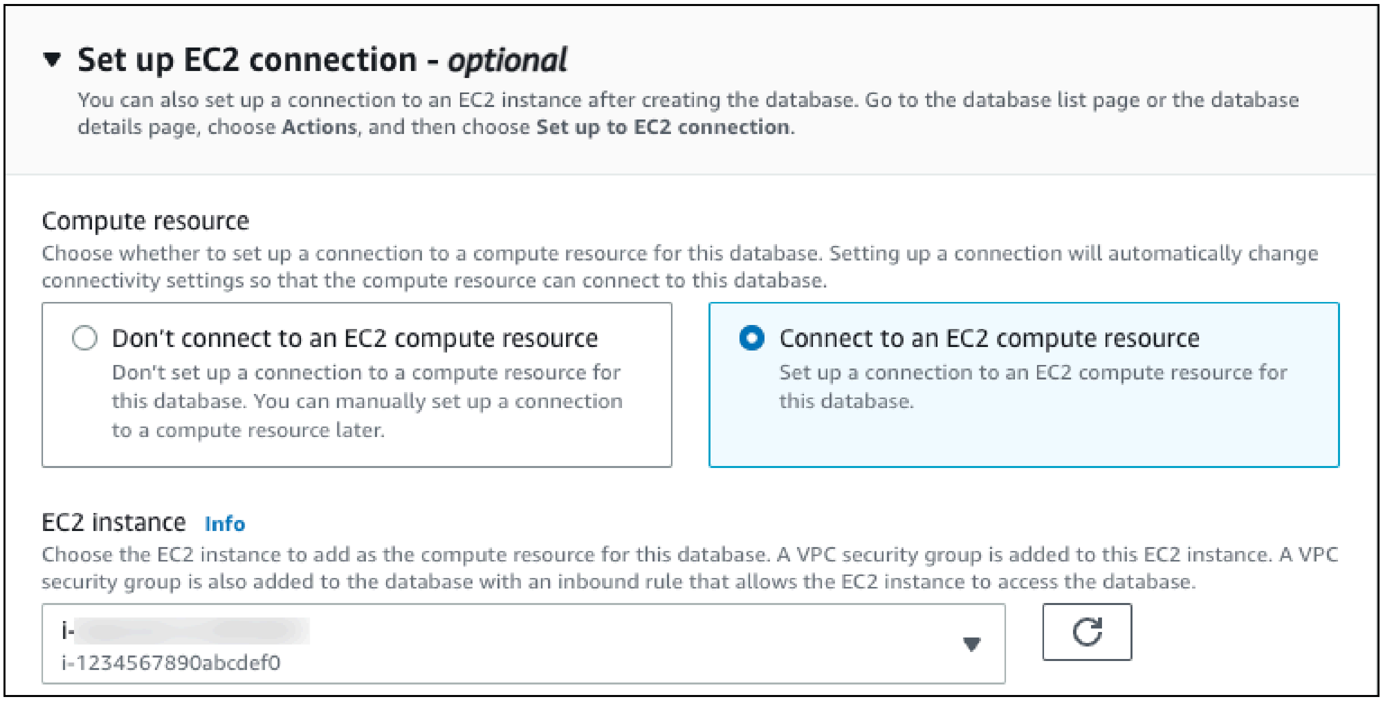
8. For **Master username**, enter a name for the user, or keep the default name (`postgres`).

9. To use an automatically generated master password for the DB cluster, select **Auto generate a password**.

   To enter your master password, make sure **Auto generate a password** is cleared, and then enter the same password in **Master password** and **Confirm password**.

10. To set up a connection with the EC2 instance you created previously, open **Set up EC2 connection - *optional***.

Select **Connect to an EC2 compute resource**. Choose the EC2 instance you created previously.



▼ **Set up EC2 connection - *optional***

You can also set up a connection to an EC2 instance after creating the database. Go to the database list page or the database details page, choose **Actions**, and then choose **Set up to EC2 connection**.

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

○ Don’t connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

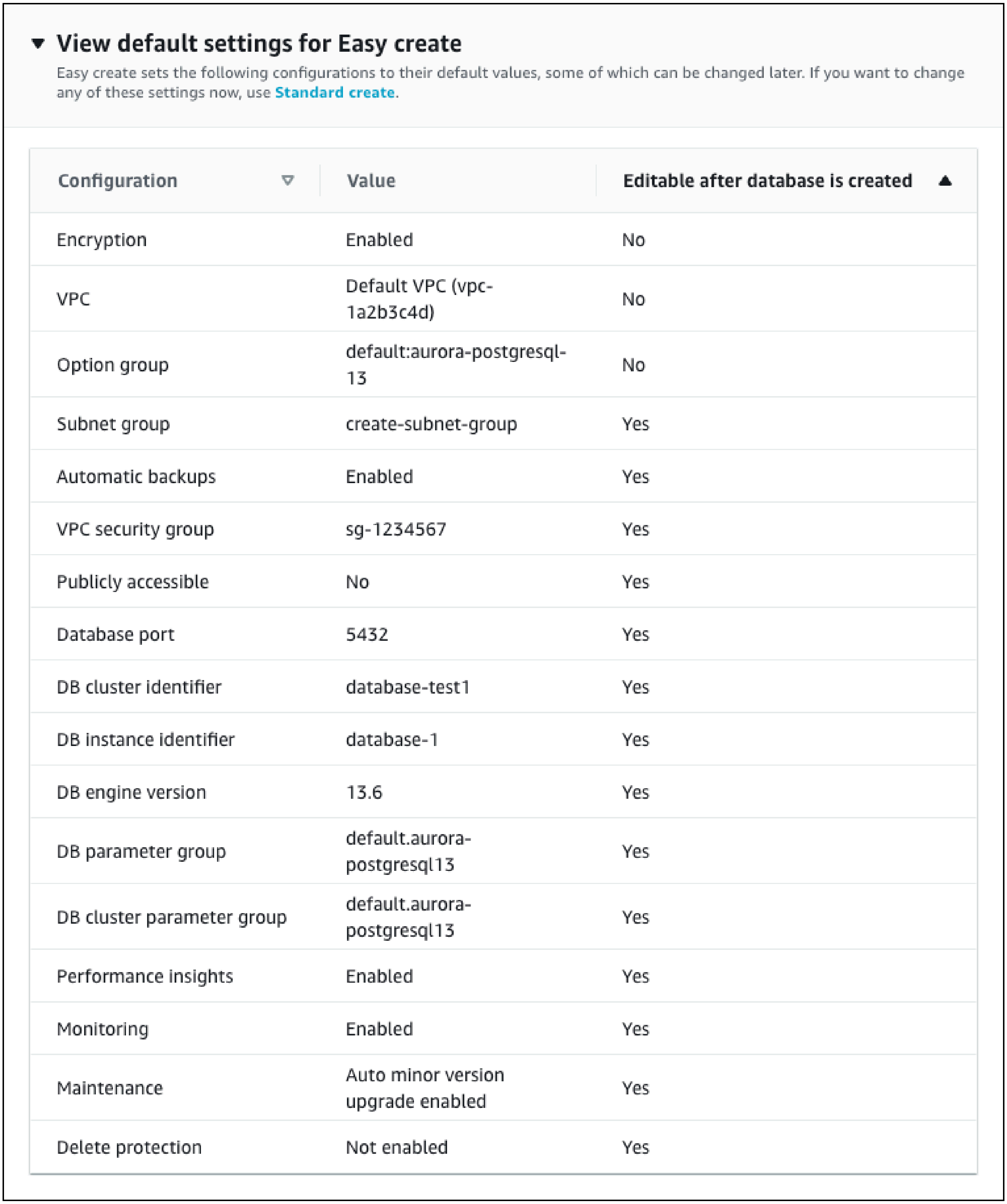
● Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

EC2 instance Info

Choose the EC2 instance to add as the compute resource for this database. A VPC security group is added to this EC2 instance. A VPC security group is also added to the database with an inbound rule that allows the EC2 instance to access the database.

i-[redacted]
i-1234567890abcdef0

11. Open **View default settings for Easy create**.



▼ View default settings for Easy create

Easy create sets the following configurations to their default values, some of which can be changed later. If you want to change any of these settings now, use **Standard create**.

Configuration ▽	Value	Editable after database is created ▲
Encryption	Enabled	No
VPC	Default VPC (vpc-1a2b3c4d)	No
Option group	default:aurora-postgresql-13	No
Subnet group	create-subnet-group	Yes
Automatic backups	Enabled	Yes
VPC security group	sg-1234567	Yes
Publicly accessible	No	Yes
Database port	5432	Yes
DB cluster identifier	database-test1	Yes
DB instance identifier	database-1	Yes
DB engine version	13.6	Yes
DB parameter group	default.aurora-postgresql13	Yes
DB cluster parameter group	default.aurora-postgresql13	Yes
Performance insights	Enabled	Yes
Monitoring	Enabled	Yes
Maintenance	Auto minor version upgrade enabled	Yes
Delete protection	Not enabled	Yes

You can examine the default settings used with **Easy create**. The **Editable after database is created** column shows which options you can change after you create the database.

- If a setting has **No** in that column, and you want a different setting, you can use **Standard create** to create the DB cluster.
- If a setting has **Yes** in that column, and you want a different setting, you can either use **Standard create** to create the DB cluster, or modify the DB cluster after you create it to change the setting.

12. Choose **Create database**.

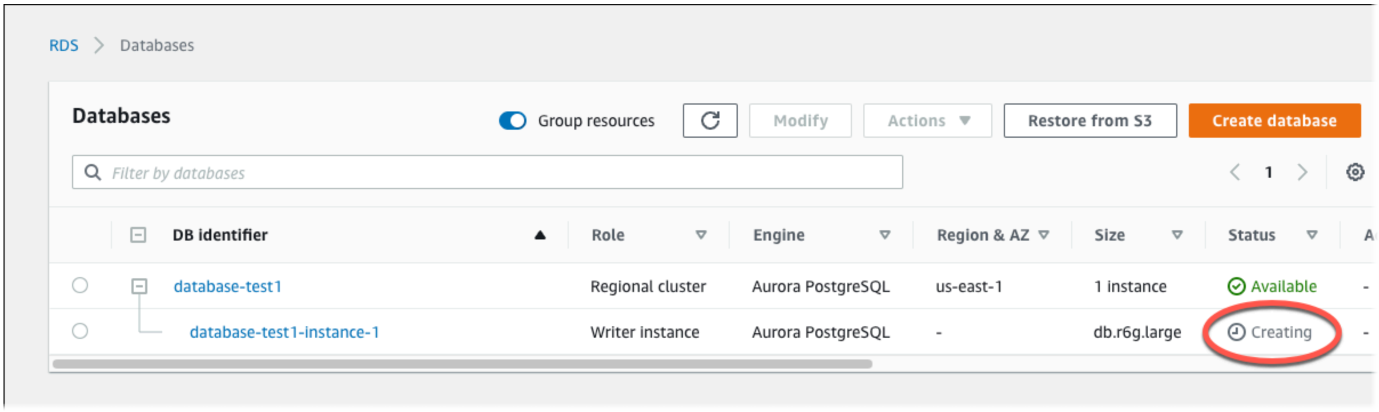
    To view the master username and password for the DB cluster, choose **View credential details**.

    You can use the username and password that appears to connect to the DB cluster as the master user.

    > **Important**
    >
    > You can't view the master user password again. If you don't record it, you might have to change it.
    > If you need to change the master user password after the DB cluster is available, you can modify the DB cluster to do so. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

13. In the **Databases** list, choose the name of the new Aurora PostgreSQL DB cluster to show its details.

    The writer instance has a status of **Creating** until the DB cluster is ready to use.



		DB identifier ▲	Role ▽	Engine ▽	Region & AZ ▽	Size ▽	Status ▽	A
○	⊟	database-test1	Regional cluster	Aurora PostgreSQL	us-east-1	1 instance	✓ Available	-
○		database-test1-instance-1	Writer instance	Aurora PostgreSQL	-	db.r6g.large	Creating	-

When the status of the writer instance changes to **Available**, you can connect to the DB cluster. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new DB cluster is available.

(Optional) Create VPC, EC2 instance, and Aurora PostgreSQL cluster using AWS CloudFormation

Instead of using the console to create your VPC, EC2 instance, and Aurora PostgreSQL DB cluster, you can use AWS CloudFormation to provision AWS resources by treating infrastructure as code. To help you organize your AWS resources into smaller and more manageable units, you can use the AWS CloudFormation nested stack functionality. For more information, see Creating a stack on the AWS CloudFormation console and Working with nested stacks.

⚠ **Important**

AWS CloudFormation is free, but the resources that CloudFormation creates are live. You incur the standard usage fees for these resources until you terminate them. The total charges will be minimal. For information about how you might minimize any charges, go to AWS Free Tier.

To create your resources using the AWS CloudFormation console, complete the following steps:

- Step 1: Download the CloudFormation template
- Step 2: Configure your resources using CloudFormation

Download the CloudFormation template

A CloudFormation template is a JSON or YAML text file that contains the configuration information about the resources you want to create in the stack. This template also creates a VPC and a bastion host for you along with the Aurora cluster.

To download the template file, open the following link, Aurora PostgreSQL CloudFormation template.

In the Github page, click the *Download raw file* button to save the template YAML file.

Configure your resources using CloudFormation

> **Note**
>
> Before starting this process, make sure you have a Key pair for an EC2 instance in your AWS account. For more information, see Amazon EC2 key pairs and Linux instances.

When you use the AWS CloudFormation template, you must select the correct parameters to make sure your resources are created properly. Follow the steps below:

1. Sign in to the AWS Management Console and open the AWS CloudFormation console at https://console.aws.amazon.com/cloudformation.
2. Choose **Create Stack**.
3. In the Specify template section, select **Upload a template file from your computer**, and then choose **Next**.
4. In the **Specify stack details** page, set the following parameters:
   a. Set **Stack name** to **AurPostgreSQLTestStack**.
   b. Under **Parameters**, set **Availability Zones** by selecting two availability zones.
   c. Under **Linux Bastion Host configuration**, for **Key Name**, select a key pair to login to your EC2 instance.
   d. In **Linux Bastion Host configuration** settings, set the **Permitted IP range** to your IP address. To connect to EC2 instances in your VPC using Secure Shell (SSH), determine your public IP address using the service at https://checkip.amazonaws.com. An example of an IP address is 192.0.2.1/32.

   > **Warning**
   >
   > If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public EC2 instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your EC2 instances using SSH.

   e. Under **Database General configuration**, set **Database instance class** to **db.t4g.large**.
   f. Set **Database name** to **`database-test1`**.

g. For **Database master username**, enter a name for the master user.

h. Set **Manage DB master user password with Secrets Manager** to `false` for this tutorial.

i. For **Database password**, set a password of your choice. Remember this password for further steps in the tutorial.

j. Set **Multi-AZ deployment** to `false`.

k. Leave all other settings as the default values. Click **Next** to continue.

5. In the **Configure stack options** page, leave all the default options. Click **Next** to continue.

6. In the **Review stack** page, select **Submit** after checking the database and Linux bastion host options.

After the stack creation process completes, view the stacks with names *BastionStack* and *APGNS* to note the information you need to connect to the database. For more information, see Viewing AWS CloudFormation stack data and resources on the AWS Management Console.

Step 3: Connect to an Aurora PostgreSQL DB cluster

You can use any standard PostgreSQL client application to connect to the DB cluster. In this example, you connect to the Aurora PostgreSQL DB cluster using the psql command line client.

To connect to the Aurora PostgreSQL DB cluster

1. Find the endpoint (DNS name) and port number of the writer instance for your DB cluster.

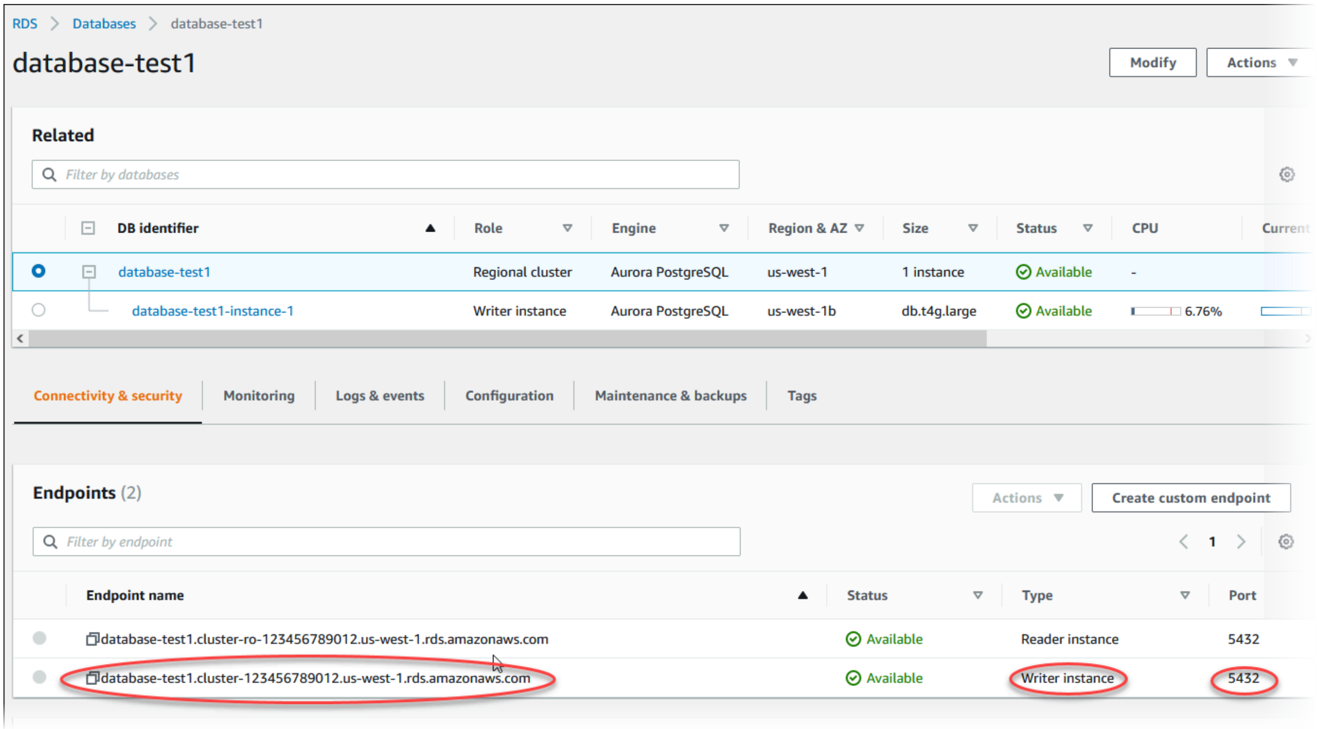
   a. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   b. In the upper-right corner of the Amazon RDS console, choose the AWS Region for the DB cluster.

   c. In the navigation pane, choose **Databases**.

   d. Choose the Aurora PostgreSQL DB cluster name to display its details.

   e. On the **Connectivity & security** tab, copy the endpoint of the writer instance. Also, note the port number. You need both the endpoint and the port number to connect to the DB cluster.



database-test1

DB identifier	Role	Engine	Region & AZ	Size	Status	CPU	Current
database-test1	Regional cluster	Aurora PostgreSQL	us-west-1	1 instance	✓ Available	-	
database-test1-instance-1	Writer instance	Aurora PostgreSQL	us-west-1b	db.t4g.large	✓ Available	6.76%	

Endpoints (2)

Endpoint name	Status	Type	Port
database-test1.cluster-ro-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Reader instance	5432
database-test1.cluster-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Writer instance	5432

2. Connect to the EC2 instance that you created earlier by following the steps in Connect to your Linux instance in the *Amazon EC2 User Guide*.

   We recommend that you connect to your EC2 instance using SSH. If the SSH client utility is installed on Windows, Linux, or Mac, you can connect to the instance using the following command format:

   ```
   ssh -i location_of_pem_file ec2-user@ec2-instance-public-dns-name
   ```

   For example, assume that `ec2-database-connect-key-pair.pem` is stored in `/dir1` on Linux, and the public IPv4 DNS for your EC2 instance is `ec2-12-345-678-90.compute-1.amazonaws.com`. Your SSH command would look as follows:

   ```
   ssh -i /dir1/ec2-database-connect-key-pair.pem ec2-
   user@ec2-12-345-678-90.compute-1.amazonaws.com
   ```

3. Get the latest bug fixes and security updates by updating the software on your EC2 instance. To do so, use the following command.

> **Note**
>
> The `-y` option installs the updates without asking for confirmation. To examine updates before installing, omit this option.

```
sudo dnf update -y
```

4. Install the psql command line client from PostgreSQL on Amazon Linux 2023, using the following command:

```
sudo dnf install postgresql15
```

5. Connect to the Aurora PostgreSQL DB cluster. For example, enter the following command. This action lets you connect to the Aurora PostgreSQL DB cluster using the psql client.

   Substitute the endpoint of the writer instance for *`endpoint`*, substitute the database name `--dbname` that you want to connect to for *`postgres`*, and substitute the master username that you used for *`postgres`*. Provide the master password that you used when prompted for a password.

```
psql --host=endpoint --port=5432 --dbname=postgres --username=postgres
```

   After you enter the password for the user, you should see output similar to the following.

```
psql (14.3, server 14.6)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256,
 compression: off)
Type "help" for help.

postgres=>
```

   For more information about connecting to an Aurora PostgreSQL DB cluster, see Connecting to an Amazon Aurora PostgreSQL DB cluster. If you can't connect to your DB cluster, see Can't connect to Amazon RDS DB instance.

   For security, it is a best practice to use encrypted connections. Only use an unencrypted PostgreSQL connection when the client and server are in the same VPC and the network is trusted. For information about using encrypted connections, see Securing Aurora PostgreSQL data with SSL/TLS.

6. Run SQL commands.

   For example, the following SQL command shows the current date and time:

   ```
   SELECT CURRENT_TIMESTAMP;
   ```

Step 4: Delete the EC2 instance and DB cluster

After you connect to and explore the sample EC2 instance and DB cluster that you created, delete them so you're no longer charged for them.

If you used AWS CloudFormation to create resources, skip this step and go to the next step.

To delete the EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Instances**.
3. Select the EC2 instance, and choose **Instance state, Terminate instance**.
4. Choose **Terminate** when prompted for confirmation.

For more information about deleting an EC2 instance, see Terminate your instance in the *Amazon EC2 User Guide*.

To delete a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** and then choose the DB instance associated with the DB cluster.
3. For **Actions**, choose **Delete**.
4. Choose **Delete**.

   After all of the DB instances associated with a DB cluster are deleted, the DB cluster is deleted automatically.

(Optional) Delete the EC2 instance and DB cluster created with CloudFormation

If you used AWS CloudFormation to create resources, delete the CloudFormation stack after you connect to and explore the sample EC2 instance and DB cluster, so you're no longer charged for them.

To delete the CloudFormation resources

1. Open the AWS CloudFormation console.
2. On the **Stacks** page in the CloudFormation console, select the root stack (the stack without the name VPCStack, BastionStack or APGNS).
3. Choose **Delete**.
4. Select **Delete stack** when prompted for confirmation.

For more information about deleting a stack in CloudFormation, see Deleting a stack on the AWS CloudFormation console in the *AWS CloudFormation User Guide*.

(Optional) Connect your DB cluster to a Lambda function

You can also connect your Aurora PostgreSQL DB cluster to a Lambda serverless compute resource. Lambda functions allow you to run code without provisioning or managing infrastructure. A Lambda function also allows you to automatically respond to code execution requests at any scale, from a dozen events a day to hundreds of per second. For more information, see Automatically connecting a Lambda function and an Aurora DB cluster.

Tutorial: Create a web server and an Amazon Aurora DB cluster

This tutorial shows you how to install an Apache web server with PHP and create a MariaDB, MySQL, or PostgreSQL database. The web server runs on an Amazon EC2 instance using Amazon Linux 2023, and you can choose between an Aurora MySQL or Aurora PostgreSQL DB cluster. Both the Amazon EC2 instance and the DB cluster run in a virtual private cloud (VPC) based on the Amazon VPC service.

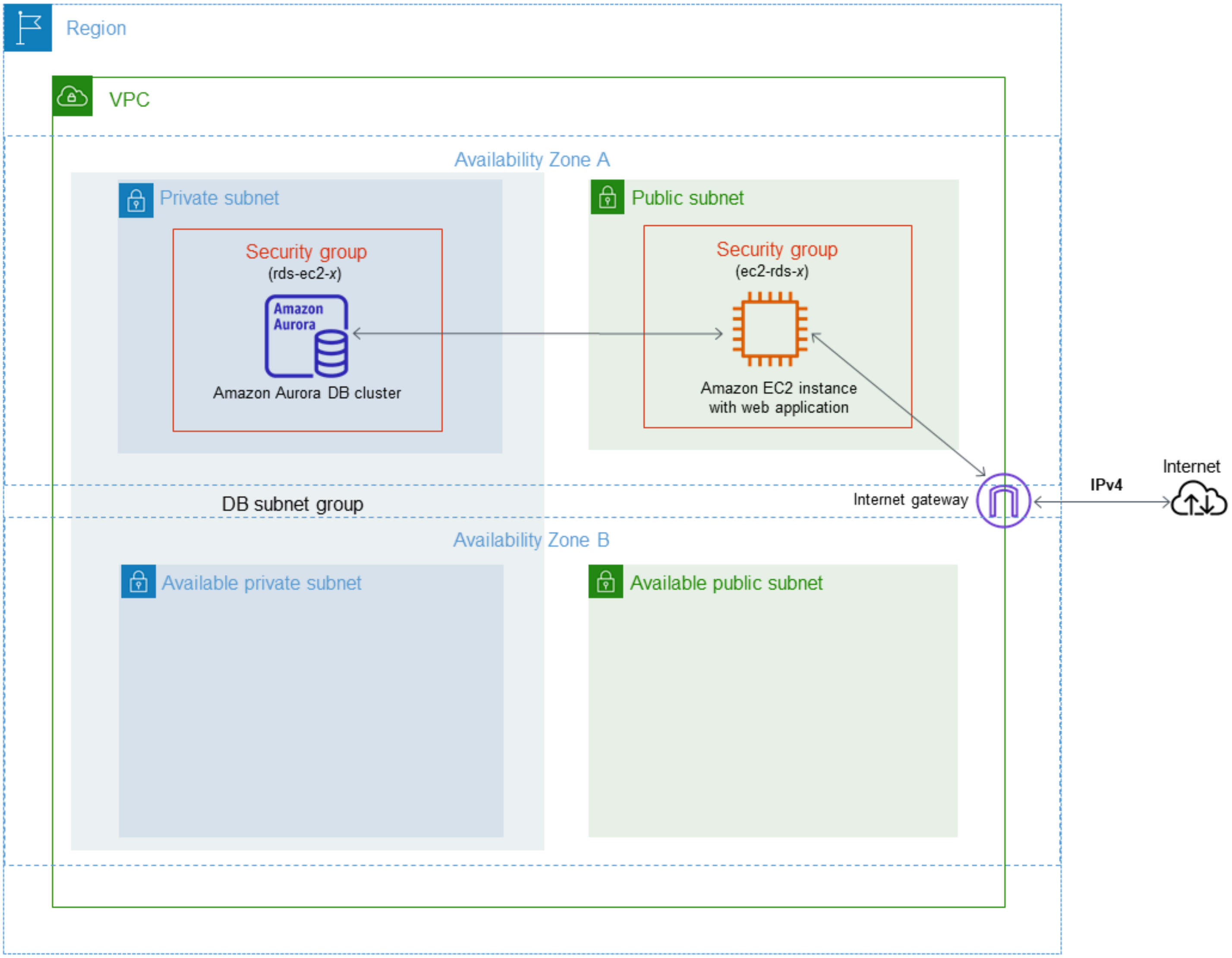
> ⚠ **Important**
>
> There's no charge for creating an AWS account. However, by completing this tutorial, you might incur costs for the AWS resources you use. You can delete these resources after you complete the tutorial if they are no longer needed.

> ⓘ **Note**
>
> This tutorial works with Amazon Linux 2023 and might not work for other versions of Linux.

In the tutorial that follows, you create an EC2 instance that uses the default VPC, subnets, and security group for your AWS account. This tutorial shows you how to create the DB cluster and automatically set up connectivity with the EC2 instance that you created. The tutorial then shows you how to install the web server on the EC2 instance. You connect your web server to your DB cluster in the VPC using the DB cluster writer endpoint.

1. Launch an EC2 instance
2. Create an Amazon Aurora DB cluster
3. Install a web server on your EC2 instance

The following diagram shows the configuration when the tutorial is complete.



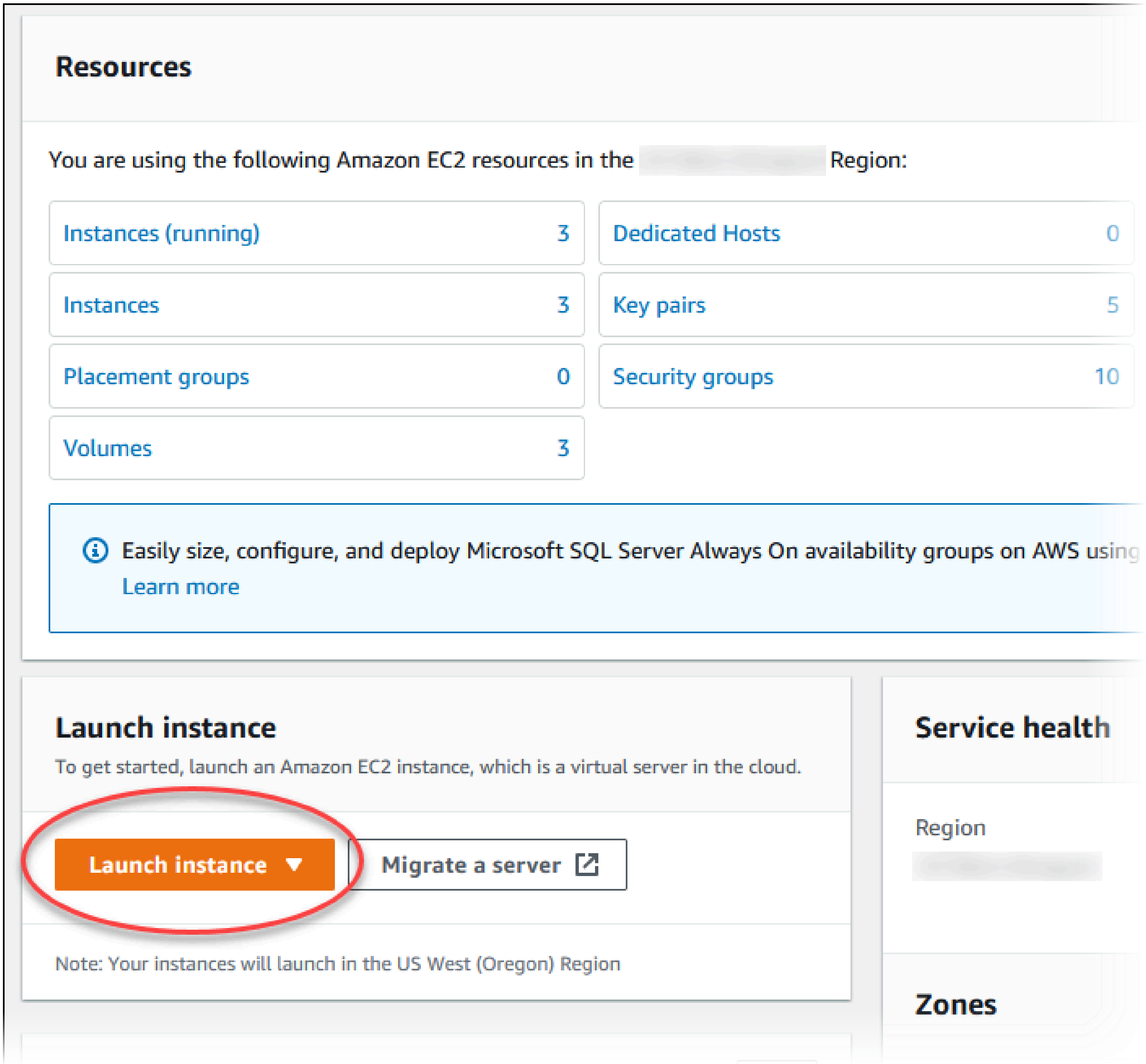
> **Note**
>
> After you complete the tutorial, there is a public and private subnet in each Availability Zone in your VPC. This tutorial uses the default VPC for your AWS account and automatically sets up connectivity between your EC2 instance and DB cluster. If you would rather configure a new VPC for this scenario instead, complete the tasks in Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

Launch an EC2 instance

Create an Amazon EC2 instance in the public subnet of your VPC.

To launch an EC2 instance

1. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region where you want to create the EC2 instance.
3. Choose **EC2 Dashboard**, and then choose **Launch instance**, as shown following.



Resources

You are using the following Amazon EC2 resources in the [redacted] Region:

Instances (running)	3	Dedicated Hosts	0
Instances	3	Key pairs	5
Placement groups	0	Security groups	10
Volumes	3		

Easily size, configure, and deploy Microsoft SQL Server Always On availability groups on AWS using

Learn more

Launch instance

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

Note: Your instances will launch in the US West (Oregon) Region

Service health

Region

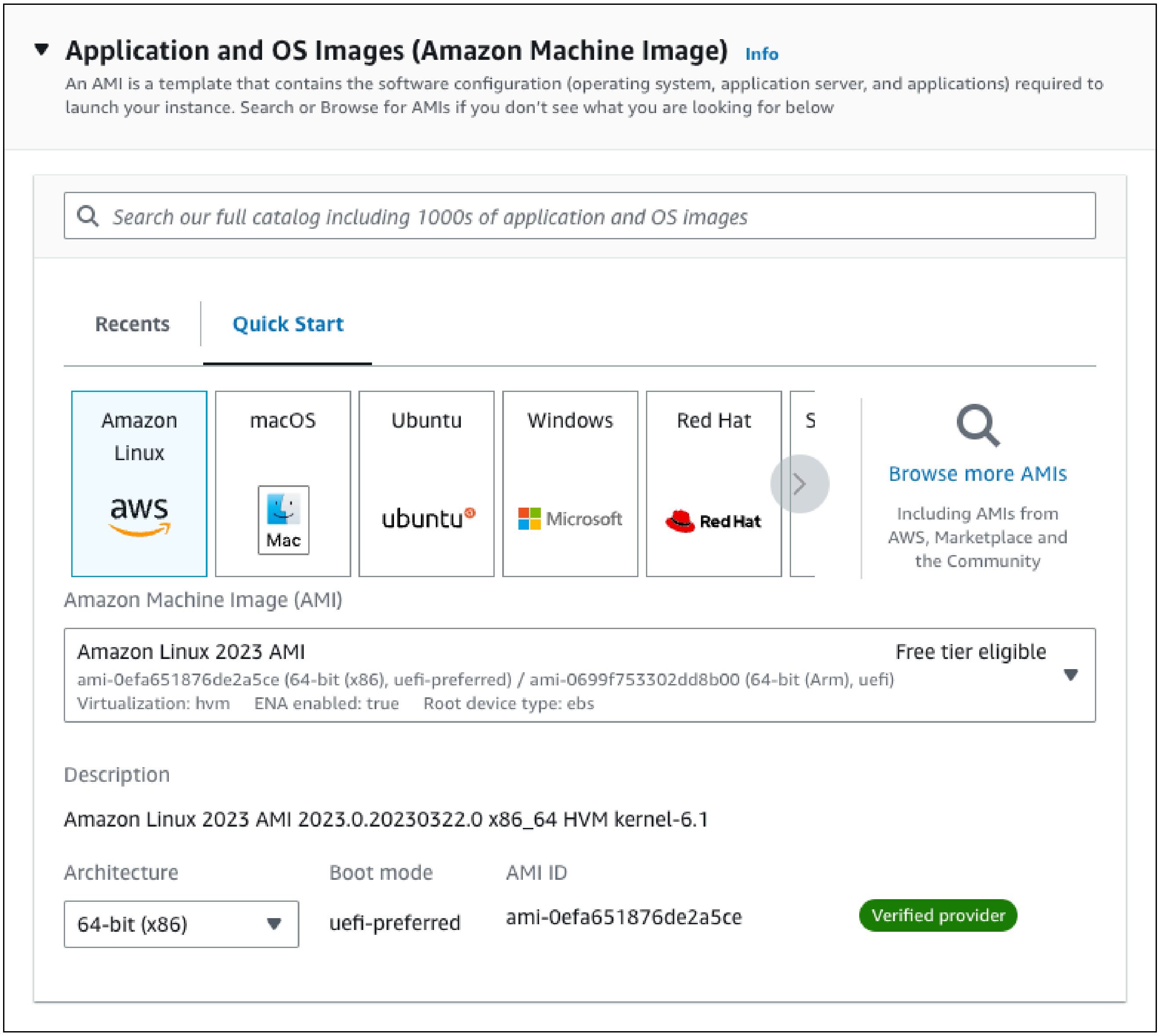
[redacted]

Zones

4. Choose the following settings in the **Launch an instance** page.

a. Under **Name and tags**, for **Name**, enter **`tutorial-ec2-instance-web-server`**.

b. Under **Application and OS Images (Amazon Machine Image)**, choose **Amazon Linux**, and then choose the **Amazon Linux 2023 AMI**. Keep the defaults for the other choices.



▼ Application and OS Images (Amazon Machine Image) Info

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below

Search our full catalog including 1000s of application and OS images

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI — Free tier eligible
ami-0efa651876de2a5ce (64-bit (x86), uefi-preferred) / ami-0699f753302dd8b00 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs

Description

Amazon Linux 2023 AMI 2023.0.20230322.0 x86_64 HVM kernel-6.1

Architecture: 64-bit (x86)

Boot mode: uefi-preferred

AMI ID: ami-0efa651876de2a5ce

c. Under **Instance type**, choose **t2.micro**.

d. Under **Key pair (login)**, choose a **Key pair name** to use an existing key pair. To create a new key pair for the Amazon EC2 instance, choose **Create new key pair** and then use the **Create key pair** window to create it.

For more information about creating a new key pair, see [Create a key pair]() in the *Amazon EC2 User Guide*.

e. Under **Network settings**, set these values and keep the other values as their defaults:

- For **Allow SSH traffic from**, choose the source of SSH connections to the EC2 instance.

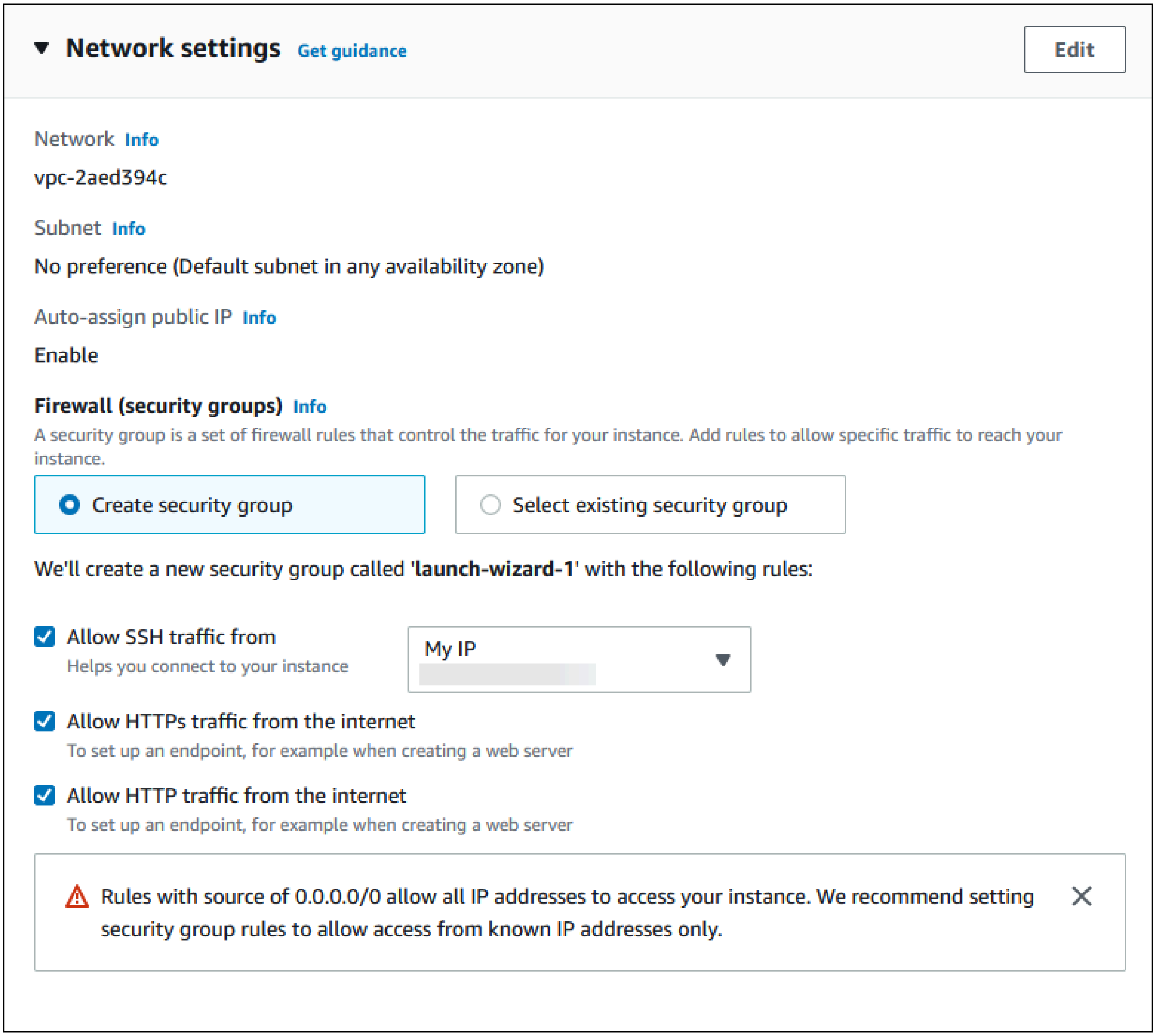
  You can choose **My IP** if the displayed IP address is correct for SSH connections.

  Otherwise, you can determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com. An example of an IP address is `203.0.113.25/32`.

  In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, make sure to determine the range of IP addresses used by client computers.

  > ⚠ **Warning**
  >
  > If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your instances using SSH.

- Turn on **Allow HTTPs traffic from the internet**.
- Turn on **Allow HTTP traffic from the internet**.



▼ Network settings Get guidance Edit

Network Info

vpc-2aed394c

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

● Create security group ○ Select existing security group

We'll create a new security group called '**launch-wizard-1**' with the following rules:

☑ Allow SSH traffic from My IP
Helps you connect to your instance

☑ Allow HTTPs traffic from the internet
To set up an endpoint, for example when creating a web server

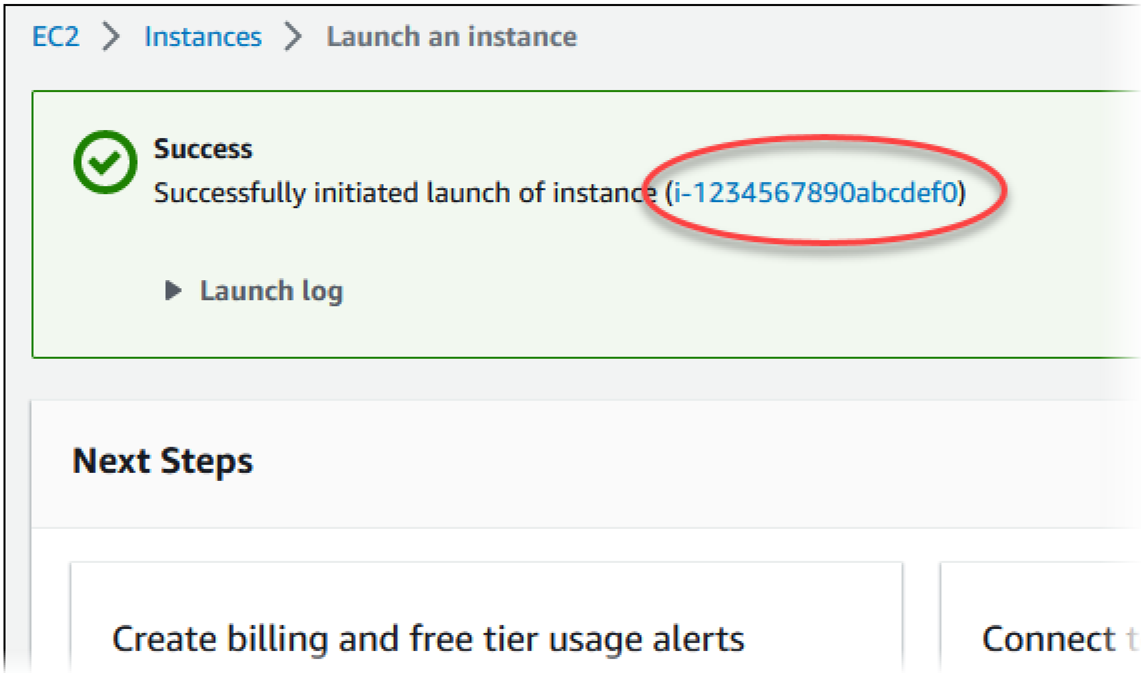
☑ Allow HTTP traffic from the internet
To set up an endpoint, for example when creating a web server

⚠ Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only. ✕

f. Leave the default values for the remaining sections.

g. Review a summary of your instance configuration in the **Summary** panel, and when you're ready, choose **Launch instance**.

5. On the **Launch Status** page, note the identifier for your new EC2 instance, for example: `i-1234567890abcdef0`.



Success

Successfully initiated launch of instance (i-1234567890abcdef0)

Next Steps

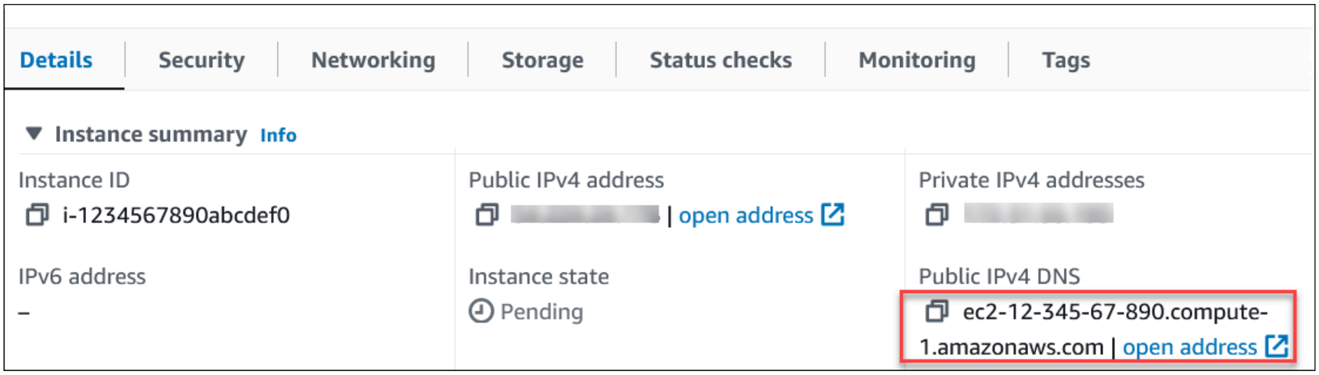
Create billing and free tier usage alerts

Connect t

6. Choose the EC2 instance identifier to open the list of EC2 instances, and then select your EC2 instance.

7. In the **Details** tab, note the following values, which you need when you connect using SSH:

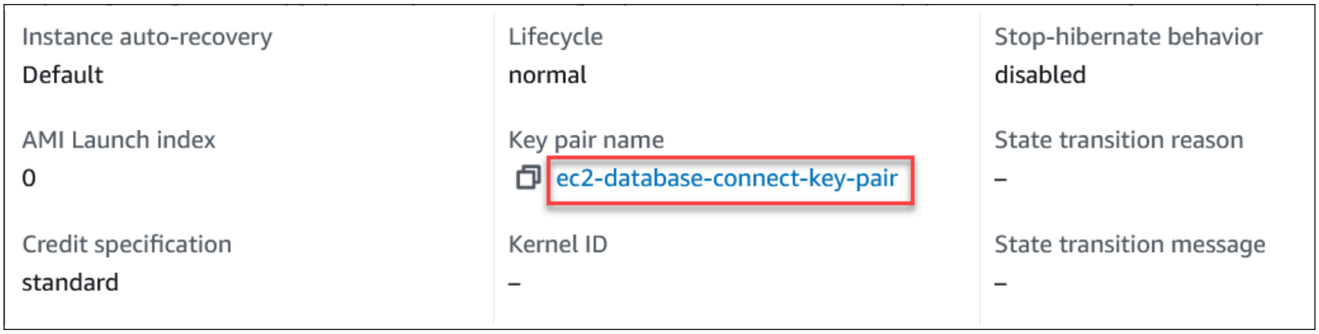
   a. In **Instance summary**, note the value for **Public IPv4 DNS**.



▼ Instance summary Info

Instance ID i-1234567890abcdef0	Public IPv4 address [redacted] | open address	Private IPv4 addresses [redacted]
IPv6 address –	Instance state Pending	Public IPv4 DNS ec2-12-345-67-890.compute-1.amazonaws.com | open address

   b. In **Instance details**, note the value for **Key pair name**.



Instance auto-recovery Default	Lifecycle normal	Stop-hibernate behavior disabled
AMI Launch index 0	Key pair name ec2-database-connect-key-pair	State transition reason –
Credit specification standard	Kernel ID –	State transition message –

8. Wait until **Instance state** for your instance is **Running** before continuing.
9. Complete Create an Amazon Aurora DB cluster.

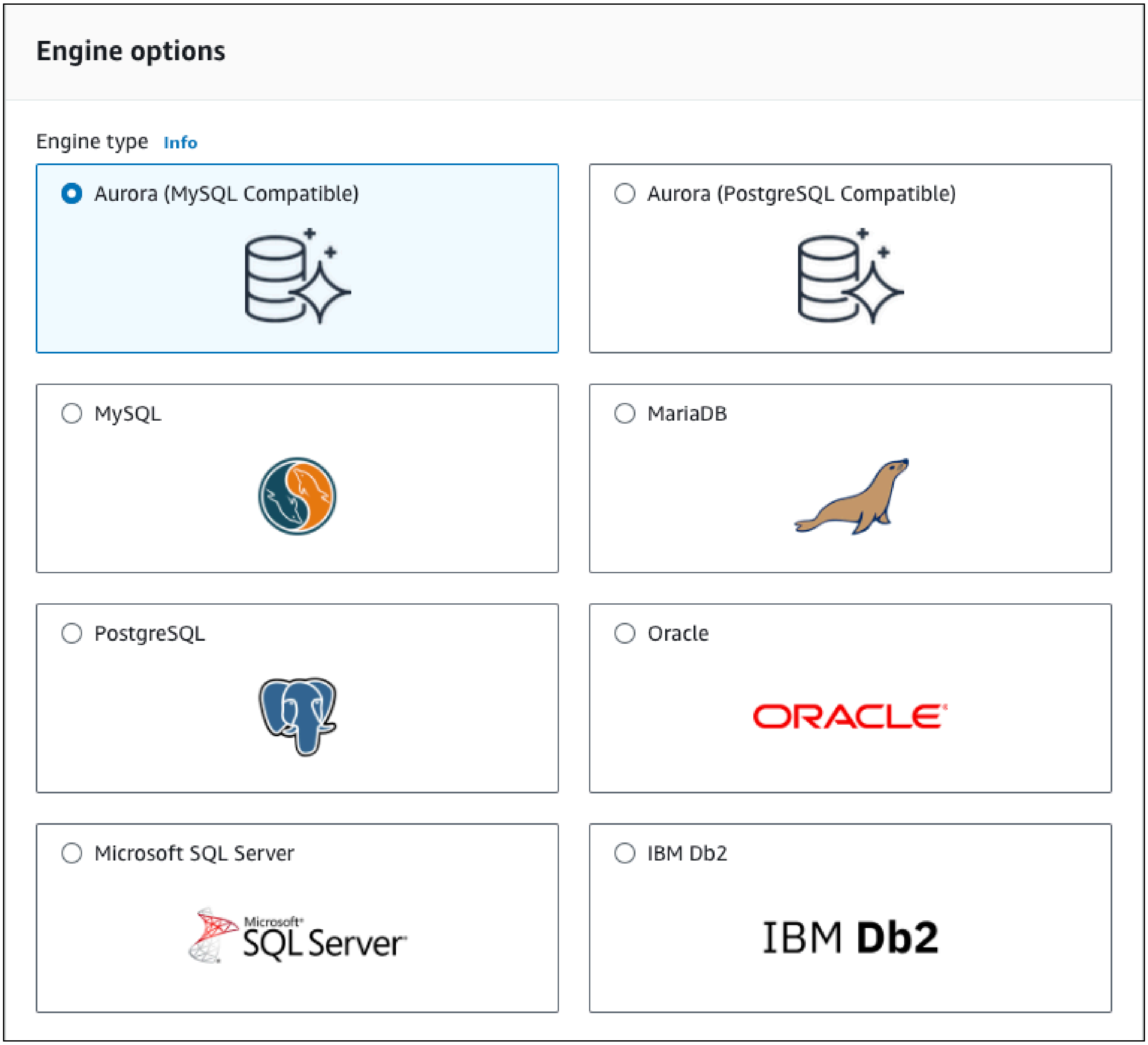
Create an Amazon Aurora DB cluster

Create an Amazon Aurora MySQL or Aurora PostgreSQL DB cluster that maintains the data used by a web application.

Aurora MySQL

To create an Aurora MySQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, make sure the AWS Region is the same as the one where you created your EC2 instance.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database**.
5. On the **Create database** page, choose **Standard create**.
6. For **Engine options**, choose **Aurora (MySQL Compatible)**.



Engine options

Engine type Info

● Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

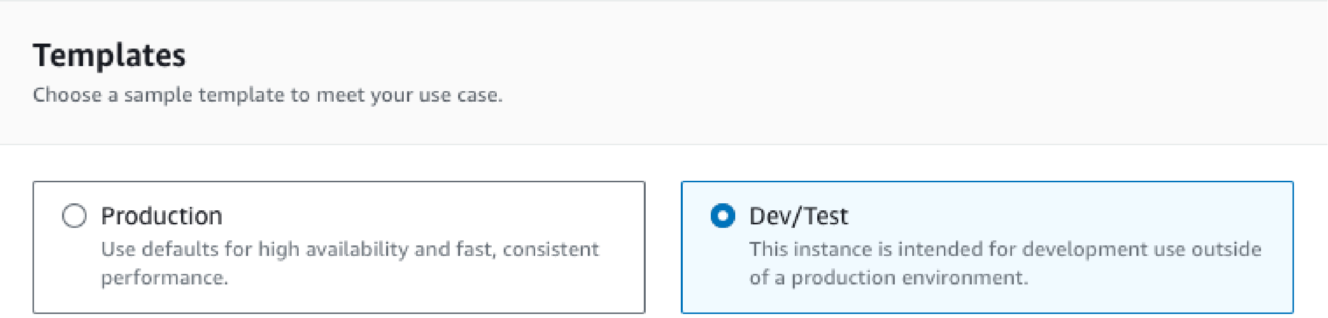
○ Oracle

○ Microsoft SQL Server

○ IBM Db2

Keep the default values for **Version** and the other engine options.

7. In the **Templates** section, choose **Dev/Test**.



Templates

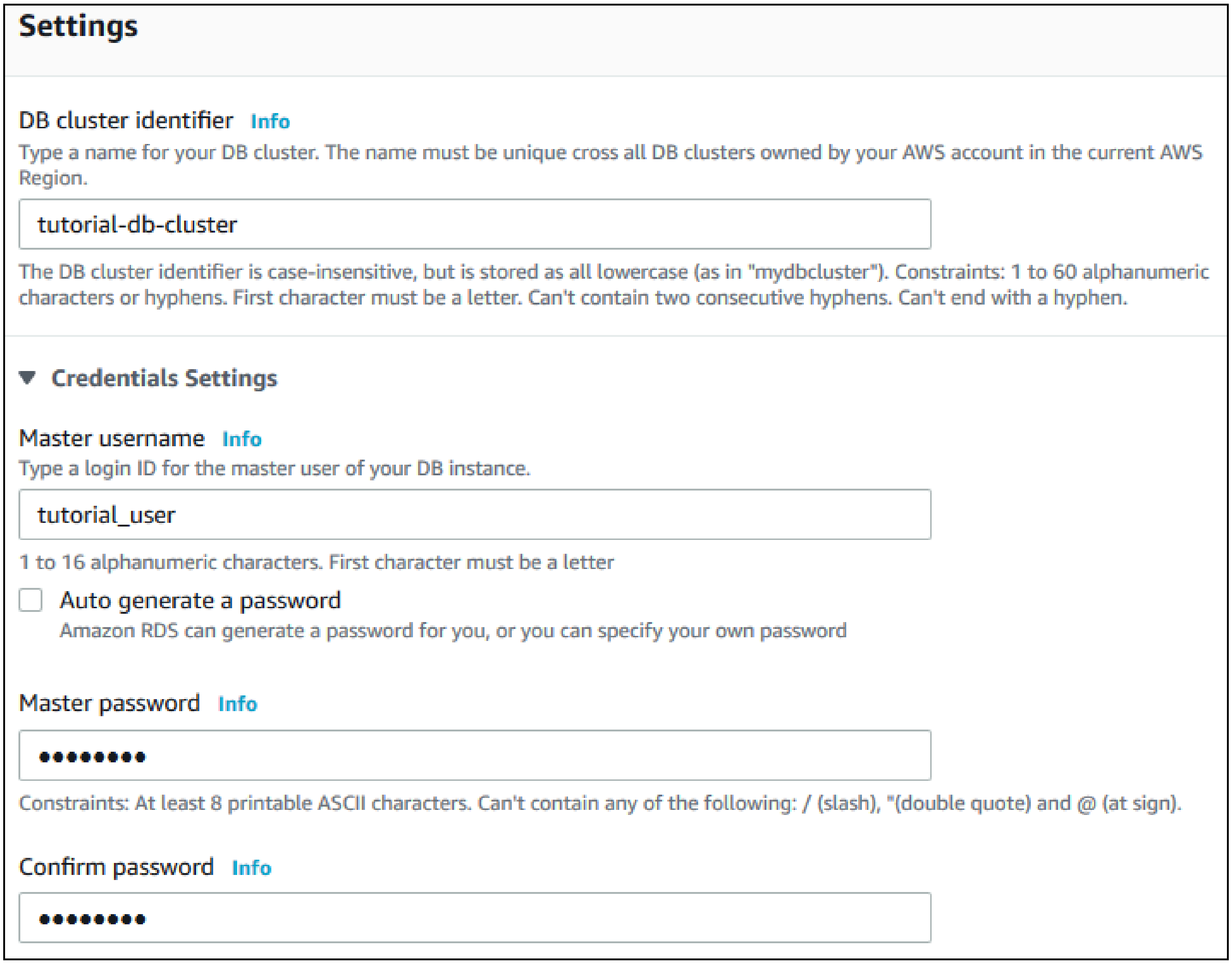
Choose a sample template to meet your use case.

○ Production
Use defaults for high availability and fast, consistent performance.

● Dev/Test
This instance is intended for development use outside of a production environment.

8. In the **Settings** section, set these values:

   - **DB cluster identifier** – Type **`tutorial-db-cluster`**.
   - **Master username** – Type **`tutorial_user`**.
   - **Auto generate a password** – Leave the option turned off.
   - **Master password** – Type a password.
   - **Confirm password** – Retype the password.



Settings

DB cluster identifier Info

Type a name for your DB cluster. The name must be unique cross all DB clusters owned by your AWS account in the current AWS Region.

tutorial-db-cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

tutorial_user

1 to 16 alphanumeric characters. First character must be a letter

□ Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password

Master password Info

••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), "(double quote) and @ (at sign).

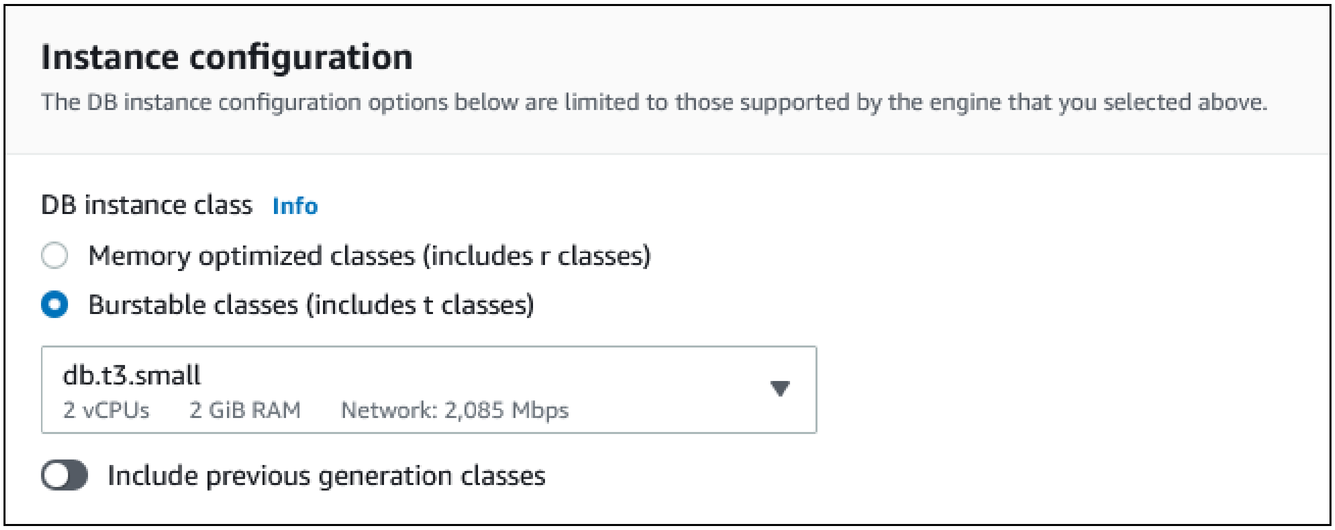
Confirm password Info

••••••••

9. In the **Instance configuration** section, set these values:

   - **Burstable classes (includes t classes)**
   - **db.t3.small** or **db.t3.medium**

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

○ Memory optimized classes (includes r classes)

● Burstable classes (includes t classes)

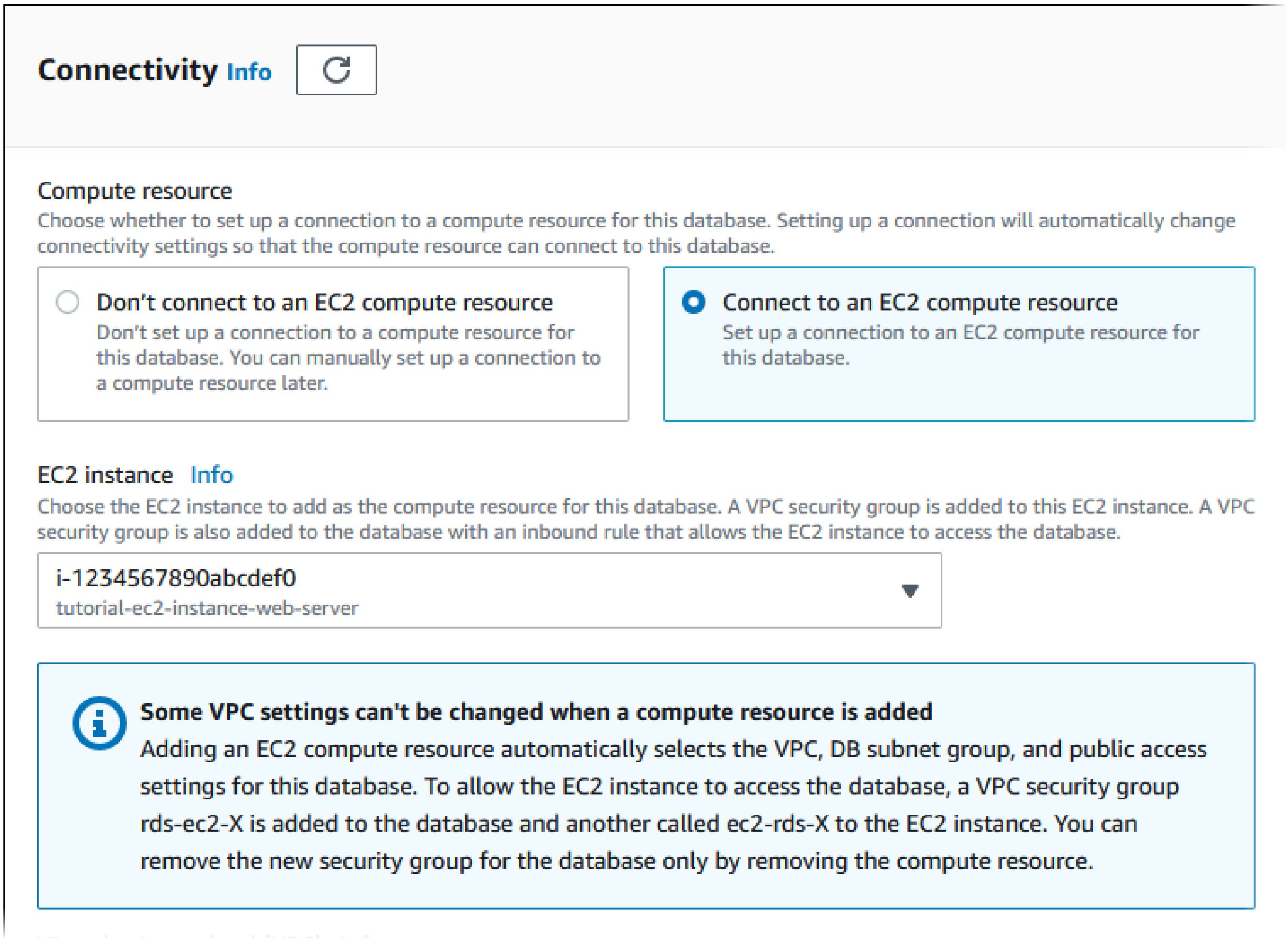
db.t3.small
2 vCPUs 2 GiB RAM Network: 2,085 Mbps

Include previous generation classes

10. In the **Availability and durability** section, use the default values.

11. In the **Connectivity** section, set these values and keep the other values as their defaults:

    - For **Compute resource**, choose **Connect to an EC2 compute resource**.
    - For **EC2 instance**, choose the EC2 instance you created previously, such as **tutorial-ec2-instance-web-server**.



Connectivity Info

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

○ Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

● Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

EC2 instance Info

Choose the EC2 instance to add as the compute resource for this database. A VPC security group is added to this EC2 instance. A VPC security group is also added to the database with an inbound rule that allows the EC2 instance to access the database.

i-1234567890abcdef0
tutorial-ec2-instance-web-server

Some VPC settings can't be changed when a compute resource is added

Adding an EC2 compute resource automatically selects the VPC, DB subnet group, and public access settings for this database. To allow the EC2 instance to access the database, a VPC security group rds-ec2-X is added to the database and another called ec2-rds-X to the EC2 instance. You can remove the new security group for the database only by removing the compute resource.

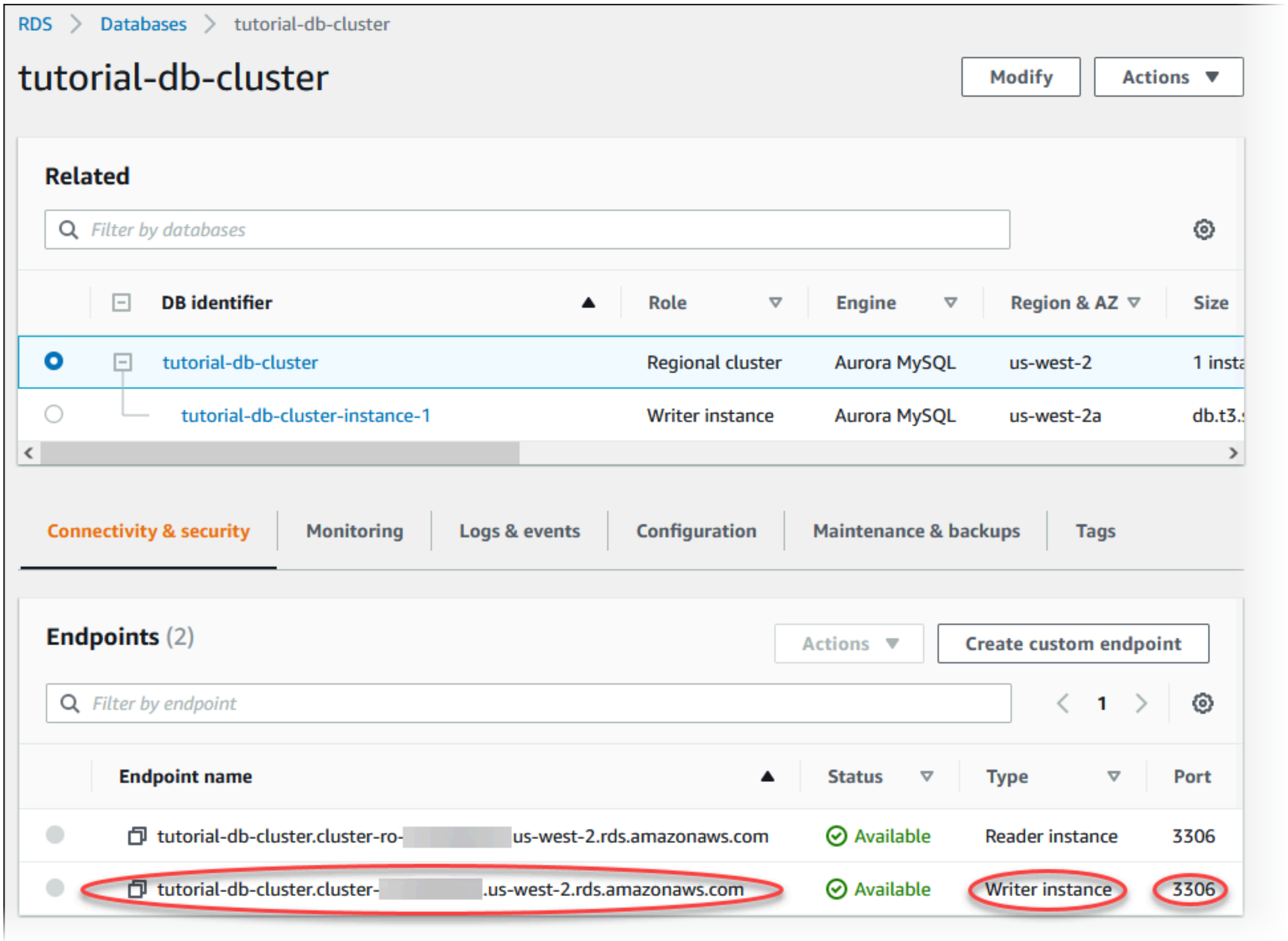
12. Open the **Additional configuration** section, and enter **`sample`** for **Initial database name**. Keep the default settings for the other options.

13. To create your Aurora MySQL DB cluster, choose **Create database**.

    Your new DB cluster appears in the **Databases** list with the status **Creating**.

14. Wait for the **Status** of your new DB cluster to show as **Available**. Then choose the DB cluster name to show its details.

15. In the **Connectivity & security** section, view the **Endpoint** and **Port** of the writer DB instance.



	DB identifier	Role	Engine	Region & AZ	Size
●	tutorial-db-cluster	Regional cluster	Aurora MySQL	us-west-2	1 insta
○	tutorial-db-cluster-instance-1	Writer instance	Aurora MySQL	us-west-2a	db.t3.

	Endpoint name	Status	Type	Port
○	tutorial-db-cluster.cluster-ro-[redacted]us-west-2.rds.amazonaws.com	✓ Available	Reader instance	3306
○	tutorial-db-cluster.cluster-[redacted].us-west-2.rds.amazonaws.com	✓ Available	Writer instance	3306

Note the endpoint and port for your writer DB instance. You use this information to connect your web server to your DB cluster.

16. Complete Install a web server on your EC2 instance.

Aurora PostgreSQL

To create an Aurora PostgreSQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

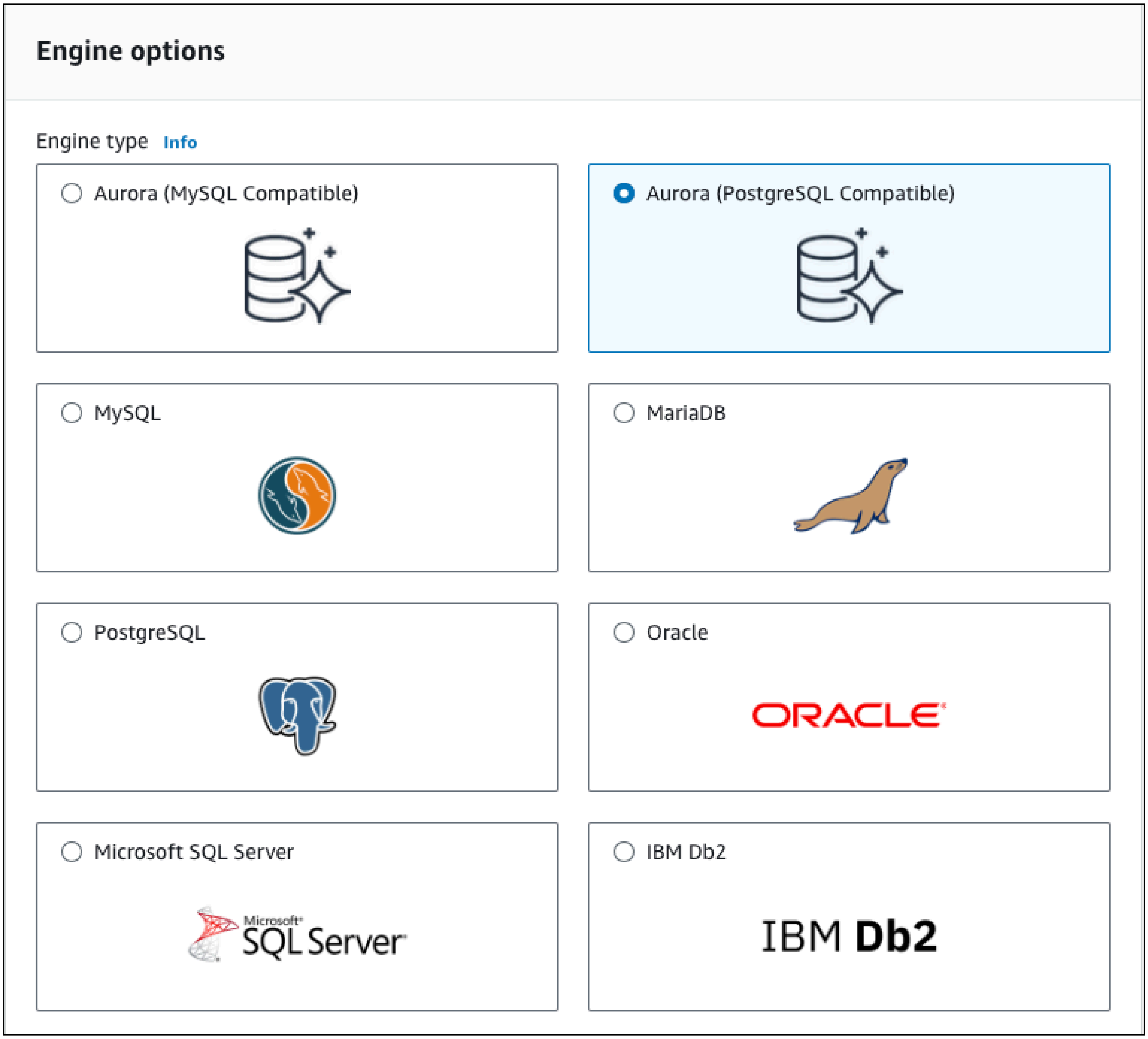
2. In the upper-right corner of the AWS Management Console, make sure the AWS Region is the same as the one where you created your EC2 instance.

3. In the navigation pane, choose **Databases**.

4. Choose **Create database**.

5. On the **Create database** page, choose **Standard create**.

6. For **Engine options**, choose **Aurora (PostgreSQL Compatible)**.



Engine options

Engine type Info

○ Aurora (MySQL Compatible)

● Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

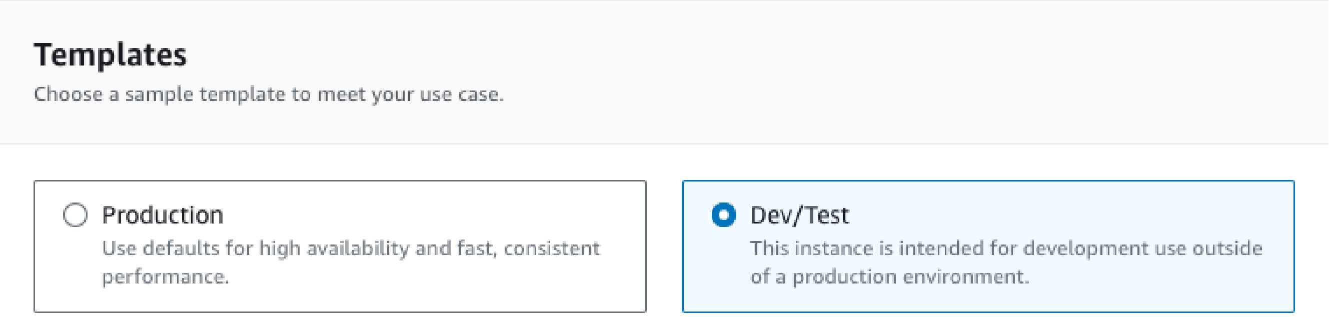
○ Oracle

○ Microsoft SQL Server

○ IBM Db2

Keep the default values for **Version** and the other engine options.

7. In the **Templates** section, choose **Dev/Test**.



Templates

Choose a sample template to meet your use case.

○ Production

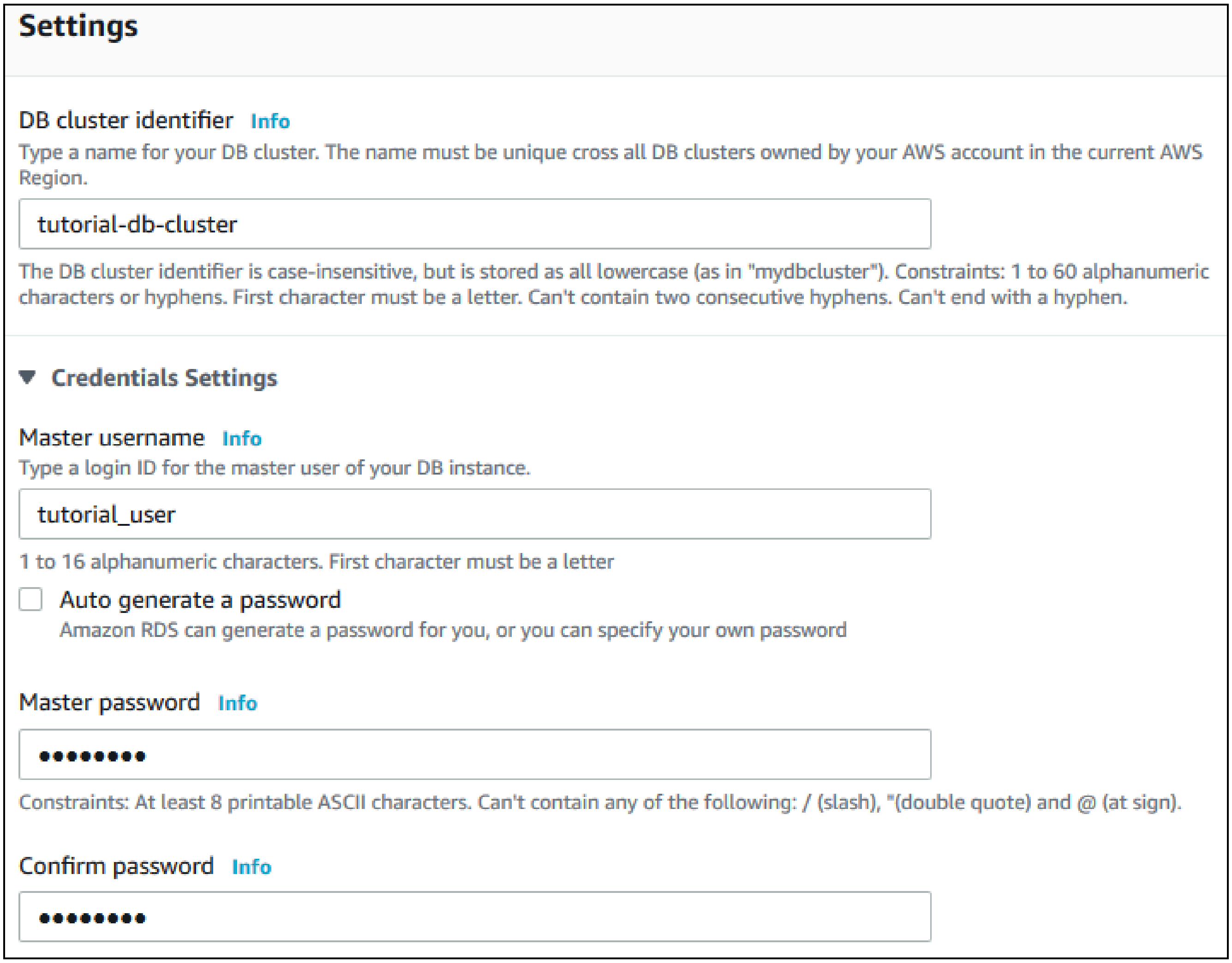
Use defaults for high availability and fast, consistent performance.

● Dev/Test

This instance is intended for development use outside of a production environment.

8. In the **Settings** section, set these values:

   - **DB cluster identifier** – Type **`tutorial-db-cluster`**.
   - **Master username** – Type **`tutorial_user`**.
   - **Auto generate a password** – Leave the option turned off.
   - **Master password** – Type a password.
   - **Confirm password** – Retype the password.



Settings

DB cluster identifier Info

Type a name for your DB cluster. The name must be unique cross all DB clusters owned by your AWS account in the current AWS Region.

tutorial-db-cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

tutorial_user

1 to 16 alphanumeric characters. First character must be a letter

□ Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password

Master password Info

••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), "(double quote) and @ (at sign).

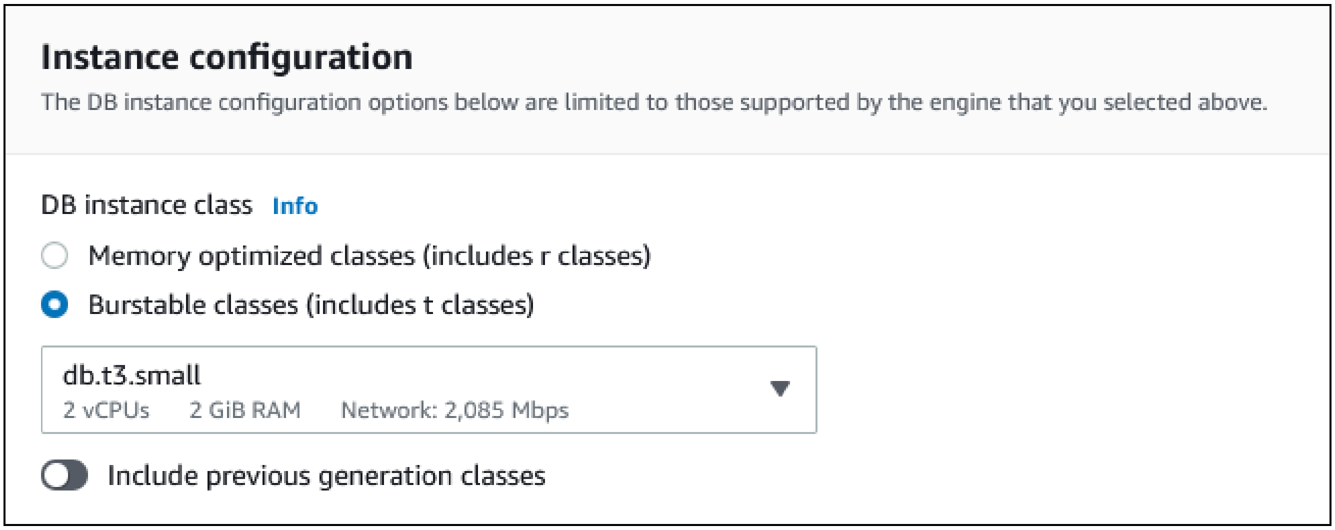
Confirm password Info

••••••••

9. In the **Instance configuration** section, set these values:

- **Burstable classes (includes t classes)**
- **db.t3.small** or **db.t3.medium**

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

○ Memory optimized classes (includes r classes)

● Burstable classes (includes t classes)

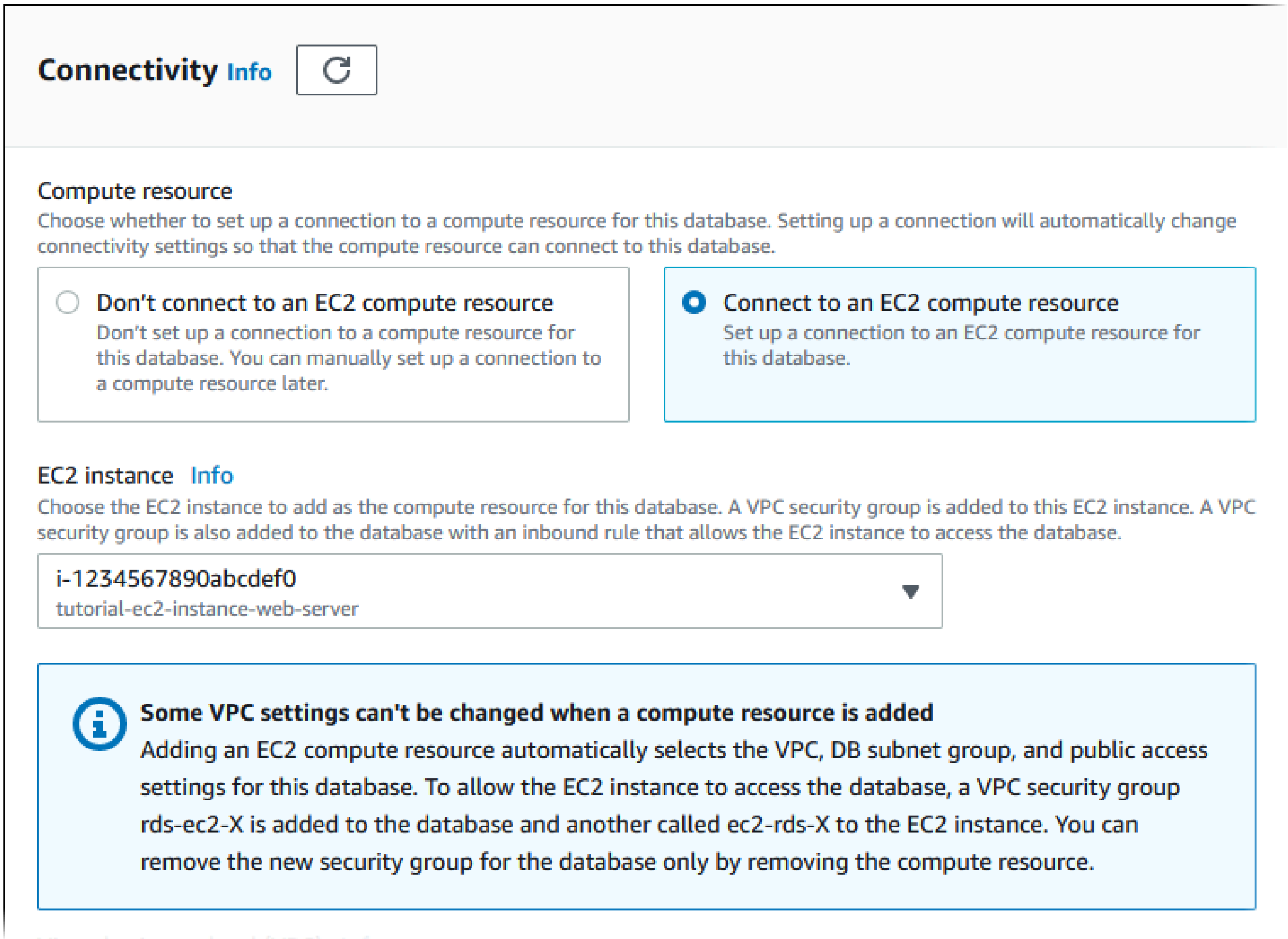
db.t3.small
2 vCPUs 2 GiB RAM Network: 2,085 Mbps

Include previous generation classes

10. In the **Availability and durability** section, use the default values.

11. In the **Connectivity** section, set these values and keep the other values as their defaults:

    - For **Compute resource**, choose **Connect to an EC2 compute resource**.
    - For **EC2 instance**, choose the EC2 instance you created previously, such as **tutorial-ec2-instance-web-server**.



Connectivity Info

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

○ Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

● Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

EC2 instance Info

Choose the EC2 instance to add as the compute resource for this database. A VPC security group is added to this EC2 instance. A VPC security group is also added to the database with an inbound rule that allows the EC2 instance to access the database.

i-1234567890abcdef0
tutorial-ec2-instance-web-server

Some VPC settings can't be changed when a compute resource is added
Adding an EC2 compute resource automatically selects the VPC, DB subnet group, and public access settings for this database. To allow the EC2 instance to access the database, a VPC security group rds-ec2-X is added to the database and another called ec2-rds-X to the EC2 instance. You can remove the new security group for the database only by removing the compute resource.

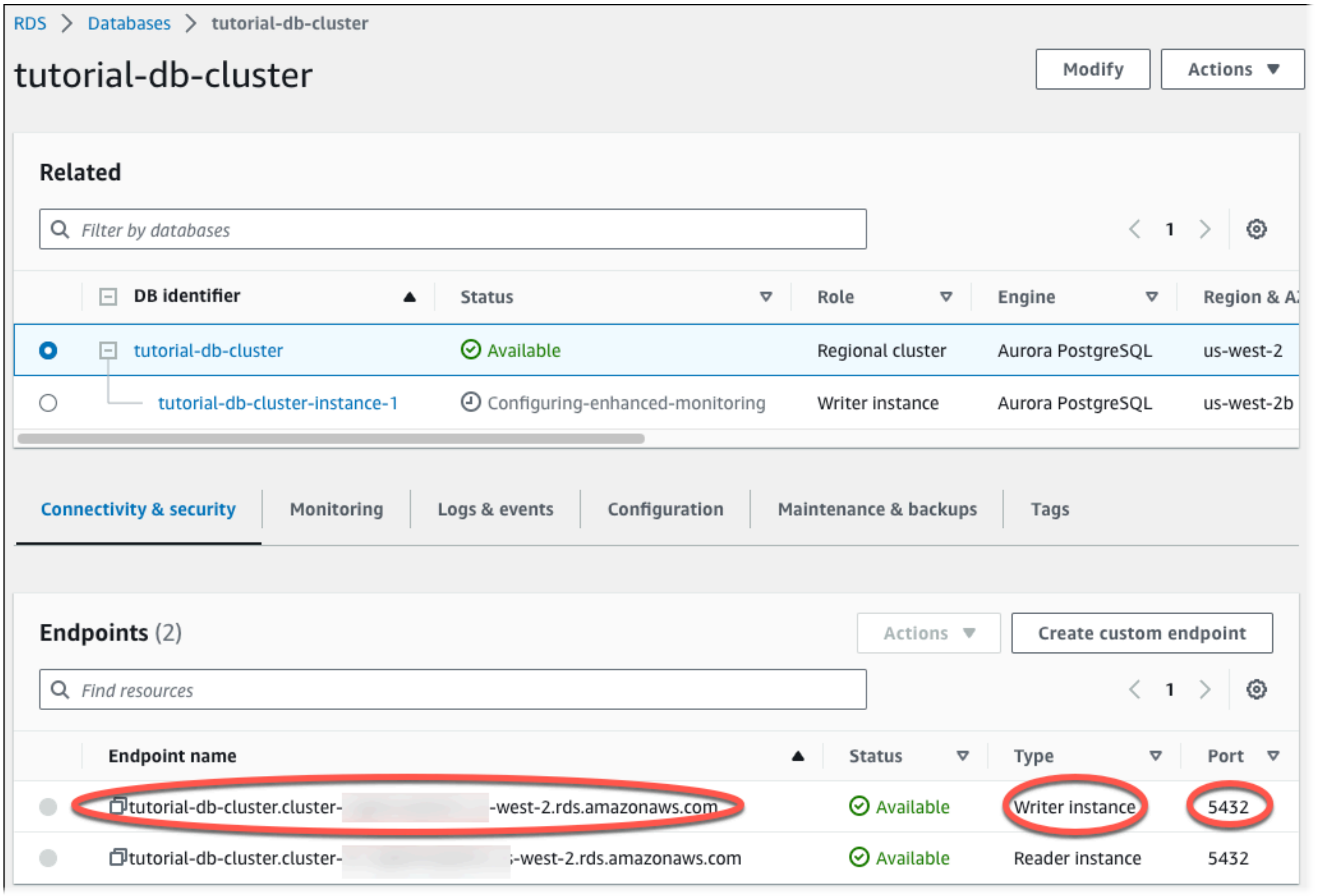
12. Open the **Additional configuration** section, and enter **`sample`** for **Initial database name**. Keep the default settings for the other options.

13. To create your Aurora PostgreSQL DB cluster, choose **Create database**.

    Your new DB cluster appears in the **Databases** list with the status **Creating**.

14. Wait for the **Status** of your new DB cluster to show as **Available**. Then choose the DB cluster name to show its details.

15. In the **Connectivity & security** section, view the **Endpoint** and **Port** of the writer DB instance.



tutorial-db-cluster

Related

	DB identifier ▲	Status	Role	Engine	Region & A
●	⊟ tutorial-db-cluster	✓ Available	Regional cluster	Aurora PostgreSQL	us-west-2
○	tutorial-db-cluster-instance-1	Configuring-enhanced-monitoring	Writer instance	Aurora PostgreSQL	us-west-2b

Endpoints (2)

	Endpoint name ▲	Status	Type	Port
○	tutorial-db-cluster.cluster-[redacted]-west-2.rds.amazonaws.com	✓ Available	Writer instance	5432
○	tutorial-db-cluster.cluster-[redacted]-west-2.rds.amazonaws.com	✓ Available	Reader instance	5432

Note the endpoint and port for your writer DB instance. You use this information to connect your web server to your DB cluster.

16. Complete Install a web server on your EC2 instance.

Install a web server on your EC2 instance

Install a web server on the EC2 instance you created in Launch an EC2 instance. The web server connects to the Amazon Aurora DB cluster that you created in Create an Amazon Aurora DB cluster.

Install an Apache web server with PHP and MariaDB

Connect to your EC2 instance and install the web server.

To connect to your EC2 instance and install the Apache web server with PHP

1. Connect to the EC2 instance that you created earlier by following the steps in Connect to your Linux instance in the *Amazon EC2 User Guide*.

We recommend that you connect to your EC2 instance using SSH. If the SSH client utility is installed on Windows, Linux, or Mac, you can connect to the instance using the following command format:

```
ssh -i location_of_pem_file ec2-user@ec2-instance-public-dns-name
```

For example, assume that `ec2-database-connect-key-pair.pem` is stored in `/dir1` on Linux, and the public IPv4 DNS for your EC2 instance is `ec2-12-345-678-90.compute-1.amazonaws.com`. Your SSH command would look as follows:

```
ssh -i /dir1/ec2-database-connect-key-pair.pem ec2-
user@ec2-12-345-678-90.compute-1.amazonaws.com
```

2. Get the latest bug fixes and security updates by updating the software on your EC2 instance. To do this, use the following command.

> **Note**
>
> The `-y` option installs the updates without asking for confirmation. To examine updates before installing, omit this option.

```
sudo dnf update -y
```

3. After the updates complete, install the Apache web server, PHP, and MariaDB or PostgreSQL software using the following commands. This command installs multiple software packages and related dependencies at the same time.

MariaDB & MySQL

```
sudo dnf install -y httpd php php-mysqli mariadb105
```

PostgreSQL

```
sudo dnf install -y httpd php php-pgsql postgresql15
```

If you receive an error, your instance probably wasn't launched with an Amazon Linux 2023 AMI. You might be using the Amazon Linux 2 AMI instead. You can view your version of Amazon Linux using the following command.

```
cat /etc/system-release
```

For more information, see Updating instance software.

4. Start the web server with the command shown following.

```
sudo systemctl start httpd
```

You can test that your web server is properly installed and started. To do this, enter the public Domain Name System (DNS) name of your EC2 instance in the address bar of a web browser, for example: `http://ec2-42-8-168-21.us-west-1.compute.amazonaws.com`. If your web server is running, then you see the Apache test page.

If you don't see the Apache test page, check your inbound rules for the VPC security group that you created in Tutorial: Create a VPC for use with a DB cluster (IPv4 only). Make sure that your inbound rules include one allowing HTTP (port 80) access for the IP address to connect to the web server.

> **Note**
>
> The Apache test page appears only when there is no content in the document root directory, `/var/www/html`. After you add content to the document root directory, your content appears at the public DNS address of your EC2 instance. Before this point, it appears on the Apache test page.

5. Configure the web server to start with each system boot using the `systemctl` command.

```
sudo systemctl enable httpd
```

To allow `ec2-user` to manage files in the default root directory for your Apache web server, modify the ownership and permissions of the `/var/www` directory. There are many ways to accomplish this task. In this tutorial, you add `ec2-user` to the `apache` group, to give the `apache` group ownership of the `/var/www` directory and assign write permissions to the group.

To set file permissions for the Apache web server

1. Add the `ec2-user` user to the `apache` group.

   ```
   sudo usermod -a -G apache ec2-user
   ```

2. Log out to refresh your permissions and include the new `apache` group.

   ```
   exit
   ```

3. Log back in again and verify that the `apache` group exists with the `groups` command.

   ```
   groups
   ```

   Your output looks similar to the following:

   ```
   ec2-user adm wheel apache systemd-journal
   ```

4. Change the group ownership of the `/var/www` directory and its contents to the `apache` group.

   ```
   sudo chown -R ec2-user:apache /var/www
   ```

5. Change the directory permissions of `/var/www` and its subdirectories to add group write permissions and set the group ID on subdirectories created in the future.

   ```
   sudo chmod 2775 /var/www
   find /var/www -type d -exec sudo chmod 2775 {} \;
   ```

6. Recursively change the permissions for files in the `/var/www` directory and its subdirectories to add group write permissions.

   ```
   find /var/www -type f -exec sudo chmod 0664 {} \;
   ```

Now, `ec2-user` (and any future members of the `apache` group) can add, delete, and edit files in the Apache document root. This makes it possible for you to add content, such as a static website or a PHP application.

> **Note**
>
> A web server running the HTTP protocol provides no transport security for the data that it sends or receives. When you connect to an HTTP server using a web browser, much information is visible to eavesdroppers anywhere along the network pathway. This information includes the URLs that you visit, the content of web pages that you receive, and the contents (including passwords) of any HTML forms.
> The best practice for securing your web server is to install support for HTTPS (HTTP Secure). This protocol protects your data with SSL/TLS encryption. For more information, see Tutorial: Configure SSL/TLS with the Amazon Linux AMI in the *Amazon EC2 User Guide*.

Connect your Apache web server to your DB cluster

Next, you add content to your Apache web server that connects to your Amazon Aurora DB cluster.

To add content to the Apache web server that connects to your DB cluster

1. While still connected to your EC2 instance, change the directory to `/var/www` and create a new subdirectory named `inc`.

   ```
   cd /var/www
   mkdir inc
   cd inc
   ```

2. Create a new file in the `inc` directory named `dbinfo.inc`, and then edit the file by calling nano (or the editor of your choice).

   ```
   >dbinfo.inc
   nano dbinfo.inc
   ```

3. Add the following contents to the `dbinfo.inc` file. Here, *`db_instance_endpoint`* is DB cluster writer endpoint, without the port, for your DB cluster.

Note

We recommend placing the user name and password information in a folder that isn't part of the document root for your web server. Doing this reduces the possibility of your security information being exposed.

Make sure to change `master password` to a suitable password in your application.

```
<?php

define('DB_SERVER', 'db_cluster_writer_endpoint');
define('DB_USERNAME', 'tutorial_user');
define('DB_PASSWORD', 'master password');
define('DB_DATABASE', 'sample');
?>
```

4. Save and close the `dbinfo.inc` file. If you are using nano, save and close the file by using Ctrl+S and Ctrl+X.

5. Change the directory to `/var/www/html`.

```
cd /var/www/html
```

6. Create a new file in the `html` directory named `SamplePage.php`, and then edit the file by calling nano (or the editor of your choice).

```
>SamplePage.php
nano SamplePage.php
```

7. Add the following contents to the `SamplePage.php` file:

MariaDB & MySQL

```
<?php include "../inc/dbinfo.inc"; ?>
<html>
<body>
<h1>Sample page</h1>
<?php

  /* Connect to MySQL and select the database. */
  $connection = mysqli_connect(DB_SERVER, DB_USERNAME, DB_PASSWORD);

  if (mysqli_connect_errno()) echo "Failed to connect to MySQL: " .
 mysqli_connect_error();

  $database = mysqli_select_db($connection, DB_DATABASE);

  /* Ensure that the EMPLOYEES table exists. */
  VerifyEmployeesTable($connection, DB_DATABASE);

  /* If input fields are populated, add a row to the EMPLOYEES table. */
  $employee_name = htmlentities($_POST['NAME']);
  $employee_address = htmlentities($_POST['ADDRESS']);

  if (strlen($employee_name) || strlen($employee_address)) {
    AddEmployee($connection, $employee_name, $employee_address);
  }
?>

<!-- Input form -->
<form action="<?PHP echo $_SERVER['SCRIPT_NAME'] ?>" method="POST">
  <table border="0">
    <tr>
      <td>NAME</td>
      <td>ADDRESS</td>
    </tr>
    <tr>
      <td>
        <input type="text" name="NAME" maxlength="45" size="30" />
      </td>
      <td>
        <input type="text" name="ADDRESS" maxlength="90" size="60" />
      </td>
      <td>
        <input type="submit" value="Add Data" />
      </td>
    </tr>
  </table>
</form>

<!-- Display table data. -->
<table border="1" cellpadding="2" cellspacing="2">
  <tr>
    <td>ID</td>
    <td>NAME</td>
    <td>ADDRESS</td>
  </tr>

<?php

$result = mysqli_query($connection, "SELECT * FROM EMPLOYEES");

while($query_data = mysqli_fetch_row($result)) {
  echo "<tr>";
  echo "<td>",$query_data[0], "</td>",
       "<td>",$query_data[1], "</td>",
       "<td>",$query_data[2], "</td>";
  echo "</tr>";
}
?>

</table>

<!-- Clean up. -->
<?php

  mysqli_free_result($result);
  mysqli_close($connection);

?>

</body>
</html>


<?php

/* Add an employee to the table. */
function AddEmployee($connection, $name, $address) {
   $n = mysqli_real_escape_string($connection, $name);
   $a = mysqli_real_escape_string($connection, $address);

   $query = "INSERT INTO EMPLOYEES (NAME, ADDRESS) VALUES ('$n', '$a');";

   if(!mysqli_query($connection, $query)) echo("<p>Error adding employee data.</
p>");
}
```

```
/* Check whether the table exists and, if not, create it. */
function VerifyEmployeesTable($connection, $dbName) {
  if(!TableExists("EMPLOYEES", $connection, $dbName))
  {
     $query = "CREATE TABLE EMPLOYEES (
         ID int(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
         NAME VARCHAR(45),
         ADDRESS VARCHAR(90)
       )";

     if(!mysqli_query($connection, $query)) echo("<p>Error creating table.</
p>");
  }
}

/* Check for the existence of a table. */
function TableExists($tableName, $connection, $dbName) {
  $t = mysqli_real_escape_string($connection, $tableName);
  $d = mysqli_real_escape_string($connection, $dbName);

  $checktable = mysqli_query($connection,
      "SELECT TABLE_NAME FROM information_schema.TABLES WHERE TABLE_NAME = '$t'
 AND TABLE_SCHEMA = '$d'");

  if(mysqli_num_rows($checktable) > 0) return true;

  return false;
}
?>
```

PostgreSQL

```
<?php include "../inc/dbinfo.inc"; ?>

<html>
<body>
<h1>Sample page</h1>
<?php

/* Connect to PostgreSQL and select the database. */
$constring = "host=" . DB_SERVER . " dbname=" . DB_DATABASE . " user=" .
 DB_USERNAME . " password=" . DB_PASSWORD ;
```

```
$connection = pg_connect($constring);

if (!$connection){
 echo "Failed to connect to PostgreSQL";
 exit;
}

/* Ensure that the EMPLOYEES table exists. */
VerifyEmployeesTable($connection, DB_DATABASE);

/* If input fields are populated, add a row to the EMPLOYEES table. */
$employee_name = htmlentities($_POST['NAME']);
$employee_address = htmlentities($_POST['ADDRESS']);

if (strlen($employee_name) || strlen($employee_address)) {
  AddEmployee($connection, $employee_name, $employee_address);
}

?>

<!-- Input form -->
<form action="<?PHP echo $_SERVER['SCRIPT_NAME'] ?>" method="POST">
  <table border="0">
    <tr>
      <td>NAME</td>
      <td>ADDRESS</td>
    </tr>
    <tr>
      <td>
    <input type="text" name="NAME" maxlength="45" size="30" />
      </td>
      <td>
    <input type="text" name="ADDRESS" maxlength="90" size="60" />
      </td>
      <td>
    <input type="submit" value="Add Data" />
      </td>
    </tr>
  </table>
</form>
<!-- Display table data. -->
<table border="1" cellpadding="2" cellspacing="2">
  <tr>
    <td>ID</td>
    <td>NAME</td>
    <td>ADDRESS</td>
  </tr>

<?php

$result = pg_query($connection, "SELECT * FROM EMPLOYEES");

while($query_data = pg_fetch_row($result)) {
  echo "<tr>";
  echo "<td>",$query_data[0], "</td>",
       "<td>",$query_data[1], "</td>",
       "<td>",$query_data[2], "</td>";
  echo "</tr>";
}
?>
</table>

<!-- Clean up. -->
<?php

  pg_free_result($result);
  pg_close($connection);
?>
</body>
</html>


<?php

/* Add an employee to the table. */
function AddEmployee($connection, $name, $address) {
   $n = pg_escape_string($name);
   $a = pg_escape_string($address);
   echo "Forming Query";
   $query = "INSERT INTO EMPLOYEES (NAME, ADDRESS) VALUES ('$n', '$a');";

   if(!pg_query($connection, $query)) echo("<p>Error adding employee data.</
p>");
}

/* Check whether the table exists and, if not, create it. */
function VerifyEmployeesTable($connection, $dbName) {
  if(!TableExists("EMPLOYEES", $connection, $dbName))
  {
     $query = "CREATE TABLE EMPLOYEES (
         ID serial PRIMARY KEY,
         NAME VARCHAR(45),
         ADDRESS VARCHAR(90)
       )";

     if(!pg_query($connection, $query)) echo("<p>Error creating table.</p>");
  }
}
/* Check for the existence of a table. */
function TableExists($tableName, $connection, $dbName) {
  $t = strtolower(pg_escape_string($tableName)); //table name is case sensitive
  $d = pg_escape_string($dbName); //schema is 'public' instead of 'sample' db
 name so not using that

  $query = "SELECT TABLE_NAME FROM information_schema.TABLES WHERE TABLE_NAME =
 '$t';";
  $checktable = pg_query($connection, $query);

  if (pg_num_rows($checktable) >0) return true;
  return false;

}
?>
```

8. Save and close the `SamplePage.php` file.
9. Verify that your web server successfully connects to your DB cluster by opening a web browser and browsing to `http://`*EC2 instance endpoint*`/SamplePage.php`, for example: `http://ec2-12-345-67-890.us-west-2.compute.amazonaws.com/SamplePage.php`.

You can use `SamplePage.php` to add data to your DB cluster. The data that you add is then displayed on the page. To verify that the data was inserted into the table, install MySQL client on the Amazon EC2 instance. Then connect to the DB cluster and query the table.

For information about connecting to a DB cluster, see Connecting to an Amazon Aurora DB cluster.

To make sure that your DB cluster is as secure as possible, verify that sources outside of the VPC can't connect to your DB cluster.

After you have finished testing your web server and your database, you should delete your DB cluster and your Amazon EC2 instance.

- To delete a DB cluster, follow the instructions in Deleting Aurora DB clusters and DB instances. You don't need to create a final snapshot.
- To terminate an Amazon EC2 instance, follow the instruction in Terminate your instance in the *Amazon EC2 User Guide*.

Amazon Aurora tutorials and sample code

The AWS documentation includes several tutorials that guide you through common Amazon Aurora use cases. Many of these tutorials show you how to use Amazon Aurora with other AWS services. In addition, you can access sample code in GitHub.

> **Note**
>
> You can find more tutorials at the AWS Database Blog. For information about training, see AWS Training and Certification.

Topics

- Tutorials in this guide
- Tutorials in other AWS guides
- AWS workshop and lab content portal for Amazon Aurora PostgreSQL
- AWS workshop and lab content portal for Amazon Aurora MySQL
- Tutorials and sample code in GitHub
- Using this service with an AWS SDK

Tutorials in this guide

The following tutorials in this guide show you how to perform common tasks with Amazon Aurora:

- Tutorial: Create a VPC for use with a DB cluster (IPv4 only)

  Learn how to include a DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. In this case, the VPC shares data with a web server that is running on an Amazon EC2 instance in the same VPC.

- Tutorial: Create a VPC for use with a DB cluster (dual-stack mode)

  Learn how to include a DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. In this case, the VPC shares data with an Amazon EC2 instance in the same VPC. In this tutorial, you create the VPC for this scenario that works with a database running in dual-stack mode.

- Tutorial: Create a web server and an Amazon Aurora DB cluster

  Learn how to install an Apache web server with PHP and create a MySQL database. The web server runs on an Amazon EC2 instance using Amazon Linux, and the MySQL database is an Aurora MySQL DB cluster. Both the Amazon EC2 instance and the DB cluster run in an Amazon VPC.
- Tutorial: Restore an Amazon Aurora DB cluster from a DB cluster snapshot

  Learn how to restore a DB cluster from a DB cluster snapshot.
- Tutorial: Use tags to specify which Aurora DB clusters to stop

  Learn how to use tags to specify which Aurora DB clusters to stop.
- Tutorial: Log DB instance state changes using Amazon EventBridge

  Learn how to log a DB instance state change using Amazon EventBridge and AWS Lambda.

Tutorials in other AWS guides

The following tutorials in other AWS guides show you how to perform common tasks with Amazon Aurora:

> **Note**
>
> Some of the tutorials use Amazon RDS DB instances, but they can be adapted to use Aurora DB clusters.

- Tutorial: Aurora Serverless in the *AWS AppSync Developer Guide*

  Learn how to use AWS AppSync to provide a data source for running SQL commands against Aurora Serverless DB clusters with the Data API enabled. You can use AWS AppSync resolvers to run SQL statements against the Data API with GraphQL queries, mutations, and subscriptions.
- Tutorial: Rotating a Secret for an AWS Database in the *AWS Secrets Manager User Guide*

  Learn how to create a secret for an AWS database and configure the secret to rotate on a schedule. You trigger one rotation manually, and then confirm that the new version of the secret continues to provide access.
- Tutorials and samples in the *AWS Elastic Beanstalk Developer Guide*

Learn how to deploy applications that use Amazon RDS databases with AWS Elastic Beanstalk.

- Using Data from an Amazon RDS Database to Create an Amazon ML Datasource in the *Amazon Machine Learning Developer Guide*

  Learn how to create an Amazon Machine Learning (Amazon ML) datasource object from data stored in a MySQL DB instance.

- Manually Enabling Access to an Amazon RDS Instance in a VPC in the *Amazon QuickSight User Guide*

  Learn how to enable Amazon QuickSight access to an Amazon RDS DB instance in a VPC.

AWS workshop and lab content portal for Amazon Aurora PostgreSQL

The following collection of workshops and other hands-on content helps you to gain an understanding of the Amazon Aurora PostgreSQL features and capabilities:

- Creating an Aurora Cluster

  Learn how to create an Amazon Aurora PostgreSQL cluster manually.

- Creating a Cloud9 Cloud-based IDE environment to connect to your database

  Learn how to configure Cloud9 and initialize the PostgreSQL database.

- Fast Cloning

  Learn how to create an Aurora fast clone.

- Query Plan Management

  Learn how to control execution plans for a set of statements using query plan management.

- Cluster Cache Management

  Learn about Cluster Cache Management feature in Aurora PostgreSQL.

- Database Activity Streaming

  Learn how to monitor and audit your database activity with this feature.

- Using Performance Insights

Learn how to monitor and tune your DB instance using Performance insights.

- Performance Monitoring with RDS Tools

  Learn how to use AWS and Postgres tools(Cloudwatch, Enhanced Monitoring, Slow Query Logs, Performance Insights, PostgreSQL Catalog Views) to understand performance issues and identify ways to improve performance of your database.
- Auto Scaling Read Replicas

  Learn how Aurora read replica auto scaling works in practice using a load generator script.
- Testing Fault Tolerance

  Learn how a DB cluster can tolerate a failure.
- Aurora Global Database

  Learn about Aurora Global Database.
- Using Machine Learning

  Learn about Aurora Machine Learning.
- Aurora Serverless v2

  Learn about Aurora Serverless v2.
- Trusted Language Extensions for Aurora PostgreSQL

  Learn how to build high-performance extensions that run safely on Aurora PostgreSQL.

AWS workshop and lab content portal for Amazon Aurora MySQL

The following collection of workshops and other hands-on content helps you to gain an understanding of the Amazon Aurora MySQL features and capabilities:

- Creating an Aurora Cluster

  Learn how to create an Amazon Aurora MySQL cluster manually.
- Creating a Cloud9 Cloud-based IDE environment to connect to your database

  Learn how to configure Cloud9 and initialize the MySQL database.

- Fast Cloning

  Learn how to create an Aurora fast clone.
- Backtrack a Cluster

  Learn how to backtrack a DB cluster.
- Using Performance Insights

  Learn how to monitor and tune your DB instance using Performance insights.
- Performance Monitoring with RDS Tools

  Learn how to use AWS and SQL tools to understand performance issues and identify ways to improve performance of your database.
- Analyze Query Performance

  Learn how to troubleshoot SQL performance related issues using different tools.
- Auto Scaling Read Replicas

  Learn how auto scaling read replicas work.
- Testing Fault Tolerance

  Learn about high availability and fault tolerance features in Aurora MySQL.
- Aurora Global Database

  Learn about Aurora Global Database.
- Aurora Serverless v2

  Learn about Aurora Serverless v2.
- Using Machine Learning

  Learn about Aurora Machine Learning.

Tutorials and sample code in GitHub

The following tutorials and sample code in GitHub show you how to perform common tasks with Amazon Aurora:

- Creating an Aurora Serverless v2 lending library

Learn how to create a lending library application where patrons can borrow and return books. The example uses Aurora Serverless v2 and AWS SDK for Python (Boto3).

- Creating an Amazon Aurora item tracker application with a Spring REST API that queries Aurora Serverless v2 data using SDK for Java 2.x

  Learn how to create a Spring REST API that queries Aurora Serverless v2 data. It's for use by a React application using SDK for Java 2.x.
- Creating an Amazon Aurora item tracker application that queries Aurora Serverless v2 data using AWS SDK for PHP

  Learn how to create an application that uses the `RdsDataClient` of the Data API and Aurora Serverless v2 to track and report on work items. The example uses AWS SDK for PHP.
- Creating an Amazon Aurora item tracker application that queries Aurora Serverless v2 data using AWS SDK for Python (Boto3)

  Learn how to create an application that uses the `RdsDataClient` of the Data API and Aurora Serverless v2 to track and report on work items. The example uses AWS SDK for Python (Boto3).

Using this service with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples

SDK documentation	Code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to this service, see Code examples for Aurora using AWS SDKs.

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Configuring your Amazon Aurora DB cluster

This section shows how to set up your Aurora DB cluster. Before creating an Aurora DB cluster, decide on the DB instance class that will run the DB cluster. Also, decide where the DB cluster will run by choosing an AWS Region. Next, create the DB cluster. If you have data outside of Aurora, you can migrate the data into an Aurora DB cluster.

Topics

Creating an Amazon Aurora DB cluster

An Amazon Aurora DB cluster consists of a DB instance, compatible with either MySQL or PostgreSQL, and a cluster volume that holds the data for the DB cluster, copied across three Availability Zones as a single, virtual volume. By default, an Aurora DB cluster contains a primary DB instance that performs reads and writes, and, optionally, up to 15 Aurora Replicas (reader DB instances). For more information about Aurora DB clusters, see Amazon Aurora DB clusters.

Aurora has two main types of DB cluster:

- Aurora provisioned – You choose the DB instance class for the writer and reader instances based on your expected workload. For more information, see Aurora DB instance classes. Aurora provisioned has several options, including Aurora global databases. For more information, see Using Amazon Aurora global databases.
- Aurora Serverless – Aurora Serverless v1 and Aurora Serverless v2 are on-demand automatic scaling configurations for Aurora. Capacity is adjusted automatically based on application demand. You're charged only for the resources that your DB cluster consumes. This automation is especially useful for environments with highly variable and unpredictable workloads. For more information, see Using Amazon Aurora Serverless v1 and Using Aurora Serverless v2.

Following, you can find out how to create an Aurora DB cluster. To get started, first see DB cluster prerequisites.

For instructions on connecting to your Aurora DB cluster, see Connecting to an Amazon Aurora DB cluster.

Contents

- Settings that don't apply to Amazon Aurora DB instances

DB cluster prerequisites

⚠ **Important**

Before you can create an Aurora DB cluster, you must complete the tasks in Setting up your environment for Amazon Aurora.

The following are prerequisites to complete before creating a DB cluster.

Topics

- Configure the network for the DB cluster
- Additional prerequisites

Configure the network for the DB cluster

You can create an Amazon Aurora DB cluster only in a virtual private cloud (VPC) based on the Amazon VPC service, in an AWS Region that has at least two Availability Zones. The DB subnet group that you choose for the DB cluster must cover at least two Availability Zones. This configuration ensures that your DB cluster always has at least one DB instance available for failover, in the unlikely event of an Availability Zone failure.

If you plan to set up connectivity between your new DB cluster and an EC2 instance in the same VPC, you can do so during DB cluster creation. If you plan to connect to your DB cluster from resources other than EC2 instances in the same VPC, you can configure the network connections manually.

Topics

- Configure automatic network connectivity with an EC2 instance
- Configure the network manually

Configure automatic network connectivity with an EC2 instance

When you create an Aurora DB cluster, you can use the AWS Management Console to set up connectivity between an Amazon EC2 instance and the new DB cluster. When you do so, RDS configures your VPC and network settings automatically. The DB cluster is created in the same VPC as the EC2 instance so that the EC2 instance can access the DB cluster.

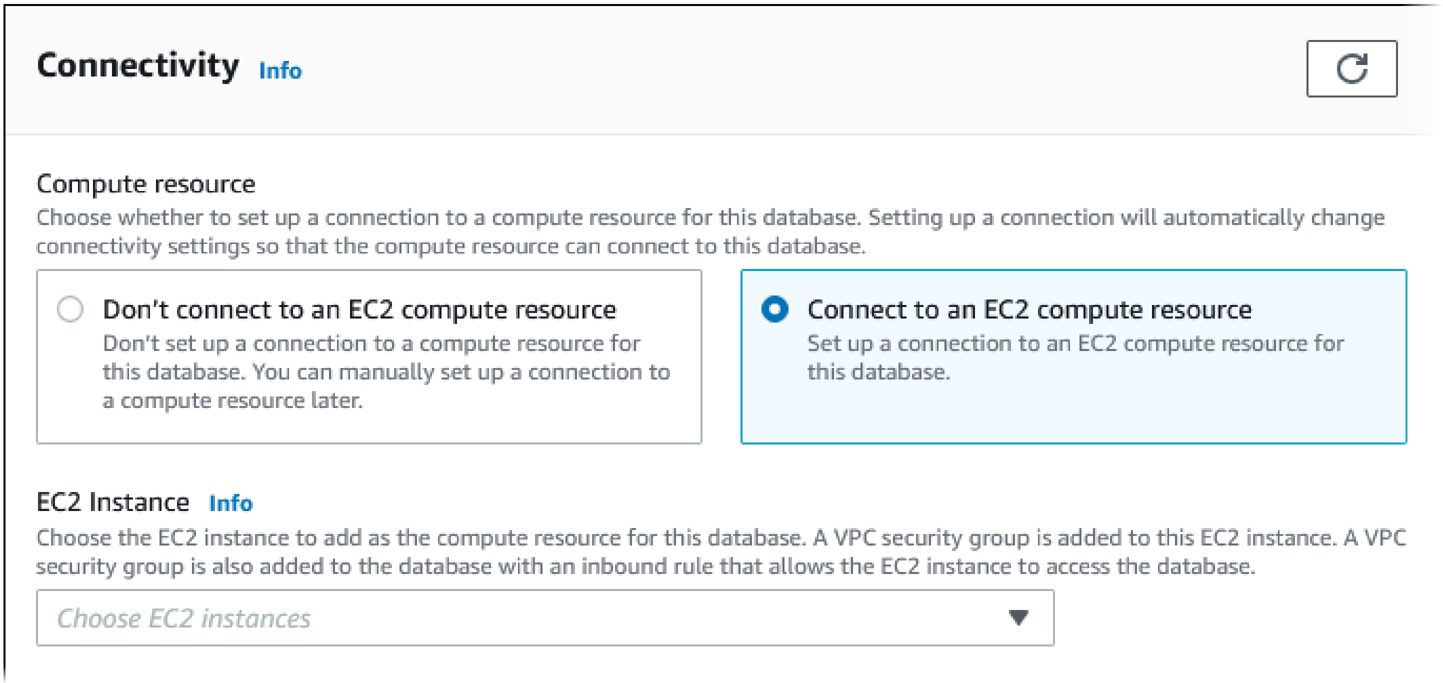
The following are requirements for connecting an EC2 instance with the DB cluster:

- The EC2 instance must exist in the AWS Region before you create the DB cluster.

  If no EC2 instances exist in the AWS Region, the console provides a link to create one.
- Currently, the DB cluster can't be an Aurora Serverless DB cluster or part of an Aurora global database.
- The user who is creating the DB instance must have permissions to perform the following operations:
  - `ec2:AssociateRouteTable`
  - `ec2:AuthorizeSecurityGroupEgress`
  - `ec2:AuthorizeSecurityGroupIngress`
  - `ec2:CreateRouteTable`
  - `ec2:CreateSubnet`
  - `ec2:CreateSecurityGroup`
  - `ec2:DescribeInstances`
  - `ec2:DescribeNetworkInterfaces`
  - `ec2:DescribeRouteTables`
  - `ec2:DescribeSecurityGroups`
  - `ec2:DescribeSubnets`
  - `ec2:ModifyNetworkInterfaceAttribute`
  - `ec2:RevokeSecurityGroupEgress`

Using this option creates a private DB cluster. The DB cluster uses a DB subnet group with only private subnets to restrict access to resources within the VPC.

To connect an EC2 instance to the DB cluster, choose **Connect to an EC2 compute resource** in the **Connectivity** section on the **Create database** page.



Connectivity Info

Compute resource

Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

○ Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

● Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

EC2 Instance Info

Choose the EC2 instance to add as the compute resource for this database. A VPC security group is added to this EC2 instance. A VPC security group is also added to the database with an inbound rule that allows the EC2 instance to access the database.

Choose EC2 instances ▼

When you choose **Connect to an EC2 compute resource**, RDS sets the following options automatically. You can't change these settings unless you choose not to set up connectivity with an EC2 instance by choosing **Don't connect to an EC2 compute resource**.

Console option	Automatic setting
Network type	RDS sets network type to **IPv4**. Currently, dual-stack mode isn't supported when you set up a connection between an EC2 instance and the DB cluster.
Virtual Private Cloud (VPC)	RDS sets the VPC to the one associated with the EC2 instance.
DB subnet group	RDS requires a DB subnet group with a private subnet in the same Availability Zone as the EC2 instance. If a DB subnet group that meets this requirement exists, then RDS uses the existing DB subnet group. By default, this option is set to **Automatic setup**. When you choose **Automatic setup** and there is no DB subnet group that meets this requirement, the following action happens. RDS uses three available private subnets in three Availability Zones where one of the Availability Zones is the

Console option	Automatic setting
	same as the EC2 instance. If a private subnet isn’t available in an Availability Zone, RDS creates a private subnet in the Availability Zone. Then RDS creates the DB subnet group. When a private subnet is available, RDS uses the route table associated with the subnet and adds any subnets it creates to this route table. When no private subnet is available, RDS creates a route table without internet gateway access and adds the subnets it creates to the route table. RDS also allows you to use existing DB subnet groups. Select **Choose existing** if you want to use an existing DB subnet group of your choice.
Public access	RDS chooses **No** so that the DB cluster isn't publicly accessible. For security, it is a best practice to keep the database private and make sure it isn't accessible from the internet.

Console option	Automatic setting
VPC security group (firewall)	RDS creates a new security group that is associated with the DB cluster. The security group is named `rds-ec2-`*n*, where *n* is a number. This security group includes an inbound rule with the EC2 VPC security group (firewall) as the source. This security group that is associated with the DB cluster allows the EC2 instance to access the DB cluster. RDS also creates a new security group that is associated with the EC2 instance. The security group is named `ec2-rds-`*n*, where *n* is a number. This security group includes an outbound rule with the VPC security group of the DB cluster as the source. This security group allows the EC2 instance to send traffic to the DB cluster. You can add another new security group by choosing **Create new** and typing the name of the new security group. You can add existing security groups by choosing **Choose existing** and selecting security groups to add.
Availability Zone	When you don't create an Aurora Replica in **Availability & durability** during DB cluster creation (Single-AZ deployment), RDS chooses the Availability Zone of the EC2 instance. When you create an Aurora Replica during DB cluster creation (Multi-AZ deployment), RDS chooses the Availability Zone of the EC2 instance for one DB instance in the DB cluster. RDS randomly chooses a different Availability Zone for the other DB instance in the DB cluster. Either the primary DB instance or the Aurora Replica is created in the same Availability Zone as the EC2 instance. There is the possibility of cross Availability Zone costs if the primary DB instance and EC2 instance are in different Availability Zones.

For more information about these settings, see Settings for Aurora DB clusters.

If you make any changes to these settings after the DB cluster is created, the changes might affect the connection between the EC2 instance and the DB cluster.

Configure the network manually

If you plan to connect to your DB cluster from resources other than EC2 instances in the same VPC, you can configure the network connections manually. If you use the AWS Management Console to create your DB cluster, you can have Amazon RDS automatically create a VPC for you. Or you can use an existing VPC or create a new VPC for your Aurora DB cluster. Whichever approach you take, your VPC must have at least one subnet in each of at least two Availability Zones for you to use it with an Amazon Aurora DB cluster.

By default, Amazon RDS creates the primary DB instance and the Aurora Replica in the Availability Zones automatically for you. To choose a specific Availability Zone, you need to change the **Availability & durability** Multi-AZ deployment setting to **Don't create an Aurora Replica**. Doing so exposes an **Availability Zone** setting that lets you choose from among the Availability Zones in your VPC. However, we strongly recommend that you keep the default setting and let Amazon RDS create a Multi-AZ deployment and choose Availability Zones for you. By doing so, your Aurora DB cluster is created with the fast failover and high availability features that are two of Aurora's key benefits.

If you don't have a default VPC or you haven't created a VPC, you can have Amazon RDS automatically create a VPC for you when you create a DB cluster using the console. Otherwise, you must do the following:

- Create a VPC with at least one subnet in each of at least two of the Availability Zones in the AWS Region where you want to deploy your DB cluster. For more information, see Working with a DB cluster in a VPC and Tutorial: Create a VPC for use with a DB cluster (IPv4 only).
- Specify a VPC security group that authorizes connections to your DB cluster. For more information, see Provide access to the DB cluster in the VPC by creating a security group and Controlling access with security groups.
- Specify an RDS DB subnet group that defines at least two subnets in the VPC that can be used by the DB cluster. For more information, see Working with DB subnet groups.

For information on VPCs, see Amazon VPC VPCs and Amazon Aurora. For a tutorial that configures the network for a private DB cluster, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

If you want to connect to a resource that isn't in the same VPC as the Aurora DB cluster, see the appropriate scenarios in Scenarios for accessing a DB cluster in a VPC.

Additional prerequisites

Before you create your DB cluster, consider the following additional prerequisites:

- If you are connecting to AWS using AWS Identity and Access Management (IAM) credentials, your AWS account must have IAM policies that grant the permissions required to perform Amazon RDS operations. For more information, see Identity and access management for Amazon Aurora.

  If you are using IAM to access the Amazon RDS console, you must first sign on to the AWS Management Console with your user credentials. Then go to the Amazon RDS console at https://console.aws.amazon.com/rds/.
- If you want to tailor the configuration parameters for your DB cluster, you must specify a DB cluster parameter group and DB parameter group with the required parameter settings. For information about creating or modifying a DB cluster parameter group or DB parameter group, see Working with parameter groups.
- Determine the TCP/IP port number to specify for your DB cluster. The firewalls at some companies block connections to the default ports (3306 for MySQL, 5432 for PostgreSQL) for Aurora. If your company firewall blocks the default port, choose another port for your DB cluster. All instances in a DB cluster use the same port.
- If the major engine version for your database has reached the RDS end of standard support date, you must use the Extended Support CLI option or the RDS API parameter. For more information, see RDS Extended Support in Settings for Aurora DB clusters.

Creating a DB cluster

You can create an Aurora DB cluster using the AWS Management Console, the AWS CLI, or the RDS API.

Console

You can create a DB cluster using the AWS Management Console with **Easy create** enabled or not enabled. With **Easy create** enabled, you specify only the DB engine type, DB instance size, and DB instance identifier. **Easy create** uses the default setting for other configuration options. With **Easy create** not enabled, you specify more configuration options when you create a database, including ones for availability, security, backups, and maintenance.

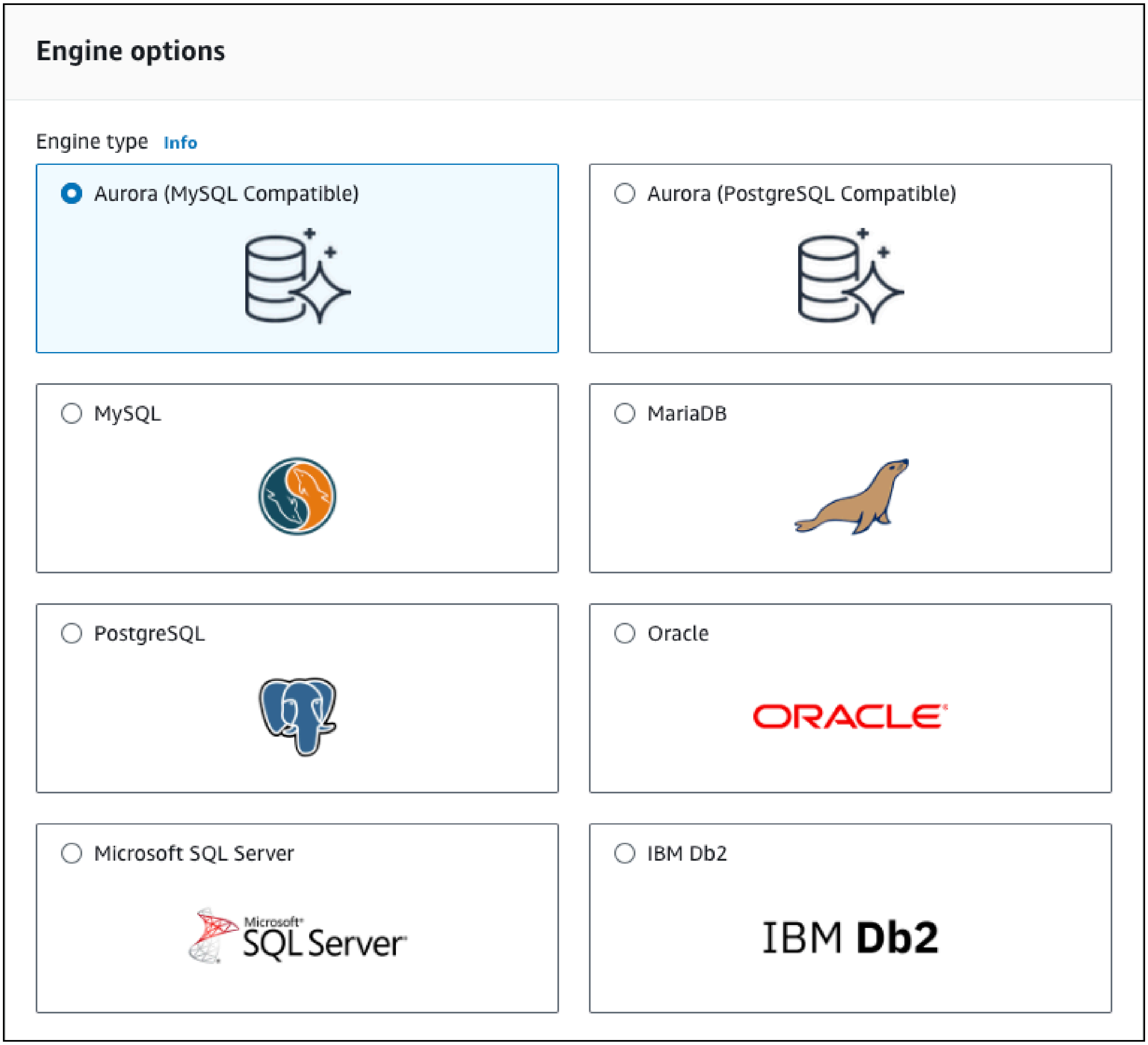
Note

For this example, **Standard create** is enabled, and **Easy create** isn't enabled. For information about creating a DB cluster with **Easy create** enabled, see Getting started with Amazon Aurora.

To create an Aurora DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you want to create the DB cluster.

   Aurora is not available in all AWS Regions. For a list of AWS Regions where Aurora is available, see Region availability.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database**.
5. For **Choose a database creation method**, choose **Standard create**.
6. For **Engine type**, choose one of the following:
   - **Aurora (MySQL Compatible)**
   - **Aurora (PostgreSQL Compatible)**



Engine options

Engine type Info

● Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

○ IBM Db2

7. Choose the **Engine version**.

   For more information, see [Amazon Aurora versions](). You can use the filters to choose versions that are compatible with features that you want, such as Aurora Serverless v2. For more information, see [Using Aurora Serverless v2]().

8. In **Templates**, choose the template that matches your use case.

9. To enter your master password, do the following:

   a. In the **Settings** section, expand **Credential Settings**.

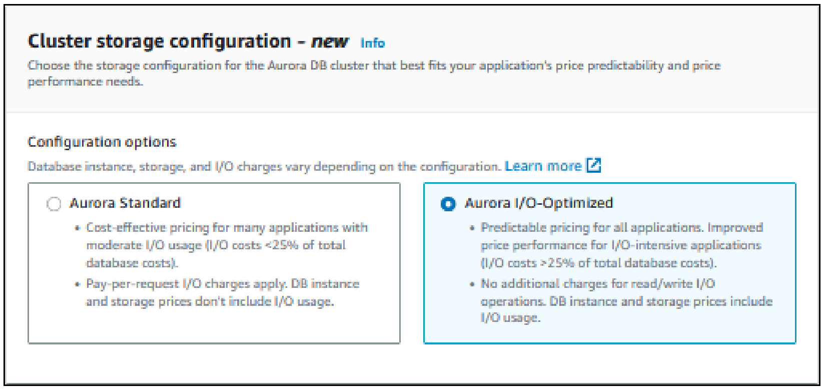
   b. Clear the **Auto generate a password** check box.

   c. (Optional) Change the **Master username** value and enter the same password in **Master password** and **Confirm password**.

   By default, the new DB instance uses an automatically generated password for the master user.

10. In the **Connectivity** section under **VPC security group (firewall)**, if you select **Create new**, a VPC security group is created with an inbound rule that allows your local computer's IP address to access the database.

11. For **Cluster storage configuration**, choose either **Aurora I/O-Optimized** or **Aurora Standard**. For more information, see Storage configurations for Amazon Aurora DB clusters.



Cluster storage configuration - *new* Info

Choose the storage configuration for the Aurora DB cluster that best fits your application's price predictability and price performance needs.

Configuration options

Database instance, storage, and I/O charges vary depending on the configuration. Learn more

○ Aurora Standard

- Cost-effective pricing for many applications with moderate I/O usage (I/O costs <25% of total database costs).
- Pay-per-request I/O charges apply. DB instance and storage prices don't include I/O usage.

● Aurora I/O-Optimized

- Predictable pricing for all applications. Improved price performance for I/O-intensive applications (I/O costs >25% of total database costs).
- No additional charges for read/write I/O operations. DB instance and storage prices include I/O usage.

12. (Optional) Set up a connection to a compute resource for this DB cluster.

    You can configure connectivity between an Amazon EC2 instance and the new DB cluster during DB cluster creation. For more information, see Configure automatic network connectivity with an EC2 instance.

13. For the remaining sections, specify your DB cluster settings. For information about each setting, see Settings for Aurora DB clusters.

14. Choose **Create database**.

    If you chose to use an automatically generated password, the **View credential details** button appears on the **Databases** page.

    To view the master user name and password for the DB cluster, choose **View credential details**.

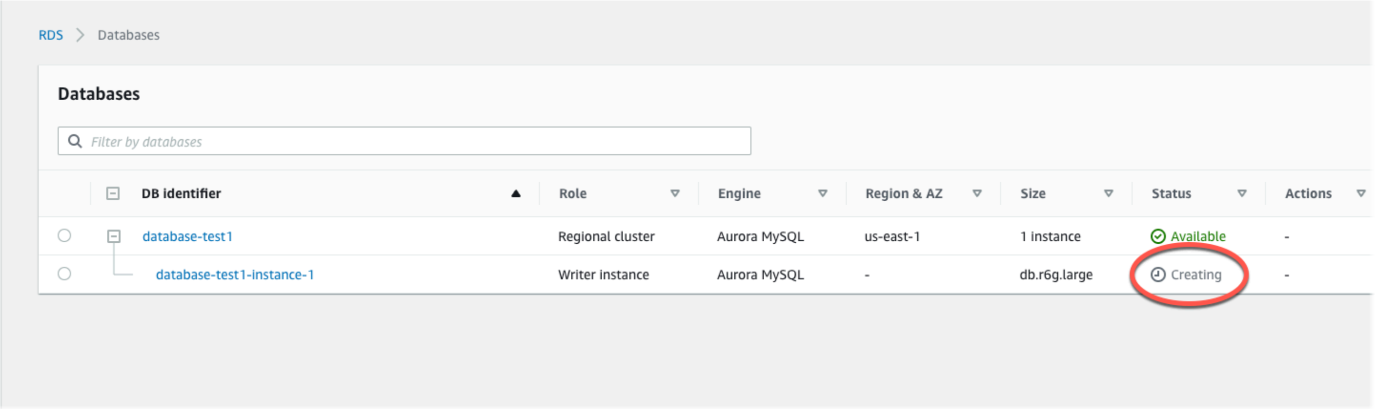
    To connect to the DB instance as the master user, use the user name and password that appear.

Important

You can't view the master user password again. If you don't record it, you might have to change it. If you need to change the master user password after the DB instance is available, you can modify the DB instance to do so. For more information about modifying a DB instance, see Modifying an Amazon Aurora DB cluster.

15. For **Databases**, choose the name of the new Aurora DB cluster.

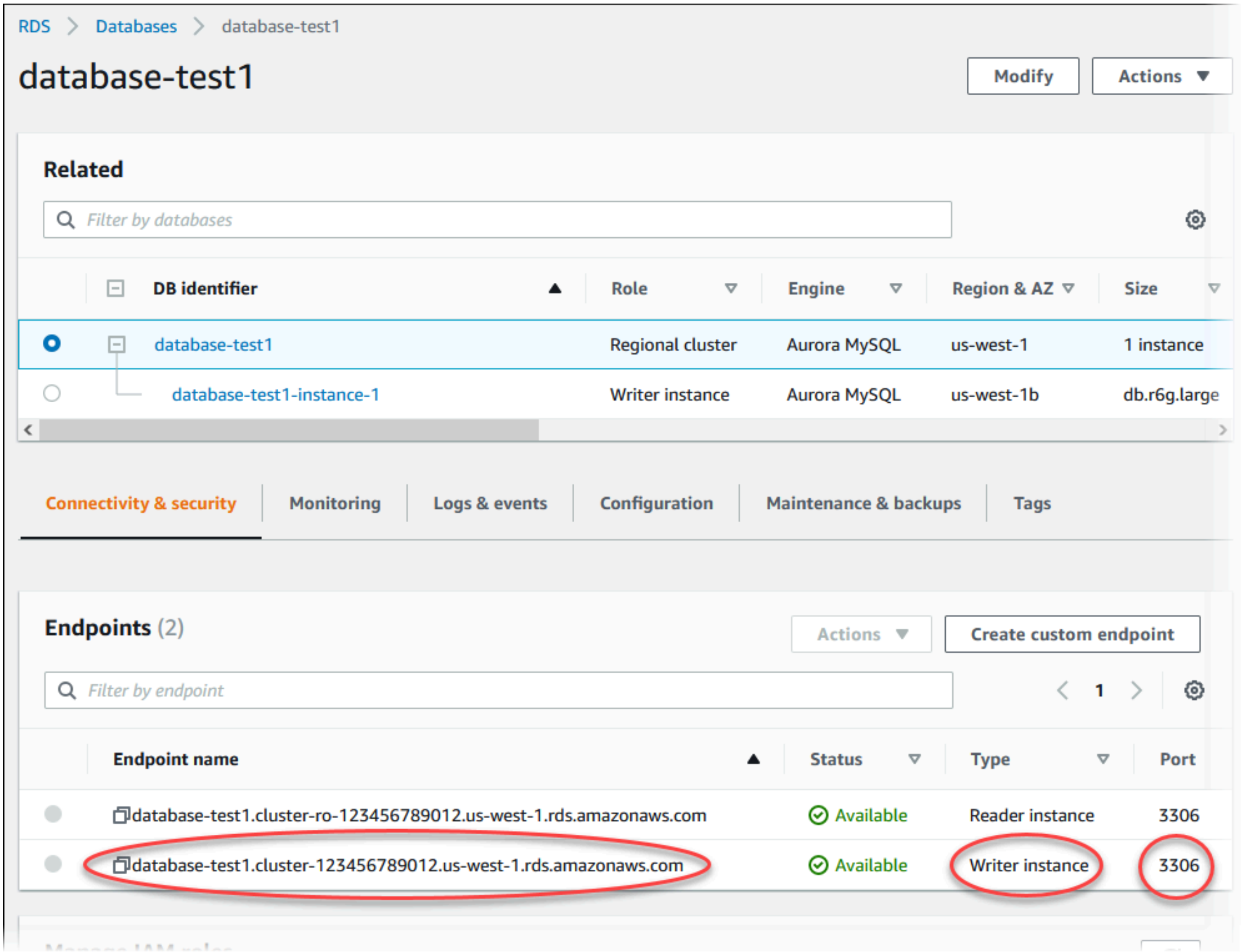
    On the RDS console, the details for new DB cluster appear. The DB cluster and its DB instance have a status of **creating** until the DB cluster is ready to use.



DB identifier	Role	Engine	Region & AZ	Size	Status	Actions
database-test1	Regional cluster	Aurora MySQL	us-east-1	1 instance	Available	-
database-test1-instance-1	Writer instance	Aurora MySQL	-	db.r6g.large	Creating	-

    When the state changes to **available** for both, you can connect to the DB cluster. Depending on the DB instance class and the amount of storage, it can take up to 20 minutes before the new DB cluster is available.

    To view the newly created cluster, choose **Databases** from the navigation pane in the Amazon RDS console. Then choose the DB cluster to show the DB cluster details. For more information, see Viewing an Amazon Aurora DB cluster.



database-test1

Related

	DB identifier	Role	Engine	Region & AZ	Size
●	database-test1	Regional cluster	Aurora MySQL	us-west-1	1 instance
○	database-test1-instance-1	Writer instance	Aurora MySQL	us-west-1b	db.r6g.large

Endpoints (2)

	Endpoint name	Status	Type	Port
○	database-test1.cluster-ro-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Reader instance	3306
○	database-test1.cluster-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Writer instance	3306

On the **Connectivity & security** tab, note the port and the endpoint of the writer DB instance. Use the endpoint and port of the cluster in your JDBC and ODBC connection strings for any application that performs write or read operations.

AWS CLI

> **Note**
>
> Before you can create an Aurora DB cluster using the AWS CLI, you must fulfill the required prerequisites, such as creating a VPC and an RDS DB subnet group. For more information, see DB cluster prerequisites.

You can use the AWS CLI to create an Aurora MySQL DB cluster or an Aurora PostgreSQL DB cluster.

To create an Aurora MySQL DB cluster using the AWS CLI

When you create an Aurora MySQL 8.0-compatible or 5.7-compatible DB cluster or DB instance, you specify `aurora-mysql` for the `--engine` option.

Complete the following steps:

1. Identify the DB subnet group and VPC security group ID for your new DB cluster, and then call the create-db-cluster AWS CLI command to create the Aurora MySQL DB cluster.

   For example, the following command creates a new MySQL 8.0–compatible DB cluster named `sample-cluster`. The cluster uses the default engine version and the Aurora I/O-Optimized storage type.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster --db-cluster-identifier sample-cluster \
       --engine aurora-mysql --engine-version 8.0 \
       --storage-type aurora-iopt1 \
       --master-username user-name --manage-master-user-password \
       --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
   ```

   For Windows:

   ```
   aws rds create-db-cluster --db-cluster-identifier sample-cluster ^
       --engine aurora-mysql --engine-version 8.0 ^
       --storage-type aurora-iopt1 ^
       --master-username user-name --manage-master-user-password ^
       --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
   ```

   The following command creates a new MySQL 5.7–compatible DB cluster named `sample-cluster`. The cluster uses the default engine version and the Aurora Standard storage type.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster --db-cluster-identifier sample-cluster  \
       --engine aurora-mysql --engine-version 5.7 \
       --storage-type aurora \
       --master-username user-name --manage-master-user-password \
       --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
   ```

For Windows:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster sample-cluster  ^
    --engine aurora-mysql --engine-version 5.7 ^
    --storage-type aurora ^
    --master-username user-name --manage-master-user-password ^
    --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
```

2. If you use the console to create a DB cluster, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the AWS CLI to create a DB cluster, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster. Until you create the primary DB instance, the DB cluster endpoints remain in the `Creating` status.

   Call the create-db-instance AWS CLI command to create the primary instance for your DB cluster. Include the name of the DB cluster as the `--db-cluster-identifier` option value.

   > **Note**
   >
   > You can't set the `--storage-type` option for DB instances. You can set it only for DB clusters.

   For example, the following command creates a new MySQL 5.7–compatible or MySQL 8.0–compatible DB instance named `sample-instance`.

   For Linux, macOS, or Unix:

```
aws rds create-db-instance --db-instance-identifier sample-instance \
     --db-cluster-identifier sample-cluster --engine aurora-mysql --db-instance-
class db.r5.large
```

   For Windows:

```
aws rds create-db-instance --db-instance-identifier sample-instance ^
     --db-cluster-identifier sample-cluster --engine aurora-mysql --db-instance-
class db.r5.large
```

To create an Aurora PostgreSQL DB cluster using the AWS CLI

1. Identify the DB subnet group and VPC security group ID for your new DB cluster, and then call the create-db-cluster AWS CLI command to create the Aurora PostgreSQL DB cluster.

   For example, the following command creates a new DB cluster named `sample-cluster`. The cluster uses the default engine version and the Aurora I/O-Optimized storage type.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster --db-cluster-identifier sample-cluster \
       --engine aurora-postgresql \
       --storage-type aurora-iopt1 \
       --master-username user-name --manage-master-user-password \
       --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
   ```

   For Windows:

   ```
   aws rds create-db-cluster --db-cluster-identifier sample-cluster ^
       --engine aurora-postgresql ^
       --storage-type aurora-iopt1 ^
       --master-username user-name --manage-master-user-password ^
       --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
   ```

2. If you use the console to create a DB cluster, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the AWS CLI to create a DB cluster, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster. Until you create the primary DB instance, the DB cluster endpoints remain in the `Creating` status.

   Call the create-db-instance AWS CLI command to create the primary instance for your DB cluster. Include the name of the DB cluster as the `--db-cluster-identifier` option value.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-instance --db-instance-identifier sample-instance \
        --db-cluster-identifier sample-cluster --engine aurora-postgresql --db-
   instance-class db.r5.large
   ```

   For Windows:

```
aws rds create-db-instance --db-instance-identifier sample-instance ^
     --db-cluster-identifier sample-cluster --engine aurora-postgresql --db-
instance-class db.r5.large
```

These examples specify the `--manage-master-user-password` option to generate the master user password and manage it in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

RDS API

> **Note**
>
> Before you can create an Aurora DB cluster using the AWS CLI, you must fulfill the required prerequisites, such as creating a VPC and an RDS DB subnet group. For more information, see DB cluster prerequisites.

Identify the DB subnet group and VPC security group ID for your new DB cluster, and then call the CreateDBCluster operation to create the DB cluster.

When you create an Aurora MySQL version 2 or 3 DB cluster or DB instance, specify `aurora-mysql` for the `Engine` parameter.

When you create an Aurora PostgreSQL DB cluster or DB instance, specify `aurora-postgresql` for the `Engine` parameter.

If you use the console to create a DB cluster, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the RDS API to create a DB cluster, you must explicitly create the primary instance for your DB cluster using the CreateDBInstance. The primary instance is the first instance that is created in a DB cluster. Until you create the primary DB instance, the DB cluster endpoints remain in the `Creating` status.

Creating a primary (writer) DB instance

If you use the AWS Management Console to create a DB cluster, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the AWS CLI or RDS API to create a DB cluster, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster. Until you create the primary DB instance, the DB cluster endpoints remain in the `Creating` status.

For more information, see Creating a DB cluster.

> **Note**
>
> If you have a DB cluster without a writer DB instance, also called a *headless* cluster, you can't use the console to create a writer instance. You must use the AWS CLI or RDS API.

The following example uses the create-db-instance AWS CLI command to create a writer instance for an Aurora PostgreSQL DB cluster named `headless-test`.

```
aws rds create-db-instance \
    --db-instance-identifier no-longer-headless \
    --db-cluster-identifier headless-test \
    --engine aurora-postgresql \
    --db-instance-class db.t4g.medium
```

Settings for Aurora DB clusters

The following table contains details about settings that you choose when you create an Aurora DB cluster.

> **Note**
>
> Additional settings are available if you are creating an Aurora Serverless v1 DB cluster. For information about these settings, see Creating an Aurora Serverless v1 DB cluster. Also, some settings aren't available for Aurora Serverless v1 because of Aurora Serverless v1 limitations. For more information, see Limitations of Aurora Serverless v1.

Console setting	Setting description	CLI option and RDS API parameter
Auto minor version upgrade	Choose **Enable auto minor version upgrade** if you want to	Set this value for every DB instance in your Aurora cluster.

Console setting	Setting description	CLI option and RDS API parameter
	enable your Aurora DB cluster to receive preferred minor version upgrades to the DB engine automatically when they become available. The **Auto minor version upgrade** setting applies to both Aurora PostgreSQL and Aurora MySQL DB clusters. For more information about engine updates for Aurora PostgreSQL, see Amazon Aurora PostgreSQL updates. For more information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.	If any DB instance in your cluster has this setting turned off, the cluster isn't automatically upgraded. Using the AWS CLI, run `create-db-instance` and set the `--auto-minor-version-upgrade|--no-auto-minor-version-upgrade` option. Using the RDS API, call `CreateDBInstance` and set the `AutoMinorVersionUpgrade` parameter.
AWS KMS key	Only available if **Encryption** is set to **Enable encryption**. Choose the AWS KMS key to use for encrypting this DB cluster. For more information, see Encrypting Amazon Aurora resources.	Using the AWS CLI, run `create-db-cluster` and set the `--kms-key-id` option. Using the RDS API, call `CreateDBCluster` and set the `KmsKeyId` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Backtrack	Applies only to Aurora MySQL. Choose **Enable Backtrack** to enable backtracking or **Disable Backtrack** to disable backtracking. Using backtracking, you can rewind a DB cluster to a specific time, without creating a new DB cluster. It is disabled by default. If you enable backtracking, also specify the amount of time that you want to be able to backtrack your DB cluster (the target backtrack window). For more information, see Backtracking an Aurora DB cluster.	Using the AWS CLI, run `create-db-cluster` and set the `--backtrack-window` option. Using the RDS API, call `CreateDBCluster` and set the `BacktrackWindow` parameter.
Certificate authority	The certificate authority (CA) for the server certificate used by the DB instances in the DB cluster. For more information, see Using SSL/TLS to encrypt a connection to a DB cluster.	Using the AWS CLI, run `create-db-instance` and set the `--ca-certificate-identifier` option. Using the RDS API, call `CreateDBInstance` and set the `CACertificateIdentifier` parameter.
Cluster storage configuration	The storage type for the DB cluster: **Aurora I/O-Optimized** or **Aurora Standard**. For more information, see Storage configurations for Amazon Aurora DB clusters.	Using the AWS CLI, run `create-db-cluster` and set the `--storage-type` option. Using the RDS API, call `CreateDBCluster` and set the `StorageType` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Copy tags to snapshots	Choose this option to copy any DB instance tags to a DB snapshot when you create a snapshot. For more information, see Tagging Amazon RDS resources.	Using the AWS CLI, run `create-db-cluster` and set the `--copy-tags-to-snapshot | --no-copy-tags-to-snapshot` option. Using the RDS API, call `CreateDBCluster` and set the `CopyTagsToSnapshot` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Database authentication	The database authentication you want to use. For MySQL: - Choose **Password authentication** to authenticate database users with database passwords only. - Choose **Password and IAM database authentication** to authenticate database users with database passwords and user credentials through IAM users and roles. For more information, see IAM database authentication. For PostgreSQL: - Choose **IAM database authentication** to authenticate database users with database passwords and user credentials through users and roles. For more information, see IAM database authentication. - Choose **Kerberos authentication** to authenticate database passwords and user credentials using Kerberos authentication. For more information, see Using	To use IAM database authentication with the AWS CLI, run `create-db-cluster` and set the `--enable-iam-database-authentication | --no-enable-iam-database-authentication` option. To use IAM database authentication with the RDS API, call `CreateDBCluster` and set the `EnableIAMDatabaseAuthentication` parameter. To use Kerberos authentication with the AWS CLI, run `create-db-cluster` and set the `--domain` and `--domain-iam-role-name` options. To use Kerberos authentication with the RDS API, call `CreateDBCluster` and set the `Domain` and `DomainIAMRoleName` parameters.

Console setting	Setting description	CLI option and RDS API parameter
	Kerberos authentication with Aurora PostgreSQL.	
Database port	Specify the port for applications and utilities to use to access the database. Aurora MySQL DB clusters default to the default MySQL port, 3306, and Aurora PostgreSQL DB clusters default to the default PostgreSQL port, 5432. The firewalls at some companies block connections to these default ports. If your company firewall blocks the default port, choose another port for the new DB cluster.	Using the AWS CLI, run `create-db-cluster` and set the `--port` option. Using the RDS API, call `CreateDBCluster` and set the `Port` parameter.

Console setting	Setting description	CLI option and RDS API parameter
DB cluster identifier	Enter a name for your DB cluster that is unique for your account in the AWS Region that you chose. This identifier is used in the cluster endpoint address for your DB cluster. For information on the cluster endpoint, see Amazon Aurora connection management. The DB cluster identifier has the following constraints: • It must contain from 1 to 63 alphanumeric characters or hyphens. • Its first character must be a letter. • It cannot end with a hyphen or contain two consecutive hyphens. • It must be unique for all DB clusters per AWS account, per AWS Region.	Using the AWS CLI, run `create-db-cluster` and set the `--db-cluster-identifier` option. Using the RDS API, call `CreateDBCluster` and set the `DBClusterIdentifier` parameter.
DB cluster parameter group	Choose a DB cluster parameter group. Aurora has a default DB cluster parameter group you can use, or you can create your own DB cluster parameter group. For more information about DB cluster parameter groups, see Working with parameter groups.	Using the AWS CLI, run `create-db-cluster` and set the `--db-cluster-parameter-group-name` option. Using the RDS API, call `CreateDBCluster` and set the `DBClusterParameterGroupName` parameter.

Console setting	Setting description	CLI option and RDS API parameter
DB instance class	Applies only to the provisioned capacity type. Choose a DB instance class that defines the processing and memory requirements for each instance in the DB cluster. For more information about DB instance classes, see Aurora DB instance classes.	Set this value for every DB instance in your Aurora cluster. Using the AWS CLI, run `create-db-instance` and set the `--db-instance-class` option. Using the RDS API, call `CreateDBInstance` and set the `DBInstanceClass` parameter.
DB parameter group	Choose a parameter group. Aurora has a default parameter group you can use, or you can create your own parameter group. For more information about parameter groups, see Working with parameter groups.	Set this value for every DB instance in your Aurora cluster. Using the AWS CLI, run `create-db-instance` and set the `--db-parameter-group-name` option. Using the RDS API, call `CreateDBInstance` and set the `DBParameterGroupName` parameter.

Console setting	Setting description	CLI option and RDS API parameter
DB subnet group	The DB subnet group you want to use for the DB cluster. Select **Choose existing** to use an existing DB subnet group. Then choose the required subnet group from the **Existing DB subnet groups** dropdown list. Choose **Automatic setup** to let RDS select a compatible DB subnet group. If none exist, RDS creates a new subnet group for your cluster. For more information, see DB cluster prerequisites.	Using the AWS CLI, run `create-db-cluster` and set the `--db-subnet-group-name` option. Using the RDS API, call `CreateDBCluster` and set the `DBSubnetGroupName` parameter.
Enable deletion protection	Choose **Enable deletion protection** to prevent your DB cluster from being deleted. If you create a production DB cluster with the console, deletion protection is enabled by default.	Using the AWS CLI, run `create-db-cluster` and set the `--deletion-protection | --no-deletion-protection` option. Using the RDS API, call `CreateDBCluster` and set the `DeletionProtection` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Enable encryption	Choose `Enable encryption` to enable encryption at rest for this DB cluster. For more information, see [Encrypting Amazon Aurora resources]().	Using the AWS CLI, run [`create-db-cluster`]() and set the `--storage-encrypted | --no-storage-encrypted` option. Using the RDS API, call [`CreateDBCluster`]() and set the `StorageEncrypted` parameter.
Enable Enhanced Monitoring	Choose **Enable enhanced monitoring** to enable gathering metrics in real time for the operating system that your DB cluster runs on. For more information, see [Monitoring OS metrics with Enhanced Monitoring]().	Set these values for every DB instance in your Aurora cluster. Using the AWS CLI, run [`create-db-instance`]() and set the `--monitoring-interval` and `--monitoring-role-arn` options. Using the RDS API, call [`CreateDBInstance`]() and set the `MonitoringInterval` and `MonitoringRoleArn` parameters.
Enable the RDS Data API	Choose **Enable the RDS Data API** to enable RDS Data API (Data API). Data API provides a secure HTTP endpoint for running SQL statements without managing connections. For more information, see [*Using RDS Data API*]().	Using the AWS CLI, run [`create-db-cluster`]() and set the `--enable-http-endpoint | --no-enable-http-endpoint` option. Using the RDS API, call [`CreateDBCluster`]() and set the `EnableHttpEndpoint` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Engine type	Choose the database engine to be used for this DB cluster.	Using the AWS CLI, run create-db-cluster and set the --engine option. Using the RDS API, call CreateDBCluster and set the Engine parameter.
Engine version	Applies only to the provisioned capacity type. Choose the version number of your DB engine.	Using the AWS CLI, run create-db-cluster and set the --engine-version option. Using the RDS API, call CreateDBCluster and set the EngineVersion parameter.
Failover priority	Choose a failover priority for the instance. If you don't choose a value, the default is **tier-1**. This priority determines the order in which Aurora Replicas are promoted when recovering from a primary instance failure. For more information, see Fault tolerance for an Aurora DB cluster.	Set this value for every DB instance in your Aurora cluster. Using the AWS CLI, run create-db-instance and set the --promotion-tier option. Using the RDS API, call CreateDBInstance and set the PromotionTier parameter.

Console setting	Setting description	CLI option and RDS API parameter
Initial database name	Enter a name for your default database. If you don't provide a name for an Aurora MySQL DB cluster, Amazon RDS doesn't create a database on the DB cluster you are creating. If you don't provide a name for an Aurora PostgreSQL DB cluster, Amazon RDS creates a database named `postgres`. For Aurora MySQL, the default database name has these constraints: • It must contain 1–64 alphanumeric characters. • It can't be a word reserved by the database engine. For Aurora PostgreSQL, the default database name has these constraints: • It must contain 1–63 alphanumeric characters. • It must begin with a letter. Subsequent characters can be letters, underscores, or digits (0–9). • It can't be a word reserved by the database engine.	Using the AWS CLI, run `create-db-cluster` and set the `--database-name` option. Using the RDS API, call `CreateDBCluster` and set the `DatabaseName` parameter.

Console setting	Setting description	CLI option and RDS API parameter
	To create additional databases, connect to the DB cluster and use the SQL command CREATE DATABASE. For more information about connecting to the DB cluster, see Connecting to an Amazon Aurora DB cluster.	
Log exports	In the **Log exports** section, choose the logs that you want to start publishing to Amazon CloudWatch Logs. For more information about publishing Aurora MySQL logs to CloudWatch Logs, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs. For more information about publishing Aurora PostgreSQL logs to CloudWatch Logs, see Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs.	Using the AWS CLI, run `create-db-cluster` and set the `--enable-cloudwatch-logs-exports` option. Using the RDS API, call `CreateDBCluster` and set the `EnableCloudwatchLogsExports` parameter.
Maintenance window	Choose **Select window** and specify the weekly time range during which system maintenance can occur. Or choose **No preference** for Amazon RDS to assign a period randomly.	Using the AWS CLI, run `create-db-cluster` and set the `--preferred-maintenance-window` option. Using the RDS API, call `CreateDBCluster` and set the `PreferredMaintenanceWindow` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Manage master credentials in AWS Secrets Manager	Select **Manage master credentials in AWS Secrets Manager** to manage the master user password in a secret in Secrets Manager. Optionally, choose a KMS key to use to protect the secret. Choose from the KMS keys in your account, or enter the key from a different account. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.	Using the AWS CLI, run `create-db-cluster` and set the `--manage-master-user-password | --no-manage-master-user-password` and `--master-user-secret-kms-key-id` options. Using the RDS API, call `CreateDBCluster` and set the `ManageMasterUserPassword` and `MasterUserSecretKmsKeyId` parameters.
Master password	Enter a password to log on to your DB cluster: • For Aurora MySQL, the password must contain 8–41 printable ASCII characters. • For Aurora PostgreSQL, it must contain 8–99 printable ASCII characters. • It can't contain `/`, `"`, `@`, or a space.	Using the AWS CLI, run `create-db-cluster` and set the `--master-user-password` option. Using the RDS API, call `CreateDBCluster` and set the `MasterUserPassword` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Master username	Enter a name to use as the master user name to log on to your DB cluster: - For Aurora MySQL, the name must contain 1–16 alphanume ric characters. - For Aurora PostgreSQL, it must contain 1–63 alphanumeric characters. - The first character must be a letter. - The name can't be a word reserved by the database engine. You can't change the master user name after the DB cluster is created.	Using the AWS CLI, run `create-db-cluster` and set the `--master-username` option. Using the RDS API, call `CreateDBCluster` and set the `MasterUsername` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Multi-AZ deployment	Applies only to the provisioned capacity type. Determine if you want to create Aurora Replicas in other Availability Zones for failover support. If you choose **Create Replica in Different Zone**, then Amazon RDS creates an Aurora Replica for you in your DB cluster in a different Availability Zone than the primary instance for your DB cluster. For more information about multiple Availability Zones, see Regions and Availability Zones.	Using the AWS CLI, run `create-db-cluster` and set the `--availability-zones` option. Using the RDS API, call `CreateDBCluster` and set the `AvailabilityZones` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Network type	The IP addressing protocols supported by the DB cluster. **IPv4** to specify that resources can communicate with the DB cluster only over the IPv4 addressing protocol. **Dual-stack mode** to specify that resources can communicate with the DB cluster over IPv4, IPv6, or both. Use dual-stack mode if you have any resources that must communicate with your DB cluster over the IPv6 addressing protocol. To use dual-stack mode, make sure at least two subnets spanning two Availability Zones that support both the IPv4 and IPv6 network protocol. Also, make sure you associate an IPv6 CIDR block with subnets in the DB subnet group you specify. For more information, see Amazon Aurora IP addressing.	Using the AWS CLI, run `create-db-cluster` and set the `-network-type` option. Using the RDS API, call `CreateDBCluster` and set the `NetworkType` parameter.

Console setting	Setting description	CLI option and RDS API parameter
Public access	Choose **Publicly accessible** to give the DB cluster a public IP address, or choose **Not publicly accessible**. The instances in your DB cluster can be a mix of both public and private DB instances. For more information about hiding instances from public access, see Hiding a DB cluster in a VPC from the internet. To connect to a DB instance from outside of its Amazon VPC, the DB instance must be publicly accessible, access must be granted using the inbound rules of the DB instance's security group, and other requirements must be met. For more information, see Can't connect to Amazon RDS DB instance. If your DB instance is isn't publicly accessible, you can also use an AWS Site-to-Site VPN connection or an AWS Direct Connect connection to access it from a private network. For more information, see Internetwork traffic privacy.	Set this value for every DB instance in your Aurora cluster. Using the AWS CLI, run `create-db-instance` and set the `--publicly-accessible | --no-publicly-accessible` option. Using the RDS API, call `CreateDBInstance` and set the `PubliclyAccessible` parameter.

Console setting	Setting description	CLI option and RDS API parameter
RDS Extended Support	Select **Enable RDS Extended Support** to allow supported major engine versions to continue running past the Aurora end of standard support date. When you create a DB cluster, Amazon Aurora defaults to RDS Extended Support. To prevent the creation of a new DB cluster after the Aurora end of standard support date and to avoid charges for RDS Extended Support, disable this setting. Your existing DB clusters won't incur charges until the RDS Extended Support pricing start date. For more information, see Using Amazon RDS Extended Support.	Using the AWS CLI, run `create-db-cluster` and set the `--engine-lifecycle-support` option. Using the RDS API, call `CreateDBCluster` and set the `EngineLifecycleSupport` parameter.
RDS Proxy	Choose **Create an RDS Proxy** to create a proxy for your DB cluster. Amazon RDS automatically creates an IAM role and a Secrets Manager secret for the proxy. For more information, see Using Amazon RDS Proxy for Aurora.	Not available when creating a DB cluster.

Console setting	Setting description	CLI option and RDS API parameter
Retention period	Choose the length of time, from 1 to 35 days, that Aurora retains backup copies of the database. Backup copies can be used for point-in-time restores (PITR) of your database down to the second.	Using the AWS CLI, run `create-db-cluster` and set the `--backup-retention-period` option. Using the RDS API, call `CreateDBCluster` and set the `BackupRetentionPeriod` parameter.
Turn on DevOps Guru	Choose **Turn on DevOps Guru** to turn on Amazon DevOps Guru for your Aurora database. For DevOps Guru for RDS to provide detailed analysis of performance anomalies, Performance Insights must be turned on. For more information, see Setting up DevOps Guru for RDS.	You can turn on DevOps Guru for RDS from within the RDS console, but not by using the RDS API or CLI. For more information about turning on DevOps Guru, see the *Amazon DevOps Guru User Guide*.

Console setting	Setting description	CLI option and RDS API parameter
Turn on Performance Insights	Choose **Turn on Performance Insights** to turn on Amazon RDS Performance Insights. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.	Set these values for every DB instance in your Aurora cluster. Using the AWS CLI, run `create-db-instance` and set the `--enable-performance-insights | --no-enable-performance-insights`, `--performance-insights-kms-key-id`, and `--performance-insights-retention-period` options. Using the RDS API, call `CreateDBInstance` and set the `EnablePerformanceInsights`, `PerformanceInsightsKMSKeyId`, and `PerformanceInsightsRetentionPeriod` parameters.
Virtual Private Cloud (VPC)	Choose the VPC to host the DB cluster. Choose **Create a New VPC** to have Amazon RDS create a VPC for you. For more information, see DB cluster prerequisites.	For the AWS CLI and API, you specify the VPC security group IDs.

Console setting	Setting description	CLI option and RDS API parameter
VPC security group (firewall)	Choose **Create new** to have Amazon RDS create a VPC security group for you. Or choose **Choose existing** and specify one or more VPC security groups to secure network access to the DB cluster. When you choose **Create new** in the RDS console, a new security group is created with an inbound rule that allows access to the DB instance from the IP address detected in your browser. For more information, see DB cluster prerequisites.	Using the AWS CLI, run `create-db-cluster` and set the `--vpc-security-group-ids` option. Using the RDS API, call `CreateDBCluster` and set the `VpcSecurityGroupIds` parameter.

Settings that don't apply to Amazon Aurora for DB clusters

The following settings in the AWS CLI command `create-db-cluster` and the RDS API operation `CreateDBCluster` don't apply to Amazon Aurora DB clusters.

Note

The AWS Management Console doesn't show these settings for Aurora DB clusters.

AWS CLI setting	RDS API setting
`--allocated-storage`	`AllocatedStorage`
`--auto-minor-version-upgrade | --no-auto-minor-version-upgrade`	`AutoMinorVersionUpgrade`

AWS CLI setting	RDS API setting
`--db-cluster-instance-class`	`DBClusterInstanceClass`
`--enable-performance-insights | --no-enable-performance-insights`	`EnablePerformanceInsights`
`--iops`	`Iops`
`--monitoring-interval`	`MonitoringInterval`
`--monitoring-role-arn`	`MonitoringRoleArn`
`--option-group-name`	`OptionGroupName`
`--performance-insights-kms-key-id`	`PerformanceInsightsKMSKeyId`
`--performance-insights-retention-period`	`PerformanceInsightsRetentionPeriod`
`--publicly-accessible | --no-publicly-accessible`	`PubliclyAccessible`

Settings that don't apply to Amazon Aurora DB instances

The following settings in the AWS CLI command `create-db-instance` and the RDS API operation `CreateDBInstance` don't apply to DB instances Amazon Aurora DB cluster.

> **Note**
>
> The AWS Management Console doesn't show these settings for Aurora DB instances.

AWS CLI setting	RDS API setting
`--allocated-storage`	`AllocatedStorage`

AWS CLI setting	RDS API setting
`--availability-zone`	`AvailabilityZone`
`--backup-retention-period`	`BackupRetentionPeriod`
`--backup-target`	`BackupTarget`
`--character-set-name`	`CharacterSetName`
`--character-set-name`	`CharacterSetName`
`--custom-iam-instance-profile`	`CustomIamInstanceProfile`
`--db-security-groups`	`DBSecurityGroups`
`--deletion-protection | --no-deletion-protection`	`DeletionProtection`
`--domain`	`Domain`
`--domain-iam-role-name`	`DomainIAMRoleName`
`--enable-cloudwatch-logs-exports`	`EnableCloudwatchLogsExports`
`--enable-customer-owned-ip | --no-enable-customer-owned-ip`	`EnableCustomerOwnedIp`
`--enable-iam-database-authentication | --no-enable-iam-database-authentication`	`EnableIAMDatabaseAuthentication`
`--engine-version`	`EngineVersion`
`--iops`	`Iops`
`--kms-key-id`	`KmsKeyId`
`--master-username`	`MasterUsername`
`--master-user-password`	`MasterUserPassword`

AWS CLI setting	RDS API setting
`--max-allocated-storage`	`MaxAllocatedStorage`
`--multi-az | --no-multi-az`	`MultiAZ`
`--nchar-character-set-name`	`NcharCharacterSetName`
`--network-type`	`NetworkType`
`--option-group-name`	`OptionGroupName`
`--preferred-backup-window`	`PreferredBackupWindow`
`--processor-features`	`ProcessorFeatures`
`--storage-encrypted | --no-storage-encrypted`	`StorageEncrypted`
`--storage-type`	`StorageType`
`--tde-credential-arn`	`TdeCredentialArn`
`--tde-credential-password`	`TdeCredentialPassword`
`--timezone`	`Timezone`
`--vpc-security-group-ids`	`VpcSecurityGroupIds`

Creating Amazon Aurora resources with AWS CloudFormation

Amazon Aurora is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as DB clusters and DB cluster parameter groups), and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your Aurora resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

Aurora and AWS CloudFormation templates

To provision and configure resources for Aurora and related services, you must understand AWS CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see What is AWS CloudFormation Designer? in the *AWS CloudFormation User Guide*.

Aurora supports creating resources in AWS CloudFormation. For more information, including examples of JSON and YAML templates for these resources, see the RDS resource type reference in the *AWS CloudFormation User Guide*.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- AWS CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Connecting to an Amazon Aurora DB cluster

You can connect to an Aurora DB cluster using the same tools that you use to connect to a MySQL or PostgreSQL database. You specify a connection string with any script, utility, or application that connects to a MySQL or PostgreSQL DB instance. You use the same public key for Secure Sockets Layer (SSL) connections.

In the connection string, you typically use the host and port information from special endpoints associated with the DB cluster. With these endpoints, you can use the same connection parameters regardless of how many DB instances are in the cluster. You also use the host and port information from a specific DB instance in your Aurora DB cluster for specialized tasks, such as troubleshooting.

> **Note**
>
> For Aurora Serverless DB clusters, you connect to the database endpoint rather than to the DB instance. You can find the database endpoint for an Aurora Serverless DB cluster on the **Connectivity & security** tab of the AWS Management Console. For more information, see Using Amazon Aurora Serverless v1.

Regardless of the Aurora DB engine and specific tools you use to work with the DB cluster or instance, the endpoint must be accessible. An Aurora DB cluster can be created only in a virtual private cloud (VPC) based on the Amazon VPC service. That means that you access the endpoint from either inside the VPC or outside the VPC using one of the following approaches.

- **Access the Aurora DB cluster inside the VPC** – Enable access to the Aurora DB cluster through the VPC. You do so by editing the Inbound rules on the Security group for the VPC to allow access to your specific Aurora DB cluster. To learn more, including how to configure your VPC for different Aurora DB cluster scenarios, see Amazon Virtual Private Cloud VPCs and Amazon Aurora.
- **Access the Aurora DB cluster outside the VPC** – To access an Aurora DB cluster from outside the VPC, use the public endpoint address of the DB cluster.

For more information, see Troubleshooting Aurora connection failures.

Contents

Connecting to Aurora DB clusters with the AWS drivers

The AWS suite of drivers has been designed to provide support for faster switchover and failover times, and authentication with AWS Secrets Manager, AWS Identity and Access Management (IAM), and Federated Identity. The AWS drivers rely on monitoring DB cluster status and being aware of the cluster topology to determine the new writer. This approach reduces switchover and failover times to single-digit seconds, compared to tens of seconds for open-source drivers.

The following table lists the features supported for each of the drivers. As new service features are introduced, the goal of the AWS suite of drivers is to have built-in support for these service features.

Feature	AWS JDBC Driver	AWS Python Driver
Failover support	Yes	Yes
Enhanced failover monitoring	Yes	Yes
Read/write splitting	Yes	Yes
Aurora connection tracker	Yes	Yes
Driver metadata connection	Yes	N/A

Feature	AWS JDBC Driver	AWS Python Driver
Telemetry	Yes	Yes
Secrets Manager	Yes	Yes
IAM authentication	Yes	Yes
Federated Identity (AD FS)	Yes	Yes
Federated Identity (Okta)	Yes	No

For more information on the AWS drivers, see the corresponding language driver for your Aurora MySQL or Aurora PostgreSQL DB cluster.

Connecting to an Amazon Aurora MySQL DB cluster

To authenticate to your Aurora MySQL DB cluster, you can use either MySQL user name and password authentication or AWS Identity and Access Management (IAM) database authentication. For more information on using MySQL user name and password authentication, see Access control and account management in the MySQL documentation. For more information on using IAM database authentication, see IAM database authentication.

When you have a connection to your Amazon Aurora DB cluster with MySQL 8.0 compatibility, you can run SQL commands that are compatible with MySQL version 8.0. The minimum compatible version is MySQL 8.0.23. For more information about MySQL 8.0 SQL syntax, see the MySQL 8.0 reference manual. For information about limitations that apply to Aurora MySQL version 3, see Comparison of Aurora MySQL version 3 and MySQL 8.0 Community Edition.

When you have a connection to your Amazon Aurora DB cluster with MySQL 5.7 compatibility, you can run SQL commands that are compatible with MySQL version 5.7. For more information about MySQL 5.7 SQL syntax, see the MySQL 5.7 reference manual. For information about limitations that apply to Aurora MySQL 5.7, see Aurora MySQL version 2 compatible with MySQL 5.7.

Note

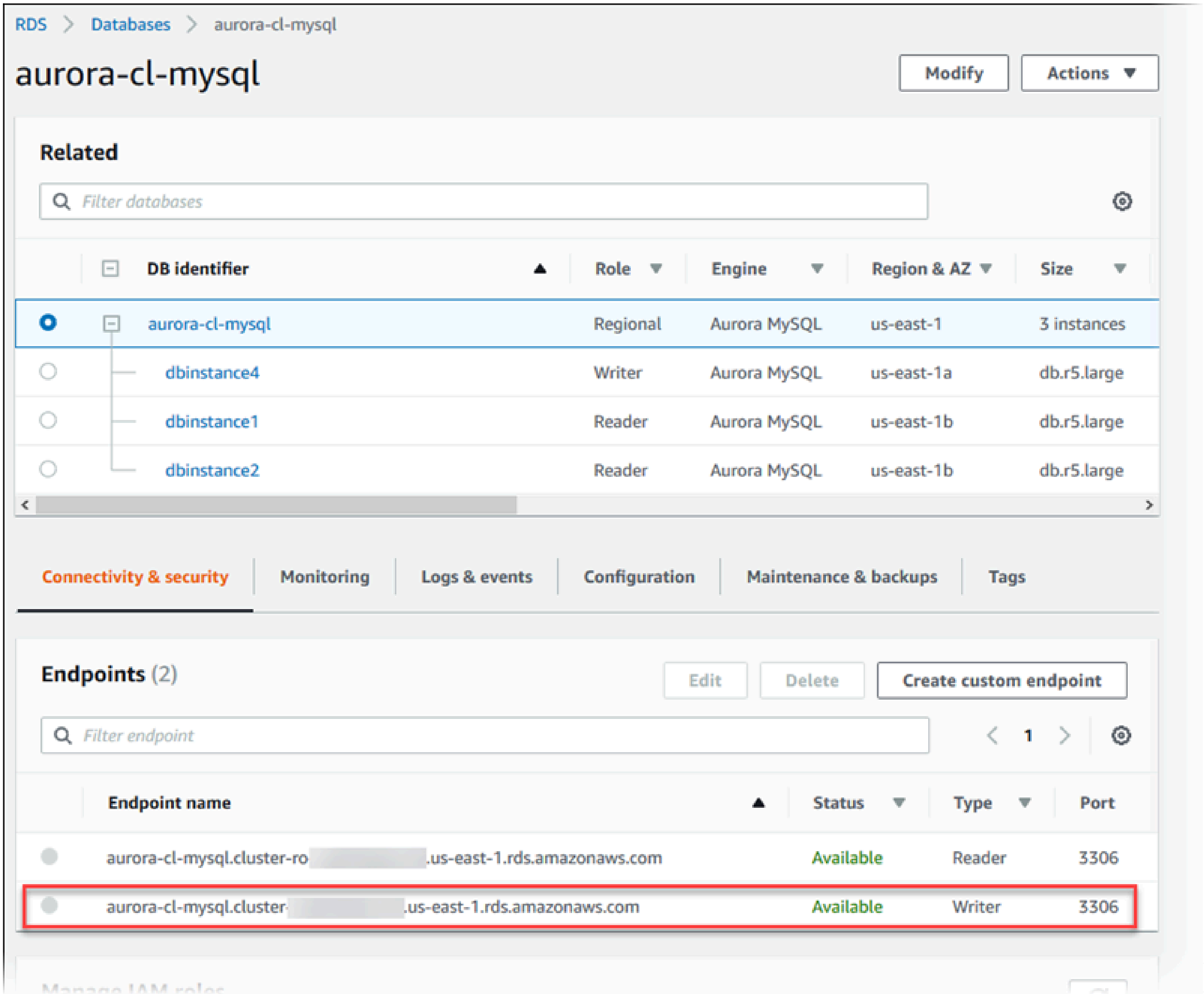
For a helpful and detailed guide on connecting to an Amazon Aurora MySQL DB cluster, you can see the Aurora connection management handbook.

In the details view for your DB cluster, you can find the cluster endpoint, which you can use in your MySQL connection string. The endpoint is made up of the domain name and port for your DB cluster. For example, if an endpoint value is `mycluster.cluster-123456789012.us-east-1.rds.amazonaws.com:3306`, then you specify the following values in a MySQL connection string:

- For host or host name, specify `mycluster.cluster-123456789012.us-east-1.rds.amazonaws.com`
- For port, specify `3306` or the port value you used when you created the DB cluster

The cluster endpoint connects you to the primary instance for the DB cluster. You can perform both read and write operations using the cluster endpoint. Your DB cluster can also have up to 15 Aurora Replicas that support read-only access to the data in your DB cluster. The primary instance and each Aurora Replica has a unique endpoint that is independent of the cluster endpoint and allows you to connect to a specific DB instance in the cluster directly. The cluster endpoint always points to the primary instance. If the primary instance fails and is replaced, then the cluster endpoint points to the new primary instance.

To view the cluster endpoint (writer endpoint), choose **Databases** on the Amazon RDS console and choose the name of the DB cluster to show the DB cluster details.



RDS > Databases > aurora-cl-mysql

aurora-cl-mysql

Related

	DB identifier ▲	Role ▼	Engine ▼	Region & AZ ▼	Size ▼
●	⊟ aurora-cl-mysql	Regional	Aurora MySQL	us-east-1	3 instances
○	dbinstance4	Writer	Aurora MySQL	us-east-1a	db.r5.large
○	dbinstance1	Reader	Aurora MySQL	us-east-1b	db.r5.large
○	dbinstance2	Reader	Aurora MySQL	us-east-1b	db.r5.large

Endpoints (2)

	Endpoint name ▲	Status ▼	Type ▼	Port
○	aurora-cl-mysql.cluster-ro[redacted].us-east-1.rds.amazonaws.com	Available	Reader	3306
○	aurora-cl-mysql.cluster-[redacted].us-east-1.rds.amazonaws.com	Available	Writer	3306

Manage IAM roles

Topics

- Connection utilities for Aurora MySQL
- Connecting to Aurora MySQL with the MySQL utility
- Connecting to Aurora MySQL with the Amazon Web Services (AWS) JDBC Driver
- Connecting to Aurora MySQL with the Amazon Web Services (AWS) Python Driver
- Connecting to Aurora MySQL using SSL

Connection utilities for Aurora MySQL

Some connection utilities you can use are the following:

- **Command line** – You can connect to an Amazon Aurora DB cluster by using tools like the MySQL command line utility. For more information on using the MySQL utility, see mysql — the MySQL command-line client in the MySQL documentation.
- **GUI** – You can use the MySQL Workbench utility to connect by using a UI interface. For more information, see the Download MySQL workbench page.
- **AWS drivers**:
  - Connecting to Aurora MySQL with the Amazon Web Services (AWS) JDBC Driver
  - Connecting to Aurora MySQL with the Amazon Web Services (AWS) Python Driver

Connecting to Aurora MySQL with the MySQL utility

Use the following procedure. It assumes that you configured your DB cluster in a private subnet in your VPC. You connect using an Amazon EC2 instance that you configured according to the tutorials in Tutorial: Create a web server and an Amazon Aurora DB cluster.

> **Note**
>
> This procedure doesn't require installing the web server in the tutorial, but it does require installing MariaDB 10.5.

To connect to a DB cluster using the MySQL utility

1. Log in to the EC2 instance that you're using to connect to your DB cluster.

   You should see output similar to the following.

   ```
   Last login: Thu Jun 23 13:32:52 2022 from xxx.xxx.xxx.xxx

          __|  __|_  )
          _|  (     /   Amazon Linux 2 AMI
         ___|\___|___|

   https://aws.amazon.com/amazon-linux-2/
   ```

```
[ec2-user@ip-10-0-xxx.xxx ~]$
```

2. Type the following command at the command prompt to connect to the primary DB instance of your DB cluster.

   For the -h parameter, substitute the endpoint DNS name for your primary instance. For the -u parameter, substitute the user ID of a database user account.

   ```
   mysql -h primary-instance-endpoint.AWS_account.AWS_Region.rds.amazonaws.com -P 3306
    -u database_user -p
   ```

   For example:

   ```
   mysql -h my-aurora-cluster-instance.c1xy5example.123456789012.eu-
   central-1.rds.amazonaws.com -P 3306 -u admin -p
   ```

3. Enter the password for the database user.

   You should see output similar to the following.

   ```
   Welcome to the MariaDB monitor.  Commands end with ; or \g.
   Your MySQL connection id is 1770
   Server version: 8.0.23 Source distribution

   Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

   Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

   MySQL [(none)]>
   ```

4. Enter your SQL commands.

Connecting to Aurora MySQL with the Amazon Web Services (AWS) JDBC Driver

The Amazon Web Services (AWS) JDBC Driver is designed as an advanced JDBC wrapper. This wrapper is complementary to and extends the functionality of an existing JDBC driver to help applications take advantage of the features of clustered databases such as Aurora MySQL. The driver is drop-in compatible with the community MySQL Connector/J driver and the community MariaDB Connector/J driver.

To install the AWS JDBC Driver, append the AWS JDBC Driver .jar file (located in the application CLASSPATH), and keep references to the respective community driver. Update the respective connection URL prefix as follows:

- jdbc:mysql:// to jdbc:aws-wrapper:mysql://
- jdbc:mariadb:// to jdbc:aws-wrapper:mariadb://

For more information about the AWS JDBC Driver and complete instructions for using it, see the Amazon Web Services (AWS) JDBC Driver GitHub repository.

> **Note**
>
> Version 3.0.3 of the MariaDB Connector/J utility drops support for Aurora DB clusters, so we highly recommend moving to the AWS JDBC Driver.

Connecting to Aurora MySQL with the Amazon Web Services (AWS) Python Driver

The Amazon Web Services (AWS) Python Driver is designed as an advanced Python wrapper. This wrapper is complementary to and extends the functionality of the open-source Psycopg driver. The AWS Python Driver supports Python versions 3.8 and higher. You can install the aws-advanced-python-wrapper package using the pip command, along with the psycopg open-source packages.

For more information about the AWS Python Driver and complete instructions for using it, see the Amazon Web Services (AWS) Python Driver GitHub repository.

Connecting to Aurora MySQL using SSL

You can use SSL encryption on connections to an Aurora MySQL DB instance. For information, see Using TLS with Aurora MySQL DB clusters.

To connect using SSL, use the MySQL utility as described in the following procedure. If you are using IAM database authentication, you must use an SSL connection. For information, see IAM database authentication.

> **Note**
>
> To connect to the cluster endpoint using SSL, your client connection utility must support Subject Alternative Names (SAN). If your client connection utility doesn't support SAN, you can connect directly to the instances in your Aurora DB cluster. For more information on Aurora endpoints, see Amazon Aurora connection management.

To connect to a DB cluster with SSL using the MySQL utility

1. Download the public key for the Amazon RDS signing certificate.

   For information about downloading certificates, see Using SSL/TLS to encrypt a connection to a DB cluster.

2. Type the following command at a command prompt to connect to the primary instance of a DB cluster with SSL using the MySQL utility. For the `-h` parameter, substitute the endpoint DNS name for your primary instance. For the `-u` parameter, substitute the user ID of a database user account. For the `--ssl-ca` parameter, substitute the SSL certificate file name as appropriate. Type the master user password when prompted.

   ```
   mysql -h mycluster-primary.123456789012.us-east-1.rds.amazonaws.com -u
   admin_user -p --ssl-ca=[full path]global-bundle.pem --ssl-verify-server-
   cert
   ```

You should see output similar to the following.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 350
Server version: 8.0.26-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

For general instructions on constructing RDS for MySQL connection strings and finding the public key for SSL connections, see Connecting to a DB instance running the MySQL database engine.

Connecting to an Amazon Aurora PostgreSQL DB cluster

You can connect to a DB instance in your Amazon Aurora PostgreSQL DB cluster using the same tools that you use to connect to a PostgreSQL database. As part of this, you use the same public key for Secure Sockets Layer (SSL) connections. You can use the endpoint and port information from the primary instance or Aurora Replicas in your Aurora PostgreSQL DB cluster in the connection string of any script, utility, or application that connects to a PostgreSQL DB instance. In the connection string, specify the DNS address from the primary instance or Aurora Replica endpoint as the host parameter. Specify the port number from the endpoint as the port parameter.

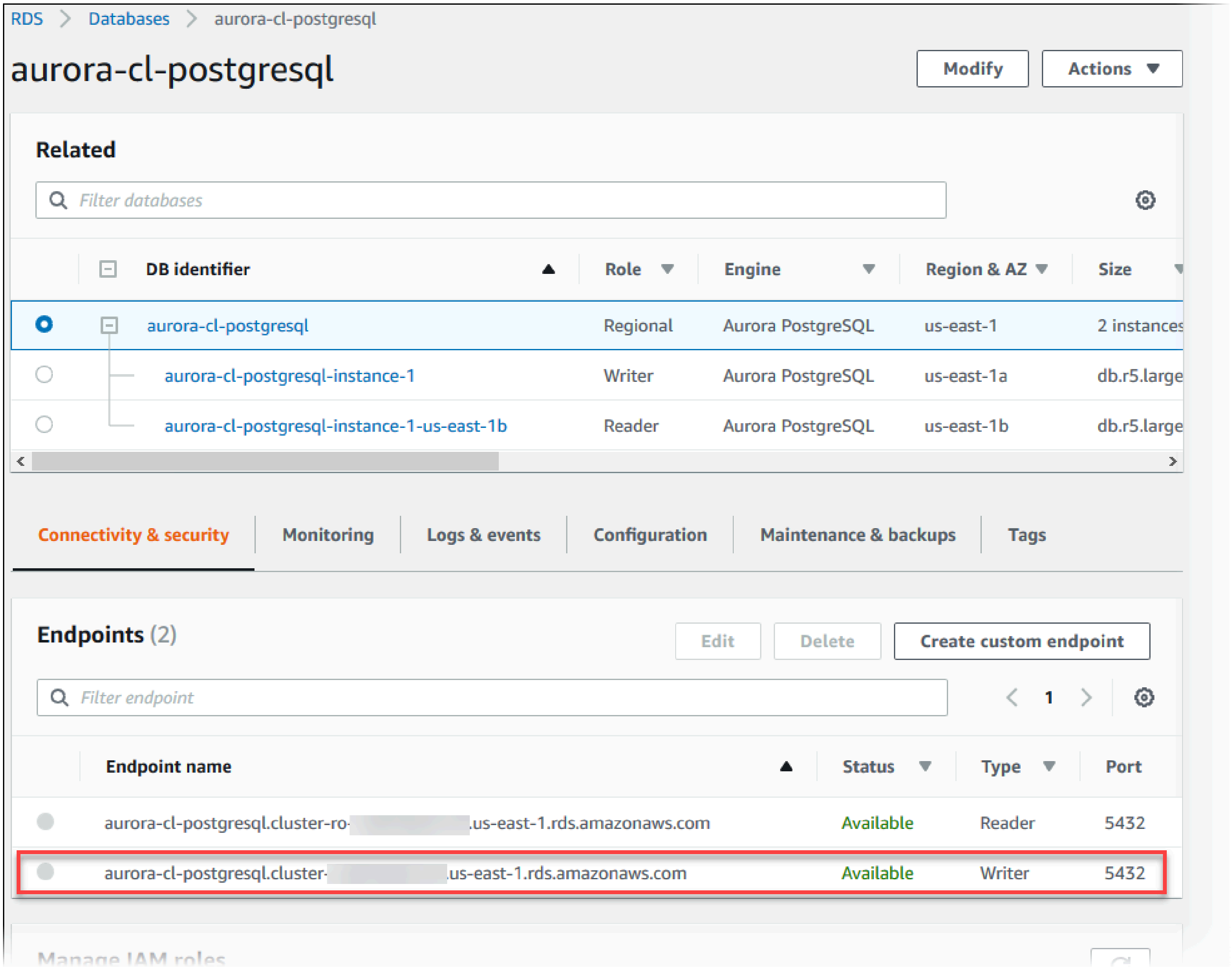
When you have a connection to a DB instance in your Amazon Aurora PostgreSQL DB cluster, you can run any SQL command that is compatible with PostgreSQL.

In the details view for your Aurora PostgreSQL DB cluster you can find the cluster endpoint name, status, type, and port number. You use the endpoint and port number in your PostgreSQL connection string. For example, if an endpoint value is `mycluster.cluster-123456789012.us-east-1.rds.amazonaws.com`, then you specify the following values in a PostgreSQL connection string:

- For host or host name, specify `mycluster.cluster-123456789012.us-east-1.rds.amazonaws.com`
- For port, specify `5432` or the port value you used when you created the DB cluster

The cluster endpoint connects you to the primary instance for the DB cluster. You can perform both read and write operations using the cluster endpoint. Your DB cluster can also have up to 15 Aurora Replicas that support read-only access to the data in your DB cluster. Each DB instance in the Aurora cluster (that is, the primary instance and each Aurora Replica) has a unique endpoint that is independent of the cluster endpoint. This unique endpoint allows you to connect to a specific DB instance in the cluster directly. The cluster endpoint always points to the primary instance. If the primary instance fails and is replaced, the cluster endpoint points to the new primary instance.

To view the cluster endpoint (writer endpoint), choose **Databases** on the Amazon RDS console and choose the name of the DB cluster to show the DB cluster details.



	DB identifier ▲	Role ▼	Engine ▼	Region & AZ ▼	Size ▼
●	⊟ aurora-cl-postgresql	Regional	Aurora PostgreSQL	us-east-1	2 instances
○	aurora-cl-postgresql-instance-1	Writer	Aurora PostgreSQL	us-east-1a	db.r5.large
○	aurora-cl-postgresql-instance-1-us-east-1b	Reader	Aurora PostgreSQL	us-east-1b	db.r5.large

	Endpoint name ▲	Status ▼	Type ▼	Port
○	aurora-cl-postgresql.cluster-ro-[redacted].us-east-1.rds.amazonaws.com	Available	Reader	5432
○	aurora-cl-postgresql.cluster-[redacted].us-east-1.rds.amazonaws.com	Available	Writer	5432

Connection utilities for Aurora PostgreSQL

Some connection utilities you can use are the following:

- **Command line** – You can connect to Aurora PostgreSQL DB clusters by using tools like psql, the PostgreSQL interactive terminal. For more information on using the PostgreSQL interactive terminal, see psql in the PostgreSQL documentation.
- **GUI** – You can use the pgAdmin utility to connect to Aurora PostgreSQL DB clusters by using a UI interface. For more information, see the Download page from the pgAdmin website.
- **AWS drivers**:
  - Connecting to Aurora PostgreSQL with the Amazon Web Services (AWS) JDBC Driver
  - Connecting to Aurora PostgreSQL with the Amazon Web Services (AWS) Python Driver

Connecting to Aurora PostgreSQL with the Amazon Web Services (AWS) JDBC Driver

The Amazon Web Services (AWS) JDBC Driver is designed as an advanced JDBC wrapper. This wrapper is complementary to and extends the functionality of an existing JDBC driver to help applications take advantage of the features of clustered databases such as Aurora PostgreSQL. The driver is drop-in compatible with the community pgJDBC driver.

To install the AWS JDBC Driver, append the AWS JDBC Driver .jar file (located in the application `CLASSPATH`), and keep references to the pgJDBC community driver. Update the connection URL prefix from `jdbc:postgresql://` to `jdbc:aws-wrapper:postgresql://`.

For more information about the AWS JDBC Driver and complete instructions for using it, see the Amazon Web Services (AWS) JDBC Driver GitHub repository.

Connecting to Aurora PostgreSQL with the Amazon Web Services (AWS) Python Driver

The Amazon Web Services (AWS) Python Driver is designed as an advanced Python wrapper. This wrapper is complementary to and extends the functionality of the open-source Psycopg driver. The AWS Python Driver supports Python versions 3.8 and higher. You can install the `aws-advanced-python-wrapper` package using the `pip` command, along with the `psycopg` open-source packages.

For more information about the AWS Python Driver and complete instructions for using it, see the Amazon Web Services (AWS) Python Driver GitHub repository.

Troubleshooting Aurora connection failures

Common causes of connection failures to a new Aurora DB cluster include the following:

- **Security group in the VPC doesn't allow access** – Your VPC needs to allow connections from your device or from an Amazon EC2 instance by proper configuration of the security group in the VPC. To resolve, modify your VPC's Security group Inbound rules to allow connections. For an example, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).
- **Port blocked by firewall rules** – Check the value of the port configured for your Aurora DB cluster. If a firewall rule blocks that port, you can re-create the instance using a different port.

- **Incomplete or incorrect IAM configuration** – If you created your Aurora DB instance to use IAM–based authentication, make sure that it's properly configured. For more information, see IAM database authentication.

For more information about troubleshooting Aurora DB connection issues, see Can't connect to Amazon RDS DB instance.

Working with parameter groups

Database parameters specify how the database is configured. For example, database parameters can specify the amount of resources, such as memory, to allocate to a database.

You manage your database configuration by associating your DB instances and Aurora DB clusters with parameter groups. Aurora defines parameter groups with default settings. You can also define your own parameter groups with customized settings.

Topics

- Overview of parameter groups
- Working with DB cluster parameter groups
- Working with DB parameter groups in a DB instance
- Comparing DB parameter groups
- Specifying DB parameters

Overview of parameter groups

A *DB cluster parameter group* acts as a container for engine configuration values that apply to every DB instance in an Aurora DB cluster. For example, the Aurora shared storage model requires that every DB instance in an Aurora cluster use the same setting for parameters such as `innodb_file_per_table`. Thus, parameters that affect the physical storage layout are part of the cluster parameter group. The DB cluster parameter group also includes default values for all instance-level parameters.

A *DB parameter group* acts as a container for engine configuration values that are applied to one or more DB instances. DB parameter groups apply to DB instances in both Amazon RDS and Aurora. These configuration settings apply to properties that can vary among the DB instances within an Aurora cluster, such as the sizes for memory buffers.

Topics

- Default and custom parameter groups
- Static and dynamic DB cluster parameters
- Static and dynamic DB instance parameters
- Character set parameters

- Supported parameters and parameter values

Default and custom parameter groups

If you create a DB instance without specifying a DB parameter group, the DB instance uses a default DB parameter group. Likewise, if you create an Aurora DB cluster without specifying a DB cluster parameter group, the DB cluster uses a default DB cluster parameter group. Each default parameter group contains database engine defaults and Amazon RDS system defaults based on the engine, compute class, and allocated storage of the instance.

You can't modify the parameter settings of a default parameter group. Instead, you can do the following:

1. Create a new parameter group.
2. Change the settings of your desired parameters. Not all DB engine parameters in a parameter group are eligible to be modified.
3. Modify your DB instance or DB cluster to associate the new parameter group.

   For information about modifying a DB cluster or DB instance, see Modifying an Amazon Aurora DB cluster.

> **Note**
>
> If you have modified your DB instance to use a custom parameter group, and you start the DB instance, RDS automatically reboots the DB instance as part of the startup process.

RDS applies the modified static and dynamic parameters in a newly associated parameter group only after the DB instance is rebooted. However, if you modify dynamic parameters in the DB parameter group after you associate it with the DB instance, these changes are applied immediately without a reboot. For more information about changing the DB parameter group, see Modifying an Amazon Aurora DB cluster.

If you update parameters within a DB parameter group, the changes apply to all DB instances that are associated with that parameter group. Likewise, if you update parameters within an Aurora DB cluster parameter group, the changes apply to all Aurora DB clusters that are associated with that DB cluster parameter group.

If you don't want to create a parameter group from scratch, you can copy an existing parameter group with the AWS CLI copy-db-parameter-group command or copy-db-cluster-parameter-group command. You might find that copying a parameter group is useful in some cases. For example, you might want to include most of an existing parameter group's custom parameters and values in a new parameter group.

Static and dynamic DB cluster parameters

DB cluster parameters are either static or dynamic. They differ in the following ways:

- When you change a static parameter and save the DB cluster parameter group, the parameter change takes effect after you manually reboot the DB instances in each associated DB cluster. When you use the AWS Management Console to change static DB cluster parameter values, it always uses `pending-reboot` for the `ApplyMethod`.
- When you change a dynamic parameter, by default the parameter change takes effect immediately, without requiring a reboot. When you use the console, it always uses `immediate` for the `ApplyMethod`. To defer the parameter change until after you reboot the DB instances in an associated DB cluster, use the AWS CLI or RDS API. Set the `ApplyMethod` to `pending-reboot` for the parameter change.

For more information about using the AWS CLI to change a parameter value, see modify-db-cluster-parameter-group. For more information about using the RDS API to change a parameter value, see ModifyDBClusterParameterGroup.

If you change the DB cluster parameter group associated with a DB cluster, reboot the DB instances in the DB cluster. The reboot applies the changes to all DB instances in the DB cluster. To determine whether the DB instances of a DB cluster must be rebooted to apply changes, run the following AWS CLI command.

```
aws rds describe-db-clusters --db-cluster-identifier db_cluster_identifier
```

Check the `DBClusterParameterGroupStatus` value for the primary DB instance in the output. If the value is `pending-reboot`, then reboot the DB instances of the DB cluster.

Static and dynamic DB instance parameters

DB instance parameters are either static or dynamic. They differ as follows:

- When you change a static parameter and save the DB parameter group, the parameter change takes effect after you manually reboot the associated DB instances. For static parameters, the console always uses `pending-reboot` for the `ApplyMethod`.
- When you change a dynamic parameter, by default the parameter change takes effect immediately, without requiring a reboot. When you use the AWS Management Console to change DB instance parameter values, it always uses `immediate` for the `ApplyMethod` for dynamic parameters. To defer the parameter change until after you reboot an associated DB instance, use the AWS CLI or RDS API. Set the `ApplyMethod` to `pending-reboot` for the parameter change.

For more information about using the AWS CLI to change a parameter value, see modify-db-parameter-group. For more information about using the RDS API to change a parameter value, see ModifyDBParameterGroup.

If a DB instance isn't using the latest changes to its associated DB parameter group, the console shows a status of **pending-reboot** for the DB parameter group. This status doesn't result in an automatic reboot during the next maintenance window. To apply the latest parameter changes to that DB instance, manually reboot the DB instance.

Character set parameters

Before you create a DB cluster, set any parameters that relate to the character set or collation of your database in your parameter group. Also do so before you create a database in it. In this way, you ensure that the default database and new databases use the character set and collation values that you specify. If you change character set or collation parameters, the parameter changes aren't applied to existing databases.

For some DB engines, you can change character set or collation values for an existing database using the `ALTER DATABASE` command, for example:

```
ALTER DATABASE database_name CHARACTER SET character_set_name COLLATE collation;
```

For more information about changing the character set or collation values for a database, check the documentation for your DB engine.

Supported parameters and parameter values

To determine the supported parameters for your DB engine, view the parameters in the DB parameter group and DB cluster parameter group used by the DB instance or DB cluster. For more information, see Viewing parameter values for a DB parameter group and Viewing parameter values for a DB cluster parameter group.

In many cases, you can specify integer and Boolean parameter values using expressions, formulas, and functions. Functions can include a mathematical log expression. However, not all parameters support expressions, formulas, and functions for parameter values. For more information, see Specifying DB parameters.

For an Aurora global database, you can specify different configuration settings for the individual Aurora clusters. Make sure that the settings are similar enough to produce consistent behavior if you promote a secondary cluster to be the primary cluster. For example, use the same settings for time zones and character sets across all the clusters of an Aurora global database.

Improperly setting parameters in a parameter group can have unintended adverse effects, including degraded performance and system instability. Always be cautious when modifying database parameters, and back up your data before modifying a parameter group. Try parameter group setting changes on a test DB instance or DB cluster before applying those parameter group changes to a production DB instance or DB cluster.

Working with DB cluster parameter groups

Amazon Aurora DB clusters use DB cluster parameter groups. The following sections describe configuring and managing DB cluster parameter groups.

Topics

Amazon Aurora DB cluster and DB instance parameters

Aurora uses a two-level system of configuration settings:

- Parameters in a *DB cluster parameter group* apply to every DB instance in a DB cluster. Your data is stored in the Aurora shared storage subsystem. Because of this, all parameters related to physical layout of table data must be the same for all DB instances in an Aurora cluster. Likewise, because Aurora DB instances are connected by replication, all the parameters for replication settings must be identical throughout an Aurora cluster.
- Parameters in a *DB parameter group* apply to a single DB instance in an Aurora DB cluster. These parameters are related to aspects such as memory usage that you can vary across DB instances in the same Aurora cluster. For example, a cluster often contains DB instances with different AWS instance classes.

Every Aurora cluster is associated with a DB cluster parameter group. This parameter group assigns default values for every configuration value for the corresponding DB engine. The cluster parameter group includes defaults for both the cluster-level and instance-level parameters. Each DB instance within a provisioned or Aurora Serverless v2 cluster inherits the settings from that DB cluster parameter group.

Each DB instance is also associated with a DB parameter group. The values in the DB parameter group can override default values from the cluster parameter group. For example, if one instance in a cluster experienced issues, you might assign a custom DB parameter group to that instance. The custom parameter group might have specific settings for parameters related to debugging or performance tuning.

Aurora assigns default parameter groups when you create a cluster or a new DB instance, based on the specified database engine and version. You can specify custom parameter groups instead. You create those parameter groups yourself, and you can edit the parameter values. You can specify these custom parameter groups at creation time. You can also modify a DB cluster or instance later to use a custom parameter group.

For provisioned and Aurora Serverless v2 instances, any configuration values that you modify in the DB cluster parameter group override default values in the DB parameter group. If you edit the corresponding values in the DB parameter group, those values override the settings from the DB cluster parameter group.

Any DB parameter settings that you modify take precedence over the DB cluster parameter group values, even if you change the configuration parameters back to their default values. You can see which parameters are overridden by using the describe-db-parameters AWS CLI command or the DescribeDBParameters RDS API operation. The `Source` field contains the value `user` if you modified that parameter. To reset one or more parameters so that the value from the DB cluster parameter group takes precedence, use the reset-db-parameter-group AWS CLI command or the ResetDBParameterGroup RDS API operation.

The DB cluster and DB instance parameters available to you in Aurora vary depending on database engine compatibility.

Database engine	Parameters
Aurora MySQL	See Aurora MySQL configuration parameters. For Aurora Serverless clusters, see additional details in Working with parameter groups for Aurora Serverless v2 and Parameter groups for Aurora Serverless v1.
Aurora PostgreSQL	See Amazon Aurora PostgreSQL parameters. For Aurora Serverless clusters, see additional details in Working with parameter groups for Aurora Serverless v2 and Parameter groups for Aurora Serverless v1.

Note

Aurora Serverless v1 clusters have only DB cluster parameter groups, not DB parameter groups. For Aurora Serverless v2 clusters, you make all your changes to custom parameters in the DB cluster parameter group.
Aurora Serverless v2 uses both DB cluster parameter groups and DB parameter groups. With Aurora Serverless v2, you can modify almost all of the configuration parameters. Aurora Serverless v2 overrides the settings of some capacity-related configuration parameters so that your workload isn't interrupted when Aurora Serverless v2 instances scale down.

To learn more about Aurora Serverless configuration settings and which settings you can modify, see Working with parameter groups for Aurora Serverless v2 and Parameter groups for Aurora Serverless v1.

Creating a DB cluster parameter group

You can create a new DB cluster parameter group using the AWS Management Console, the AWS CLI, or the RDS API.

After you create a DB cluster parameter group, wait at least 5 minutes before creating a DB cluster that uses that DB cluster parameter group. Doing this allows Amazon RDS to fully create the parameter group before it is used by the new DB cluster. You can use the **Parameter groups** page in the Amazon RDS console or the describe-db-cluster-parameters command to verify that your DB cluster parameter group is created.

The following limitations apply to the DB cluster parameter group name:

- The name must be 1 to 255 letters, numbers, or hyphens.

  Default parameter group names can include a period, such as `default.aurora-mysql5.7`. However, custom parameter group names can't include a period.
- The first character must be a letter.
- The name can't end with a hyphen or contain two consecutive hyphens.

Console

To create a DB cluster parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. Choose **Create parameter group**.

   The **Create parameter group** window appears.
4. In the **Parameter group family** list, select a DB parameter group family
5. In the **Type** list, select **DB cluster parameter group**.

6. In the **Group name** box, enter the name of the new DB cluster parameter group.
7. In the **Description** box, enter a description for the new DB cluster parameter group.
8. Choose **Create**.

AWS CLI

To create a DB cluster parameter group, use the AWS CLI `create-db-cluster-parameter-group` command.

The following example creates a DB cluster parameter group named *mydbclusterparametergroup* for Aurora MySQL version 5.7 with a description of "*My new cluster parameter group*."

Include the following required parameters:

- `--db-cluster-parameter-group-name`
- `--db-parameter-group-family`
- `--description`

To list all of the available parameter group families, use the following command:

```
aws rds describe-db-engine-versions --query "DBEngineVersions[].DBParameterGroupFamily"
```

> **Note**
>
> The output contains duplicates.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-cluster-parameter-group \
    --db-cluster-parameter-group-name mydbclusterparametergroup \
    --db-parameter-group-family aurora-mysql5.7 \
    --description "My new cluster parameter group"
```

For Windows:

```
aws rds create-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name mydbclusterparametergroup ^
    --db-parameter-group-family aurora-mysql5.7 ^
    --description "My new cluster parameter group"
```

This command produces output similar to the following:

```
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupName": "mydbclusterparametergroup",
        "DBParameterGroupFamily": "aurora-mysql5.7",
        "Description": "My new cluster parameter group",
        "DBClusterParameterGroupArn": "arn:aws:rds:us-east-1:123456789012:cluster-
pg:mydbclusterparametergroup"
    }
}
```

RDS API

To create a DB cluster parameter group, use the RDS API `CreateDBClusterParameterGroup` action.

Include the following required parameters:

- `DBClusterParameterGroupName`
- `DBParameterGroupFamily`
- `Description`

Associating a DB cluster parameter group with a DB cluster

You can create your own DB cluster parameter groups with customized settings. You can associate a DB cluster parameter group with a DB cluster using the AWS Management Console, the AWS CLI, or the RDS API. You can do so when you create or modify a DB cluster.

For information about creating a DB cluster parameter group, see Creating a DB cluster parameter group. For information about creating a DB cluster, see Creating an Amazon Aurora DB cluster. For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

> ⓘ **Note**
>
> For Aurora PostgreSQL 15.2, 14.7, 13.10, 12.14, and all 11 versions, when you change the DB cluster parameter group associated with a DB cluster, reboot each replica instance to apply the changes.
> To determine whether the primary DB instance of a DB cluster must be rebooted to apply changes, run the following AWS CLI command:
> ```
> aws rds describe-db-clusters --db-cluster-identifier
> db_cluster_identifier
> ```
> Check the `DBClusterParameterGroupStatus` value for the primary DB instance in the output. If the value is `pending-reboot`, then reboot the primary DB instance of the DB cluster.

Console

To associate a DB cluster parameter group with a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB cluster that you want to modify.
3. Choose **Modify**. The **Modify DB cluster** page appears.
4. Change the **DB cluster parameter group** setting.
5. Choose **Continue** and check the summary of modifications.

   The change is applied immediately regardless of the **Scheduling of modifications** setting.
6. On the confirmation page, review your changes. If they are correct, choose **Modify cluster** to save your changes.

   Alternatively, choose **Back** to edit your changes, or choose **Cancel** to cancel your changes.

AWS CLI

To associate a DB cluster parameter group with a DB cluster, use the AWS CLI `modify-db-cluster` command with the following options:

- `--db-cluster-name`
- `--db-cluster-parameter-group-name`

The following example associates the `mydbclpg` DB parameter group with the `mydbcluster` DB cluster.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --db-cluster-parameter-group-name mydbclpg
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --db-cluster-parameter-group-name mydbclpg
```

RDS API

To associate a DB cluster parameter group with a DB cluster, use the RDS API `ModifyDBCluster` operation with the following parameters:

- `DBClusterIdentifier`
- `DBClusterParameterGroupName`

Modifying parameters in a DB cluster parameter group

You can modify parameter values in a customer-created DB cluster parameter group. You can't change the parameter values in a default DB cluster parameter group. Changes to parameters in a customer-created DB cluster parameter group are applied to all DB clusters that are associated with the DB cluster parameter group.

Console

To modify a DB cluster parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the parameter group that you want to modify.
4. For **Parameter group actions**, choose **Edit**.
5. Change the values of the parameters you want to modify. You can scroll through the parameters using the arrow keys at the top right of the dialog box.

   You can't change values in a default parameter group.
6. Choose **Save changes**.
7. Reboot the primary (writer) DB instance in the cluster to apply the changes to it.
8. Then reboot the reader DB instances to apply the changes to them.

AWS CLI

To modify a DB cluster parameter group, use the AWS CLI `modify-db-cluster-parameter-group` command with the following required parameters:

- `--db-cluster-parameter-group-name`
- `--parameters`

The following example modifies the `server_audit_logging` and `server_audit_logs_upload` values in the DB cluster parameter group named *mydbclusterparametergroup*.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name mydbclusterparametergroup \
    --parameters
 "ParameterName=server_audit_logging,ParameterValue=1,ApplyMethod=immediate" \
 "ParameterName=server_audit_logs_upload,ParameterValue=1,ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name mydbclusterparametergroup ^
    --parameters
 "ParameterName=server_audit_logging,ParameterValue=1,ApplyMethod=immediate" ^

 "ParameterName=server_audit_logs_upload,ParameterValue=1,ApplyMethod=immediate"
```

The command produces output like the following:

```
DBCLUSTERPARAMETERGROUP  mydbclusterparametergroup
```

RDS API

To modify a DB cluster parameter group, use the RDS API `ModifyDBClusterParameterGroup` command with the following required parameters:

- `DBClusterParameterGroupName`
- `Parameters`

Resetting parameters in a DB cluster parameter group

You can reset parameters to their default values in a customer-created DB cluster parameter group. Changes to parameters in a customer-created DB cluster parameter group are applied to all DB clusters that are associated with the DB cluster parameter group.

Note

In a default DB cluster parameter group, parameters are always set to their default values.

Console

To reset parameters in a DB cluster parameter group to their default values

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the parameter group.
4. For **Parameter group actions**, choose **Edit**.
5. Choose the parameters that you want to reset to their default values. You can scroll through the parameters using the arrow keys at the top right of the dialog box.

   You can't reset values in a default parameter group.
6. Choose **Reset** and then confirm by choosing **Reset parameters**.
7. Reboot the primary DB instance in the DB cluster to apply the changes to all of the DB instances in the DB cluster.

AWS CLI

To reset parameters in a DB cluster parameter group to their default values, use the AWS CLI `reset-db-cluster-parameter-group` command with the following required option: `--db-cluster-parameter-group-name`.

To reset all of the parameters in the DB cluster parameter group, specify the `--reset-all-parameters` option. To reset specific parameters, specify the `--parameters` option.

The following example resets all of the parameters in the DB parameter group named *mydbparametergroup* to their default values.

Example

For Linux, macOS, or Unix:

```
aws rds reset-db-cluster-parameter-group \
    --db-cluster-parameter-group-name mydbparametergroup \
    --reset-all-parameters
```

For Windows:

```
aws rds reset-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name mydbparametergroup ^
    --reset-all-parameters
```

The following example resets the `server_audit_logging` and `server_audit_logs_upload` to their default values in the DB cluster parameter group named *mydbclusterparametergroup*.

Example

For Linux, macOS, or Unix:

```
aws rds reset-db-cluster-parameter-group \
    --db-cluster-parameter-group-name mydbclusterparametergroup \
    --parameters "ParameterName=server_audit_logging,ApplyMethod=immediate" \
                 "ParameterName=server_audit_logs_upload,ApplyMethod=immediate"
```

For Windows:

```
aws rds reset-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name mydbclusterparametergroup ^
    --parameters
 "ParameterName=server_audit_logging,ParameterValue=1,ApplyMethod=immediate" ^

 "ParameterName=server_audit_logs_upload,ParameterValue=1,ApplyMethod=immediate"
```

The command produces output like the following:

```
DBClusterParameterGroupName  mydbclusterparametergroup
```

RDS API

To reset parameters in a DB cluster parameter group to their default values, use the RDS API `ResetDBClusterParameterGroup` command with the following required parameter: `DBClusterParameterGroupName`.

To reset all of the parameters in the DB cluster parameter group, set the `ResetAllParameters` parameter to `true`. To reset specific parameters, specify the `Parameters` parameter.

Copying a DB cluster parameter group

You can copy custom DB cluster parameter groups that you create. Copying a parameter group is a convenient solution when you have already created a DB cluster parameter group and you want to include most of the custom parameters and values from that group in a new DB cluster parameter group. You can copy a DB cluster parameter group by using the AWS CLI copy-db-cluster-parameter-group command or the RDS API CopyDBClusterParameterGroup operation.

After you copy a DB cluster parameter group, wait at least 5 minutes before creating a DB cluster that uses that DB cluster parameter group. Doing this allows Amazon RDS to fully copy the parameter group before it is used by the new DB cluster. You can use the **Parameter groups** page in the Amazon RDS console or the describe-db-cluster-parameters command to verify that your DB cluster parameter group is created.

> **Note**
>
> You can't copy a default parameter group. However, you can create a new parameter group that is based on a default parameter group.
>
> You can't copy a DB cluster parameter group to a different AWS account or AWS Region.

Console

To copy a DB cluster parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the custom parameter group that you want to copy.
4. For **Parameter group actions**, choose **Copy**.
5. In **New DB parameter group identifier**, enter a name for the new parameter group.
6. In **Description**, enter a description for the new parameter group.
7. Choose **Copy**.

AWS CLI

To copy a DB cluster parameter group, use the AWS CLI copy-db-cluster-parameter-group command with the following required parameters:

- --source-db-cluster-parameter-group-identifier
- --target-db-cluster-parameter-group-identifier
- --target-db-cluster-parameter-group-description

The following example creates a new DB cluster parameter group named mygroup2 that is a copy of the DB cluster parameter group mygroup1.

Example

For Linux, macOS, or Unix:

```
aws rds copy-db-cluster-parameter-group \
    --source-db-cluster-parameter-group-identifier mygroup1 \
    --target-db-cluster-parameter-group-identifier mygroup2 \
    --target-db-cluster-parameter-group-description "DB parameter group 2"
```

For Windows:

```
aws rds copy-db-cluster-parameter-group ^
    --source-db-cluster-parameter-group-identifier mygroup1 ^
    --target-db-cluster-parameter-group-identifier mygroup2 ^
    --target-db-cluster-parameter-group-description "DB parameter group 2"
```

RDS API

To copy a DB cluster parameter group, use the RDS API CopyDBClusterParameterGroup operation with the following required parameters:

- SourceDBClusterParameterGroupIdentifier
- TargetDBClusterParameterGroupIdentifier
- TargetDBClusterParameterGroupDescription

Listing DB cluster parameter groups

You can list the DB cluster parameter groups you've created for your AWS account.

> **Note**
>
> Default parameter groups are automatically created from a default parameter template when you create a DB cluster for a particular DB engine and version. These default parameter groups contain preferred parameter settings and can't be modified. When you create a custom parameter group, you can modify parameter settings.

Console

To list all DB cluster parameter groups for an AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The DB cluster parameter groups appear in the list with **DB cluster parameter group** for **Type**.

AWS CLI

To list all DB cluster parameter groups for an AWS account, use the AWS CLI `describe-db-cluster-parameter-groups` command.

Example

The following example lists all available DB cluster parameter groups for an AWS account.

```
aws rds describe-db-cluster-parameter-groups
```

The following example describes the *mydbclusterparametergroup* parameter group.

For Linux, macOS, or Unix:

```
aws rds describe-db-cluster-parameter-groups \
    --db-cluster-parameter-group-name mydbclusterparametergroup
```

For Windows:

```
aws rds describe-db-cluster-parameter-groups ^
    --db-cluster-parameter-group-name mydbclusterparametergroup
```

The command returns a response like the following:

```
{
    "DBClusterParameterGroups": [
        {
            "DBClusterParameterGroupName": "mydbclusterparametergroup",
            "DBParameterGroupFamily": "aurora-mysql5.7",
            "Description": "My new cluster parameter group",
            "DBClusterParameterGroupArn": "arn:aws:rds:us-east-1:123456789012:cluster-
pg:mydbclusterparametergroup"
        }
    ]
}
```

RDS API

To list all DB cluster parameter groups for an AWS account, use the RDS API `DescribeDBClusterParameterGroups` action.

Viewing parameter values for a DB cluster parameter group

You can get a list of all parameters in a DB cluster parameter group and their values.

Console

To view the parameter values for a DB cluster parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The DB cluster parameter groups appear in the list with **DB cluster parameter group** for **Type**.
3. Choose the name of the DB cluster parameter group to see its list of parameters.

AWS CLI

To view the parameter values for a DB cluster parameter group, use the AWS CLI `describe-db-cluster-parameters` command with the following required parameter.

- `--db-cluster-parameter-group-name`

Example

The following example lists the parameters and parameter values for a DB cluster parameter group named *mydbclusterparametergroup*, in JSON format.

The command returns a response like the following:

```
aws rds describe-db-cluster-parameters --db-cluster-parameter-group-
name mydbclusterparametergroup
```

```
{
    "Parameters": [
        {
            "ParameterName": "allow-suspicious-udfs",
            "Description": "Controls whether user-defined functions that have only an
 xxx symbol for the main function can be loaded",
            "Source": "engine-default",
            "ApplyType": "static",
            "DataType": "boolean",
            "AllowedValues": "0,1",
            "IsModifiable": false,
            "ApplyMethod": "pending-reboot",
            "SupportedEngineModes": [
                "provisioned"
            ]
        },
        {
            "ParameterName": "aurora_binlog_read_buffer_size",
            "ParameterValue": "5242880",
            "Description": "Read buffer size used by master dump thread when the switch
 aurora_binlog_use_large_read_buffer is ON.",
            "Source": "engine-default",
            "ApplyType": "dynamic",
            "DataType": "integer",
            "AllowedValues": "8192-536870912",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot",
            "SupportedEngineModes": [
                "provisioned"
            ]
        },

...
```

RDS API

To view the parameter values for a DB cluster parameter group, use the RDS API `DescribeDBClusterParameters` command with the following required parameter.

- `DBClusterParameterGroupName`

In some cases, the allowed values for a parameter aren't shown. These are always parameters where the source is the database engine default.

To view the values of these parameters, you can run the following SQL statements:

- MySQL:

```
-- Show the value of a particular parameter
mysql$ SHOW VARIABLES LIKE '%parameter_name%';

-- Show the values of all parameters
mysql$ SHOW VARIABLES;
```

- PostgreSQL:

```
-- Show the value of a particular parameter
postgresql=> SHOW parameter_name;

-- Show the values of all parameters
postgresql=> SHOW ALL;
```

Deleting a DB cluster parameter group

You can delete a DB cluster parameter group using the AWS Management Console, AWS CLI, or RDS API. A DB cluster parameter group parameter group is eligible for deletion only if it isn't associated with a DB cluster.

Console

To delete parameter groups

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The parameter groups appear in a list.
3. Choose the name of the DB cluster parameter groups to be deleted.
4. Choose **Actions** and then **Delete**.
5. Review the parameter group names and then choose **Delete**.

AWS CLI

To delete a DB cluster parameter group, use the AWS CLI `delete-db-cluster-parameter-group` command with the following required parameter.

- `--db-parameter-group-name`

Example

The following example deletes a DB cluster parameter group named *mydbparametergroup.*

```
aws rds delete-db-cluster-parameter-group --db-parameter-group-name mydbparametergroup
```

RDS API

To delete a DB cluster parameter group, use the RDS API `DeleteDBClusterParameterGroup` command with the following required parameter.

- `DBParameterGroupName`

Working with DB parameter groups in a DB instance

DB instances use DB parameter groups. The following sections describe configuring and managing DB instance parameter groups.

Topics

Creating a DB parameter group

You can create a new DB parameter group using the AWS Management Console, the AWS CLI, or the RDS API.

The following limitations apply to the DB parameter group name:

- The name must be 1 to 255 letters, numbers, or hyphens.

  Default parameter group names can include a period, such as `default.mysql8.0`. However, custom parameter group names can't include a period.
- The first character must be a letter.
- The name can't end with a hyphen or contain two consecutive hyphens.

Console

To create a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

3. Choose **Create parameter group**.
4. For **Parameter group name**, enter the name of your new DB parameter group.
5. For **Description**, enter a description for your new DB parameter group.
6. For **Engine type**, choose your DB engine.
7. For **Parameter group family**, choose a DB parameter group family.
8. For **Type**, if applicable, choose **DB Parameter Group**.
9. Choose **Create**.

AWS CLI

To create a DB parameter group, use the AWS CLI `create-db-parameter-group` command. The following example creates a DB parameter group named *mydbparametergroup* for MySQL version 8.0 with a description of "*My new parameter group*."

Include the following required parameters:

- `--db-parameter-group-name`
- `--db-parameter-group-family`
- `--description`

To list all of the available parameter group families, use the following command:

```
aws rds describe-db-engine-versions --query "DBEngineVersions[].DBParameterGroupFamily"
```

> **Note**
>
> The output contains duplicates.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-parameter-group \
    --db-parameter-group-name mydbparametergroup \
    --db-parameter-group-family aurora-mysql5.7 \
    --description "My new parameter group"
```

For Windows:

```
aws rds create-db-parameter-group ^
    --db-parameter-group-name mydbparametergroup ^
    --db-parameter-group-family aurora-mysql5.7 ^
    --description "My new parameter group"
```

This command produces output similar to the following:

```
DBPARAMETERGROUP  mydbparametergroup  aurora-mysql5.7  My new parameter group
```

RDS API

To create a DB parameter group, use the RDS API `CreateDBParameterGroup` operation.

Include the following required parameters:

- `DBParameterGroupName`
- `DBParameterGroupFamily`
- `Description`

Associating a DB parameter group with a DB instance

You can create your own DB parameter groups with customized settings. You can associate a DB parameter group with a DB instance using the AWS Management Console, the AWS CLI, or the RDS API. You can do so when you create or modify a DB instance.

For information about creating a DB parameter group, see Creating a DB parameter group. For information about modifying a DB instance, see Modifying a DB instance in a DB cluster.

> **Note**
>
> When you associate a new DB parameter group with a DB instance, the modified static and dynamic parameters are applied only after the DB instance is rebooted. However, if you modify dynamic parameters in the DB parameter group after you associate it with the DB instance, these changes are applied immediately without a reboot.

Console

To associate a DB parameter group with a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB instance that you want to modify.
3. Choose **Modify**. The **Modify DB instance** page appears.
4. Change the **DB parameter group** setting.
5. Choose **Continue** and check the summary of modifications.
6. (Optional) Choose **Apply immediately** to apply the changes immediately. Choosing this option can cause an outage in some cases.
7. On the confirmation page, review your changes. If they are correct, choose **Modify DB instance** to save your changes.

   Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

AWS CLI

To associate a DB parameter group with a DB instance, use the AWS CLI `modify-db-instance` command with the following options:

- `--db-instance-identifier`
- `--db-parameter-group-name`

The following example associates the `mydbpg` DB parameter group with the `database-1` DB instance. The changes are applied immediately by using `--apply-immediately`. Use `--no-apply-immediately` to apply the changes during the next maintenance window.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier database-1 \
    --db-parameter-group-name mydbpg \
    --apply-immediately
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier database-1 ^
    --db-parameter-group-name mydbpg ^
    --apply-immediately
```

RDS API

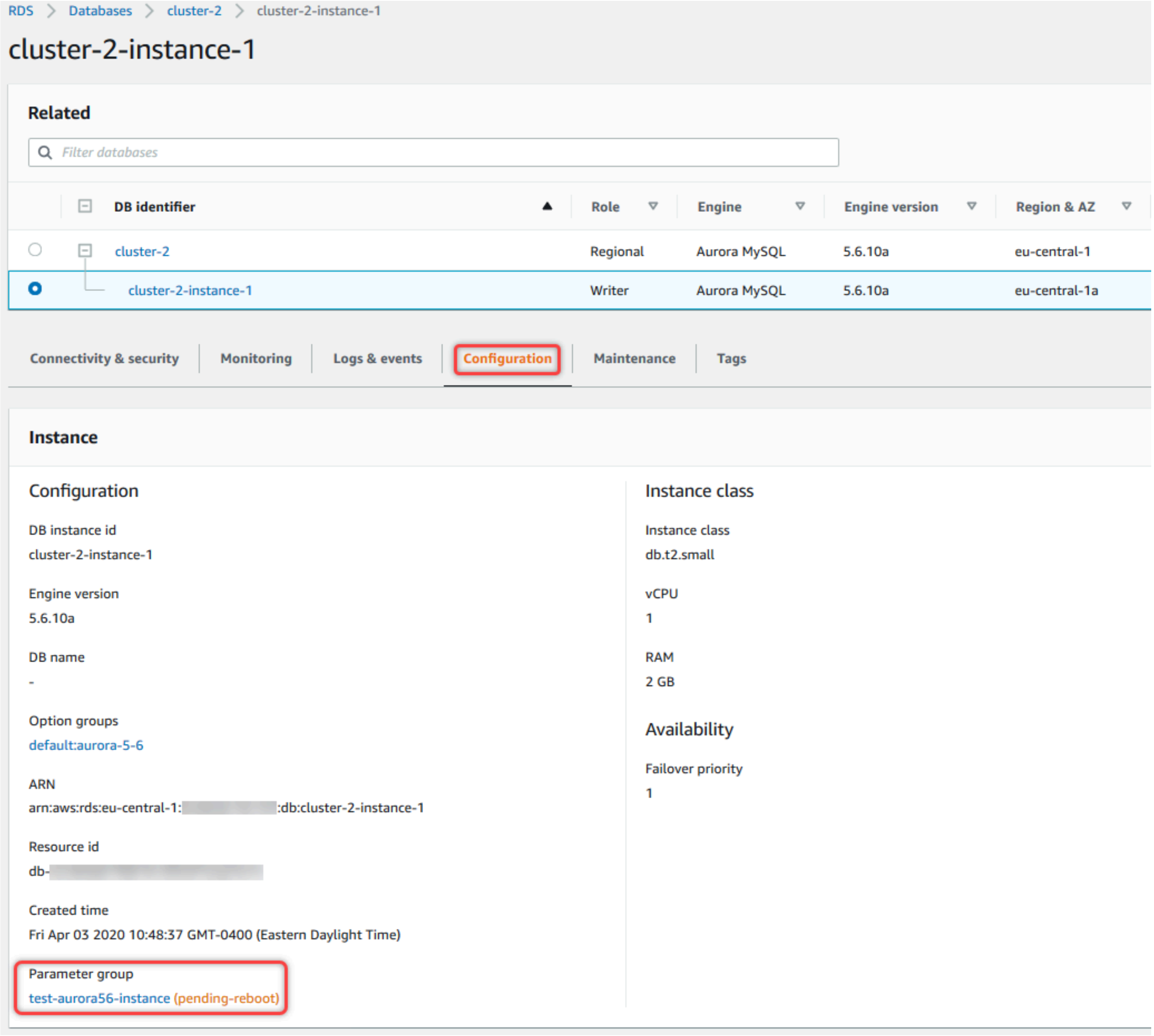
To associate a DB parameter group with a DB instance, use the RDS API `ModifyDBInstance` operation with the following parameters:

- `DBInstanceName`
- `DBParameterGroupName`

Modifying parameters in a DB parameter group

You can modify parameter values in a customer-created DB parameter group; you can't change the parameter values in a default DB parameter group. Changes to parameters in a customer-created DB parameter group are applied to all DB instances that are associated with the DB parameter group.

Changes to some parameters are applied to the DB instance immediately without a reboot. Changes to other parameters are applied only after the DB instance is rebooted. The RDS console shows the status of the DB parameter group associated with a DB instance on the **Configuration** tab. For example, suppose that the DB instance isn't using the latest changes to its associated DB parameter group. If so, the RDS console shows the DB parameter group with a status of **pending-reboot**. To apply the latest parameter changes to that DB instance, manually reboot the DB instance.



RDS > Databases > cluster-2 > cluster-2-instance-1

cluster-2-instance-1

Related

Filter databases

DB identifier	Role	Engine	Engine version	Region & AZ
○ cluster-2	Regional	Aurora MySQL	5.6.10a	eu-central-1
● cluster-2-instance-1	Writer	Aurora MySQL	5.6.10a	eu-central-1a

Instance

Configuration

DB instance id
cluster-2-instance-1

Engine version
5.6.10a

DB name
-

Option groups
default:aurora-5-6

ARN
arn:aws:rds:eu-central-1:[redacted]:db:cluster-2-instance-1

Resource id
db-[redacted]

Created time
Fri Apr 03 2020 10:48:37 GMT-0400 (Eastern Daylight Time)

Parameter group
test-aurora56-instance (pending-reboot)

Instance class

Instance class
db.t2.small

vCPU
1

RAM
2 GB

Availability

Failover priority
1

Console

To modify the parameters in a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the name of the parameter group that you want to modify.
4. For **Parameter group actions**, choose **Edit**.

5. Change the values of the parameters that you want to modify. You can scroll through the parameters using the arrow keys at the top right of the dialog box.

   You can't change values in a default parameter group.

6. Choose **Save changes**.

AWS CLI

To modify a DB parameter group, use the AWS CLI `modify-db-parameter-group` command with the following required options:

- `--db-parameter-group-name`
- `--parameters`

The following example modifies the `max_connections` and `max_allowed_packet` values in the DB parameter group named *mydbparametergroup*.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-parameter-group \
    --db-parameter-group-name mydbparametergroup \
    --parameters
 "ParameterName=max_connections,ParameterValue=250,ApplyMethod=immediate" \

 "ParameterName=max_allowed_packet,ParameterValue=1024,ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-parameter-group ^
    --db-parameter-group-name mydbparametergroup ^
    --parameters
 "ParameterName=max_connections,ParameterValue=250,ApplyMethod=immediate" ^

 "ParameterName=max_allowed_packet,ParameterValue=1024,ApplyMethod=immediate"
```

The command produces output like the following:

```
DBPARAMETERGROUP  mydbparametergroup
```

RDS API

To modify a DB parameter group, use the RDS API `ModifyDBParameterGroup` operation with the following required parameters:

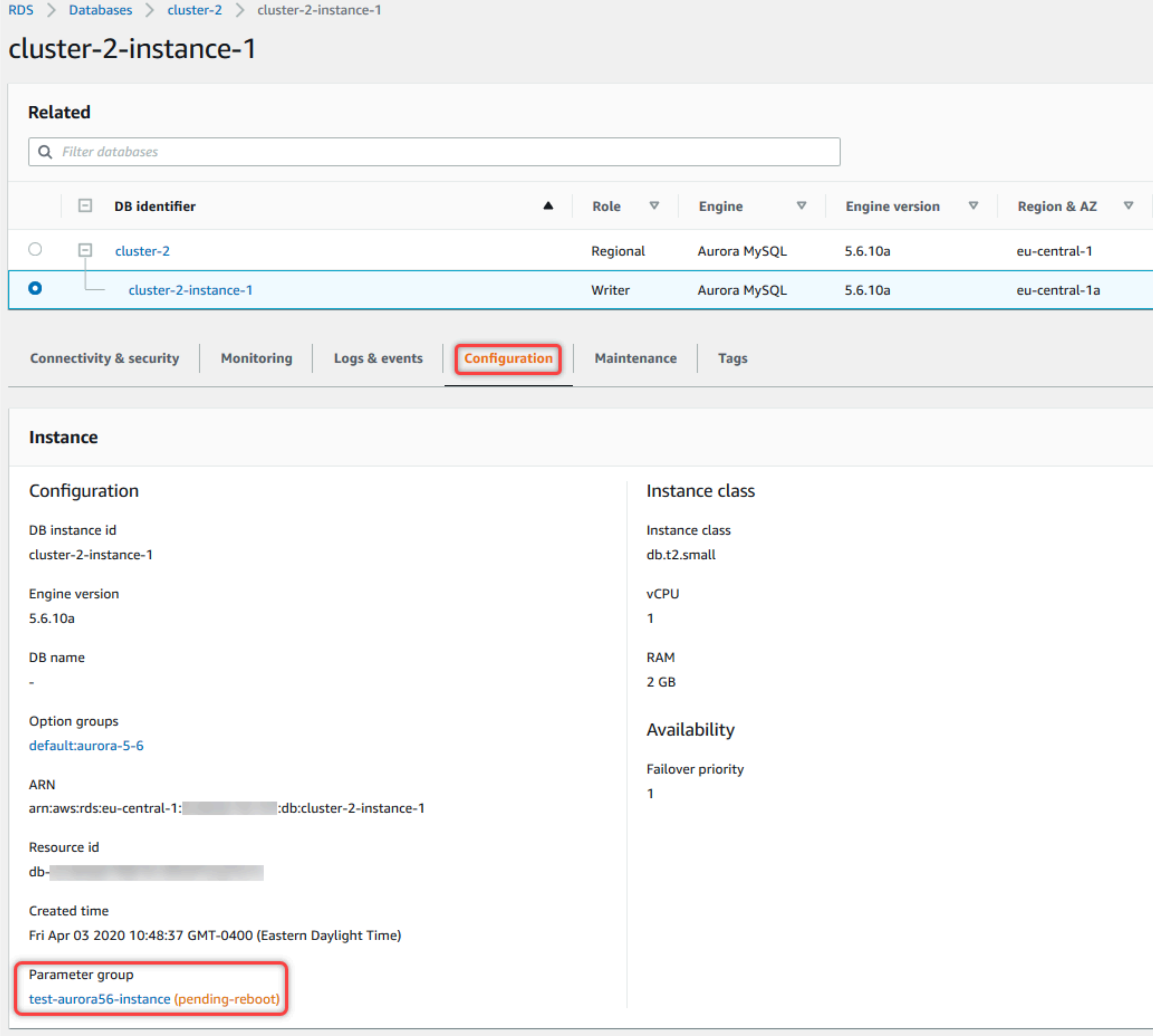
- `DBParameterGroupName`
- `Parameters`

Resetting parameters in a DB parameter group to their default values

You can reset parameter values in a customer-created DB parameter group to their default values. Changes to parameters in a customer-created DB parameter group are applied to all DB instances that are associated with the DB parameter group.

When you use the console, you can reset specific parameters to their default values. However, you can't easily reset all of the parameters in the DB parameter group at once. When you use the AWS CLI or RDS API, you can reset specific parameters to their default values. You can also reset all of the parameters in the DB parameter group at once.

Changes to some parameters are applied to the DB instance immediately without a reboot. Changes to other parameters are applied only after the DB instance is rebooted. The RDS console shows the status of the DB parameter group associated with a DB instance on the **Configuration** tab. For example, suppose that the DB instance isn't using the latest changes to its associated DB parameter group. If so, the RDS console shows the DB parameter group with a status of **pending-reboot**. To apply the latest parameter changes to that DB instance, manually reboot the DB instance.



RDS > Databases > cluster-2 > cluster-2-instance-1

cluster-2-instance-1

Related

Filter databases

	DB identifier	Role	Engine	Engine version	Region & AZ
○	cluster-2	Regional	Aurora MySQL	5.6.10a	eu-central-1
●	cluster-2-instance-1	Writer	Aurora MySQL	5.6.10a	eu-central-1a

Instance

Configuration

DB instance id
cluster-2-instance-1

Engine version
5.6.10a

DB name
-

Option groups
default:aurora-5-6

ARN
arn:aws:rds:eu-central-1:[redacted]:db:cluster-2-instance-1

Resource id
db-[redacted]

Created time
Fri Apr 03 2020 10:48:37 GMT-0400 (Eastern Daylight Time)

Parameter group
test-aurora56-instance (pending-reboot)

Instance class

Instance class
db.t2.small

vCPU
1

RAM
2 GB

Availability

Failover priority
1

Note

In a default DB parameter group, parameters are always set to their default values.

Console

To reset parameters in a DB parameter group to their default values

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the parameter group.
4. For **Parameter group actions**, choose **Edit**.
5. Choose the parameters that you want to reset to their default values. You can scroll through the parameters using the arrow keys at the top right of the dialog box.

   You can't reset values in a default parameter group.
6. Choose **Reset** and then confirm by choosing **Reset parameters**.

AWS CLI

To reset some or all of the parameters in a DB parameter group, use the AWS CLI `reset-db-parameter-group` command with the following required option: `--db-parameter-group-name`.

To reset all of the parameters in the DB parameter group, specify the `--reset-all-parameters` option. To reset specific parameters, specify the `--parameters` option.

The following example resets all of the parameters in the DB parameter group named *mydbparametergroup* to their default values.

Example

For Linux, macOS, or Unix:

```
aws rds reset-db-parameter-group \
    --db-parameter-group-name mydbparametergroup \
    --reset-all-parameters
```

For Windows:

```
aws rds reset-db-parameter-group ^
```

```
    --db-parameter-group-name mydbparametergroup ^
    --reset-all-parameters
```

The following example resets the `max_connections` and `max_allowed_packet` options to their default values in the DB parameter group named *mydbparametergroup*.

Example

For Linux, macOS, or Unix:

```
aws rds reset-db-parameter-group \
    --db-parameter-group-name mydbparametergroup \
    --parameters "ParameterName=max_connections,ApplyMethod=immediate" \
                 "ParameterName=max_allowed_packet,ApplyMethod=immediate"
```

For Windows:

```
aws rds reset-db-parameter-group ^
    --db-parameter-group-name mydbparametergroup ^
    --parameters "ParameterName=max_connections,ApplyMethod=immediate" ^
                 "ParameterName=max_allowed_packet,ApplyMethod=immediate"
```

The command produces output like the following:

```
DBParameterGroupName  mydbparametergroup
```

RDS API

To reset parameters in a DB parameter group to their default values, use the RDS API `ResetDBParameterGroup` command with the following required parameter: `DBParameterGroupName`.

To reset all of the parameters in the DB parameter group, set the `ResetAllParameters` parameter to `true`. To reset specific parameters, specify the `Parameters` parameter.

Copying a DB parameter group

You can copy custom DB parameter groups that you create. Copying a parameter group can be convenient solution. An example is when you have created a DB parameter group and want to include most of its custom parameters and values in a new DB parameter group. You can copy a DB parameter group by using the AWS Management Console. You can also use the AWS CLI copy-db-parameter-group command or the RDS API CopyDBParameterGroup operation.

After you copy a DB parameter group, wait at least 5 minutes before creating your first DB instance that uses that DB parameter group as the default parameter group. Doing this allows Amazon RDS to fully complete the copy action before the parameter group is used. This is especially important for parameters that are critical when creating the default database for a DB instance. An example is the character set for the default database defined by the `character_set_database` parameter. Use the **Parameter Groups** option of the Amazon RDS console or the describe-db-parameters command to verify that your DB parameter group is created.

> **Note**
>
> You can't copy a default parameter group. However, you can create a new parameter group that is based on a default parameter group.
> You can't copy a DB parameter group to a different AWS account or AWS Region.

Console

To copy a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the custom parameter group that you want to copy.
4. For **Parameter group actions**, choose **Copy**.
5. In **New DB parameter group identifier**, enter a name for the new parameter group.
6. In **Description**, enter a description for the new parameter group.
7. Choose **Copy**.

AWS CLI

To copy a DB parameter group, use the AWS CLI `copy-db-parameter-group` command with the following required options:

- `--source-db-parameter-group-identifier`

- `--target-db-parameter-group-identifier`
- `--target-db-parameter-group-description`

The following example creates a new DB parameter group named `mygroup2` that is a copy of the DB parameter group `mygroup1`.

Example

For Linux, macOS, or Unix:

```
aws rds copy-db-parameter-group \
    --source-db-parameter-group-identifier mygroup1 \
    --target-db-parameter-group-identifier mygroup2 \
    --target-db-parameter-group-description "DB parameter group 2"
```

For Windows:

```
aws rds copy-db-parameter-group ^
    --source-db-parameter-group-identifier mygroup1 ^
    --target-db-parameter-group-identifier mygroup2 ^
    --target-db-parameter-group-description "DB parameter group 2"
```

RDS API

To copy a DB parameter group, use the RDS API [CopyDBParameterGroup] operation with the following required parameters:

- `SourceDBParameterGroupIdentifier`
- `TargetDBParameterGroupIdentifier`
- `TargetDBParameterGroupDescription`

Listing DB parameter groups

You can list the DB parameter groups you've created for your AWS account.

> **Note**
>
> Default parameter groups are automatically created from a default parameter template when you create a DB instance for a particular DB engine and version. These default

parameter groups contain preferred parameter settings and can't be modified. When you create a custom parameter group, you can modify parameter settings.

Console

To list all DB parameter groups for an AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The DB parameter groups appear in a list.

AWS CLI

To list all DB parameter groups for an AWS account, use the AWS CLI `describe-db-parameter-groups` command.

Example

The following example lists all available DB parameter groups for an AWS account.

```
aws rds describe-db-parameter-groups
```

The command returns a response like the following:

```
DBPARAMETERGROUP  default.mysql8.0     mysql8.0  Default parameter group for MySQL8.0
DBPARAMETERGROUP  mydbparametergroup   mysql8.0  My new parameter group
```

The following example describes the *mydbparamgroup1* parameter group.

For Linux, macOS, or Unix:

```
aws rds describe-db-parameter-groups \
    --db-parameter-group-name mydbparamgroup1
```

For Windows:

```
aws rds describe-db-parameter-groups ^
    --db-parameter-group-name mydbparamgroup1
```

The command returns a response like the following:

```
DBPARAMETERGROUP  mydbparametergroup1  mysql8.0  My new parameter group
```

RDS API

To list all DB parameter groups for an AWS account, use the RDS API `DescribeDBParameterGroups` operation.

Viewing parameter values for a DB parameter group

You can get a list of all parameters in a DB parameter group and their values.

Console

To view the parameter values for a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The DB parameter groups appear in a list.
3. Choose the name of the parameter group to see its list of parameters.

AWS CLI

To view the parameter values for a DB parameter group, use the AWS CLI `describe-db-parameters` command with the following required parameter.

- `--db-parameter-group-name`

Example

The following example lists the parameters and parameter values for a DB parameter group named *mydbparametergroup.*

```
aws rds describe-db-parameters --db-parameter-group-name mydbparametergroup
```

The command returns a response like the following:

```
DBPARAMETER  Parameter Name            Parameter Value  Source           Data Type
 Apply Type  Is Modifiable
DBPARAMETER  allow-suspicious-udfs                      engine-default   boolean
 static      false
DBPARAMETER  auto_increment_increment                   engine-default   integer
 dynamic     true
DBPARAMETER  auto_increment_offset                      engine-default   integer
 dynamic     true
DBPARAMETER  binlog_cache_size         32768            system           integer
 dynamic     true
DBPARAMETER  socket                    /tmp/mysql.sock  system           string
 static      false
```

RDS API

To view the parameter values for a DB parameter group, use the RDS API `DescribeDBParameters` command with the following required parameter.

- `DBParameterGroupName`

Deleting a DB parameter group

You can delete a DB parameter group using the AWS Management Console, AWS CLI, or RDS API. A parameter group is eligible for deletion only if it isn't associated with a DB instance.

Console

To delete a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.

   The DB parameter groups appear in a list.
3. Choose the name of the parameter groups to be deleted.
4. Choose **Actions** and then **Delete**.

5. Review the parameter group names and then choose **Delete**.

AWS CLI

To delete a DB parameter group, use the AWS CLI `delete-db-parameter-group` command with the following required parameter.

- `--db-parameter-group-name`

Example

The following example deletes a DB parameter group named *mydbparametergroup*.

```
aws rds delete-db-parameter-group --db-parameter-group-name mydbparametergroup
```

RDS API

To delete a DB parameter group, use the RDS API `DeleteDBParameterGroup` command with the following required parameter.

- `DBParameterGroupName`

Comparing DB parameter groups

You can use the AWS Management Console to view the differences between two DB parameter groups.

The specified parameter groups must both be DB parameter groups, or they both must be DB cluster parameter groups. This is true even when the DB engine and version are the same. For example, you can't compare an `aurora-mysql8.0` (Aurora MySQL version 3) DB parameter group and an `aurora-mysql8.0` DB cluster parameter group.

You can compare Aurora MySQL and RDS for MySQL DB parameter groups, even for different versions, but you can't compare Aurora PostgreSQL and RDS for PostgreSQL DB parameter groups.

To compare two DB parameter groups

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the two parameter groups that you want to compare.

   > **Note**
   >
   > To compare a default parameter group to a custom parameter group, first choose the default parameter group on the **Default** tab, then choose the custom parameter group on the **Custom** tab.

4. From **Actions**, choose **Compare**.

Specifying DB parameters

DB parameter types include the following:

- Integer
- Boolean
- String
- Long
- Double
- Timestamp
- Object of other defined data types
- Array of values of type integer, Boolean, string, long, double, timestamp, or object

You can also specify integer and Boolean parameters using expressions, formulas, and functions.

Contents

DB parameter formulas

A DB parameter formula is an expression that resolves to an integer value or a Boolean value. You enclose the expression in braces: **{}**. You can use a formula for either a DB parameter value or as an argument to a DB parameter function.

Syntax

```
{FormulaVariable}
{FormulaVariable*Integer}
{FormulaVariable*Integer/Integer}
{FormulaVariable/Integer}
```

DB parameter formula variables

Each formula variable returns an integer or a Boolean value. The names of the variables are case-sensitive.

AllocatedStorage

Returns an integer representing the size, in bytes, of the data volume.

DBInstanceClassMemory

Returns an integer for the number of bytes of memory available to the database process. This number is internally calculated by starting with the total amount of memory for the DB instance class. From this, the calculation subtracts memory reserved for the operating system and the RDS processes that manage the instance. Therefore, the number is always somewhat lower than the memory figures shown in the instance class tables in Aurora DB instance classes. The exact value depends on a combination of factors. These include instance class, DB engine, and whether it applies to an RDS instance or an instance that's part of an Aurora cluster.

EndPointPort

Returns an integer representing the port used when connecting to the DB instance.

TrueIfReplica

Returns 1 if the DB instance is a read replica and `0` if it is not. This is the default value for the `read_only` parameter in Aurora MySQL.

DB parameter formula operators

DB parameter formulas support two operators: division and multiplication.

Division operator: /

Divides the dividend by the divisor, returning an integer quotient. Decimals in the quotient are truncated, not rounded.

Syntax

```
dividend / divisor
```

The dividend and divisor arguments must be integer expressions.

*Multiplication operator: **

Multiplies the expressions, returning the product of the expressions. Decimals in the expressions are truncated, not rounded.

Syntax

```
expression * expression
```

Both expressions must be integers.

DB parameter functions

You specify the arguments of DB parameter functions as either integers or formulas. Each function must have at least one argument. Specify multiple arguments as a comma-separated list. The list can't have any empty members, such as *argument1*,,*argument3*. Function names are case-insensitive.

IF

Returns an argument.

Syntax

```
IF(argument1, argument2, argument3)
```

Returns the second argument if the first argument evaluates to true. Returns the third argument otherwise.

GREATEST

Returns the largest value from a list of integers or parameter formulas.

Syntax

```
GREATEST(argument1, argument2,...argumentn)
```

Returns an integer.

LEAST

Returns the smallest value from a list of integers or parameter formulas.

Syntax

```
LEAST(argument1, argument2,...argumentn)
```

Returns an integer.

SUM

Adds the values of the specified integers or parameter formulas.

Syntax

```
SUM(argument1, argument2,...argumentn)
```

Returns an integer.

DB parameter log expressions

You can set an integer DB parameter value to a log expression. You enclose the expression in braces: {}. For example:

```
{log(DBInstanceClassMemory/8187281418)*1000}
```

The `log` function represents log base 2. This example also uses the `DBInstanceClassMemory` formula variable. See DB parameter formula variables.

DB parameter value examples

These examples show using formulas, functions, and expressions for the values of DB parameters.

> ⚠ **Warning**
>
> Improperly setting parameters in a DB parameter group can have unintended adverse effects. These might include degraded performance and system instability. Use caution when modifying database parameters and back up your data before modifying your DB parameter group. Try out parameter group changes on a test DB instance, created using point-in-time-restores, before applying those parameter group changes to your production DB instances.

Example using the DB parameter function LEAST

You can specify the `LEAST` function in an Aurora MySQL `table_definition_cache` parameter value. Use it to set the number of table definitions that can be stored in the definition cache to the lesser of `DBInstanceClassMemory`/393040 or 20,000.

```
LEAST({DBInstanceClassMemory/393040}, 20000)
```

Migrating data to an Amazon Aurora DB cluster

You have several options for migrating data from your existing database to an Amazon Aurora DB cluster, depending on database engine compatibility. Your migration options also depend on the database that you are migrating from and the size of the data that you are migrating.

Migrating data to an Amazon Aurora MySQL DB cluster

You can migrate data from one of the following sources to an Amazon Aurora MySQL DB cluster.

- An RDS for MySQL DB instance
- A MySQL database external to Amazon RDS
- A database that is not MySQL-compatible

For more information, see Migrating data to an Amazon Aurora MySQL DB cluster.

Migrating data to an Amazon Aurora PostgreSQL DB cluster

You can migrate data from one of the following sources to an Amazon Aurora PostgreSQL DB cluster.

- An Amazon RDS PostgreSQL DB instance
- A database that is not PostgreSQL-compatible

For more information, see Migrating data to Amazon Aurora with PostgreSQL compatibility.

Creating an Amazon ElastiCache cache using Aurora DB cluster settings

ElastiCache is a fully managed, in-memory caching service that provides microsecond read and write latencies that support flexible, real-time use cases. ElastiCache can help you accelerate application and database performance. You can use ElastiCache as a primary data store for use cases that don't require data durability, such as gaming leaderboards, streaming, and data analytics. ElastiCache helps remove the complexity associated with deploying and managing a distributed computing environment. For more information, see Common ElastiCache Use Cases and How ElastiCache Can Help for Memcached and Common ElastiCache Use Cases and How ElastiCache Can Help for Redis. You can use the Amazon RDS console for creating ElastiCache cache.

You can operate Amazon ElastiCache in two formats. You can get started with a serverless cache or choose to design your own cache cluster. If you choose to design your own cache cluster, ElastiCache works with both the Redis and Memcached engines. If you're unsure which engine you want to use, see Comparing Memcached and Redis. For more information about Amazon ElastiCache, see the Amazon ElastiCache User Guide.

Topics

Overview of ElastiCache cache creation with Aurora DB cluster settings

You can create an ElastiCache cache from Amazon RDS using the same configuration settings as a newly created or existing Aurora DB cluster.

Some use cases to associate an ElastiCache cache with your DB cluster:

- You can save costs and improve your performance by using ElastiCache with RDS versus running on RDS alone.
- You can use the ElastiCache cache as a primary data store for applications that don't require data durability. Your applications that use Redis or Memcached can use ElastiCache with almost no modification.

When you create an ElastiCache cache from RDS, the ElastiCache cache inherits the following settings from the associated Aurora DB cluster:

- ElastiCache connectivity settings
- ElastiCache security settings

You can also set the cache configuration settings according to your requirements.

Setting up ElastiCache in your applications

Your applications must be set up to utilize ElastiCache cache. You can also optimize and improve cache performance by setting up your applications to use caching strategies depending on your requirements.

- To access your ElastiCache cache and get started, see Getting started with Amazon ElastiCache for Redis and Getting started with Amazon ElastiCache for Memcached.
- For more information about caching strategies, see Caching strategies and best practices for Memcached and Caching strategies and best practices for Redis.
- For more information about high availability in ElastiCache for Redis clusters, see High availability using replication groups.
- You might incur costs associated with backup storage, data transfer within or across regions, or use of AWS Outposts. For pricing details, see Amazon ElastiCache pricing.

Creating an ElastiCache cache with settings from an Aurora DB cluster

You can create an ElastiCache cache for your Aurora DB clusters with settings for inherited from the DB cluster.

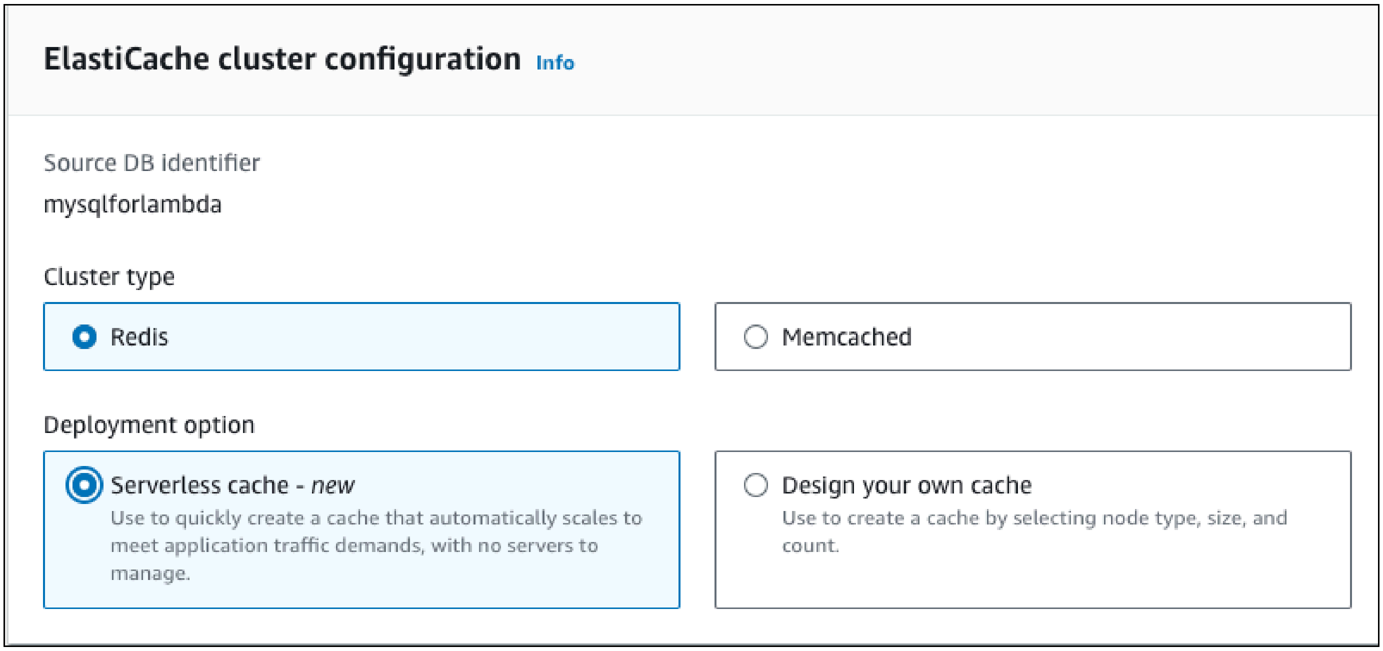
Create an ElastiCache cache with settings from a DB cluster

1. To create a DB cluster, follow the instructions in Creating an Amazon Aurora DB cluster.
2. After creating an Aurora DB cluster, the console displays the **Suggested add-ons** window. Select **Create an ElastiCache cluster from RDS using your DB settings**.

   For an existing database, in the **Databases** page, select the required DB cluster. In the **Actions** dropdown menu, choose **Create ElastiCache cluster** to create an ElastiCache cache in RDS that has the same settings as your existing Aurora DB cluster.

In the **ElastiCache configuration section**, the **Source DB identifier** displays which DB cluster the ElastiCache cache inherits settings from.

3. Choose whether you want to create a Redis or Memcached cluster. For more information, see Comparing Memcached and Redis.



ElastiCache cluster configuration Info

Source DB identifier
mysqlforlambda

Cluster type

● Redis ○ Memcached

Deployment option

● Serverless cache - *new*
Use to quickly create a cache that automatically scales to meet application traffic demands, with no servers to manage.

○ Design your own cache
Use to create a cache by selecting node type, size, and count.

4. After this, choose whether you want to create a **Serverless cache** or **Design your own cache**. For more information, see Choosing between deployment options.

   If you choose **Serverless cache**:

   a. In **Cache settings**, enter values for **Name** and **Description**.

   b. Under **View default settings**, leave the default settings to establish the connection between your cache and DB cluster.

   c. You can also edit the default settings by choosing **Customize default settings**. Select the **ElastiCache connectivity settings**, **ElastiCache security settings**, and **Maximum usage limits.**

5. If you choose **Design your own cache**:

   a. If you chose **Redis cluster**, choose whether you want to keep the cluster mode **Enabled** or **Disabled**. For more information, see Replication: Redis (Cluster Mode Disabled) vs. Redis (Cluster Mode Enabled).

   b. Enter values for **Name**, **Description**, and **Engine version**.

For **Engine version**, the recommended default value is the latest engine version. You can also choose an **Engine version** for the ElastiCache cache that best meets your requirements.

c. Choose the node type in the **Node type** option. For more information, see Managing nodes.

   If you choose to create a Redis cluster with the **Cluster mode** set to **Enabled**, then enter the number of shards (partitions/node groups) in the **Number of shards** option.

   Enter the number of replicas of each shard in **Number of replicas**.

   > **Note**
   >
   > The selected node type, the number of shards, and the number of replicas all affect your cache performance and resource costs. Be sure these settings match your database needs. For pricing information, see Amazon ElastiCache pricing.

d. Select the **ElastiCache connectivity settings** and **ElastiCache security settings**. You can keep the default settings or customize these settings per your requirements.

6. Verify the default and inherited settings of your ElastiCache cache. Some settings can't be changed after creation.

   > **Note**
   >
   > RDS might adjust the backup window of your ElastiCache cache to meet the minimum window requirement of 60 minutes. The backup window of your source database remains the same.

7. When you're ready, choose **Create ElastiCache cache**.

The console displays a confirmation banner for the ElastiCache cache creation. Follow the link in the banner to the ElastiCache console to view the cache details. The ElastiCache console displays the newly created ElastiCache cache.

Managing an Amazon Aurora DB cluster

This section shows how to manage and maintain your Aurora DB cluster. Aurora involves clusters of database servers that are connected in a replication topology. Thus, managing Aurora often involves deploying changes to multiple servers and making sure that all Aurora Replicas are keeping up with the master server. Because Aurora transparently scales the underlying storage as your data grows, managing Aurora requires relatively little management of disk storage. Likewise, because Aurora automatically performs continuous backups, an Aurora cluster does not require extensive planning or downtime for performing backups.

Topics

Stopping and starting an Amazon Aurora DB cluster

Stopping and starting Amazon Aurora clusters helps you manage costs for development and test environments. You can temporarily stop all the DB instances in your cluster, instead of setting up and tearing down all the DB instances each time that you use the cluster.

Topics

Overview of stopping and starting an Aurora DB cluster

During periods where you don't need an Aurora cluster, you can stop all instances in that cluster at once. You can start the cluster again anytime you need to use it. Starting and stopping simplifies the setup and teardown processes for clusters used for development, testing, or similar activities that don't require continuous availability. You can perform all the AWS Management Console procedures involved with only a single action, regardless of how many instances are in the cluster.

While your DB cluster is stopped, you are charged only for cluster storage, manual snapshots, and automated backup storage within your specified retention window. You aren't charged for any DB instance hours.

> ⚠ **Important**
>
> You can stop a DB cluster for up to seven days. If you don't manually start your DB cluster after seven days, your DB cluster is automatically started so that it doesn't fall behind any required maintenance updates.

To minimize charges for a lightly loaded Aurora cluster, you can stop the cluster instead of deleting all of its Aurora Replicas. For clusters with more than one or two instances, frequently deleting and recreating the DB instances is only practical using the AWS CLI or Amazon RDS API. Such a sequence of operations can also be difficult to perform in the right order, for example to delete all Aurora Replicas before deleting the primary instance to avoid activating the failover mechanism.

Don't use starting and stopping if you need to keep your DB cluster running but it has more capacity than you need. If your cluster is too costly or not very busy, delete one or more DB instances or change all your DB instances to a small instance class. You can't stop an individual Aurora DB instance.

Limitations for stopping and starting Aurora DB clusters

Some Aurora clusters can't be stopped and started:

- You can't stop and start a cluster that's part of an Aurora global database.
- You can't stop and start a cluster that has a cross-Region read replica.
- You can't stop and start a cluster that is part of a blue/green deployment.
- For a cluster that uses the Aurora parallel query feature, the minimum Aurora MySQL version is 2.09.0.
- You can't stop and start an Aurora Serverless v1 cluster. With Aurora Serverless v2, you can stop and start the cluster.

If an existing cluster can't be stopped and started, the **Stop** action isn't available from the **Actions** menu on the **Databases** page or the details page.

Stopping an Aurora DB cluster

To use an Aurora DB cluster or perform administration, you always begin with a running Aurora DB cluster, then stop the cluster, and then start the cluster again. While your cluster is stopped, you are charged for cluster storage, manual snapshots, and automated backup storage within your specified retention window, but not for DB instance hours.

The stop operation stops the Aurora Replica instances first, then the primary instance, to avoid activating the failover mechanism.

You can't stop a DB cluster that acts as the replication target for data from another DB cluster, or acts as the replication master and transmits data to another cluster.

You can't stop certain special kinds of clusters. Currently, you can't stop a cluster that's part of an Aurora global database.

Console

To stop an Aurora cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose a cluster. You can perform the stop operation from this page, or navigate to the details page for the DB cluster that you want to stop.
3. For **Actions**, choose **Stop temporarily**.

   If a DB cluster can't be stopped and started, the **Stop temporarily** action isn't available from the **Actions** menu on the **Databases** page or the details page. For the kinds of clusters that you can't start and stop, see Limitations for stopping and starting Aurora DB clusters.
4. In the **Stop DB cluster temporarily** window, select the acknowledgement that the DB cluster will restart automatically after 7 days.
5. Choose **Stop temporarily** to stop the DB cluster, or choose **Cancel** to cancel the operation.

AWS CLI

To stop a DB instance by using the AWS CLI, call the stop-db-cluster command with the following parameters:

- `--db-cluster-identifier` – the name of the Aurora cluster.

Example

```
aws rds stop-db-cluster --db-cluster-identifier mydbcluster
```

RDS API

To stop a DB instance by using the Amazon RDS API, call the StopDBCluster operation with the following parameter:

- `DBClusterIdentifier` – the name of the Aurora cluster.

Possible operations while an Aurora DB cluster is stopped

While an Aurora cluster is stopped, you can do a point-in-time restore to any point within your specified automated backup retention window. For details about doing a point-in-time restore, see Restoring data.

You can't modify the configuration of an Aurora DB cluster, or any of its DB instances, while the cluster is stopped. You also can't add or remove DB instances from the cluster, or delete the cluster if it still has any associated DB instances. You must start the cluster before performing any such administrative actions.

Stopping a DB cluster removes pending actions, except for the DB cluster parameter group or for the DB parameter groups of the DB cluster instances.

Aurora applies any scheduled maintenance to your stopped cluster after it's started again. Remember that after seven days, Aurora automatically starts any stopped clusters so that they don't fall too far behind in their maintenance status.

Aurora also doesn't perform any automated backups, because the underlying data can't change while the cluster is stopped. Aurora doesn't extend the backup retention period while the cluster is stopped.

Starting an Aurora DB cluster

You always start an Aurora DB cluster beginning with an Aurora cluster that is already in the stopped state. When you start the cluster, all its DB instances become available again. The cluster keeps its configuration settings such as endpoints, parameter groups, and VPC security groups.

Restarting a DB cluster usually takes several minutes.

Console

To start an Aurora cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose a cluster. You can perform the start operation from this page, or navigate to the details page for the DB cluster that you want to start.
3. For **Actions**, choose **Start**.

AWS CLI

To start a DB cluster by using the AWS CLI, call the start-db-cluster command with the following parameters:

- `--db-cluster-identifier` – the name of the Aurora cluster. This name is either a specific cluster identifier you chose when creating the cluster, or the DB instance identifier you chose with `-cluster` appended to the end.

Example

```
aws rds start-db-cluster --db-cluster-identifier mydbcluster
```

RDS API

To start an Aurora DB cluster by using the Amazon RDS API, call the StartDBCluster operation with the following parameter:

- `DBCluster` – the name of the Aurora cluster. This name is either a specific cluster identifier you chose when creating the cluster, or the DB instance identifier you chose with `-cluster` appended to the end.

Automatically connecting an AWS compute resource and an Aurora DB cluster

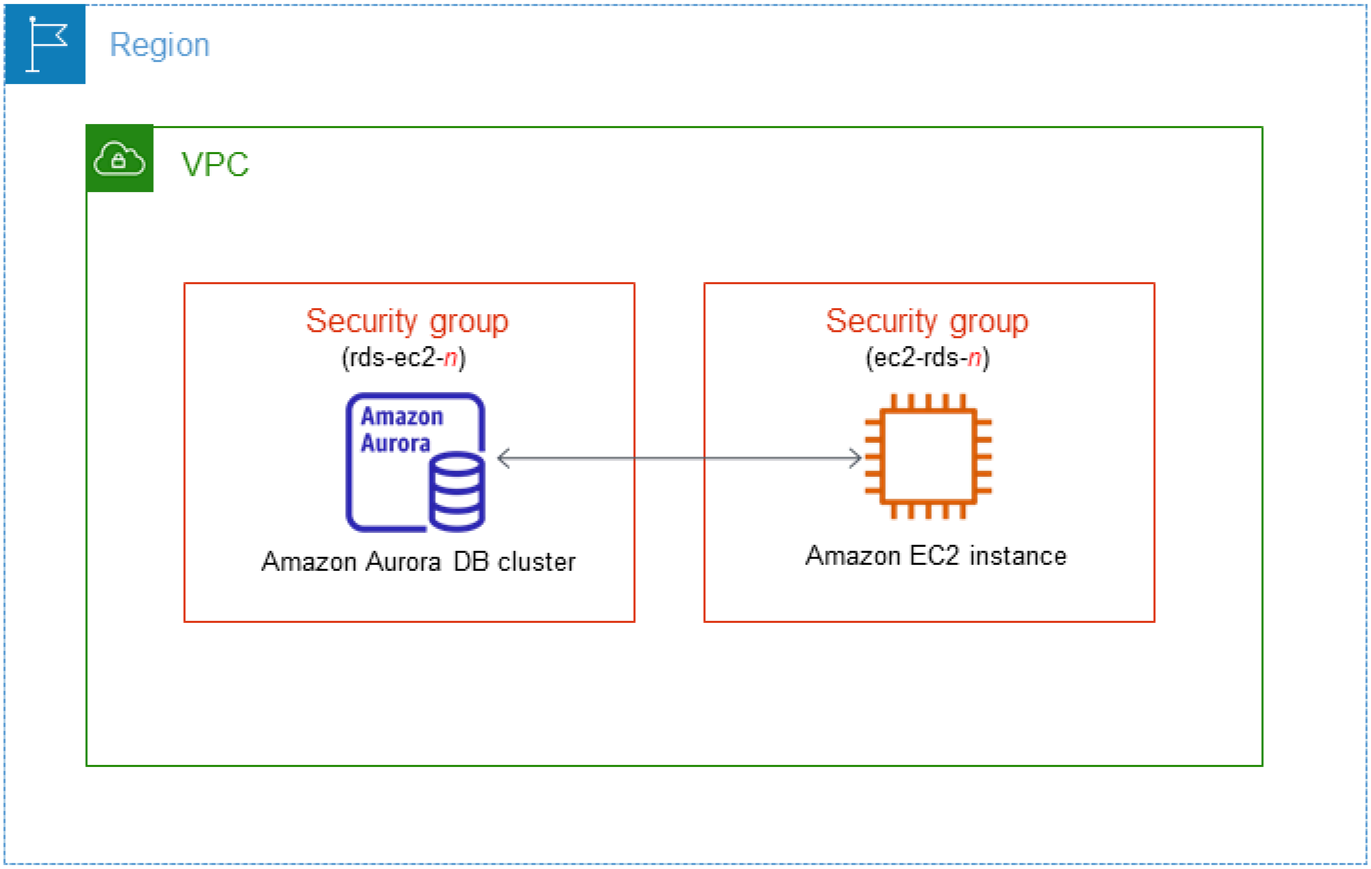
You can automatically connect an Aurora DB cluster and AWS compute resources such as Amazon Elastic Compute Cloud (Amazon EC2) instances and AWS Lambda functions.

Topics

- Automatically connecting an EC2 instance and an Aurora DB cluster
- Automatically connecting a Lambda function and an Aurora DB cluster

Automatically connecting an EC2 instance and an Aurora DB cluster

You can use the Amazon RDS console to simplify setting up a connection between an Amazon Elastic Compute Cloud (Amazon EC2) instance and an Aurora DB cluster. Often, your DB cluster is in a private subnet and your EC2 instance is in a public subnet within a VPC. You can use a SQL client on your EC2 instance to connect to your DB cluster. The EC2 instance can also run web servers or applications that access your private DB cluster.



If you want to connect to an EC2 instance that isn't in the same VPC as the Aurora DB cluster, see the scenarios in Scenarios for accessing a DB cluster in a VPC.

Topics

- Overview of automatic connectivity with an EC2 instance
- Automatically connecting an EC2 instance and an Aurora DB cluster
- Viewing connected compute resources
- Connecting to a DB instance that is running a specific DB engine

Overview of automatic connectivity with an EC2 instance

When you set up a connection between an EC2 instance and an Aurora DB cluster, Amazon RDSautomatically configures the VPC security group for your EC2 instance and for your DB cluster.

The following are requirements for connecting an EC2 instance with an Aurora DB cluster:

- The EC2 instance must exist in the same VPC as the DB cluster.

  If no EC2 instances exist in the same VPC, then the console provides a link to create one.
- Currently, the DB cluster can't be an Aurora Serverless DB cluster or part of an Aurora global database.
- The user who sets up connectivity must have permissions to perform the following Amazon EC2 operations:
  - `ec2:AuthorizeSecurityGroupEgress`
  - `ec2:AuthorizeSecurityGroupIngress`
  - `ec2:CreateSecurityGroup`
  - `ec2:DescribeInstances`
  - `ec2:DescribeNetworkInterfaces`
  - `ec2:DescribeSecurityGroups`
  - `ec2:ModifyNetworkInterfaceAttribute`
  - `ec2:RevokeSecurityGroupEgress`

If the DB instance and EC2 instance are in different Availability Zones, your account may incur cross-Availability Zone costs.

When you set up a connection to an EC2 instance, Amazon RDS acts according to the current configuration of the security groups associated with the DB cluster and EC2 instance, as described in the following table.

Current RDS security group configuration	Current EC2 security group configuration	RDS action
There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-ec2-n` (where *n* is a number). A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	There are one or more security groups associated with the EC2 instance with a name that matches the pattern `ec2-rds-n` (where *n* is a number). A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with the VPC security group of the DB cluster as the source.	RDS takes no action. A connection was already configured automatically between the EC2 instance and DB cluster. Because a connection already exists between the EC2 instance and the RDS database, the security groups aren't modified.
Either of the following conditions apply: • There is no security group associated with the DB cluster with a name that matches the pattern `rds-ec2-n`. • There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-ec2-n`. However, Amazon RDS can't use any of these security groups for the connection with the EC2 instance.	Either of the following conditions apply: • There is no security group associated with the EC2 instance with a name that matches the pattern `ec2-rds-n`. • There are one or more security groups associated with the EC2 instance with a name that matches the pattern `ec2-rds-n`. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster.	RDS action: create new security groups

Current RDS security group configuration	Current EC2 security group configuration	RDS action
Amazon RDS can't use a security group that doesn't have one inbound rule with the VPC security group of the EC2 instance as the source. Amazon RDS also can't use a security group that has been modified. Examples of modifications include adding a rule or changing the port of an existing rule.	Amazon RDS can't use a security group that doesn't have one outbound rule with the VPC security group of the DB cluster as the source. Amazon RDS also can't use a security group that has been modified.	
There are one or more security groups associate d with the DB cluster with a name that matches the pattern `rds-ec2-`*n*. A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	There are one or more security groups associate d with the EC2 instance with a name that matches the pattern `ec2-rds-`*n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster. Amazon RDS can't use a security group that doesn't have one outbound rule with the VPC security group of the DB cluster as the source. Amazon RDS also can't use a security group that has been modified.	RDS action: create new security groups

Current RDS security group configuration	Current EC2 security group configuration	RDS action
There are one or more security groups associate d with the DB cluster with a name that matches the pattern `rds-ec2-n`. A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	A valid EC2 security group for the connection exists, but it is not associated with the EC2 instance. This security group has a name that matches the pattern `ec2-rds-n`. It hasn't been modified. It has only one outbound rule with the VPC security group of the DB cluster as the source.	RDS action: associate EC2 security group

Current RDS security group configuration	Current EC2 security group configuration	RDS action
Either of the following conditions apply: • There is no security group associated with the DB cluster with a name that matches the pattern `rds-ec2-n`. • There are one or more security groups associate d with the DB cluster with a name that matches the pattern `rds-ec2-n`. However, Amazon RDS can't use any of these security groups for the connectio n with the EC2 instance. Amazon RDS can't use a security group that doesn't have one inbound rule with the VPC security group of the EC2 instance as the source. Amazon RDS also can't use security group that has been modified.	There are one or more security groups associate d with the EC2 instance with a name that matches the pattern `ec2-rds-n`. A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with the VPC security group of the DB cluster as the source.	RDS action: create new security groups

RDS action: create new security groups

Amazon RDS takes the following actions:

- Creates a new security group that matches the pattern `rds-ec2-n`. This security group has an inbound rule with the VPC security group of the EC2 instance as the source. This security group is associated with the DB cluster and allows the EC2 instance to access the DB cluster.

- Creates a new security group that matches the pattern `ec2-rds-n`. This security group has an outbound rule with the VPC security group of the DB cluster as the target. This security group is associated with the EC2 instance and allows the EC2 instance to send traffic to the DB cluster.

RDS action: associate EC2 security group

Amazon RDS associates the valid, existing EC2 security group with the EC2 instance. This security group allows the EC2 instance to send traffic to the DB cluster.

Automatically connecting an EC2 instance and an Aurora DB cluster

Before setting up a connection between an EC2 instance and an Aurora DB cluster, make sure you meet the requirements described in Overview of automatic connectivity with an EC2 instance.

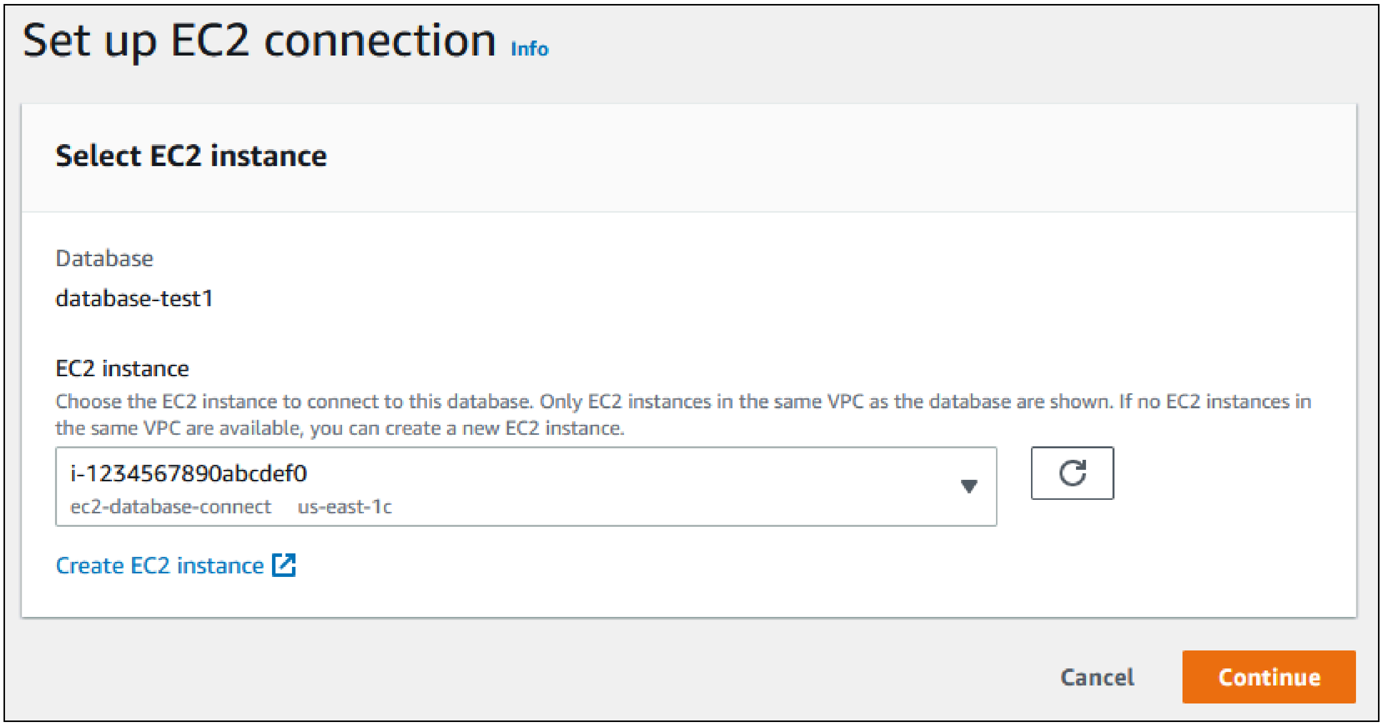
If you make changes to security groups after you configure connectivity, the changes might affect the connection between the EC2 instance and the Aurora DB cluster.

> **Note**
>
> You can only set up a connection between an EC2 instance and an Aurora DB cluster automatically by using the AWS Management Console. You can't set up a connection automatically with the AWS CLI or RDS API.

To connect an EC2 instance and an Aurora DB cluster automatically

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster.
3. From **Actions**, choose **Set up EC2 connection**.

   The **Set up EC2 connection** page appears.
4. On the **Set up EC2 connection** page, choose the EC2 instance.



Set up EC2 connection Info

Select EC2 instance

Database

database-test1

EC2 instance

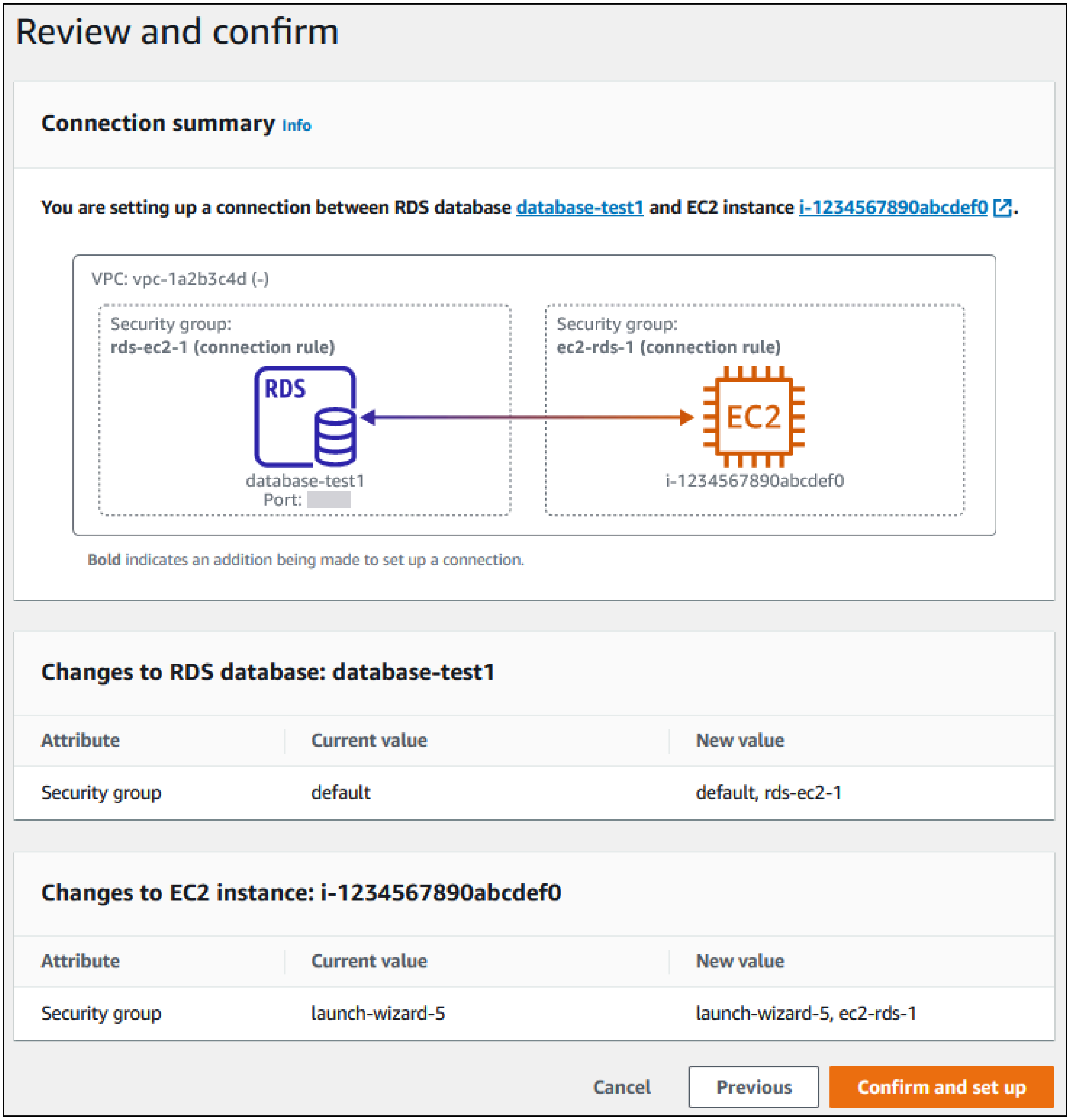
Choose the EC2 instance to connect to this database. Only EC2 instances in the same VPC as the database are shown. If no EC2 instances in the same VPC are available, you can create a new EC2 instance.

i-1234567890abcdef0
ec2-database-connect us-east-1c

If no EC2 instances exist in the same VPC, choose **Create EC2 instance** to create one. In this case, make sure the new EC2 instance is in the same VPC as the DB cluster.

5. Choose **Continue**.

   The **Review and confirm** page appears.



Review and confirm

Connection summary Info

You are setting up a connection between RDS database database-test1 and EC2 instance i-1234567890abcdef0.

VPC: vpc-1a2b3c4d (-)

Security group:
rds-ec2-1 (connection rule)

database-test1
Port:

Security group:
ec2-rds-1 (connection rule)

i-1234567890abcdef0

Bold indicates an addition being made to set up a connection.

Changes to RDS database: database-test1

Attribute	Current value	New value
Security group	default	default, rds-ec2-1

Changes to EC2 instance: i-1234567890abcdef0

Attribute	Current value	New value
Security group	launch-wizard-5	launch-wizard-5, ec2-rds-1

6. On the **Review and confirm** page, review the changes that RDS will make to set up connectivity with the EC2 instance.

   If the changes are correct, choose **Confirm and set up**.

   If the changes aren't correct, choose **Previous** or **Cancel**.

Viewing connected compute resources

You can use the AWS Management Console to view the compute resources that are connected to an Aurora DB cluster. The resources shown include compute resource connections that were set up automatically. You can set up connectivity with compute resources automatically in the following ways:

- You can select the compute resource when you create the database.

  For more information, see Creating an Amazon Aurora DB cluster.
- You can set up connectivity between an existing database and a compute resource.

  For more information, see Automatically connecting an EC2 instance and an Aurora DB cluster.

The listed compute resources don't include ones that were connected to the database manually. For example, you can allow a compute resource to access a database manually by adding a rule to the VPC security group associated with the database.

For a compute resource to be listed, the following conditions must apply:

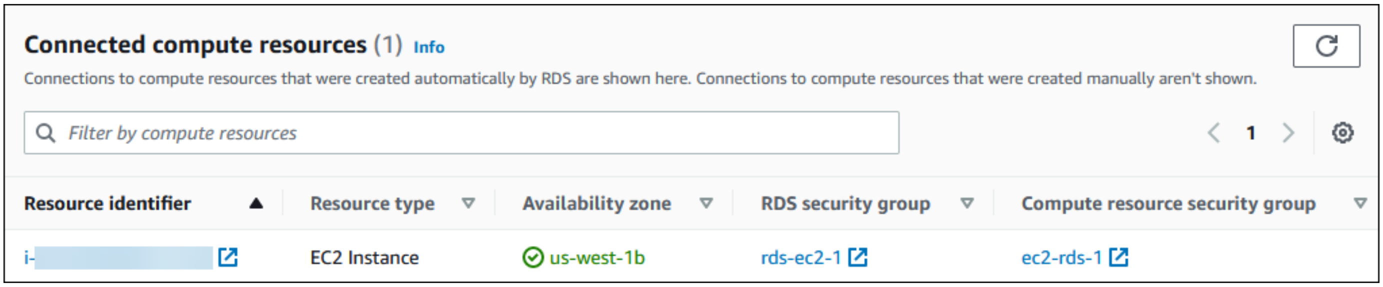
- The name of the security group associated with the compute resource matches the pattern `ec2-rds-n` (where *n* is a number).
- The security group associated with the compute resource has an outbound rule with the port range set to the port that the DB cluster uses.
- The security group associated with the compute resource has an outbound rule with the source set to a security group associated with the DB cluster.
- The name of the security group associated with the DB cluster matches the pattern `rds-ec2-n` (where *n* is a number).
- The security group associated with the DB cluster has an inbound rule with the port range set to the port that the DB cluster uses.
- The security group associated with the DB cluster has an inbound rule with the source set to a security group associated with the compute resource.

To view compute resources connected to an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Databases**, and then choose the name of the DB cluster.

3. On the **Connectivity & security** tab, view the compute resources in the **Connected compute resources**.



Connected compute resources (1) Info

Connections to compute resources that were created automatically by RDS are shown here. Connections to compute resources that were created manually aren't shown.

Filter by compute resources

Resource identifier ▲	Resource type ▽	Availability zone ▽	RDS security group ▽	Compute resource security group ▽
i-[redacted]	EC2 Instance	✓ us-west-1b	rds-ec2-1	ec2-rds-1

Connecting to a DB instance that is running a specific DB engine

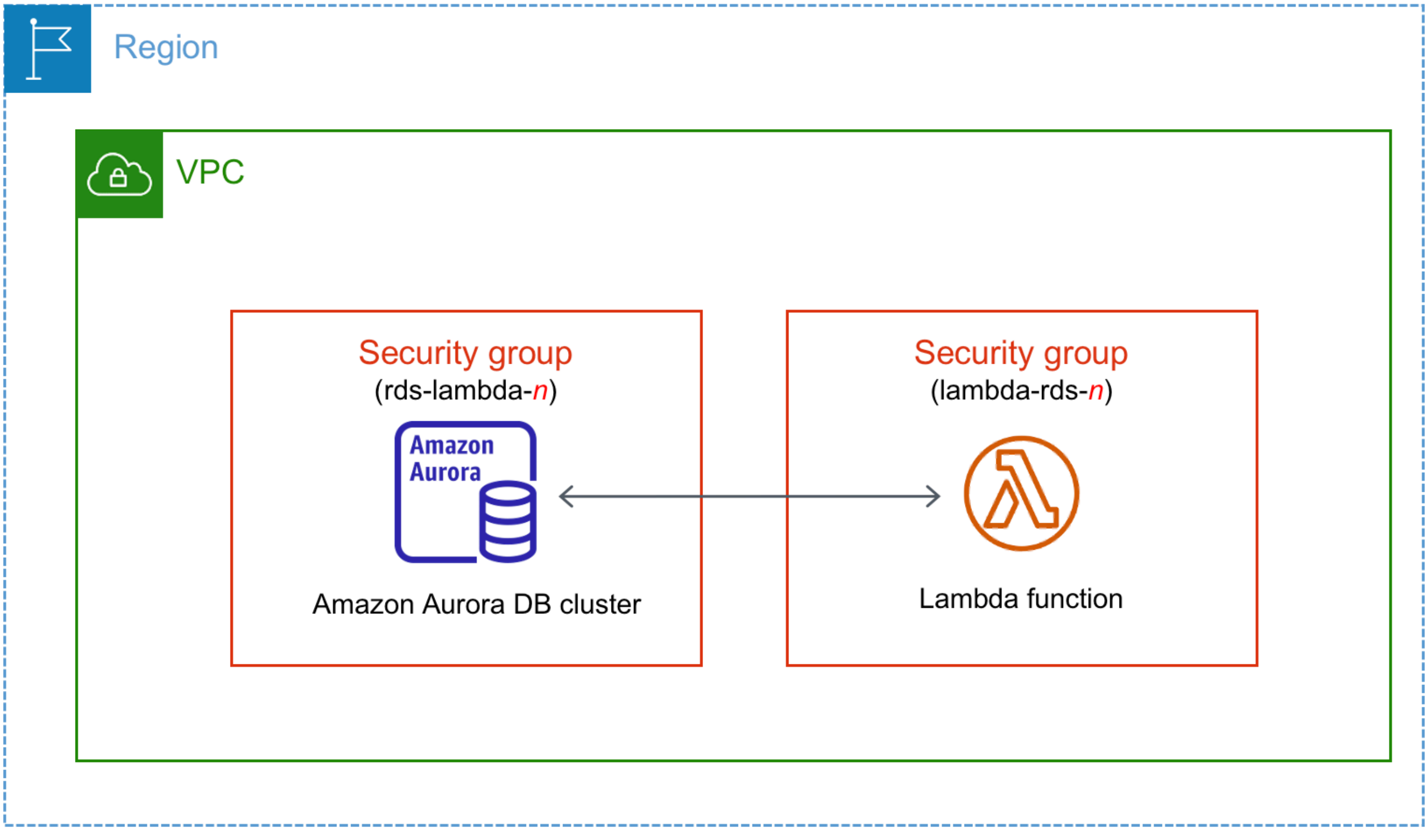
For information about connecting to a DB instance that is running a specific DB engine, follow the instructions for your DB engine:

- Connecting to an Amazon Aurora MySQL DB cluster
- Connecting to an Amazon Aurora PostgreSQL DB cluster

Automatically connecting a Lambda function and an Aurora DB cluster

You can use the Amazon RDS console to simplify setting up a connection between a Lambda function and an Aurora DB cluster. Often, your DB cluster is in a private subnet within a VPC. The Lambda function can be used by applications to access your private DB cluster.

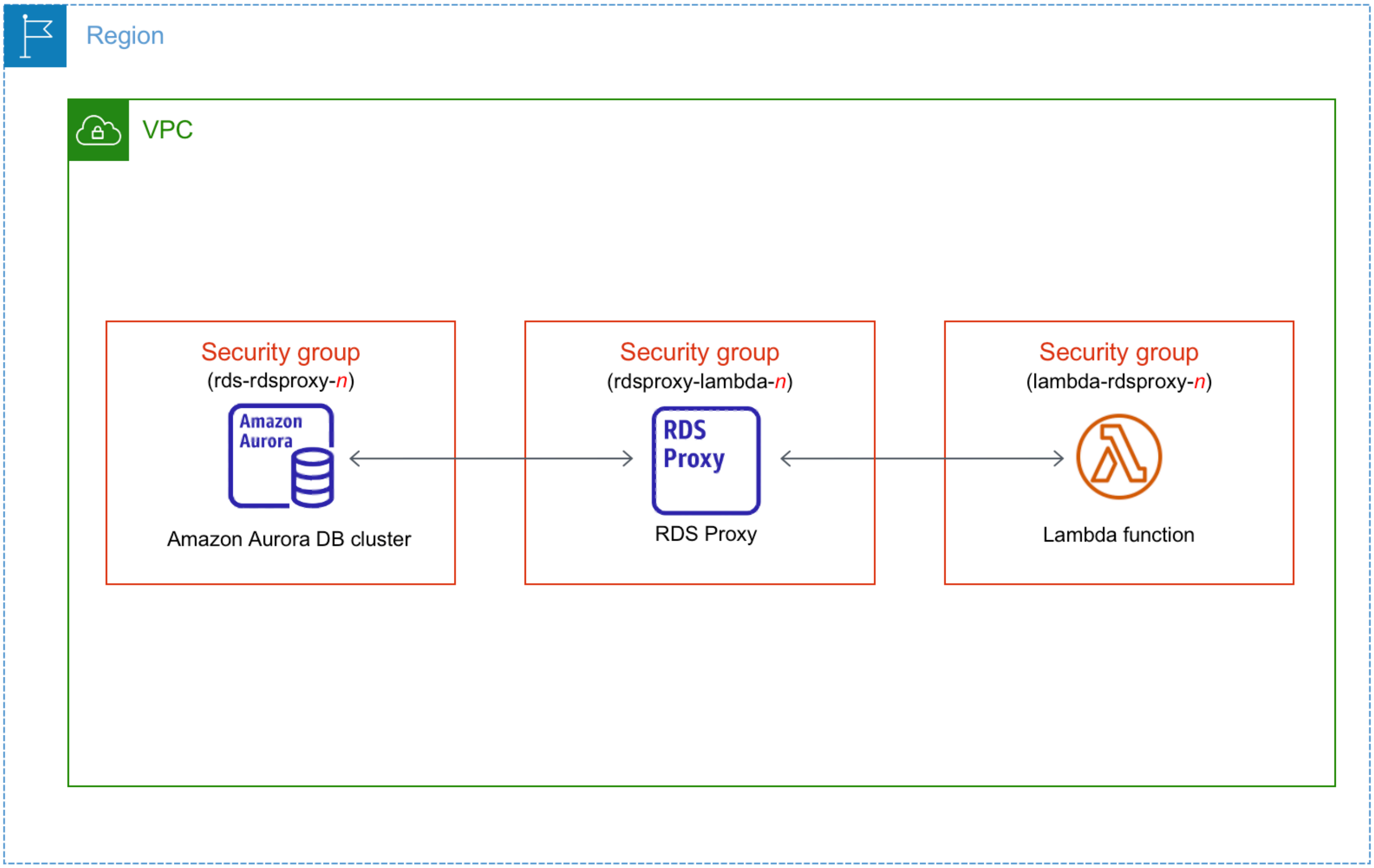
The following image shows a direct connection between your DB cluster and your Lambda function.



You can set up the connection between your Lambda function and your DB cluster through RDS Proxy to improve your database performance and resiliency. Often, Lambda functions make frequent, short database connections that benefit from connection pooling that RDS Proxy offers. You can take advantage of any AWS Identity and Access Management (IAM) authentication that you already have for Lambda functions, instead of managing database credentials in your Lambda application code. For more information, see Using Amazon RDS Proxy for Aurora.

When you use the console to connect with an existing proxy, Amazon RDS updates the proxy security group to allow connections from your DB cluster and Lambda function.

You can also create a new proxy from the same console page. When you create a proxy in the console, to access the DB cluster, you must input your database credentials or select an AWS Secrets Manager secret.



Topics

- Overview of automatic connectivity with a Lambda function
- Automatically connecting a Lambda function and an Aurora DB cluster
- Viewing connected compute resources

Overview of automatic connectivity with a Lambda function

The following are requirements for connecting a Lambda function with an Aurora DB cluster:

- The Lambda function must exist in the same VPC as the DB cluster.
- Currently, the DB cluster can't be an Aurora Serverless DB cluster or part of an Aurora global database.
- The user who sets up connectivity must have permissions to perform the following Amazon RDS, Amazon EC2, Lambda, Secrets Manager, and IAM operations:
  - Amazon RDS

  - `rds:CreateDBProxies`
  - `rds:DescribeDBClusters`
  - `rds:DescribeDBProxies`
  - `rds:ModifyDBCluster`
  - `rds:ModifyDBProxy`
  - `rds:RegisterProxyTargets`
- Amazon EC2
  - `ec2:AuthorizeSecurityGroupEgress`
  - `ec2:AuthorizeSecurityGroupIngress`
  - `ec2:CreateSecurityGroup`
  - `ec2:DeleteSecurityGroup`
  - `ec2:DescribeSecurityGroups`
  - `ec2:RevokeSecurityGroupEgress`
  - `ec2:RevokeSecurityGroupIngress`
- Lambda
  - `lambda:CreateFunctions`
  - `lambda:ListFunctions`
  - `lambda:UpdateFunctionConfiguration`
- Secrets Manager
  - `secretsmanager:CreateSecret`
  - `secretsmanager:DescribeSecret`
- IAM
  - `iam:AttachPolicy`
  - `iam:CreateRole`
  - `iam:CreatePolicy`
- AWS KMS
  - `kms:describeKey`

Note

If the DB cluster and Lambda function are in different Availability Zones, your account might incur cross-Availability Zone costs.

When you set up a connection between a Lambda function and an Aurora DB cluster, Amazon RDS configures the VPC security group for your function and for your DB cluster. If you use RDS Proxy, then Amazon RDS also configures the VPC security group for the proxy. Amazon RDS acts according to the current configuration of the security groups associated with the DB cluster, Lambda function, and proxy, as described in the following table.

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda-` *n* or if a proxy is already connected to your DB cluster, RDS checks if the `TargetHealth` of an associated proxy is `AVAILABLE`. A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC	There are one or more security groups associated with the Lambda function with a name that matches the pattern `lambda-rds-` *n* or `lambda-rdsproxy-` *n* (where *n* is a number). A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with either the VPC security group of the DB cluster or the	There are one or more security groups associated with the proxy with a name that matches the pattern `rdsproxy-lambda-` *n* (where *n* is a number). A security group that matches the pattern hasn't been modified. This security group has inbound and outbound rules with the VPC security groups of the Lambda function and the DB cluster.	Amazon RDS takes no action. A connection was already configured automatically between the Lambda function, the proxy (optional), and DB cluster. Because a connection already exists between the function, proxy, and the database, the security groups aren't modified.

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
security group of the Lambda function or proxy as the source.	proxy as the destinati on.		

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
Either of the following conditions apply: - There is no security group associated with the DB cluster with a name that matches the pattern `rds-lambda-`*n* or if the `TargetHealth` of an associated proxy is `AVAILABLE`. - There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda-`*n* or if the `TargetHealth` of an associated proxy is `AVAILABLE`. However, none of these security groups can be used for the connectio	Either of the following conditions apply: - There is no security group associated with the Lambda function with a name that matches the pattern `lambda-rds-`*n* or `lambda-rdsproxy-`*n*. - There are one or more security groups associated with the Lambda function with a name that matches the pattern `lambda-rds-`*n* or `lambda-rdsproxy-`*n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster. Amazon RDS can't use a security group	Either of the following conditions apply: - There is no security group associated with the proxy with a name that matches the pattern `rdsproxy-lambda-`*n*. - There are one or more security groups associated with the proxy with a name that matches `rdsproxy-lambda-`*n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster or Lambda function. Amazon RDS can't use a security group that doesn't have inbound and	[RDS action: create new security groups](#)

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
n with the Lambda function. Amazon RDS can't use a security group that doesn't have one inbound rule with the VPC security group of the Lambda function or proxy as the source. Amazon RDS also can't use a security group that has been modified. Examples of modifications include adding a rule or changing the port of an existing rule.	that doesn't have one outbound rule with the VPC security group of the DB cluster or proxy as the destination. Amazon RDS also can't use a security group that has been modified.	outbound rules with the VPC security group of the DB cluster and the Lambda function. Amazon RDS also can't use a security group that has been modified.	

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda-` *n* or if the `TargetHealth` of an associated proxy is `AVAILABLE` . A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the Lambda function or proxy as the source.	There are one or more security groups associated with the Lambda function with a name that matches the pattern `lambda-rds-` *n* or `lambda-rdsproxy-` *n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster. Amazon RDS can't use a security group that doesn't have one outbound rule with the VPC security group of the DB cluster or proxy as the destination. Amazon RDS also can't use a security group that has been modified.	There are one or more security groups associated with the proxy with a name that matches the pattern `rdsproxy-lambda-` *n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster or Lambda function. Amazon RDS can't use a security group that doesn't have inbound and outbound rules with the VPC security group of the DB cluster and the Lambda function. Amazon RDS also can't use a security group that has been modified.	RDS action: create new security groups

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda-` *n* or if the `TargetHealth` of an associated proxy is `AVAILABLE` . A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the Lambda function or proxy as the source.	A valid Lambda security group for the connection exists, but it isn't associated with the Lambda function. This security group has a name that matches the pattern `lambda-rds-` *n* or `lambda-rdsproxy-` *n*. It hasn't been modified. It has only one outbound rule with the VPC security group of the DB cluster or proxy as the destination.	A valid proxy security group for the connection exists, but it isn't associated with the proxy. This security group has a name that matches the pattern `rdsproxy-lambda-` *n*. It hasn't been modified. It has inbound and outbound rules with the VPC security group of the DB cluster and the Lambda function.	RDS action: associate Lambda security group

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
Either of the following conditions apply: - There is no security group associated with the DB cluster with a name that matches the pattern `rds-lambda- n` or if the `TargetHealth` of an associated proxy is `AVAILABLE`. - There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda- n` or if the `TargetHealth` of an associated proxy is `AVAILABLE`. However, Amazon RDS can't use any of these security groups for the connectio	There are one or more security groups associated with the Lambda function with a name that matches the pattern `lambda-rds- n` or `lambda-rdsproxy- n`. A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with the VPC security group of the DB instance or proxy as the destination.	There are one or more security groups associated with the proxy with a name that matches the pattern `rdsproxy-lambda- n`. A security group that matches the pattern hasn't been modified. This security group has inbound and outbound rules with the VPC security group of the DB cluster and the Lambda function.	RDS action: create new security groups

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
n with the Lambda function or proxy. Amazon RDS can't use a security group that doesn't have one inbound rule with the VPC security group of the Lambda function or proxy as the source. Amazon RDS also can't use a security group that has been modified.			

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
Either of the following conditions apply: - There is no security group associated with the DB cluster with a name that matches the pattern `rds-lambda-`*n* or if the `TargetHealth` of an associated proxy is `AVAILABLE`. - There are one or more security groups associated with the DB cluster with a name that matches the pattern `rds-lambda-`*n* or if the `TargetHealth` of an associated proxy is `AVAILABLE`. However, Amazon RDS can't use any of these security groups for the connectio	Either of the following conditions apply: - There is no security group associated with the Lambda function with a name that matches the pattern `lambda-rds-`*n* or `lambda-rdsproxy-`*n*. - There are one or more security groups associated with the Lambda function with a name that matches the pattern `lambda-rds-`*n* or `lambda-rdsproxy-`*n*. However, Amazon RDS can't use any of these security groups for the connection with the DB DB cluster. Amazon RDS can't use a security group	Either of the following conditions apply: - There is no security group associated with the proxy with a name that matches the pattern `rdsproxy-lambda-`*n*. - There are one or more security groups associated with the proxy with a name that matches `rdsproxy-lambda-`*n*. However, Amazon RDS can't use any of these security groups for the connection with the DB cluster or Lambda function. Amazon RDS can't use a security group that doesn't have inbound and	[RDS action: create new security groups]()

Current RDS security group configuration	Current Lambda security group configuration	Current proxy security group configuration	RDS action
n with the Lambda function or proxy. Amazon RDS can't use a security group that doesn't have one inbound rule with the VPC security group of the Lambda function or proxy as the source. Amazon RDS also can't use a security group that has been modified.	that doesn't have one outbound rule with the VPC security group of the DB cluster or proxy as the source. Amazon RDS also can't use a security group that has been modified.	outbound rules with the VPC security group of the DB cluster and the Lambda function. Amazon RDS also can't use a security group that has been modified.	

RDS action: create new security groups

Amazon RDS takes the following actions:

- Creates a new security group that matches the pattern `rds-lambda-`*`n`* or `rds-rdsproxy-`*`n`* (if you choose to use RDS Proxy). This security group has an inbound rule with the VPC security group of the Lambda function or proxy as the source. This security group is associated with the DB cluster and allows the function or proxy to access the DB cluster.
- Creates a new security group that matches the pattern `lambda-rds-`*`n`* or `lambda-rdsproxy-`*`n`*. This security group has an outbound rule with the VPC security group of the DB cluster or proxy as the destination. This security group is associated with the Lambda function and allows the function to send traffic to the DB cluster or send traffic through a proxy.
- Creates a new security group that matches the pattern `rdsproxy-lambda-`*`n`*. This security group has inbound and outbound rules with the VPC security group of the DB cluster and the Lambda function.

RDS action: associate Lambda security group

Amazon RDS associates the valid, existing Lambda security group with the Lambda function. This security group allows the function to send traffic to the DB cluster or send traffic through a proxy.

Automatically connecting a Lambda function and an Aurora DB cluster

You can use the Amazon RDS console to automatically connect a Lambda function to your DB cluster. This simplifies the process of setting up a connection between these resources.

You can also use RDS Proxy to include a proxy in your connection. Lambda functions make frequent short database connections that benefit from the connection pooling that RDS Proxy offers. You can also use any IAM authentication that you've already set up for your Lambda function, instead of managing database credentials in your Lambda application code.

You can connect an existing DB cluster to new and existing Lambda functions using the **Set up Lambda connection** page. The setup process automatically sets up the required security groups for you.

Before setting up a connection between a Lambda function and a DB cluster, make sure that:

- Your Lambda function and DB cluster are in the same VPC.
- You have the right permissions for your user account. For more information about the requirements, see Overview of automatic connectivity with a Lambda function.

If you change security groups after you configure connectivity, the changes might affect the connection between the Lambda function and the DB cluster.

> **Note**
>
> You can automatically set up a connection between a DB cluster and a Lambda function only in the AWS Management Console. To connect a Lambda function, all instances in the DB cluster must be in the **Available** state.

To automatically connect a Lambda function and a DB cluster

<result>

After you confirm the setup, Amazon RDS begins the process of connecting your Lambda function, RDS Proxy (if you used a proxy), and DB cluster. The console shows the **Connection details** dialog box, which lists the security group changes that allow connections between your resources.

```
</result>
```

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to connect to a Lambda function.
3. For **Actions**, choose **Set up Lambda connection**.
4. On the **Set up Lambda connection** page, under **Select Lambda function**, do either of the following:
   - If you have an existing Lambda function in the same VPC as your DB cluster, choose **Choose existing function**, and then choose the function.
   - If you don't have a Lambda function in the same VPC, choose **Create new function**, and then enter a **Function name**. The default runtime is set to Nodejs.18. You can modify the settings for your new Lambda function in the Lambda console after you complete the connection setup.
5. (Optional) Under **RDS Proxy**, select **Connect using RDS Proxy**, and then do any of the following:
   - If you have an existing proxy that you want to use, choose **Choose existing proxy**, and then choose the proxy.
   - If you don't have a proxy, and you want Amazon RDS to automatically create one for you, choose **Create new proxy**. Then, for **Database credentials**, do either of the following:

     a. Choose **Database username and password**, and then enter the **Username** and **Password** for your DB cluster.

     b. Choose **Secrets Manager secret**. Then, for **Select secret**, choose an AWS Secrets Manager secret. If you don't have a Secrets Manager secret, choose **Create new Secrets Manager secret** to [create a new secret](). After you create the secret, for **Select secret**, choose the new secret.

     After you create the new proxy, choose **Choose existing proxy**, and then choose the proxy. Note that it might take some time for your proxy to be available for connection.
6. (Optional) Expand **Connection summary** and verify the highlighted updates for your resources.
7. Choose **Set up**.

Viewing connected compute resources

You can use the AWS Management Console to view the Lambda functions that are connected to your DB cluster. The resources shown include compute resource connections that Amazon RDS set up automatically.

The listed compute resources don't include those that are manually connected to the DB cluster. For example, you can allow a compute resource to access your DB cluster manually by adding a rule to your VPC security group associated with the database.

For the console to list a Lambda function, the following conditions must apply:

- The name of the security group associated with the compute resource matches the pattern `lambda-rds-`*`n`* or `lambda-rdsproxy-`*`n`* (where *n* is a number).
- The security group associated with the compute resource has an outbound rule with the port range set to the port of the DB cluster or an associated proxy. The destination for the outbound rule must be set to a security group associated with the DB cluster or an associated proxy.
- If the configuration includes a proxy, the name of the security group attached to the proxy associated with your database matches the pattern `rdsproxy-lambda-`*`n`* (where *n* is a number).
- The security group associated with the function has an outbound rule with the port set to the port that the DB cluster or associated proxy uses. The destination must be set to a security group associated with the DB cluster or associated proxy.

To view compute resources automatically connected to an DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster.
3. On the **Connectivity & security** tab, view the compute resources under **Connected compute resources**.

Modifying an Amazon Aurora DB cluster

You can change the settings of a DB cluster to accomplish tasks such as changing its backup retention period or its database port. You can also modify DB instances in a DB cluster to accomplish tasks such as changing its DB instance class or enabling Performance Insights for it. This topic guides you through modifying an Aurora DB cluster and its DB instances, and describes the settings for each.

We recommend that you test any changes on a test DB cluster or DB instance before modifying a production DB cluster or DB instance, so that you fully understand the impact of each change. This is especially important when upgrading database versions.

Topics

Modifying the DB cluster by using the console, CLI, and API

You can modify a DB cluster using the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> Most modifications can be applied immediately or during the next scheduled maintenance window. Some modifications, such as turning on deletion protection, are applied immediately—regardless of when you choose to apply them.
> Changing the master password in the AWS Management Console is always applied immediately. However, when using the AWS CLI or RDS API, you can choose whether to apply this change immediately or during the next scheduled maintenance window.
> If you're using SSL endpoints and change the DB cluster identifier, stop and restart the DB cluster to update the SSL endpoints. For more information, see Stopping and starting an Amazon Aurora DB cluster.

Console

To modify a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB cluster that you want to modify.
3. Choose **Modify**. The **Modify DB cluster** page appears.
4. Change any of the settings that you want. For information about each setting, see Settings for Amazon Aurora.

   > **Note**
   >
   > In the AWS Management Console, some instance level changes only apply to the current DB instance, while others apply to the entire DB cluster. For information about whether a setting applies to the DB instance or the DB cluster, see the scope for the setting in Settings for Amazon Aurora. To change a setting that modifies the entire DB cluster at the instance level in the AWS Management Console, follow the instructions in Modifying a DB instance in a DB cluster.

5. When all the changes are as you want them, choose **Continue** and check the summary of modifications.
6. To apply the changes immediately, select **Apply immediately**.
7. On the confirmation page, review your changes. If they are correct, choose **Modify cluster** to save your changes.

   Alternatively, choose **Back** to edit your changes, or choose **Cancel** to cancel your changes.

AWS CLI

To modify a DB cluster using the AWS CLI, call the modify-db-cluster command. Specify the DB cluster identifier, and the values for the settings that you want to modify. For information about each setting, see Settings for Amazon Aurora.

Note

Some settings only apply to DB instances. To change those settings, follow the instructions in Modifying a DB instance in a DB cluster.

Example

The following command modifies `mydbcluster` by setting the backup retention period to 1 week (7 days).

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --backup-retention-period 7
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --backup-retention-period 7
```

RDS API

To modify a DB cluster using the Amazon RDS API, call the ModifyDBCluster operation. Specify the DB cluster identifier, and the values for the settings that you want to modify. For information about each parameter, see Settings for Amazon Aurora.

Note

Some settings only apply to DB instances. To change those settings, follow the instructions in Modifying a DB instance in a DB cluster.

Modifying a DB instance in a DB cluster

You can modify a DB instance in a DB cluster using the AWS Management Console, the AWS CLI, or the RDS API.

When you modify a DB instance, you can apply the changes immediately. To apply changes immediately, you select the **Apply Immediately** option in the AWS Management Console, you use the `--apply-immediately` parameter when calling the AWS CLI, or you set the `ApplyImmediately` parameter to `true` when using the Amazon RDS API.

If you don't choose to apply changes immediately, the changes are deferred until the next maintenance window. During the next maintenance window, any of these deferred changes are applied. If you choose to apply changes immediately, your new changes and any previously deferred changes are applied.

To see the modifications that are pending for the next maintenance window, use the describe-db-clusters AWS CLI command and check the `PendingModifiedValues` field.

> ⚠ **Important**
>
> If any of the deferred modifications require downtime, choosing **Apply immediately** can cause unexpected downtime for the DB instance. There is no downtime for the other DB instances in the DB cluster.
>
> Modifications that you defer aren't listed in the output of the `describe-pending-maintenance-actions` CLI command. Maintenance actions only include system upgrades that you schedule for the next maintenance window.

Console

To modify a DB instance in a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB instance that you want to modify.
3. For **Actions**, choose **Modify**. The **Modify DB instance** page appears.
4. Change any of the settings that you want. For information about each setting, see Settings for Amazon Aurora.

> **Note**
>
> Some settings apply to the entire DB cluster and must be changed at the cluster level. To change those settings, follow the instructions in Modifying the DB cluster by using the console, CLI, and API.
> In the AWS Management Console, some instance level changes only apply to the current DB instance, while others apply to the entire DB cluster. For information about whether a setting applies to the DB instance or the DB cluster, see the scope for the setting in Settings for Amazon Aurora.

5. When all the changes are as you want them, choose **Continue** and check the summary of modifications.
6. To apply the changes immediately, select **Apply immediately**.
7. On the confirmation page, review your changes. If they are correct, choose **Modify DB instance** to save your changes.

   Alternatively, choose **Back** to edit your changes, or choose **Cancel** to cancel your changes.

AWS CLI

To modify a DB instance in a DB cluster by using the AWS CLI, call the modify-db-instance command. Specify the DB instance identifier, and the values for the settings that you want to modify. For information about each parameter, see Settings for Amazon Aurora.

> **Note**
>
> Some settings apply to the entire DB cluster. To change those settings, follow the instructions in Modifying the DB cluster by using the console, CLI, and API.

Example

The following code modifies `mydbinstance` by setting the DB instance class to `db.r4.xlarge`. The changes are applied during the next maintenance window by using `--no-apply-immediately`. Use `--apply-immediately` to apply the changes immediately.

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier mydbinstance \
    --db-instance-class db.r4.xlarge \
    --no-apply-immediately
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier mydbinstance ^
    --db-instance-class db.r4.xlarge ^
    --no-apply-immediately
```

RDS API

To modify a DB instance by using the Amazon RDS API, call the ModifyDBInstance operation. Specify the DB instance identifier, and the values for the settings that you want to modify. For information about each parameter, see Settings for Amazon Aurora.

> **Note**
>
> Some settings apply to the entire DB cluster. To change those settings, follow the instructions in Modifying the DB cluster by using the console, CLI, and API.

Changing the password for the database master user

You can use the AWS Management Console or the AWS CLI to change the master user password.

Console

You modify the writer DB instance to change the master user password using the AWS Management Console.

To change the master user password

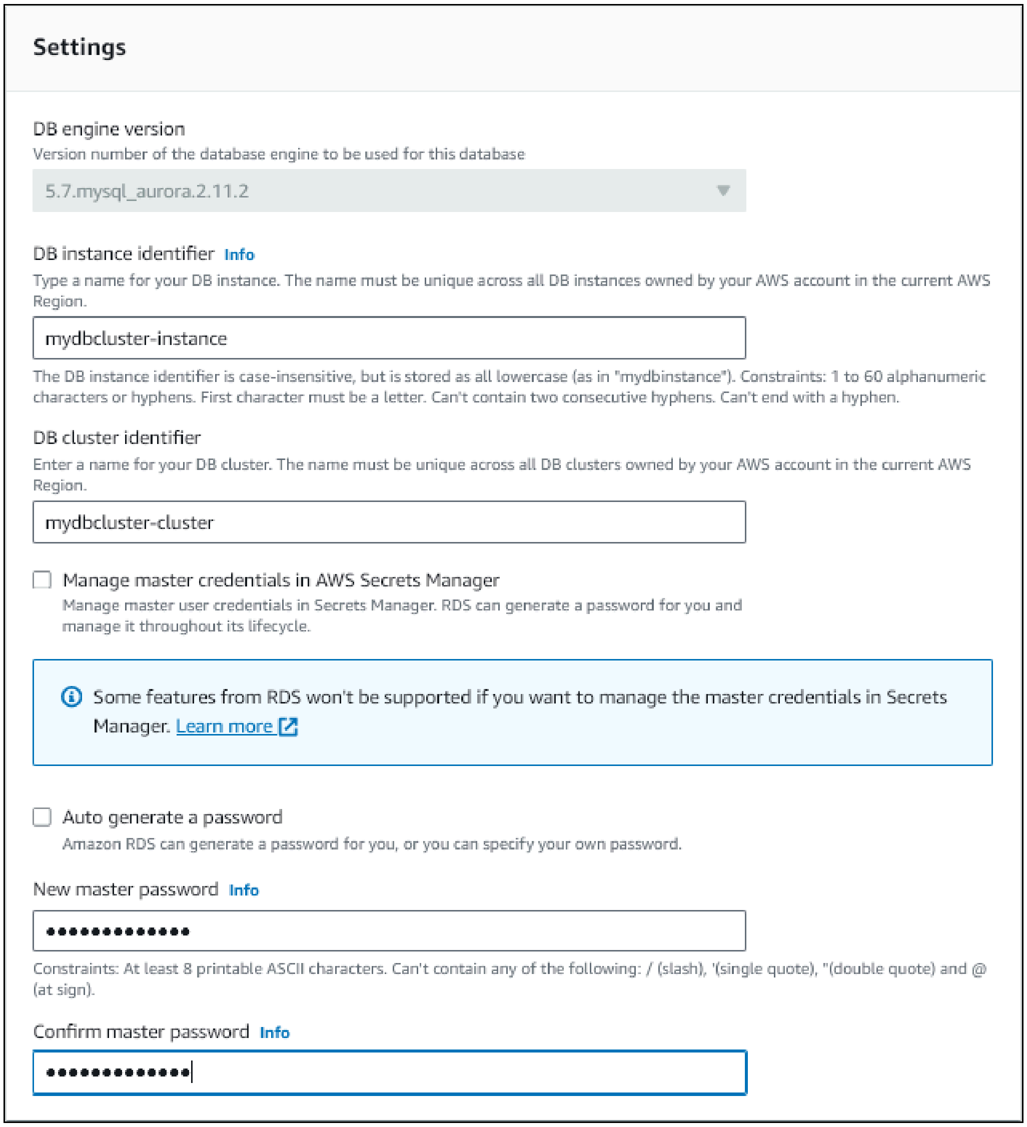
1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB instance that you want to modify.

3. For **Actions**, choose **Modify**.

   The **Modify DB instance** page appears.

4. Enter a **New master password**.

5. For **Confirm master password**, enter the same new password.



Settings

DB engine version

Version number of the database engine to be used for this database

5.7.mysql_aurora.2.11.2

DB instance identifier Info

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

mydbcluster-instance

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

DB cluster identifier

Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

mydbcluster-cluster

☐ Manage master credentials in AWS Secrets Manager

Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

Some features from RDS won't be supported if you want to manage the master credentials in Secrets Manager. Learn more

☐ Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password.

New master password Info

•••••••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password Info

•••••••••••••

6. Choose **Continue** and check the summary of modifications.

   Note

   Password changes are always applied immediately.

7. On the confirmation page, choose **Modify DB instance**.

CLI

You call the modify-db-cluster command to change the master user password using the AWS CLI. Specify the DB cluster identifier and the new password, as shown in the following examples.

You don't need to specify `--apply-immediately|--no-apply-immediately`, because password changes are always applied immediately.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --master-user-password mynewpassword
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --master-user-password mynewpassword
```

Settings for Amazon Aurora

The following table contains details about which settings you can modify, the methods for modifying the setting, and the scope of the setting. The scope determines whether the setting applies to the entire DB cluster or if it can be set only for specific DB instances.

> **Note**
>
> Additional settings are available if you are modifying an Aurora Serverless v1 or Aurora Serverless v2 DB cluster. For information about these settings, see Modifying an Aurora Serverless v1 DB cluster and Managing Aurora Serverless v2 DB clusters.
> Some settings aren't available for Aurora Serverless v1 and Aurora Serverless v2 because of their limitations. For more information, see Limitations of Aurora Serverless v1 and Requirements and limitations for Aurora Serverless v2.

Setting and description	Method	Scope	Downtime notes
Auto minor version upgrade Whether you want the DB instance to receive preferred minor engine version upgrades automatically when they become available. Upgrades are installed only during your scheduled maintenance window. For more information about engine updates, see Amazon Aurora PostgreSQL updates and Database engine updates for Amazon Aurora MySQL. For more information about the **Auto minor version upgrade** setting for Aurora MySQL, see Enabling automatic upgrades between minor Aurora MySQL versions.	**Note** This setting is enabled by default. For each new cluster, choose the appropriate value for this setting based on its importance, expected lifetime, and the amount of verification testing that you do after each upgrade. When you change this setting, perform this modification for every DB instance in your Aurora cluster. If any DB instance in your cluster has this setting turned off, the cluster isn't automatically upgraded.	The entire DB cluster	An outage doesn't occur during this change. Outages do occur during future maintenance windows when Aurora applies automatic upgrades.

Setting and description	Method	Scope	Downtime notes
	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--auto-minor-version-upgrade|--no-auto-minor-version-upgrade` option. Using the RDS API, call `ModifyDBInstance` and set the `AutoMinorVersionUpgrade` parameter.		

Setting and description	Method	Scope	Downtime notes
Backup retention period The number of days that automatic backups are retained. The minimum value is 1. For more information, see Backups.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--backup-retention-period` option. Using the RDS API, call `ModifyDBCluster` and set the `BackupRetentionPeriod` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Backup window (Start time) The time range during which automated backups of your database occurs. The backup window is a start time in Universal Coordinated Time (UTC), and a duration in hours. Aurora backups are continuous and incremental, but the backup window is used to create a daily system backup that is preserved within the backup retention period. You can copy it to preserve it outside of the retention period. The maintenance window and the backup window for the DB cluster can't overlap.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--preferred-backup-window` option. Using the RDS API, call `ModifyDBCluster` and set the `PreferredBackupWindow` parameter.	The entire DB cluster.	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
For more information, see Backup window.			
Capacity settings The scaling properties of an Aurora Serverless v1 DB cluster. You can only modify scaling properties for DB clusters in `serverless` DB engine mode. For information about Aurora Serverless v1, see Using Amazon Aurora Serverless v1.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--scaling-configuration` option. Using the RDS API, call `ModifyDBCluster` and set the `ScalingConfiguration` parameter.	The entire DB cluster	An outage doesn't occur during this change. The change occurs immediately. This setting ignores the apply immediately setting.

Setting and description	Method	Scope	Downtime notes
Certificate authority The certificate authority (CA) for the server certificate used by the DB instance.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--ca-certificate-identifier` option. Using the RDS API, call `ModifyDBInstance` and set the `CACertificateIdentifier` parameter.	Only the specified DB instance	An outage only occurs if the DB engine doesn't support rotation without restart. You can use the describe-db-engine-versions AWS CLI command to determine whether the DB engine supports rotation without restart.

Setting and description	Method	Scope	Downtime notes
Cluster storage configuration The storage type for the DB cluster: **Aurora I/O-Optimized** or **Aurora Standard**. For more information, see Storage configurations for Amazon Aurora DB clusters.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--storage-type` option. Using the RDS API, call `ModifyDBCluster` and set the `StorageType` parameter.	The entire DB cluster	Changing the storage type of an Aurora PostgreSQL DB cluster with Optimized Reads instance classes causes an outage. This does not occur when changing storage types for clusters with other instance class types. For more information on the DB instance class types, see DB instance class types.

Setting and description	Method	Scope	Downtime notes
Copy tags to snapshots Select to specify that tags defined for this DB cluster are copied to DB snapshots created from this DB cluster. For more information, see Tagging Amazon RDS resources.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--copy-tags-to-snapshot` or `--no-copy-tags-to-snapshot` option. Using the RDS API, call `ModifyDBCluster` and set the `CopyTagsToSnapshot` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Data API You can access Aurora Serverless v1 with web services–based applications, including AWS Lambda and AWS AppSync. This setting only applies to an Aurora Serverless v1 DB cluster. For more information, see Using RDS Data API.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--enable-http-endpoint` option. Using the RDS API, call `ModifyDBCluster` and set the `EnableHttpEndpoint` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Database authentication The database authentication you want to use. For MySQL: - Choose **Password authentication** to authenticate database users with database passwords only. - Choose **Password and IAM database authentication** to authenticate database users with database passwords and user credentials through IAM users and roles. For more information, see [IAM database authentication](#). For PostgreSQL: - Choose **IAM database authentication**	Using the AWS Management Console, [Modifying the DB cluster by using the console, CLI, and API](#). Using the AWS CLI, run [modify-db-cluster](#) and set the following options: - For IAM authentication, set the `--enable-iam-database-authentication|--no-enable-iam-database-authentication` option. - For Kerberos authentication, set the `--domain` and `--domain-iam-role-name` options. Using the RDS API, call [ModifyDBCluster](#) and set the following parameters:	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
to authenticate database users with database passwords and user credentials through users and roles. For more information, see IAM database authentication. - Choose **Kerberos authentication** to authenticate database passwords and user credentials using Kerberos authentication. For more information, see Using Kerberos authentication with Aurora PostgreSQL.	- For IAM authentication, set the `EnableIAMDatabaseAuthentication` parameter. - For Kerberos authentication, set the `Domain` and `DomainIAMRoleName` parameters.		

Setting and description	Method	Scope	Downtime notes
Database port The port that you want to use to access the DB cluster.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--port` option. Using the RDS API, call `ModifyDBCluster` and set the `Port` parameter.	The entire DB cluster	An outage occurs during this change. All of the DB instances in the DB cluster are rebooted immediately.

Setting and description	Method	Scope	Downtime notes
DB cluster identifier The DB cluster identifier. This value is stored as a lowercase string. When you change the DB cluster identifier, the DB cluster endpoints change. The endpoints of the DB instances in the DB cluster don't change.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--new-db-cluster-identifier` option. Using the RDS API, call `ModifyDBCluster` and set the `NewDBClusterIdentifier` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
DB cluster parameter group The DB cluster parameter group that you want associated with the DB cluster. For more information, see Working with parameter groups.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--db-cluster-parameter-group-name` option. Using the RDS API, call `ModifyDBCluster` and set the `DBClusterParameterGroupName` parameter.	The entire DB cluster	An outage doesn't occur during this change. When you change the parameter group, changes to some parameters are applied to the DB instances in the DB cluster immediately without a reboot. Changes to other parameters are applied only after the DB instances in the DB cluster are rebooted.

Setting and description	Method	Scope	Downtime notes
DB instance class The DB instance class that you want to use. For more information, see Aurora DB instance classes.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--db-instance-class` option. Using the RDS API, call `ModifyDBInstance` and set the `DBInstanceClass` parameter.	Only the specified DB instance	An outage occurs during this change.

Setting and description	Method	Scope	Downtime notes
DB instance identifier The DB instance identifier. This value is stored as a lowercase string.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--new-db-instance-identifier` option. Using the RDS API, call `ModifyDBInstance` and set the `NewDBInstanceIdentifier` parameter.	Only the specified DB instance	Downtime occurs during this change. RDS restarts the DB instance to update the following: - Aurora MySQL – `SERVER_ID` column in the `information_schema.replica_host_status` table - Aurora PostgreSQL – `server_id` column in the `aurora_replica_status()` function

Setting and description	Method	Scope	Downtime notes
DB parameter group The DB parameter group that you want associated with the DB instance. For more information, see Working with parameter groups.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--db-parameter-group-name` option. Using the RDS API, call `ModifyDBInstance` and set the `DBParameterGroupName` parameter.	Only the specified DB instance	An outage doesn't occur during this change. When you associate a new DB parameter group with a DB instance, the modified static and dynamic parameters are applied only after the DB instance is rebooted. However, if you modify dynamic parameters in the DB parameter group after you associate it with the DB instance, these changes are applied immediately without a reboot. For more information, see Working with parameter groups and Rebooting an Amazon Aurora DB cluster or Amazon Aurora DB instance.

Setting and description	Method	Scope	Downtime notes
Deletion protection **Enable deletion protection** to prevent your DB cluster from being deleted. For more information, see Deletion protection for Aurora clusters.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--deletion-protection|--no-deletion-protection` option. Using the RDS API, call `ModifyDBCluster` and set the `DeletionProtection` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Engine version The version of the DB engine that you want to use. Before you upgrade your production DB cluster, we recommend that you test the upgrade process on a test DB cluster to verify its duration and to validate your applications.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--engine-version` option. Using the RDS API, call `ModifyDBCluster` and set the `EngineVersion` parameter.	The entire DB cluster	An outage occurs during this change.

Setting and description	Method	Scope	Downtime notes
Enhanced monitoring **Enable enhanced monitoring** to enable gathering metrics in real time for the operating system that your DB instance runs on. For more information, see Monitoring OS metrics with Enhanced Monitoring.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--monitoring-role-arn` and `--monitoring-interval` options. Using the RDS API, call `ModifyDBInstance` and set the `MonitoringRoleArn` and `MonitoringInterval` parameters.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Log exports Select the log types to publish to Amazon CloudWatch Logs. For more information, see Aurora MySQL database log files.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--cloudwatch-logs-export-configuration` option. Using the RDS API, call `ModifyDBCluster` and set the `CloudwatchLogsExportConfiguration` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Maintenance window The time range during which system maintenance occurs. System maintenance includes upgrades, if applicable. The maintenance window is a start time in Universal Coordinated Time (UTC), and a duration in hours. If you set the window to the current time, there must be at least 30 minutes between the current time and end of the window to ensure any pending changes are applied. You can set the maintenance window independently for the DB cluster and for each DB instance in the DB cluster. When the scope of a modification is the entire DB cluster, the modificat	To change the maintenance window for the DB cluster using the AWS Management Console, [Modifying the DB cluster by using the console, CLI, and API](). To change the maintenance window for a DB instance using the AWS Management Console, [Modifying a DB instance in a DB cluster](). To change the maintenance window for the DB cluster using the AWS CLI, run [`modify-db-cluster`]() and set the `--preferred-maintenance-window` option. To change the maintenance window for a DB instance using the AWS CLI, run [`modify-db-`]()	The entire DB cluster or a single DB instance	If there are one or more pending actions that cause an outage, and the maintenance window is changed to include the current time, then those pending actions are applied immediately, and an outage occurs.

Setting and description	Method	Scope	Downtime notes
ion is performed during the DB cluster maintenance window. When the scope of a modification is the a DB instance, the modification is performed during maintenance window of that DB instance. The maintenance window and the backup window for the DB cluster can't overlap. For more information, see The Amazon RDS maintenance window.	instance and set the `--preferred-maintenance-window` option. To change the maintenance window for the DB cluster using the RDS API, call `ModifyDBCluster` and set the `PreferredMaintenanceWindow` parameter. To change the maintenance window for a DB instance using the RDS API, call `ModifyDBInstance` and set the `PreferredMaintenanceWindow` parameter.		

Setting and description	Method	Scope	Downtime notes
Manage master credentials in AWS Secrets Manager Select **Manage master credentials in AWS Secrets Manager** to manage the master user password in a secret in Secrets Manager. Optionally, choose a KMS key to use to protect the secret. Choose from the KMS keys in your account, or enter the key from a different account. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. If Aurora is already managing the master user password for the DB cluster, you can rotate the master user password by choosing **Rotate secret immediately**.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-cluster` and set the `--manage-master-user-password | --no-manage-master-user-password` and `--master-user-secret-kms-key-id` options. To rotate the master user password immediately, set the `--rotate-master-user-password` option. Using the RDS API, call `ModifyDBCluster` and set the `ManageMasterUserPassword` and `MasterUserSecretKmsKeyId` parameter	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
For more information, see Password management with Amazon Aurora and AWS Secrets Manager.	s. To rotate the master user password immediately, set the `RotateMasterUserPassword` parameter to `true`.		

Setting and description	Method	Scope	Downtime notes
Network type The IP addressing protocols supported by the DB cluster. **IPv4** to specify that resources can communicate with the DB cluster only over the IPv4 addressing protocol. **Dual-stack mode** to specify that resources can communicate with the DB cluster over IPv4, IPv6, or both. Use dual-stack mode if you have any resources that must communicate with your DB cluster over the IPv6 addressing protocol. To use dual-stack mode, make sure at least two subnets spanning two Availability Zones that support both the IPv4 and IPv6 network protocol. Also, make sure you associate an IPv6	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--network-type` option. Using the RDS API, call `ModifyDBCluster` and set the `NetworkType` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
CIDR block with subnets in the DB subnet group you specify. For more information, see Amazon Aurora IP addressing.			
New master password The password for your master user. - For Aurora MySQL, the password must contain 8–41 printable ASCII characters. - For Aurora PostgreSQL, it must contain 8–99 printable ASCII characters. - It can't contain /, ", @, or a space.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-cluster` and set the `--master-user-password` option. Using the RDS API, call `ModifyDBCluster` and set the `MasterUserPassword` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Performance Insights Whether to enable Performance Insights, a tool that monitors your DB instance load so that you can analyze and troubleshoot your database performance. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--enable-performance-insights|--no-enable-performance-insights` option. Using the RDS API, call `ModifyDBInstance` and set the `EnablePerformanceInsights` parameter.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Performance Insights AWS KMS key The AWS KMS key identifier for encryption of Performance Insights data. The KMS key identifier is the Amazon Resource Name (ARN), key identifier, or key alias for the KMS key. For more information, see Turning Performance Insights on and off for Aurora.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--performance-insights-kms-key-id` option. Using the RDS API, call `ModifyDBInstance` and set the `PerformanceInsightsKMSKeyId` parameter.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Performance Insights retention period The amount of time, in days, to retain Performance Insights data. The retention setting in the free tier is **Default (7 days)**. To retain your performance data for longer, specify 1–24 months. For more information about retention periods, see Pricing and data retention for Performance Insights. For more information, see Turning Performance Insights on and off for Aurora.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--performance-insights-retention-period` option. Using the RDS API, call `ModifyDBInstance` and set the `PerformanceInsightsRetentionPeriod` parameter.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Promotion tier A value that specifies the order in which an Aurora Replica is promoted to the primary instance in a DB cluster, after a failure of the existing primary instance. For more information, see Fault tolerance for an Aurora DB cluster.	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--promotion-tier` option. Using the RDS API, call `ModifyDBInstance` and set the `PromotionTier` parameter.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Public access **Publicly accessible** to give the DB instance a public IP address, meaning that it's accessible outside the VPC. To be publicly accessible, the DB instance also has to be in a public subnet in the VPC. **Not publicly accessible** to make the DB instance accessible only from inside the VPC. For more information, see Hiding a DB cluster in a VPC from the internet. To connect to a DB instance from outside of its Amazon VPC, the DB instance must be publicly accessible, access must be granted using the inbound rules of the DB instance's security group, and other requirements	Using the AWS Management Console, Modifying a DB instance in a DB cluster. Using the AWS CLI, run `modify-db-instance` and set the `--publicly-accessible|--no-publicly-accessible` option. Using the RDS API, call `ModifyDBInstance` and set the `PubliclyAccessible` parameter.	Only the specified DB instance	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
must be met. For more information, see Can't connect to Amazon RDS DB instance. If your DB instance is isn't publicly accessible, you can also use an AWS Site-to-Site VPN connection or an AWS Direct Connect connection to access it from a private network. For more information, see Internetwork traffic privacy.			

Setting and description	Method	Scope	Downtime notes
Serverless v2 capacity settings The database capacity of an Aurora Serverless v2 DB cluster, measured in Aurora Capacity Units (ACUs). For more information, see Setting the Aurora Serverless v2 capacity range for a cluster.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--serverless-v2-scaling-configuration` option. Using the RDS API, call `ModifyDBCluster` and set the `ServerlessV2ScalingConfiguration` parameter.	The entire DB cluster	An outage doesn't occur during this change. The change occurs immediately. This setting ignores the apply immediately setting.

Setting and description	Method	Scope	Downtime notes
Security group The security group you want associated with the DB cluster. For more information, see Controlling access with security groups.	Using the AWS Management Console, Modifying the DB cluster by using the console, CLI, and API. Using the AWS CLI, run `modify-db-cluster` and set the `--vpc-security-group-ids` option. Using the RDS API, call `ModifyDBCluster` and set the `VpcSecurityGroupIds` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Setting and description	Method	Scope	Downtime notes
Target Backtrack window The amount of time you want to be able to backtrack your DB cluster, in seconds. This setting is available only for Aurora MySQL and only if the DB cluster was created with Backtrack enabled.	Using the AWS Management Console, [Modifying the DB cluster by using the console, CLI, and API](#). Using the AWS CLI, run [`modify-db-cluster`](#) and set the `--backtrack-window` option. Using the RDS API, call [`ModifyDBCluster`](#) and set the `BacktrackWindow` parameter.	The entire DB cluster	An outage doesn't occur during this change.

Settings that don't apply to Amazon Aurora DB clusters

The following settings in the AWS CLI command [`modify-db-cluster`](#) and the RDS API operation [`ModifyDBCluster`](#) don't apply to Amazon Aurora DB clusters.

> **Note**
>
> You can't use the AWS Management Console to modify these settings for Aurora DB clusters.

AWS CLI setting	RDS API setting
`--allocated-storage`	`AllocatedStorage`

AWS CLI setting	RDS API setting
`--auto-minor-version-upgrade | --no-auto-minor-version-upgrade`	`AutoMinorVersionUpgrade`
`--db-cluster-instance-class`	`DBClusterInstanceClass`
`--enable-performance-insights | --no-enable-performance-insights`	`EnablePerformanceInsights`
`--iops`	`Iops`
`--monitoring-interval`	`MonitoringInterval`
`--monitoring-role-arn`	`MonitoringRoleArn`
`--option-group-name`	`OptionGroupName`
`--performance-insights-kms-key-id`	`PerformanceInsightsKMSKeyId`
`--performance-insights-retention-period`	`PerformanceInsightsRetentionPeriod`

Settings that don't apply to Amazon Aurora DB instances

The following settings in the AWS CLI command `modify-db-instance` and the RDS API operation `ModifyDBInstance` don't apply to Amazon Aurora DB instances.

> **Note**
>
> You can't use the AWS Management Console to modify these settings for Aurora DB instances.

AWS CLI setting	RDS API setting
`--allocated-storage`	`AllocatedStorage`

AWS CLI setting	RDS API setting
`--allow-major-version-upgrade|--no-allow-major-version-upgrade`	`AllowMajorVersionUpgrade`
`--copy-tags-to-snapshot|--no-copy-tags-to-snapshot`	`CopyTagsToSnapshot`
`--domain`	`Domain`
`--db-security-groups`	`DBSecurityGroups`
`--db-subnet-group-name`	`DBSubnetGroupName`
`--domain-iam-role-name`	`DomainIAMRoleName`
`--multi-az|--no-multi-az`	`MultiAZ`
`--iops`	`Iops`
`--license-model`	`LicenseModel`
`--network-type`	`NetworkType`
`--option-group-name`	`OptionGroupName`
`--processor-features`	`ProcessorFeatures`
`--storage-type`	`StorageType`
`--tde-credential-arn`	`TdeCredentialArn`
`--tde-credential-password`	`TdeCredentialPassword`
`--use-default-processor-features|--no-use-default-processor-features`	`UseDefaultProcessorFeatures`

Adding Aurora Replicas to a DB cluster

An Aurora DB cluster with replication has one primary DB instance and up to 15 Aurora Replicas. The primary DB instance supports read and write operations, and performs all data modifications to the cluster volume. Aurora Replicas connect to the same storage volume as the primary DB instance, but support read operations only. You use Aurora Replicas to offload read workloads from the primary DB instance. For more information, see Aurora Replicas.

Amazon Aurora Replicas have the following limitations:

- You can't create an Aurora Replica for an Aurora Serverless v1 DB cluster. Aurora Serverless v1 has a single DB instance that scales up and down automatically to support all database read and write operations.

  However, you can add reader instances to Aurora Serverless v2 DB clusters. For more information, see Adding an Aurora Serverless v2 reader.

We recommend that you distribute the primary instance and Aurora Replicas of your Aurora DB cluster over multiple Availability Zones to improve the availability of your DB cluster. For more information, see Region availability.

To remove an Aurora Replica from an Aurora DB cluster, delete the Aurora Replica by following the instructions in Deleting a DB instance from an Aurora DB cluster.

> **Note**
>
> Amazon Aurora also supports replication with an external database, such as an RDS DB instance. The RDS DB instance must be in the same AWS Region as Amazon Aurora. For more information, see Replication with Amazon Aurora.

You can add Aurora Replicas to a DB cluster using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To add an Aurora replica to a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB cluster where you want to add the new DB instance.
3. Make sure that both the cluster and the primary instance are in the **Available** state. If the DB cluster or the primary instance are in a transitional state such as **Creating**, you can't add a replica.

   If the cluster doesn't have a primary instance, create one using the create-db-instance AWS CLI command. This situation can arise if you used the CLI to restore a DB cluster snapshot and then view the cluster in the AWS Management Console.
4. For **Actions**, choose **Add reader**.

   The **Add reader** page appears.
5. On the **Add reader** page, specify options for your Aurora Replica. The following table shows settings for an Aurora Replica.

For this option	Do this
Availability zone	Determine if you want to specify a particular Availability Zone. The list includes only those Availability Zones that are mapped to the DB subnet group that you chose when you created the DB cluster. For more information about Availability Zones, see Regions and Availability Zones.
Publicly accessible	Select `Yes` to give the Aurora Replica a public IP address; otherwise, select `No`. For more information about hiding Aurora Replicas from public access, see Hiding a DB cluster in a VPC from the internet.

For this option	Do this
Encryption	Select `Enable encryption` to enable encryption at rest for this Aurora Replica. For more information, see Encrypting Amazon Aurora resources.
DB instance class	Select a DB instance class that defines the processing and memory requirements for the Aurora Replica. For more information about DB instance class options, see Aurora DB instance classes.
Aurora replica source	Select the identifier of the primary instance to create an Aurora Replica for.
DB instance identifier	Enter a name for the instance that is unique for your account in the AWS Region you selected. You might choose to add some intelligence to the name such as including the AWS Region and DB engine you selected, for example **`aurora-read-instance1`** .
Priority	Choose a failover priority for the instance. If you don't select a value, the default is **tier-1**. This priority determines the order in which Aurora Replicas are promoted when recovering from a primary instance failure. For more information, see Fault tolerance for an Aurora DB cluster.
Database port	The port for an Aurora Replica is the same as the port for the DB cluster.
DB parameter group	Select a parameter group. Aurora has a default parameter group you can use, or you can create your own parameter group. For more information about parameter groups, see Working with parameter groups.

For this option	Do this
Performance Insights	The **Turn on Performance Insights** check box is selected by default. The value isn't inherited from the writer instance. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.
Enhanced monitoring	Choose **Enable enhanced monitoring** to enable gathering metrics in real time for the operating system that your DB cluster runs on. For more information, see Monitoring OS metrics with Enhanced Monitoring.
Monitoring Role	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Choose the IAM role that you created to permit Amazon RDS to communicate with Amazon CloudWatch Logs for you, or choose **Default** to have RDS create a role for you named `rds-monitoring-role` . For more information, see Monitoring OS metrics with Enhanced Monitoring.
Granularity	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Set the interval, in seconds, between when metrics are collected for your DB cluster.

For this option	Do this
Auto minor version upgrade	Select **Enable auto minor version upgrade** if you want to enable your Aurora DB cluster to receive minor DB Engine version upgrades automatically when they become available. The **Auto minor version upgrade** setting applies to both Aurora PostgreSQL and Aurora MySQL DB clusters. For Aurora MySQL 2.x clusters, this setting upgrades the clusters to a maximum version of 2.07.2. For more information about engine updates for Aurora PostgreSQL, see Amazon Aurora PostgreSQL updates. For more information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.

6. Choose **Add reader** to create the Aurora Replica.

AWS CLI

To create an Aurora Replica in your DB cluster, run the create-db-instance AWS CLI command. Include the name of the DB cluster as the `--db-cluster-identifier` option. You can optionally specify an Availability Zone for the Aurora Replica using the `--availability-zone` parameter, as shown in the following examples.

For example, the following command creates a new MySQL 5.7–compatible Aurora Replica named `sample-instance-us-west-2a`.

For Linux, macOS, or Unix:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a \
    --db-cluster-identifier sample-cluster --engine aurora-mysql --db-instance-class
 db.r5.large \
    --availability-zone us-west-2a
```

For Windows:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a ^
    --db-cluster-identifier sample-cluster --engine aurora-mysql --db-instance-class
 db.r5.large ^
    --availability-zone us-west-2a
```

The following command creates a new MySQL 5.7–compatible Aurora Replica named `sample-instance-us-west-2a`.

For Linux, macOS, or Unix:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a \
    --db-cluster-identifier sample-cluster --engine aurora-mysql --db-instance-class
 db.r5.large \
    --availability-zone us-west-2a
```

For Windows:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a ^
    --db-cluster-identifier sample-cluster --engine aurora --db-instance-class
 db.r5.large ^
    --availability-zone us-west-2a
```

The following command creates a new PostgreSQL-compatible Aurora Replica named `sample-instance-us-west-2a`.

For Linux, macOS, or Unix:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a \
    --db-cluster-identifier sample-cluster --engine aurora-postgresql --db-instance-
class db.r5.large \
    --availability-zone us-west-2a
```

For Windows:

```
aws rds create-db-instance --db-instance-identifier sample-instance-us-west-2a ^
    --db-cluster-identifier sample-cluster --engine aurora-postgresql --db-instance-
class db.r5.large ^
    --availability-zone us-west-2a
```

RDS API

To create an Aurora Replica in your DB cluster, call the CreateDBInstance operation. Include the name of the DB cluster as the `DBClusterIdentifier` parameter. You can optionally specify an Availability Zone for the Aurora Replica using the `AvailabilityZone` parameter.

Managing performance and scaling for Aurora DB clusters

You can use the following options to manage performance and scaling for Aurora DB clusters and DB instances:

Topics

Storage scaling

Aurora storage automatically scales with the data in your cluster volume. As your data grows, your cluster volume storage expands up to a maximum of 128 tebibytes (TiB) or 64 TiB. The maximum size depends on the DB engine version. To learn what kinds of data are included in the cluster volume, see Amazon Aurora storage and reliability. For details about the maximum size for a specific version, see Amazon Aurora size limits.

The size of your cluster volume is evaluated on an hourly basis to determine your storage costs. For pricing information, see the Aurora pricing page.

Even though an Aurora cluster volume can scale up in size to many tebibytes, you are only charged for the space that you use in the volume. The mechanism for determining billed storage space depends on the version of your Aurora cluster.

- When Aurora data is removed from the cluster volume, the overall billed space decreases by a comparable amount. This dynamic resizing behavior happens when underlying tablespaces are dropped or reorganized to require less space. Thus, you can reduce storage charges by dropping tables and databases that you no longer need. Dynamic resizing applies to certain Aurora versions. The following are the Aurora versions where the cluster volume dynamically resizes as you remove data:

Aurora MySQL	• Version 3 (compatible with MySQL 8.0): all supported versions

	• Version 2 (compatible with MySQL 5.7): 2.11 and higher
Aurora PostgreSQL	All supported versions
Aurora Serverless v2	All supported versions
Aurora Serverless v1	All supported versions

- In Aurora versions lower than those in the preceding list, the cluster volume can reuse space that's freed up when you remove data, but the volume itself never decreases in size.
- This feature is being deployed in phases to the AWS Regions where Aurora is available. Depending on the Region where your cluster is, this feature might not be available yet.

Dynamic resizing applies to operations that physically remove or resize tablespaces within the cluster volume. Thus, it applies to SQL statements such as `DROP TABLE`, `DROP DATABASE`, `TRUNCATE TABLE`, and `ALTER TABLE ... DROP PARTITION`. It doesn't apply to deleting rows using the `DELETE` statement. If you delete a large number of rows from a table, you can run the Aurora MySQL `OPTIMIZE TABLE` statement or use the Aurora PostgreSQL `pg_repack` extension afterward to reorganize the table and dynamically resize the cluster volume.

For Aurora MySQL, the following considerations apply:

- After you upgrade your DB cluster to a DB engine version that supports dynamic resizing, and when the feature is enabled in that specific AWS Region, any space that's later freed by certain SQL statements, such as `DROP TABLE`, is reclaimable.

  If the feature is explicitly disabled in a particular AWS Region, the space might only be reusable —and not reclaimable—even on versions that support dynamic resizing.

  The feature was enabled for specific DB engine versions (1.23.0–1.23.4, 2.09.0–2.09.3, and 2.10.0) between November 2020 and March 2022, and is enabled by default on any subsequent versions.
- A table is stored internally in one or more contiguous fragments of varying sizes. While running `TRUNCATE TABLE` operations, the space corresponding to the first fragment is reusable and not reclaimable. Other fragments are reclaimable. During `DROP TABLE` operations, space corresponding to the entire tablespace is reclaimable.

- The innodb_file_per_table parameter affects how table storage is organized. When tables are part of the system tablespace, dropping the table doesn't reduce the size of the system tablespace. Thus, make sure to set innodb_file_per_table to 1 for Aurora MySQL DB clusters to take full advantage of dynamic resizing.
- In version 2.11 and higher, the InnoDB temporary tablespace is dropped and re-created on restart. This releases the space occupied by the temporary tablespace to the system, and then the cluster volume resizes. To take full advantage of the dynamic resizing feature, we recommend that you upgrade your DB cluster to version 2.11 or higher.

> **Note**
>
> The dynamic resizing feature doesn't reclaim space immediately when tables in tablespaces are dropped, but gradually at a rate of approximately 10 TB per day. Space in the system tablespace isn't reclaimed, because the system tablespace is never removed. Unreclaimed free space in a tablespace is reused when an operation needs space in that tablespace. The dynamic resizing feature can reclaim storage space only when the cluster is in an available state.

You can check how much storage space a cluster is using by monitoring the VolumeBytesUsed metric in CloudWatch. For more information on storage billing, see How Aurora data storage is billed.

- In the AWS Management Console, you can see this figure in a chart by viewing the Monitoring tab on the details page for the cluster.
- With the AWS CLI, you can run a command similar to the following Linux example. Substitute your own values for the start and end times and the name of the cluster.

```
aws cloudwatch get-metric-statistics --metric-name "VolumeBytesUsed" \
  --start-time "$(date -d '6 hours ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" \
  --statistics Average Maximum Minimum \
  --dimensions Name=DBClusterIdentifier,Value=my_cluster_identifier
```

That command produces output similar to the following.

```
{
    "Label": "VolumeBytesUsed",
    "Datapoints": [
        {
            "Timestamp": "2020-08-04T21:25:00+00:00",
            "Average": 182871982080.0,
            "Minimum": 182871982080.0,
            "Maximum": 182871982080.0,
            "Unit": "Bytes"
        }
    ]
}
```

The following examples show how you might track storage usage for an Aurora cluster over time using AWS CLI commands on a Linux system. The `--start-time` and `--end-time` parameters define the overall time interval as one day. The `--period` parameter requests the measurements at one hour intervals. It doesn't make sense to choose a `--period` value that's small, because the metrics are collected at intervals, not continuously. Also, Aurora storage operations sometimes continue for some time in the background after the relevant SQL statement finishes.

The first example returns output in the default JSON format. The data points are returned in arbitrary order, not sorted by timestamp. You might import this JSON data into a charting tool to do sorting and visualization.

```
$ aws cloudwatch get-metric-statistics --metric-name "VolumeBytesUsed" \
  --start-time "$(date -d '1 day ago')" --end-time "$(date -d 'now')" --period 3600
  --namespace "AWS/RDS" --statistics Maximum --dimensions
 Name=DBClusterIdentifier,Value=my_cluster_id
{
    "Label": "VolumeBytesUsed",
    "Datapoints": [
        {
            "Timestamp": "2020-08-04T19:40:00+00:00",
            "Maximum": 182872522752.0,
            "Unit": "Bytes"
        },
        {
            "Timestamp": "2020-08-05T00:40:00+00:00",
            "Maximum": 198573719552.0,
            "Unit": "Bytes"
        },
        {
            "Timestamp": "2020-08-05T05:40:00+00:00",
            "Maximum": 206827454464.0,
            "Unit": "Bytes"
        },
        {
            "Timestamp": "2020-08-04T17:40:00+00:00",
            "Maximum": 182872522752.0,
            "Unit": "Bytes"
        },
... output omitted ...
```

This example returns the same data as the previous one. The `--output` parameter represents the data in compact plain text format. The `aws cloudwatch` command pipes its output to the `sort` command. The `-k` parameter of the `sort` command sorts the output by the third field, which is the timestamp in Universal Coordinated Time (UTC) format.

```
$ aws cloudwatch get-metric-statistics --metric-name "VolumeBytesUsed" \
  --start-time "$(date -d '1 day ago')" --end-time "$(date -d 'now')" --period 3600 \
  --namespace "AWS/RDS" --statistics Maximum --dimensions
 Name=DBClusterIdentifier,Value=my_cluster_id \
  --output text | sort -k 3
VolumeBytesUsed
DATAPOINTS  182872522752.0  2020-08-04T17:41:00+00:00 Bytes
DATAPOINTS  182872522752.0  2020-08-04T18:41:00+00:00 Bytes
DATAPOINTS  182872522752.0  2020-08-04T19:41:00+00:00 Bytes
DATAPOINTS  182872522752.0  2020-08-04T20:41:00+00:00 Bytes
DATAPOINTS  187667791872.0  2020-08-04T21:41:00+00:00 Bytes
DATAPOINTS  190981029888.0  2020-08-04T22:41:00+00:00 Bytes
DATAPOINTS  195587244032.0  2020-08-04T23:41:00+00:00 Bytes
DATAPOINTS  201048915968.0  2020-08-05T00:41:00+00:00 Bytes
DATAPOINTS  205368492032.0  2020-08-05T01:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T02:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T03:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T04:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T05:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T06:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T07:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T08:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T09:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T10:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T11:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T12:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T13:41:00+00:00 Bytes
DATAPOINTS  206827454464.0  2020-08-05T14:41:00+00:00 Bytes
DATAPOINTS  206833664000.0  2020-08-05T15:41:00+00:00 Bytes
DATAPOINTS  206833664000.0  2020-08-05T16:41:00+00:00 Bytes
```

The sorted output shows how much storage was used at the start and end of the monitoring period. You can also find the points during that period when Aurora allocated more storage for the cluster. The following example uses Linux commands to reformat the starting and ending `VolumeBytesUsed` values as gigabytes (GB) and as gibibytes (GiB). Gigabytes represent units measured in powers of 10 and are commonly used in discussions of storage for rotational hard drives. Gibibytes represent units measured in powers of 2. Aurora storage measurements and limits are typically stated in the power-of-2 units, such as gibibytes and tebibytes.

```
$ GiB=$((1024*1024*1024))
$ GB=$((1000*1000*1000))
$ echo "Start: $((182872522752/$GiB)) GiB, End: $((206833664000/$GiB)) GiB"
Start: 170 GiB, End: 192 GiB
$ echo "Start: $((182872522752/$GB)) GB, End: $((206833664000/$GB)) GB"
Start: 182 GB, End: 206 GB
```

The `VolumeBytesUsed` metric tells you how much storage in the cluster is incurring charges. Thus, it's best to minimize this number when practical. However, this metric doesn't include some storage that Aurora uses internally in the cluster and doesn't charge for. If your cluster is approaching the storage limit and might run out of space, it's more helpful to monitor the `AuroraVolumeBytesLeftTotal` metric and try to maximize that number. The following example runs a similar calculation as the previous one, but for `AuroraVolumeBytesLeftTotal` instead of `VolumeBytesUsed`.

```
$ aws cloudwatch get-metric-statistics --metric-name "AuroraVolumeBytesLeftTotal" \
  --start-time "$(date -d '1 hour ago')" --end-time "$(date -d 'now')" --period 3600 \
  --namespace "AWS/RDS" --statistics Maximum --dimensions
 Name=DBClusterIdentifier,Value=my_old_cluster_id \
  --output text | sort -k 3
AuroraVolumeBytesLeftTotal
DATAPOINTS      140530528288768.0       2023-02-23T19:25:00+00:00       Count
$ TiB=$((1024*1024*1024*1024))
$ TB=$((1000*1000*1000*1000))
$ echo "$((69797067915264 / $TB)) TB remaining for this cluster"
69 TB remaining for this cluster
$ echo "$((69797067915264 / $TiB)) TiB remaining for this cluster"
63 TiB remaining for this cluster
```

For a cluster running Aurora MySQL version 2.09 or higher, or Aurora PostgreSQL, the free size reported by `VolumeBytesUsed` increases when data is added and decreases when data is removed. The following example shows how. This report shows the maximum and minimum storage size for a cluster at 15-minute intervals as tables with temporary data are created and dropped. The report lists the maximum value before the minimum value. Thus, to understand how storage usage changed within the 15-minute interval, interpret the numbers from right to left.

```
$ aws cloudwatch get-metric-statistics --metric-name "VolumeBytesUsed" \
  --start-time "$(date -d '4 hours ago')" --end-time "$(date -d 'now')" --period 1800 \
  --namespace "AWS/RDS" --statistics Maximum Minimum --dimensions
 Name=DBClusterIdentifier,Value=my_new_cluster_id
  --output text | sort -k 4
VolumeBytesUsed
DATAPOINTS 14545305600.0 14545305600.0 2020-08-05T20:49:00+00:00 Bytes
DATAPOINTS 14545305600.0 14545305600.0 2020-08-05T21:19:00+00:00 Bytes
DATAPOINTS 22022176768.0 14545305600.0 2020-08-05T21:49:00+00:00 Bytes
DATAPOINTS 22022176768.0 22022176768.0 2020-08-05T22:19:00+00:00 Bytes
DATAPOINTS 22022176768.0 22022176768.0 2020-08-05T22:49:00+00:00 Bytes
DATAPOINTS 22022176768.0 15614263296.0 2020-08-05T23:19:00+00:00 Bytes
DATAPOINTS 15614263296.0 15614263296.0 2020-08-05T23:49:00+00:00 Bytes
DATAPOINTS 15614263296.0 15614263296.0 2020-08-06T00:19:00+00:00 Bytes
```

The following example shows how with a cluster running Aurora MySQL version 2.09 or higher, or Aurora PostgreSQL, the free size reported by `AuroraVolumeBytesLeftTotal` reflects the 128-TiB size limit.

```
$ aws cloudwatch get-metric-statistics --region us-east-1 --metric-name
 "AuroraVolumeBytesLeftTotal" \
  --start-time "$(date -d '4 hours ago')" --end-time "$(date -d 'now')" --period 1800 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBClusterIdentifier,Value=pq-57 \
  --output text | sort -k 3
AuroraVolumeBytesLeftTotal
DATAPOINTS 140515818864640.0 2020-08-05T20:56:00+00:00 Count
DATAPOINTS 140515818864640.0 2020-08-05T21:26:00+00:00 Count
DATAPOINTS 140515818864640.0 2020-08-05T21:56:00+00:00 Count
DATAPOINTS 140514866757632.0 2020-08-05T22:26:00+00:00 Count
DATAPOINTS 140511020580864.0 2020-08-05T22:56:00+00:00 Count
DATAPOINTS 140503168843776.0 2020-08-05T23:26:00+00:00 Count
DATAPOINTS 140503168843776.0 2020-08-05T23:56:00+00:00 Count
DATAPOINTS 140515818864640.0 2020-08-06T00:26:00+00:00 Count
$ TiB=$((1024*1024*1024*1024))
$ TB=$((1000*1000*1000*1000))
$ echo "$((140515818864640 / $TB)) TB remaining for this cluster"
140 TB remaining for this cluster
$ echo "$((140515818864640 / $TiB)) TiB remaining for this cluster"
127 TiB remaining for this cluster
```

Instance scaling

You can scale your Aurora DB cluster as needed by modifying the DB instance class for each DB instance in the DB cluster. Aurora supports several DB instance classes optimized for Aurora, depending on database engine compatibility.

Database engine	Instance scaling
Amazon Aurora MySQL	See Scaling Aurora MySQL DB instances
Amazon Aurora PostgreSQL	See Scaling Aurora PostgreSQL DB instances

Read scaling

You can achieve read scaling for your Aurora DB cluster by creating up to 15 Aurora Replicas in a DB cluster. Each Aurora Replica returns the same data from the cluster volume with minimal replica lag—usually considerably less than 100 milliseconds after the primary instance has written an update. As your read traffic increases, you can create additional Aurora Replicas and connect to them directly to distribute the read load for your DB cluster. Aurora Replicas don't have to be of the same DB instance class as the primary instance.

For information about adding Aurora Replicas to a DB cluster, see Adding Aurora Replicas to a DB cluster.

Managing connections

The maximum number of connections allowed to an Aurora DB instance is determined by the `max_connections` parameter in the instance-level parameter group for the DB instance. The default value of that parameter varies depends on the DB instance class used for the DB instance and database engine compatibility.

Database engine	max_connections default value
Amazon Aurora MySQL	See Maximum connections to an Aurora MySQL DB instance
Amazon Aurora PostgreSQL	See Maximum connections to an Aurora PostgreSQL DB instance

Tip

If your applications frequently open and close connections, or keep a large number of long-lived connections open, we recommend that you use Amazon RDS Proxy. RDS Proxy is a fully managed, highly available database proxy that uses connection pooling to share database connections securely and efficiently. To learn more about RDS Proxy, see Using Amazon RDS Proxy for Aurora.

Managing query execution plans

If you use query plan management for Aurora PostgreSQL, you gain control over which plans the optimizer runs. For more information, see Managing query execution plans for Aurora PostgreSQL.

Cloning a volume for an Amazon Aurora DB cluster

By using Aurora cloning, you can create a new cluster that initially shares the same data pages as the original, but is a separate and independent volume. The process is designed to be fast and cost-effective. The new cluster with its associated data volume is known as a *clone*. Creating a clone is faster and more space-efficient than physically copying the data using other techniques, such as restoring a snapshot.

Topics

Overview of Aurora cloning

Aurora uses a *copy-on-write protocol* to create a clone. This mechanism uses minimal additional space to create an initial clone. When the clone is first created, Aurora keeps a single copy of the data that is used by the source Aurora DB cluster and the new (cloned) Aurora DB cluster. Additional storage is allocated only when changes are made to data (on the Aurora storage volume) by the source Aurora DB cluster or the Aurora DB cluster clone. To learn more about the copy-on-write protocol, see How Aurora cloning works.

Aurora cloning is especially useful for quickly setting up test environments using your production data, without risking data corruption. You can use clones for many types of applications, such as the following:

- Experiment with potential changes (schema changes and parameter group changes, for example) to assess all impacts.
- Run workload-intensive operations, such as exporting data or running analytical queries on the clone.
- Create a copy of your production DB cluster for development, testing, or other purposes.

You can create more than one clone from the same Aurora DB cluster. You can also create multiple clones from another clone.

After creating an Aurora clone, you can configure the Aurora DB instances differently from the source Aurora DB cluster. For example, you might not need a clone for development purposes to meet the same high availability requirements as the source production Aurora DB cluster. In this case, you can configure the clone with a single Aurora DB instance rather than the multiple DB instances used by the Aurora DB cluster.

When you create a clone using a different deployment configuration from the source, the clone is created using the latest minor version of the source's Aurora DB engine.

When you create clones from your Aurora DB clusters, the clones are created in your AWS account —the same account that owns the source Aurora DB cluster. However, you can also share Aurora Serverless v2 and provisioned Aurora DB clusters and clones with other AWS accounts. For more information, see Cross-account cloning with AWS RAM and Amazon Aurora.

When you finish using the clone for your testing, development, or other purposes, you can delete it.

Limitations of Aurora cloning

Aurora cloning currently has the following limitations:

- You can create as many clones as you want, up to the maximum number of DB clusters allowed in the AWS Region.

  You can create the clones using the copy-on-write protocol or the full-copy protocol. The full-copy protocol acts like a point-in-time recovery.
- You can't create a clone in a different AWS Region from the source Aurora DB cluster.
- You can't create a clone from an Aurora DB cluster without the parallel query feature to a cluster that uses parallel query. To bring data into a cluster that uses parallel query, create a snapshot of the original cluster and restore it to the cluster that's using the parallel query feature.
- You can't create a clone from an Aurora DB cluster that has no DB instances. You can only clone Aurora DB clusters that have at least one DB instance.
- You can create a clone in a different virtual private cloud (VPC) than that of the Aurora DB cluster. If you do, the subnets of the VPCs must map to the same Availability Zones.
- You can create an Aurora provisioned clone from a provisioned Aurora DB cluster.

- Clusters with Aurora Serverless v2 instances follow the same rules as provisioned clusters.
- For Aurora Serverless v1:
  - You can create a provisioned clone from an Aurora Serverless v1 DB cluster.
  - You can create an Aurora Serverless v1 clone from an Aurora Serverless v1 or provisioned DB cluster.
  - You can't create an Aurora Serverless v1 clone from an unencrypted, provisioned Aurora DB cluster.
  - Cross-account cloning currently doesn't support cloning Aurora Serverless v1 DB clusters. For more information, see Limitations of cross-account cloning.
  - A cloned Aurora Serverless v1 DB cluster has the same behavior and limitations as any Aurora Serverless v1 DB cluster. For more information, see Using Amazon Aurora Serverless v1.
  - Aurora Serverless v1 DB clusters are always encrypted. When you clone an Aurora Serverless v1 DB cluster into a provisioned Aurora DB cluster, the provisioned Aurora DB cluster is encrypted. You can choose the encryption key, but you can't disable the encryption. To clone from a provisioned Aurora DB cluster to an Aurora Serverless v1, you must start with an encrypted provisioned Aurora DB cluster.

How Aurora cloning works

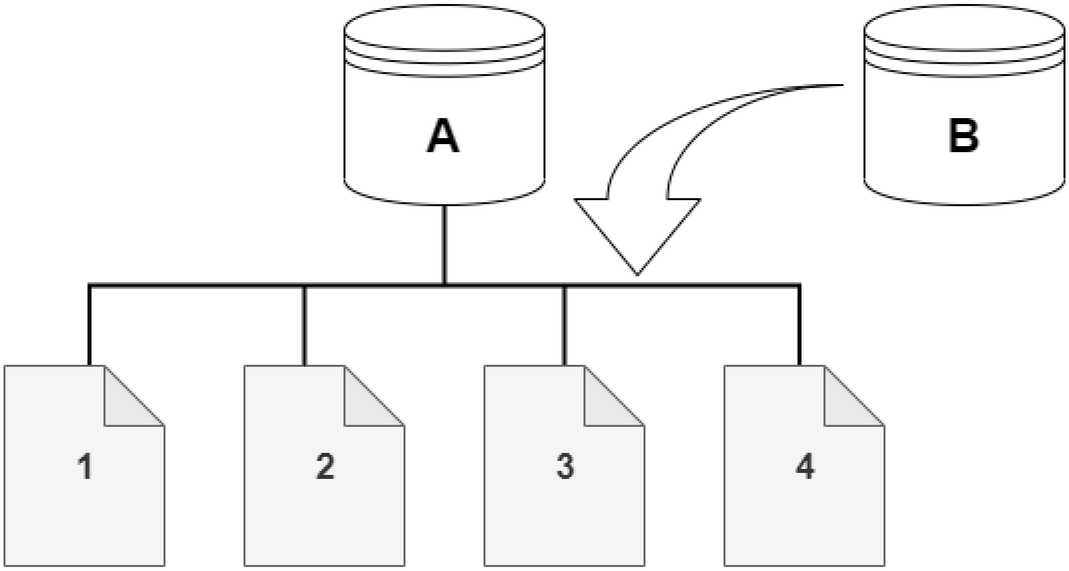
Aurora cloning works at the storage layer of an Aurora DB cluster. It uses a *copy-on-write* protocol that's both fast and space-efficient in terms of the underlying durable media supporting the Aurora storage volume. You can learn more about Aurora cluster volumes in the Overview of Amazon Aurora storage.

Topics

Understanding the copy-on-write protocol

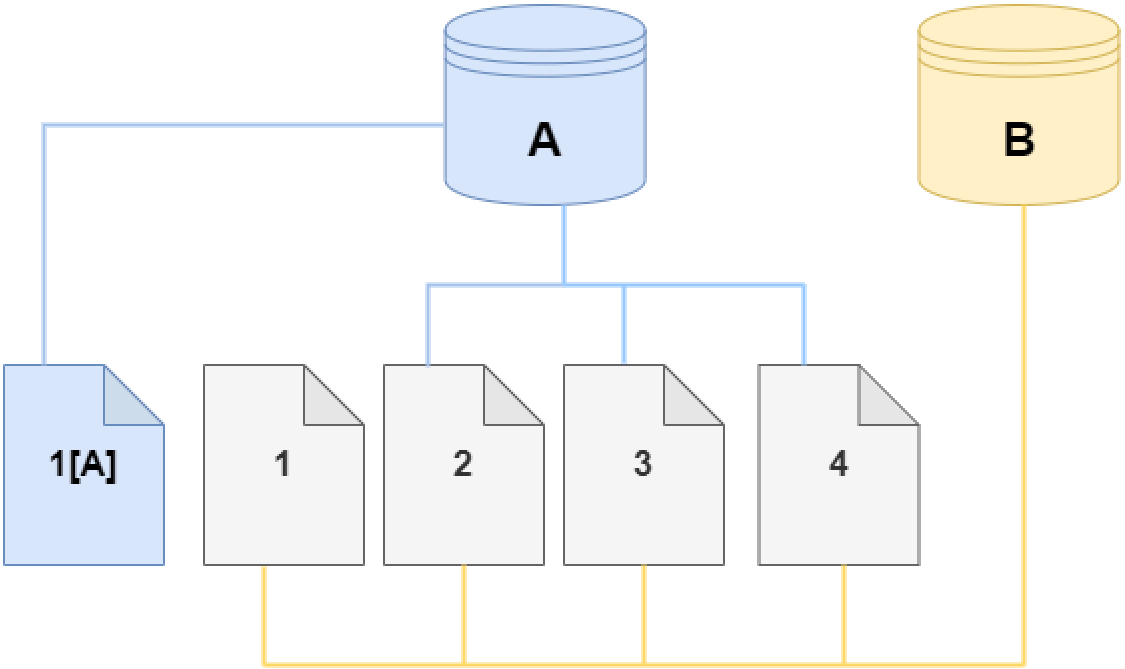
An Aurora DB cluster stores data in pages in the underlying Aurora storage volume.

For example, in the following diagram you can find an Aurora DB cluster (A) that has four data pages, 1, 2, 3, and 4. Imagine that a clone, B, is created from the Aurora DB cluster. When the clone is created, no data is copied. Rather, the clone points to the same set of pages as the source Aurora DB cluster.

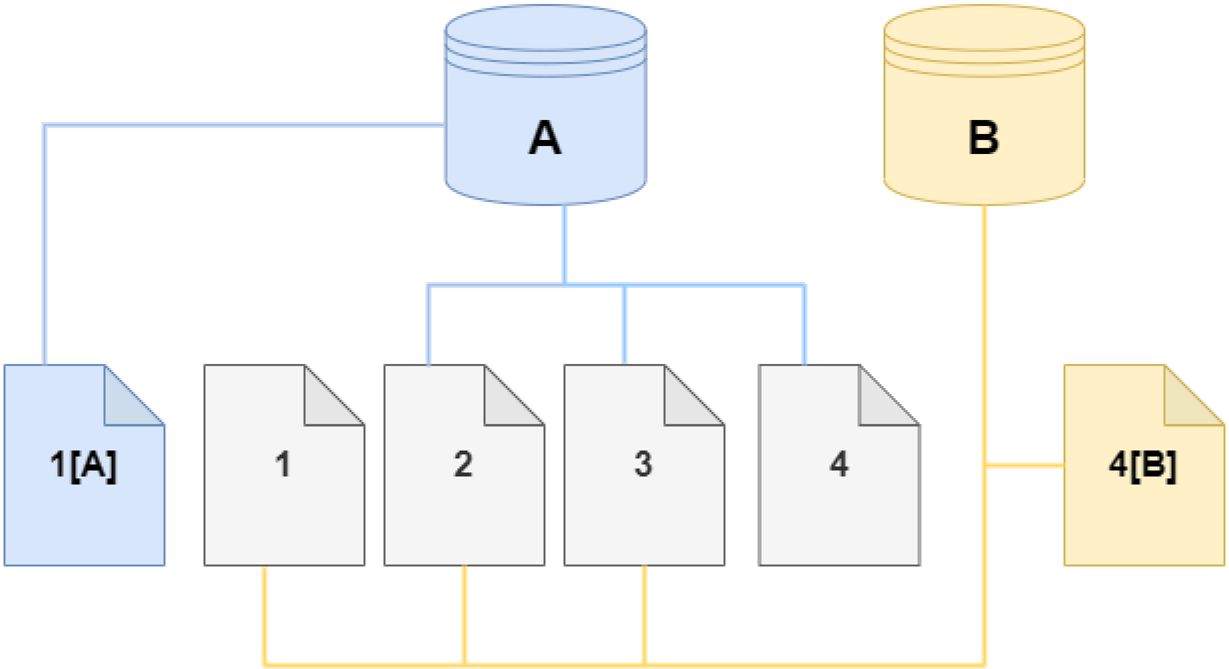


When the clone is created, no additional storage is usually needed. The copy-on-write protocol uses the same segment on the physical storage media as the source segment. Additional storage is required only if the capacity of the source segment isn't sufficient for the entire clone segment. If that's the case, the source segment is copied to another physical device.

In the following diagrams, you can find an example of the copy-on-write protocol in action using the same cluster A and its clone, B, as shown preceding. Let's say that you make a change to your Aurora DB cluster (A) that results in a change to data held on page 1. Instead of writing to the original page 1, Aurora creates a new page 1[A]. The Aurora DB cluster volume for cluster (A) now points to page 1[A], 2, 3, and 4, while the clone (B) still references the original pages.



On the clone, a change is made to page 4 on the storage volume. Instead of writing to the original page 4, Aurora creates a new page, 4[B]. The clone now points to pages 1, 2, 3, and to page 4[B], while the cluster (A) continues pointing to 1[A], 2, 3, and 4.



As more changes occur over time in both the source Aurora DB cluster volume and the clone, more storage is needed to capture and store the changes.

Deleting a source cluster volume

Initially, the clone volume shares the same data pages as the original volume from which the clone is created. As long as the original volume exists, the clone volume is only considered the owner of the pages that the clone created or modified. Thus, the `VolumeBytesUsed` metric for the clone volume starts out small and only grows as the data diverges between the original cluster and the clone. For pages that are identical between the source volume and the clone, the storage charges apply only to the original cluster. For more information about the `VolumeBytesUsed` metric, see Cluster-level metrics for Amazon Aurora.

When you delete a source cluster volume that has one or more clones associated with it, the data in the cluster volumes of the clones isn't changed. Aurora preserves the pages that were previously owned by the source cluster volume. Aurora redistributes the storage billing for the pages that were owned by the deleted cluster. For example, suppose that an original cluster had two clones and then the original cluster was deleted. Half of the data pages owned by the original cluster would now be owned by one clone. The other half of the pages would be owned by the other clone.

If you delete the original cluster, then as you create or delete more clones, Aurora continues to redistribute ownership of the data pages among all the clones that share the same pages. Thus, you might find that the value of the `VolumeBytesUsed` metric changes for the cluster volume of a clone. The metric value can decrease as more clones are created and page ownership is spread across more clusters. The metric value can also increase as clones are deleted and page ownership is assigned to a smaller number of clusters. For information about how write operations affect data pages on clone volumes, see Understanding the copy-on-write protocol.

When the original cluster and the clones are owned by the same AWS account, all the storage charges for those clusters apply to that same AWS account. If some of the clusters are cross-account clones, deleting the original cluster can result in additional storage charges to the AWS accounts that own the cross-account clones.

For example, suppose that a cluster volume has 1000 used data pages before you create any clones. When you clone that cluster, initially the clone volume has zero used pages. If the clone makes modifications to 100 data pages, only those 100 pages are stored on the clone volume and marked as used. The other 900 unchanged pages from the parent volume are shared by both clusters. In this case, the parent cluster has storage charges for 1000 pages and the clone volume for 100 pages.

If you delete the source volume, the storage charges for the clone include the 100 pages that it changed, plus the 900 shared pages from the original volume, for a total of 1000 pages.

Creating an Amazon Aurora clone

You can create a clone in the same AWS account as the source Aurora DB cluster. To do so, you can use the AWS Management Console or the AWS CLI and the procedures following.

To allow another AWS account to create a clone or to share a clone with another AWS account, use the procedures in Cross-account cloning with AWS RAM and Amazon Aurora.

Console

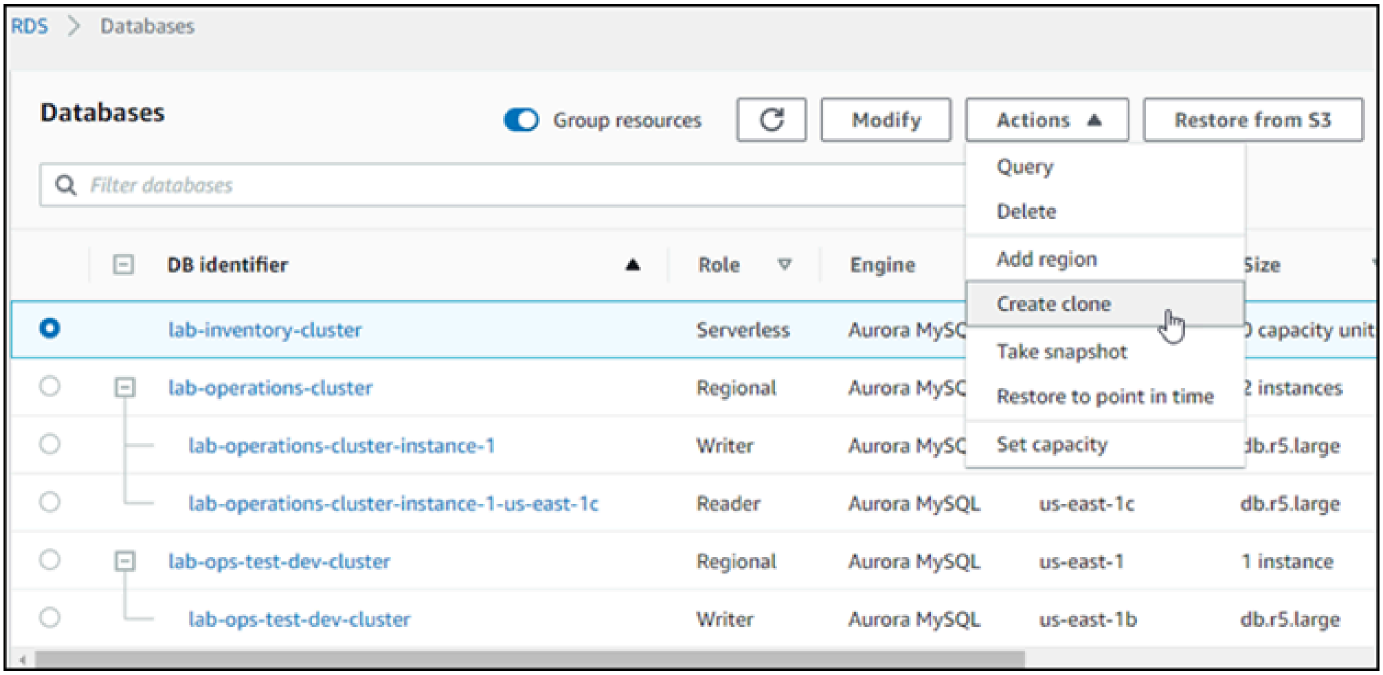
The following procedure describes how to clone an Aurora DB cluster using the AWS Management Console.

Creating a clone using the AWS Management Console results in an Aurora DB cluster with one Aurora DB instance.

These instructions apply for DB clusters owned by the same AWS account that is creating the clone. If the DB cluster is owned by a different AWS account, see [Cross-account cloning with AWS RAM and Amazon Aurora]() instead.

To create a clone of a DB cluster owned by your AWS account using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose your Aurora DB cluster from the list, and for **Actions**, choose **Create clone**.



DB identifier ▲	Role ▽	Engine		Size
● lab-inventory-cluster	Serverless	Aurora MySC		capacity unit
○ lab-operations-cluster	Regional	Aurora MySC		2 instances
○ lab-operations-cluster-instance-1	Writer	Aurora MySC		db.r5.large
○ lab-operations-cluster-instance-1-us-east-1c	Reader	Aurora MySQL	us-east-1c	db.r5.large
○ lab-ops-test-dev-cluster	Regional	Aurora MySQL	us-east-1	1 instance
○ lab-ops-test-dev-cluster	Writer	Aurora MySQL	us-east-1b	db.r5.large

The Create clone page opens, where you can configure **Settings**, **Connectivity**, and other options for the Aurora DB cluster clone.

4. For **DB instance identifier**, enter the name that you want to give to your cloned Aurora DB cluster.
5. For Aurora Serverless v1 DB clusters, choose **Provisioned** or **Serverless** for **Capacity type**.

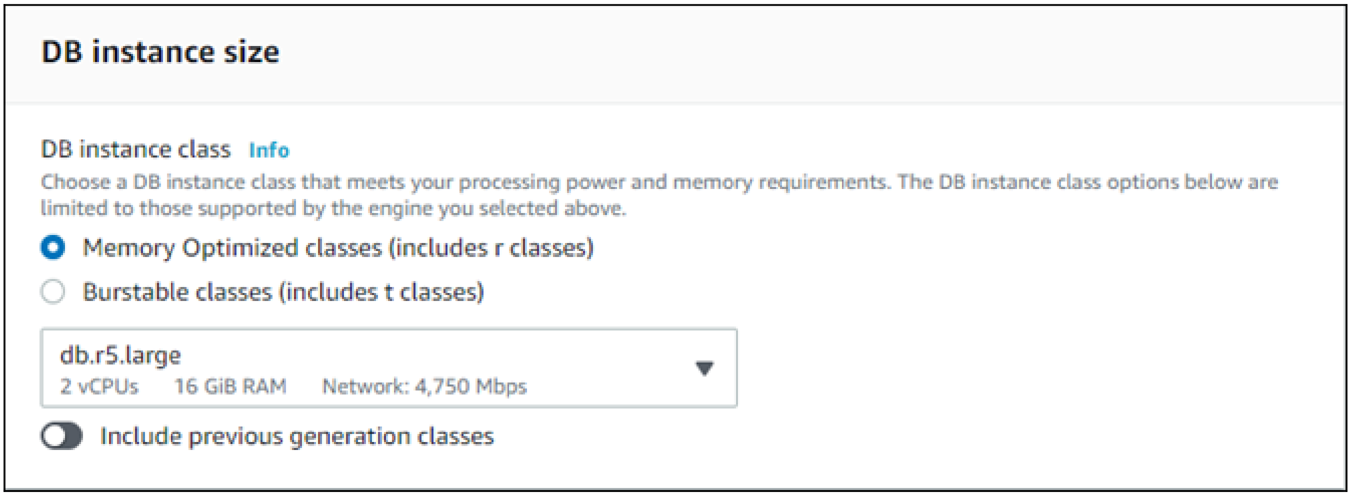
   You can choose **Serverless** only if the source Aurora DB cluster is an Aurora Serverless v1 DB cluster or is a provisioned Aurora DB cluster that is encrypted.

6. For Aurora Serverless v2 or provisioned DB clusters, choose either **Aurora I/O-Optimized** or **Aurora Standard** for **Cluster storage configuration**.

   For more information, see Storage configurations for Amazon Aurora DB clusters.

7. Choose the DB instance size or DB cluster capacity:

   - For a provisioned clone, choose a **DB instance class**.



DB instance size

DB instance class Info

Choose a DB instance class that meets your processing power and memory requirements. The DB instance class options below are limited to those supported by the engine you selected above.

● Memory Optimized classes (includes r classes)

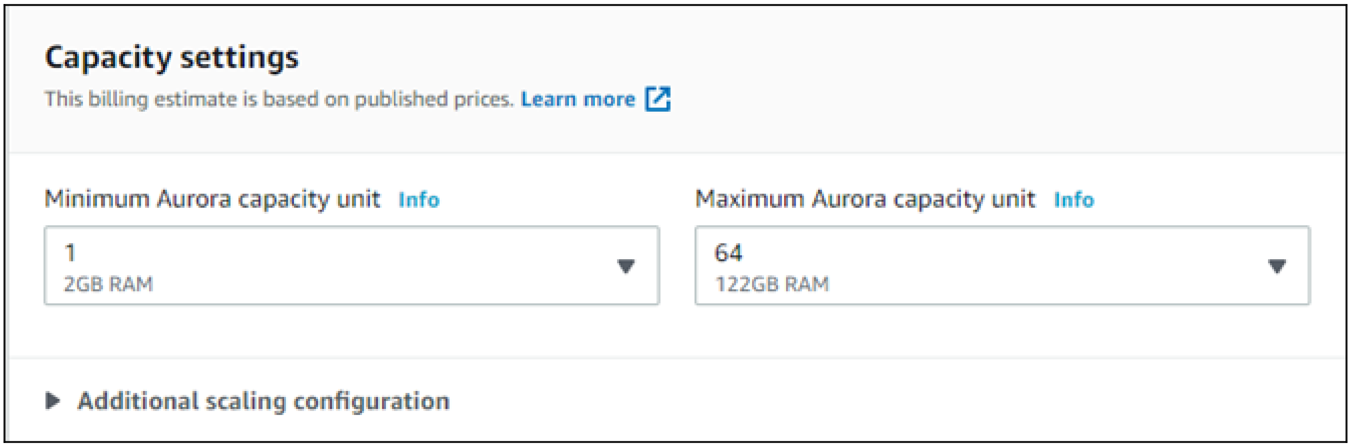
○ Burstable classes (includes t classes)

db.r5.large
2 vCPUs 16 GiB RAM Network: 4,750 Mbps

Include previous generation classes

   You can accept the provided setting, or you can use a different DB instance class for your clone.

   - For an Aurora Serverless v1 or Aurora Serverless v2 clone, choose the **Capacity settings**.



Capacity settings

This billing estimate is based on published prices. Learn more

Minimum Aurora capacity unit Info

1
2GB RAM

Maximum Aurora capacity unit Info

64
122GB RAM

▶ Additional scaling configuration

   You can accept the provided settings, or you can change them for your clone.

8. Choose other settings as needed for your clone. To learn more about Aurora DB cluster and instance settings, see Creating an Amazon Aurora DB cluster.

9. Choose **Create clone**.

When the clone is created, it's listed with your other Aurora DB clusters in the console **Databases** section and displays its current state. Your clone is ready to use when its state is **Available**.

AWS CLI

Using the AWS CLI for cloning your Aurora DB cluster involves a couple of steps.

The `restore-db-cluster-to-point-in-time` AWS CLI command that you use results in an empty Aurora DB cluster with 0 Aurora DB instances. That is, the command restores only the Aurora DB cluster, not the DB instances for that cluster. You do that separately after the clone is available. The two steps in the process are as follows:

1. Create the clone by using the restore-db-cluster-to-point-in-time CLI command. The parameters that you use with this command control the capacity type and other details of the empty Aurora DB cluster (clone) being created.
2. Create the Aurora DB instance for the clone by using the create-db-instance CLI command to recreate the Aurora DB instance in the restored Aurora DB cluster.

Topics

Creating the clone

The specific parameters that you pass to the `restore-db-cluster-to-point-in-time` CLI command vary. What you pass depends on the engine-mode type of the source DB cluster—Serverless or Provisioned—and the type of clone that you want to create.

To create a clone of the same engine mode as the source Aurora DB cluster

- Use the `restore-db-cluster-to-point-in-time` CLI command and specify values for the following parameters:

  - `--db-cluster-identifier` – Choose a meaningful name for your clone. You name the clone when you use the restore-db-cluster-to-point-in-time CLI command. You then pass the name of the clone in the create-db-instance CLI command.

  - `--restore-type` – Use `copy-on-write` to create a clone of the source DB cluster. Without this parameter, the `restore-db-cluster-to-point-in-time` restores the Aurora DB cluster rather than creating a clone.
  - `--source-db-cluster-identifier` – Use the name of the source Aurora DB cluster that you want to clone.
  - `--use-latest-restorable-time` – This value points to the latest restorable volume data for the source DB cluster. Use it to create clones.

The following example creates a clone named `my-clone` from a cluster named `my-source-cluster`.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-to-point-in-time \
    --source-db-cluster-identifier my-source-cluster \
    --db-cluster-identifier my-clone \
    --restore-type copy-on-write \
    --use-latest-restorable-time
```

For Windows:

```
aws rds restore-db-cluster-to-point-in-time ^
    --source-db-cluster-identifier my-source-cluster ^
    --db-cluster-identifier my-clone ^
    --restore-type copy-on-write ^
    --use-latest-restorable-time
```

The command returns the JSON object containing details of the clone. Check to make sure that your cloned DB cluster is available before trying to create the DB instance for your clone. For more information, see Checking the status and getting clone details.

To create a clone with a different engine mode from the source Aurora DB cluster

- Use the `restore-db-cluster-to-point-in-time` CLI command and specify values for the following parameters:

  - `--db-cluster-identifier` – Choose a meaningful name for your clone. You name the clone when you use the restore-db-cluster-to-point-in-time CLI command. You then pass the name of the clone in the create-db-instance CLI command.

- `--source-db-cluster-identifier` – Use the name of the source Aurora DB cluster that you want to clone.
- `--restore-type` – Use `copy-on-write` to create a clone of the source DB cluster. Without this parameter, the `restore-db-cluster-to-point-in-time` restores the Aurora DB cluster rather than creating a clone.
- `--use-latest-restorable-time` – This value points to the latest restorable volume data for the source DB cluster. Use it to create clones.
- `--engine-mode` – (Optional) Use this parameter only to create clones that are of a different type from the source Aurora DB cluster. Choose the value to pass with `--engine-mode` as follows:
  - Use `provisioned` to create a provisioned Aurora DB cluster clone from an Aurora Serverless DB cluster.
  - Use `serverless` to create an Aurora Serverless v1 DB cluster clone from a provisioned Aurora DB cluster. When you specify the `serverless` engine mode, you can also choose the `--scaling-configuration`.
- `--scaling-configuration` – (Optional) Use with `--engine-mode serverless` to configure the minimum and maximum capacity for an Aurora Serverless v1 clone. If you don't use this parameter, Aurora creates the clone using the default capacity values for the DB engine.
- `--serverless-v2-scaling-configuration` – (Optional) Use this parameter to configure the minimum and maximum capacity for an Aurora Serverless v2 clone. If you don't use this parameter, Aurora creates the clone using the default capacity values for the DB engine.

The following example creates an Aurora Serverless v1 clone named `my-clone`, from a provisioned Aurora DB cluster named `my-source-cluster`. The provisioned Aurora DB cluster is encrypted.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-to-point-in-time \
    --source-db-cluster-identifier my-source-cluster \
    --db-cluster-identifier my-clone \
    --engine-mode serverless \
    --scaling-configuration MinCapacity=8,MaxCapacity=64 \
    --restore-type copy-on-write \
    --use-latest-restorable-time
```

For Windows:

```
aws rds restore-db-cluster-to-point-in-time ^
    --source-db-cluster-identifier my-source-cluster ^
    --db-cluster-identifier my-clone ^
    --engine-mode serverless ^
    --scaling-configuration MinCapacity=8,MaxCapacity=64 ^
    --restore-type copy-on-write ^
    --use-latest-restorable-time
```

These commands return the JSON object containing details of the clone that you need to create the DB instance. You can't do that until the status of the clone (the empty Aurora DB cluster) has the status **Available**.

Note

The restore-db-cluster-to-point-in-time AWS CLI command only restores the DB cluster, not the DB instances for that DB cluster. You must invoke the create-db-instance command to create DB instances for the restored DB cluster, specifying the identifier of the restored DB cluster in `--db-cluster-identifier`. You can create DB instances only after the `restore-db-cluster-to-point-in-time` command has completed and the DB cluster is available.

For example, suppose you have a cluster named `tpch100g` that you want to clone. The following Linux example creates a cloned cluster named `tpch100g-clone` and a primary instance named `tpch100g-clone-instance` for the new cluster. You don't need to supply some parameters, such as `--master-username` and `--master-user-password`. Aurora automatically determines those from the original cluster. You do need to specify the DB engine to use. Thus, the example tests the new cluster to determine the right value to use for the `--engine` parameter.

```
$ aws rds restore-db-cluster-to-point-in-time \
  --source-db-cluster-identifier tpch100g \
  --db-cluster-identifier tpch100g-clone \
  --restore-type copy-on-write \
  --use-latest-restorable-time

$ aws rds describe-db-clusters \
  --db-cluster-identifier tpch100g-clone \
    --query '*[].[Engine]' \
    --output text
aurora-mysql

$ aws rds create-db-instance \
  --db-instance-identifier tpch100g-clone-instance \
  --db-cluster-identifier tpch100g-clone \
  --db-instance-class db.r5.4xlarge \
  --engine aurora-mysql
```

Checking the status and getting clone details

You can use the following command to check the status of your newly created empty DB cluster.

```
$ aws rds describe-db-clusters --db-cluster-identifier my-clone --query '*[].[Status]'
 --output text
```

Or you can obtain the status and the other values that you need to create the DB instance for your clone by using the following AWS CLI query.

For Linux, macOS, or Unix:

```
aws rds describe-db-clusters --db-cluster-identifier my-clone \
  --query '*[].
{Status:Status,Engine:Engine,EngineVersion:EngineVersion,EngineMode:EngineMode}'
```

For Windows:

```
aws rds describe-db-clusters --db-cluster-identifier my-clone ^
  --query "*[].
{Status:Status,Engine:Engine,EngineVersion:EngineVersion,EngineMode:EngineMode}"
```

This query returns output similar to the following.

```
[
  {
        "Status": "available",
        "Engine": "aurora-mysql",
        "EngineVersion": "8.0.mysql_aurora.3.04.1",
        "EngineMode": "provisioned"
    }
]
```

Creating the Aurora DB instance for your clone

Use the create-db-instance CLI command to create the DB instance for your Aurora Serverless v2 or provisioned clone. You don't create a DB instance for an Aurora Serverless v1 clone.

The DB instance inherits the `--master-username` and `--master-user-password` properties from the source DB cluster.

The following example creates a DB instance for a provisioned clone.

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
    --db-instance-identifier my-new-db \
    --db-cluster-identifier my-clone \
    --db-instance-class db.r5.4xlarge \
    --engine aurora-mysql
```

For Windows:

```
aws rds create-db-instance ^
    --db-instance-identifier my-new-db ^
    --db-cluster-identifier my-clone ^
    --db-instance-class db.r5.4xlarge ^
    --engine aurora-mysql
```

Parameters to use for cloning

The following table summarizes the various parameters used with `restore-db-cluster-to-point-in-time` to clone Aurora DB clusters.

Parameter	Description
`--source-db-cluster-identifier`	Use the name of the source Aurora DB cluster that you want to clone.
`--db-cluster-identifier`	Choose a meaningful name for your clone when you create it with the `restore-db-cluster-to-point-in-time` command. Then you pass this name to the `create-db-instance` command.

Parameter	Description
`--restore-type`	Specify `copy-on-write` as the `--restore-type` to create a clone of the source DB cluster rather than restoring the source Aurora DB cluster.
`--use-latest-restorable-time`	This value points to the latest restorable volume data for the source DB cluster. Use it to create clones.
`--engine-mode`	Use this parameter to create clones that are of a different type from the source Aurora DB cluster, with one of the following values: • Use `provisioned` to create a provisioned or Aurora Serverless v2 clone from an Aurora Serverless v1 DB cluster. • Use `serverless` to create an Aurora Serverless v1 clone from a provisioned or Aurora Serverless v2 DB cluster. When you specify the `serverless` engine mode, you can also choose the `--scaling-configuration` .
`--scaling-configuration`	Use this parameter to configure the minimum and maximum capacity for an Aurora Serverless v1 clone. If you don't specify this parameter , Aurora creates the clone using the default capacity values for the DB engine.
`--serverless-v2-scaling-configuration`	Use this parameter to configure the minimum and maximum capacity for an Aurora Serverless v2 clone. If you don't specify this parameter , Aurora creates the clone using the default capacity values for the DB engine.

Cross-VPC cloning with Amazon Aurora

Suppose that you want to impose different network access controls on the original cluster and the clone. For example, you might use cloning to make a copy of a production Aurora cluster in a different VPC for development and testing. Or you might create a clone as part of a migration from public subnets to private subnets, to enhance your database security.

The following sections demonstrate how to set up the network configuration for the clone so that the original cluster and the clone can both access the same Aurora storage nodes, even from different subnets or different VPCs. Verifying the network resources in advance can avoid errors during cloning that might be difficult to diagnose.

If you aren't familiar with how Aurora interacts with VPCs, subnets, and DB subnet groups, see Amazon VPC VPCs and Amazon Aurora first. You can work through the tutorials in that section to create these kinds of resources in the AWS console, and understand how they fit together.

Because the steps involve switching between the RDS and EC2 services, the examples use AWS CLI commands to help you understand how to automate such operations and save the output.

- Before you begin
- Gathering information about the network environment
- Creating network resources for the clone
- Creating an Aurora clone with new network settings

Before you begin

Before you start setting up a cross-VPC clone, make sure to have the following resources:

- An Aurora DB cluster to use as the source for cloning. If this is your first time creating an Aurora DB cluster, consult the tutorials in Getting started with Amazon Aurora to set up a cluster using either the MySQL or PostgreSQL database engine.
- A second VPC, if you intend to create a cross-VPC clone. If you don't have a VPC to use for the clone, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only) or Tutorial: Create a VPC for use with a DB cluster (dual-stack mode).

Gathering information about the network environment

With cross-VPC cloning, the network environment can differ substantially between the original cluster and its clone. Before you create the clone, collect and record information about the VPC, subnets, DB subnet group, and AZs used in the original cluster. That way, you can minimize the chances of problems. If a network problem does occur, you won't have to interrupt any troubleshooting activities to search for diagnostic information. The following sections show CLI examples to gather these types of information. You can save the details in whichever format is convenient to consult while creating the clone and doing any troubleshooting.

- Step 1: Check the Availability Zones of the original cluster
- Step 2: Check the DB subnet group of the original cluster
- Step 3: Check the subnets of the original cluster
- Step 4: Check the Availability Zones of the DB instances in the original cluster
- Step 5: Check the VPCs you can use for the clone

Step 1: Check the Availability Zones of the original cluster

Before you create the clone, verify which AZs the original cluster uses for its storage. As explained in Amazon Aurora storage and reliability, the storage for each Aurora cluster is associated with exactly three AZs. Because the Amazon Aurora DB clusters takes advantage of the separation of compute and storage, this rule is true regardless of how many instances are in the cluster.

For example, run a CLI command such as the following, substituting your own cluster name for *my_cluster*. The following example produces a list sorted alphabetically by the AZ name.

```
aws rds describe-db-clusters \
  --db-cluster-identifier my_cluster \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' \
  --output text
```

The following example shows sample output from the preceding `describe-db-clusters` command. It demonstrates that the storage for the Aurora cluster always uses three AZs.

```
us-east-1c
us-east-1d
us-east-1e
```

To create a clone in a network environment that doesn't have all the resources in place to connect to these AZs, you must create subnets associated with at least two of those AZs, and then create a DB subnet group containing those two or three subnets. The following examples show how.

Step 2: Check the DB subnet group of the original cluster

If you want to use the same number of subnets for the clone as in the original cluster, you can get the number of subnets from the DB subnet group of the original cluster. An Aurora DB subnet group contains at least two subnets, each associated with a different AZ. Make a note of which AZs the subnets are associated with.

The following example shows how to find the DB subnet group of the original cluster, and then work backwards to the corresponding AZs. Substitute the name of your cluster for *`my_cluster`* in the first command. Substitute the name of the DB subnet group for *`my_subnet`* in the second command.

```
aws rds describe-db-clusters --db-cluster-identifier my_cluster \
  --query '*[].DBSubnetGroup' --output text

aws rds describe-db-subnet-groups --db-subnet-group-name my_subnet_group \
  --query '*[].Subnets[].[SubnetAvailabilityZone.Name]' --output text
```

Sample output might look similar to the following, for a cluster with a DB subnet group containing containing two subnets. In this case, `two-subnets` is a name that was specified when creating the DB subnet group.

```
two-subnets

us-east-1d
us-east-1c
```

For a cluster where the DB subnet group contains three subnets, the output might look similar to the following.

```
three-subnets

us-east-1f
us-east-1d
us-east-1c
```

Step 3: Check the subnets of the original cluster

If you need more details about the subnets in the original cluster, run AWS CLI commands similar to the following. You can examine the subnet attributes such as IP address ranges, owner, and so on. That way, you can determine whether to use different subnets in the same VPC, or create subnets with similar characteristics in a different VPC.

Find the subnet IDs of all the subnets that are available in your VPC.

```
aws ec2 describe-subnets --filters Name=vpc-id,Values=my_vpc \
  --query '*[].[SubnetId]' --output text
```

Find the exact subnets used in your DB subnet group.

```
aws rds describe-db-subnet-groups --db-subnet-group-name my_subnet_group \
  --query '*[].Subnets[].[SubnetIdentifier]' --output text
```

Then specify the subnets that you want to investigate in a list, as in the following command. Substitute the names of your subnets for *`my_subnet_1`* and so on.

```
aws ec2 describe-subnets \
  --subnet-ids '["my_subnet_1","my_subnet2","my_subnet3"]'
```

The following example shows partial output from such a `describe-subnets` command. The output shows some of the important attributes you can see for each subnet, such as its associated AZ and the VPC that it’s part of.

```
{
    'Subnets': [
        {
            'AvailabilityZone': 'us-east-1d',
            'AvailableIpAddressCount': 54,
            'CidrBlock': '10.0.0.64/26',
            'State': 'available',
            'SubnetId': 'subnet-000a0bca00e0b0000',
            'VpcId': 'vpc-3f3c3fc3333b3ffb3',
            ...
        },
        {
            'AvailabilityZone': 'us-east-1c',
            'AvailableIpAddressCount': 55,
            'CidrBlock': '10.0.0.0/26',
            'State': 'available',
            'SubnetId': 'subnet-4b4dbfe4d4a4fd4c4',
            'VpcId': 'vpc-3f3c3fc3333b3ffb3',
            ...
```

Step 4: Check the Availability Zones of the DB instances in the original cluster

You can use this procedure to understand the AZs used for the DB instances in the original cluster. That way, you can set up the exact same AZs for the DB instances in the clone. You can also use more or fewer DB instances in the clone depending on whether the clone is used for production, development and testing, and so on.

For each instance in the original cluster, run a command such as the following. Make sure that the instance has finished creating and is in the `Available` state first. Substitute the instance identifier for *`my_instance`*.

```
aws rds describe-db-instances --db-instance-identifier my_instance \
  --query '*[].AvailabilityZone' --output text
```

The following example shows the output of running the preceding `describe-db-instances` command. The Aurora cluster has four database instances. Therefore, we run the command four times, substituting a different DB instance identifier each time. The output shows how those DB instances are spread across a maximum of three AZs.

```
us-east-1a
us-east-1c
us-east-1d
us-east-1a
```

After the clone is created and you are adding DB instances to it, you can specify these same AZ names in the `create-db-instance` commands. You might do so to set up DB instances in the new cluster configured for exactly the same AZs as in the original cluster.

Step 5: Check the VPCs you can use for the clone

If you intend to create the clone in a different VPC than the original, you can get a list of the VPC IDs available for your account. You might also do this step if you need to create any additional subnets in the same VPC as the original cluster. When you run the command to create a subnet, you specify the VPC ID as a parameter.

To list all the VPCs for your account, run the following CLI command:

```
aws ec2 describe-vpcs --query '*[].[VpcId]' --output text
```

The following example shows sample output from the preceding `describe-vpcs` command. The output demonstrates that there are four VPCs in the current AWS account that can be used as the source or the destination for cross-VPC cloning.

```
vpc-fd111111
vpc-2222e2cd2a222f22e
vpc-33333333a33333d33
```

```
vpc-4ae4d4de4a4444dad
```

You can use the same VPC as the destination for the clone, or a different VPC. If the original cluster and the clone are in the same VPC, you can reuse the same DB subnet group for the clone. You can also create a different DB subnet group. For example, the new DB subnet group might use private subnets, while the original cluster’s DB subnet group might use public subnets. If you create the clone in a different VPC, make sure that there are enough subnets in the new VPC and that the subnets are associated with the right AZs from the original cluster.

Creating network resources for the clone

If while collecting the network information you discovered that additional network resources are needed for the clone, you can create those resources before trying to set up the clone. For example, you might need to create more subnets, subnets associated with specific AZs, or a new DB subnet group.

- Step 1: Create the subnets for the clone
- Step 2: Create the DB subnet group for the clone

Step 1: Create the subnets for the clone

If you need to create new subnets for the clone, run a command similar to the following. You might need to do this when creating the clone in a different VPC, or when making some other network change such as using private subnets instead of public subnets.

AWS automatically generates the ID of the subnet. Substitute the name of the clone's VPC for *my_vpc*. Choose the address range for the `--cidr-block` option to allow at least 16 IP addresses in the range. You can include any other properties that you want to specify. Run the command `aws ec2 create-subnet help` to see all the choices.

```
aws ec2 create-subnet --vpc-id my_vpc \
  --availability-zone AZ_name --cidr-block IP_range
```

The following example shows some important attributes of a newly created subnet.

```
{
    'Subnet': {
        'AvailabilityZone': 'us-east-1b',
        'AvailableIpAddressCount': 59,
        'CidrBlock': '10.0.0.64/26',
        'State': 'available',
        'SubnetId': 'subnet-44b4a44f4e44db444',
        'VpcId': 'vpc-555fc5df555e555dc',
        ...
        }
}
```

Step 2: Create the DB subnet group for the clone

If you are creating the clone in a different VPC, or a different set of subnets within the same VPC, then you create a new DB subnet group and specify it when creating the clone.

Make sure that you know all the following details. You can find all of these from the output of the preceding examples.

1. VPC of the original cluster. For instructions, see Step 3: Check the subnets of the original cluster.
2. VPC of the clone, if you are creating it in a different VPC. For instructions, see Step 5: Check the VPCs you can use for the clone.
3. Three AZs associated with the Aurora storage for the original cluster. For instructions, see Step 1: Check the Availability Zones of the original cluster.
4. Two or three AZs associated with the DB subnet group for the original cluster. For instructions, see Step 2: Check the DB subnet group of the original cluster.
5. The subnet IDs and associated AZs of all the subnets in the VPC you intend to use for the clone. Use the same `describe-subnets` command as in Step 3: Check the subnets of the original cluster, substituting the VPC ID of the destination VPC.

Check how many AZs are both associated with the storage of the original cluster, and associated with subnets in the destination VPC. To successfully create the clone, there must be two or three AZs in common. If you have fewer than two AZs in common, go back to Step 1: Create the subnets for the clone. Create one, two, or three new subnets that are associated with the AZs associated with the storage of the original cluster.

Choose subnets in the destination VPC that are associated with the same AZs as the Aurora storage in the originally cluster. Ideally, choose three AZs. Doing so gives you the most flexibility to spread the DB instances of the clone across multiple AZs for high availability of compute resources.

Run a command similar to the following to create the new DB subnet group. Substitute the IDs of your subnets in the list. If you specify the subnet IDs using environment variables, be careful to quote the `--subnet-ids` parameter list in a way that preserves the double quotation marks around the IDs.

```
aws rds create-db-subnet-group --db-subnet-group-name my_subnet_group \
  --subnet-ids '["my_subnet_1","my_subnet_2","my_subnet3"]' \
  --db-subnet-group-description 'DB subnet group with 3 subnets for clone'
```

The following example shows partial output of the `create-db-subnet-group` command.

```
{
    'DBSubnetGroup': {
        'DBSubnetGroupName': 'my_subnet_group',
        'DBSubnetGroupDescription': 'DB subnet group with 3 subnets for clone',
        'VpcId': 'vpc-555fc5df555e555dc',
        'SubnetGroupStatus': 'Complete',
        'Subnets': [
            {
                'SubnetIdentifier': 'my_subnet_1',
                'SubnetAvailabilityZone': {
                    'Name': 'us-east-1c'
                },
                'SubnetStatus': 'Active'
            },
            {
                'SubnetIdentifier': 'my_subnet_2',
                'SubnetAvailabilityZone': {
                    'Name': 'us-east-1d'
                },
                'SubnetStatus': 'Active'
            }
            ...
        ],
        'SupportedNetworkTypes': [
            'IPV4'
        ]
    }
}
```

At this point, you haven't actually created the clone yet. You have created all the relevant VPC and subnet resources so that you can specify the appropriate parameters to the `restore-db-cluster-to-point-in-time` and `create-db-instance` commands when creating the clone.

Creating an Aurora clone with new network settings

Once you have made sure that the right configuration of VPCs, subnets, AZs, and subnet groups is in place for the new cluster to use, you can perform the actual cloning operation. The following CLI examples highlight the options such as `--db-subnet-group-name`, `--availability-zone`, and `--vpc-security-group-ids` that you specify on the commands to set up the clone and its DB instances.

- Step 1: Specify the DB subnet group for the clone
- Step 2: Specify network settings for instances in the clone
- Step 3: Establishing connectivity from a client system to a clone

Step 1: Specify the DB subnet group for the clone

When you create the clone, you can configure all the right VPC, subnet, and AZ settings by specifying a DB subnet group. Use the commands in the preceding examples to verify all the relationships and mappings that go into the DB subnet group.

For example, the following commands demonstrate cloning an original cluster to a clone. In the first example, the source cluster is associated with two subnets and the clone is associated with three subnets. The second example shows the opposite case, cloning from a cluster with three subnets to a cluster with two subnets.

```
aws rds restore-db-cluster-to-point-in-time \
  --source-db-cluster-identifier cluster-with-3-subnets \
  --db-cluster-identifier cluster-cloned-to-2-subnets \
  --restore-type copy-on-write --use-latest-restorable-time \
  --db-subnet-group-name two-subnets
```

If you intend to use Aurora Serverless v2 instances in the clone, include a `--serverless-v2-scaling-configuration` option when you create the clone, as shown. Doing so lets you use the `db.serverless` class when creating DB instances in the clone. You can also modify the clone later to add this scaling configuration attribute. The capacity numbers in this example allow each Serverless v2 instance in the cluster to scale between 2 and 32 Aurora Capacity Units (ACUs). For information about the Aurora Serverless v2 feature and how to choose the capacity range, see Using Aurora Serverless v2.

```
aws rds restore-db-cluster-to-point-in-time \
  --source-db-cluster-identifier cluster-with-2-subnets \
  --db-cluster-identifier cluster-cloned-to-3-subnets \
  --restore-type copy-on-write --use-latest-restorable-time \
  --db-subnet-group-name three-subnets \
  --serverless-v2-scaling-configuration 'MinCapacity=2,MaxCapacity=32'
```

Regardless of the number of subnets used by the DB instances, the Aurora storage for the source cluster and the clone is associated with three AZs. The following example lists the AZs associated with both the original cluster and the clone, for both of the `restore-db-cluster-to-point-in-time` commands in the preceding examples.

```
aws rds describe-db-clusters --db-cluster-identifier cluster-with-3-subnets \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1c
us-east-1d
us-east-1f

aws rds describe-db-clusters --db-cluster-identifier cluster-cloned-to-2-subnets \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1c
us-east-1d
us-east-1f

aws rds describe-db-clusters --db-cluster-identifier cluster-with-2-subnets \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1a
us-east-1c
us-east-1d

aws rds describe-db-clusters --db-cluster-identifier cluster-cloned-to-3-subnets \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1a
us-east-1c
us-east-1d
```

Because at least two of the AZs overlap between each pair of original and clone clusters, both clusters can access the same underlying Aurora storage.

Step 2: Specify network settings for instances in the clone

When you create DB instances in the clone, by default they inherit the DB subnet group from the cluster itself. That way, Aurora automatically assigns each instance to a particular subnet, and creates it in the AZ that's associated with the subnet. This choice is convenient, especially for development and test systems, because you don't have to keep track of the subnet IDs or the AZs while adding new instances to the clone.

As an alternative, you can specify the AZ when you create an Aurora DB instance for the clone. The AZ that you specify must be from the set of AZs that are associated with the clone. If the DB subnet group you use for the clone only contains two subnets, then you can only pick from the AZs associated with those two subnets. This choice offers flexibility and resilience for highly available systems, because you can make sure that the writer instance and the standby reader instance are in different AZs. Or if you add additional readers to the cluster, you can make sure that they are spread across three AZs. That way, even in the rare case of an AZ failure, you still have a writer instance and another reader instance in two other AZs.

The following example adds a provisioned DB instance to a cloned Aurora PostgreSQL cluster that uses a custom DB subnet group.

```
aws rds create-db-instance --db-cluster-identifier my_aurora_postgresql_clone \
  --db-instance-identifier my_postgres_instance \
  --db-subnet-group-name my_new_subnet \
  --engine aurora-postgresql \
  --db-instance-class db.t4g.medium
```

The following example shows partial output from such a command.

```
{
  'DBInstanceIdentifier': 'my_postgres_instance',
  'DBClusterIdentifier': 'my_aurora_postgresql_clone',
  'DBInstanceClass': 'db.t4g.medium',
  'DBInstanceStatus': 'creating'
  ...
}
```

The following example adds an Aurora Serverless v2 DB instance to an Aurora MySQL clone that uses a custom DB subnet group. To be able to use Serverless v2 instances, make sure to specify the `--serverless-v2-scaling-configuration` option for the `restore-db-cluster-to-point-in-time` command, as shown in preceding examples.

```
aws rds create-db-instance --db-cluster-identifier my_aurora_mysql_clone \
  --db-instance-identifier my_mysql_instance \
  --db-subnet-group-name my_other_new_subnet \
  --engine aurora-mysql \
  --db-instance-class db.serverless
```

The following example shows partial output from such a command.

```
{
  'DBInstanceIdentifier': 'my_mysql_instance',
  'DBClusterIdentifier': 'my_aurora_mysql_clone',
  'DBInstanceClass': 'db.serverless',
  'DBInstanceStatus': 'creating'
  ...
}
```

Step 3: Establishing connectivity from a client system to a clone

If you are already connecting to an Aurora cluster from a client system, you might want to allow the same type of connectivity to a new clone. For example, you might connect to the original cluster from an Amazon Cloud9 instance or EC2 instance. To allow connections from the same client systems, or new ones that you create in the destination VPC, set up equivalent DB subnet groups and VPC security groups as in the VPC. Then specify the subnet group and security groups when you create the clone.

The following examples set up an Aurora Serverless v2 clone. That configuration is based on the combination of `--engine-mode provisioned` and `--serverless-v2-scaling-configuration` when creating the DB cluster, and then `--db-instance-class db.serverless` when creating each DB instance in the cluster. The `provisioned` engine mode is the default, so you can omit that option if you prefer.

```
aws rds restore-db-cluster-to-point-in-time \
  --source-db-cluster-identifier serverless-sql-postgres\
  --db-cluster-identifier serverless-sql-postgres-clone \
  --db-subnet-group-name 'default-vpc-1234' \
  --vpc-security-group-ids 'sg-4567' \
  --serverless-v2-scaling-configuration 'MinCapacity=0.5,MaxCapacity=16' \
  --restore-type copy-on-write \
  --use-latest-restorable-time
```

Then, when creating the DB instances in the clone, specify the same `--db-subnet-group-name` option. Optionally, you can include the `--availability-zone` option and specify one of the AZs associated with the subnets in that subnet group. That AZ must also be one of the AZs associated with the original cluster.

```
aws rds create-db-instance \
  --db-cluster-identifier serverless-sql-postgres-clone \
  --db-instance-identifier serverless-sql-postgres-clone-instance \
  --db-instance-class db.serverless \
  --db-subnet-group-name 'default-vpc-987zyx654' \
  --availability-zone 'us-east-1c' \
  --engine aurora-postgresql
```

Moving a cluster from public subnets to private ones

You can use cloning to migrate a cluster between public and private subnets. You might do this when adding additional layers of security to your application before deploying it to production. For this example, you should already have the private subnets and NAT gateway set up before starting the cloning process with Aurora.

For the steps involving Aurora, you can follow the same general steps as in the preceding examples to Gathering information about the network environment and Creating an Aurora clone with new network settings. The main difference is that even if you have public subnets that map to all the AZs from the original cluster, now you must verify that you have enough private subnets for an Aurora cluster, and that those subnets are associated with all the same AZs that are used for Aurora storage in the original cluster. Similar to other cloning use cases, you can make the DB subnet group for the clone with either three or two private subnets that are associated with the required AZs. However, if you use two private subnets in the DB subnet group, you must have a third private subnet that's associated with the third AZ used for Aurora storage in the original cluster.

You can consult this checklist to verify that all the requirements are in place to perform this type of cloning operation.

- Record the three AZs that are associated with the original cluster. For instructions, see Step 1: Check the Availability Zones of the original cluster.
- Record the three or two AZs that are associated with the public subnets in the DB subnet group for the original cluster. For instructions, see Step 3: Check the subnets of the original cluster.
- Create private subnets that map to all three of the AZs that are associated with the original cluster. Also do any other networking setup, such as creating a NAT gateway, to be able to

communicate with the private subnets. For instructions, see Create a subnet in the *Amazon Virtual Private Cloud User Guide*.

- Create a new DB subnet group containing three or two of the private subnets that are associated with the AZs from the first point. For instructions, see Step 2: Create the DB subnet group for the clone.

When all the prerequisites are in place, you can pause database activity on the original cluster while you create the clone and switch your application to use it. After the clone is created and you verify that you can connect to it, run your application code, and so on, you can discontinue use of the original cluster.

End-to-end example of creating a cross-VPC clone

Creating a clone in a different VPC than the original uses the same general steps as in the preceding examples. Because the VPC ID is a property of the subnets, you don't actually specify the VPC ID as a parameter when running any of the RDS CLI commands. The main difference is that you are more likely to need to create new subnets, new subnets mapped to specific AZs, a VPC security group, and a new DB subnet group. That's especially true if this is the first Aurora cluster that you create in that VPC.

You can consult this checklist to verify that all the requirements are in place to perform this type of cloning operation.

- Record the three AZs that are associated with the original cluster. For instructions, see Step 1: Check the Availability Zones of the original cluster.
- Record the three or two AZs that are associated with the subnets in the DB subnet group for the original cluster. For instructions, see Step 2: Check the DB subnet group of the original cluster.
- Create subnets that map to all three of the AZs that are associated with the original cluster. For instructions, see Step 1: Create the subnets for the clone.
- Do any other networking setup, such as setting up a VPC security group, for client systems, application servers, and so on to be able to communicate with the DB instances in the clone. For instructions, see Controlling access with security groups.
- Create a new DB subnet group containing three or two of the subnets that are associated with the AZs from the first point. For instructions, see Step 2: Create the DB subnet group for the clone.

When all the prerequisites are in place, you can pause database activity on the original cluster while you create the clone and switch your application to use it. After the clone is created and you verify that you can connect to it, run your application code, and so on, you can consider whether to keep both the original and clones running, or discontinue use of the original cluster.

The following Linux examples show the sequence of AWS CLI operations to clone an Aurora DB cluster from one VPC to another. Some fields that aren't relevant to the examples aren't shown in the command output.

First, we check the IDs of the source and destination VPCs. The descriptive name that you assign to a VPC when you create it is represented as a tag in the VPC metadata.

```
$ aws ec2 describe-vpcs --query '*[].[VpcId,Tags]'
[
    [
        'vpc-0f0c0fc0000b0ffb0',
        [
            {
                'Key': 'Name',
                'Value': 'clone-vpc-source'
            }
        ]
    ],
    [
        'vpc-9e99d9f99a999bd99',
        [
            {
                'Key': 'Name',
                'Value': 'clone-vpc-dest'
            }
        ]
    ]
]
```

The original cluster already exists in the source VPC. To set up the clone using the same set of AZs for the Aurora storage, we check the AZs used by the original cluster.

```
$ aws rds describe-db-clusters --db-cluster-identifier original-cluster \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1c
us-east-1d
```

```
us-east-1f
```

We make sure there are subnets that correspond to the AZs used by the original cluster: `us-east-1c`, `us-east-1d`, and `us-east-1f`.

```
$ aws ec2 create-subnet --vpc-id vpc-9e99d9f99a999bd99 \
  --availability-zone us-east-1c --cidr-block 10.0.0.128/28
{
    'Subnet': {
        'AvailabilityZone': 'us-east-1c',
        'SubnetId': 'subnet-3333a33be3ef3e333',
        'VpcId': 'vpc-9e99d9f99a999bd99',
    }
}

$ aws ec2 create-subnet --vpc-id vpc-9e99d9f99a999bd99 \
--availability-zone us-east-1d --cidr-block 10.0.0.160/28
{
    'Subnet': {
        'AvailabilityZone': 'us-east-1d',
        'SubnetId': 'subnet-4eeb444cd44b4d444',
        'VpcId': 'vpc-9e99d9f99a999bd99',
    }
}

$ aws ec2 create-subnet --vpc-id vpc-9e99d9f99a999bd99 \
--availability-zone us-east-1f --cidr-block 10.0.0.224/28
{
    'Subnet': {
        'AvailabilityZone': 'us-east-1f',
        'SubnetId': 'subnet-66eea6666fb66d66c',
        'VpcId': 'vpc-9e99d9f99a999bd99',
    }
}
```

This example confirms that there are subnets that map to the necessary AZs in the destination VPC.

```
aws ec2 describe-subnets --query 'sort_by(*[] | [?VpcId == `vpc-9e99d9f99a999bd99`] |
[].{SubnetId:SubnetId,VpcId:VpcId,AvailabilityZone:AvailabilityZone},
 &AvailabilityZone)' --output table

---------------------------------------------------------------------------
|                             DescribeSubnets                             |
+------------------+----------------------------+-------------------------+
| AvailabilityZone |          SubnetId          |          VpcId          |
+------------------+----------------------------+-------------------------+
|  us-east-1a      |  subnet-000ff0e00000c0aea  |  vpc-9e99d9f99a999bd99  |
|  us-east-1b      |  subnet-1111d111111ca11b1  |  vpc-9e99d9f99a999bd99  |
|  us-east-1c      | subnet-3333a33be3ef3e333   |  vpc-9e99d9f99a999bd99  |
|  us-east-1d      | subnet-4eeb444cd44b4d444   |  vpc-9e99d9f99a999bd99  |
|  us-east-1f      | subnet-66eea6666fb66d66c   |  vpc-9e99d9f99a999bd99  |
+------------------+----------------------------+-------------------------+
```

Before creating an Aurora DB cluster in the VPC, you must have a DB subnet group with subnets that map to the AZs used for Aurora storage. When you create a regular cluster, you can use any set of three AZs. When you clone an existing cluster, the subnet group must match at least two of the three AZs that it uses for Aurora storage.

```
$ aws rds create-db-subnet-group \
  --db-subnet-group-name subnet-group-in-other-vpc \
  --subnet-ids
 '["subnet-3333a33be3ef3e333","subnet-4eeb444cd44b4d444","subnet-66eea6666fb66d66c"]' \
  --db-subnet-group-description 'DB subnet group with 3 subnets:
 subnet-3333a33be3ef3e333,subnet-4eeb444cd44b4d444,subnet-66eea6666fb66d66c'

{
    'DBSubnetGroup': {
        'DBSubnetGroupName': 'subnet-group-in-other-vpc',
        'DBSubnetGroupDescription': 'DB subnet group with 3 subnets:
 subnet-3333a33be3ef3e333,subnet-4eeb444cd44b4d444,subnet-66eea6666fb66d66c',
        'VpcId': 'vpc-9e99d9f99a999bd99',
        'SubnetGroupStatus': 'Complete',
        'Subnets': [
            {
                'SubnetIdentifier': 'subnet-4eeb444cd44b4d444',
                'SubnetAvailabilityZone': { 'Name': 'us-east-1d' }
            },
            {
                'SubnetIdentifier': 'subnet-3333a33be3ef3e333',
                'SubnetAvailabilityZone': { 'Name': 'us-east-1c' }
            },
            {
                'SubnetIdentifier': 'subnet-66eea6666fb66d66c',
                'SubnetAvailabilityZone': { 'Name': 'us-east-1f' }
            }
        ]
    }
}
```

Now the subnets and DB subnet group are in place. The following example shows the `restore-db-cluster-to-point-in-time` that clones the cluster. The `--db-subnet-group-name` option associates the clone with the correct set of subnets that map to the correct set of AZs from the original cluster.

```
$ aws rds restore-db-cluster-to-point-in-time \
  --source-db-cluster-identifier original-cluster \
  --db-cluster-identifier clone-in-other-vpc \
  --restore-type copy-on-write --use-latest-restorable-time \
  --db-subnet-group-name subnet-group-in-other-vpc

{
  'DBClusterIdentifier': 'clone-in-other-vpc',
  'DBSubnetGroup': 'subnet-group-in-other-vpc',
  'Engine': 'aurora-postgresql',
  'EngineVersion': '15.4',
  'Status': 'creating',
  'Endpoint': 'clone-in-other-vpc.cluster-c0abcdef.us-east-1.rds.amazonaws.com'
}
```

The following example confirms that the Aurora storage in the clone uses the same set of AZs as in the original cluster.

```
$ aws rds describe-db-clusters --db-cluster-identifier clone-in-other-vpc \
  --query 'sort_by(*[].AvailabilityZones[].{Zone:@},&Zone)' --output text

us-east-1c
us-east-1d
us-east-1f
```

At this point, you can create DB instances for the clone. Make sure that the VPC security group associated with each instance allows connections from the IP address ranges you use for the EC2 instances, application servers, and so on that are in the destination VPC.

Cross-account cloning with AWS RAM and Amazon Aurora

By using AWS Resource Access Manager (AWS RAM) with Amazon Aurora, you can share Aurora DB clusters and clones that belong to your AWS account with another AWS account or organization.

Such *cross-account cloning* is much faster than creating and restoring a database snapshot. You can create a clone of one of your Aurora DB clusters and share the clone. Or you can share your Aurora DB cluster with another AWS account and let the account holder create the clone. The approach that you choose depends on your use case.

For example, you might need to regularly share a clone of your financial database with your organization's internal auditing team. In this case, your auditing team has its own AWS account for the applications that it uses. You can give the auditing team's AWS account the permission to access your Aurora DB cluster and clone it as needed.

On the other hand, if an outside vendor audits your financial data you might prefer to create the clone yourself. You then give the outside vendor access to the clone only.

You can also use cross-account cloning to support many of the same use cases for cloning within the same AWS account, such as development and testing. For example, your organization might use different AWS accounts for production, development, testing, and so on. For more information, see Overview of Aurora cloning.

Thus, you might want to share a clone with another AWS account or allow another AWS account to create clones of your Aurora DB clusters. In either case, start by using AWS RAM to create a share object. For complete information about sharing AWS resources between AWS accounts, see the AWS RAM User Guide.

Creating a cross-account clone requires actions from the AWS account that owns the original cluster, and the AWS account that creates the clone. First, the original cluster owner modifies the cluster to allow one or more other accounts to clone it. If any of the accounts is in a different AWS organization, AWS generates a sharing invitation. The other account must accept the invitation before proceeding. Then each authorized account can clone the cluster. Throughout this process, the cluster is identified by its unique Amazon Resource Name (ARN).

As with cloning within the same AWS account, additional storage space is used only if changes are made to the data by the source or the clone. Charges for storage are then applied at that time. If the source cluster is deleted, storage costs are distributed equally among remaining cloned clusters.

Topics

Limitations of cross-account cloning

Aurora cross-account cloning has the following limitations:

- You can't clone an Aurora Serverless v1 cluster across AWS accounts.
- You can't view or accept invitations to shared resources with the AWS Management Console. Use the AWS CLI, the Amazon RDS API, or the AWS RAM console to view and accept invitations to shared resources.
- You can create only one new clone from a clone that's been shared with your AWS account.
- You can't share resources (clones or Aurora DB clusters) that have been shared with your AWS account.
- You can create a maximum of 15 cross-account clones from any single Aurora DB cluster.
- Each of the 15 cross-account clones must be owned by a different AWS account. That is, you can only create one cross-account clone of a cluster within any AWS account.
- After you clone a cluster, the original cluster and its clone are considered to be the same for purposes of enforcing limits on cross-account clones. You can't create cross-account clones of both the original cluster and the cloned cluster within the same AWS account. The total number of cross-account clones for the original cluster and any of its clones can't exceed 15.
- You can't share an Aurora DB cluster with other AWS accounts unless the cluster is in an `ACTIVE` state.
- You can't rename an Aurora DB cluster that's been shared with other AWS accounts.
- You can't create a cross-account clone of a cluster that is encrypted with the default RDS key.
- You can't create nonencrypted clones in one AWS account from encrypted Aurora DB clusters that have been shared by another AWS account. The cluster owner must grant permission to access the source cluster's AWS KMS key. However, you can use a different key when you create the clone.

Allowing other AWS accounts to clone your cluster

To allow other AWS accounts to clone a cluster that you own, use AWS RAM to set the sharing permission. Doing so also sends an invitation to each of the other accounts that's in a different AWS organization.

For the procedures to share resources owned by you in the AWS RAM console, see Sharing resources owned by you in the *AWS RAM User Guide*.

Topics

- Granting permission to other AWS accounts to clone your cluster
- Checking if a cluster that you own is shared with other AWS accounts

Granting permission to other AWS accounts to clone your cluster

If the cluster that you're sharing is encrypted, you also share the AWS KMS key for the cluster. You can allow AWS Identity and Access Management (IAM) users or roles in one AWS account to use a KMS key in a different account.

To do this, you first add the external account (root user) to the KMS key's key policy through AWS KMS. You don't add the individual users or roles to the key policy, only the external account that owns them. You can only share a KMS key that you create, not the default RDS service key. For information about access control for KMS keys, see Authentication and access control for AWS KMS.

Console

To grant permission to clone your cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster that you want to share to see its **Details** page, and choose the **Connectivity & security** tab.
4. In the **Share DB cluster with other AWS accounts** section, enter the numeric account ID for the AWS account that you want to allow to clone this cluster. For account IDs in the same organization, you can begin typing in the box and then choose from the menu.

   > ⚠ **Important**
   >
   > In some cases, you might want an account that is not in the same AWS organization as your account to clone a cluster. In these cases, for security reasons the console doesn't report who owns that account ID or whether the account exists.
   > Be careful entering account numbers that are not in the same AWS organization as your AWS account. Immediately verify that you shared with the intended account.

5. On the confirmation page, verify that the account ID that you specified is correct. Enter `share` in the confirmation box to confirm.

On the **Details** page, an entry appears that shows the specified AWS account ID under **Accounts that this DB cluster is shared with**. The **Status** column initially shows a status of **Pending**.

6. Contact the owner of the other AWS account, or sign in to that account if you own both of them. Instruct the owner of the other account to accept the sharing invitation and clone the DB cluster, as described following.

AWS CLI

To grant permission to clone your cluster

1. Gather the information for the required parameters. You need the ARN for your cluster and the numeric ID for the other AWS account.
2. Run the AWS RAM CLI command `create-resource-share`.

   For Linux, macOS, or Unix:

   ```
   aws ram create-resource-share --name descriptive_name \
     --region region \
     --resource-arns cluster_arn \
     --principals other_account_ids
   ```

   For Windows:

   ```
   aws ram create-resource-share --name descriptive_name ^
     --region region ^
     --resource-arns cluster_arn ^
     --principals other_account_ids
   ```

   To include multiple account IDs for the `--principals` parameter, separate IDs from each other with spaces. To specify whether the permitted account IDs can be outside your AWS organization, include the `--allow-external-principals` or `--no-allow-external-principals` parameter for `create-resource-share`.

AWS RAM API

To grant permission to clone your cluster

1. Gather the information for the required parameters. You need the ARN for your cluster and the numeric ID for the other AWS account.
2. Call the AWS RAM API operation CreateResourceShare, and specify the following values:
   - Specify the account ID for one or more AWS accounts as the `principals` parameter.
   - Specify the ARN for one or more Aurora DB clusters as the `resourceArns` parameter.
   - Specify whether the permitted account IDs can be outside your AWS organization by including a Boolean value for the `allowExternalPrincipals` parameter.

Recreating a cluster that uses the default RDS key

If the encrypted cluster that you plan to share uses the default RDS key, make sure to recreate the cluster. To do this, create a manual snapshot of your DB cluster, use an AWS KMS key, and then restore the cluster to a new cluster. Then share the new cluster. To perform this process, take the following steps.

To recreate an encrypted cluster that uses the default RDS key

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Snapshots** from the navigation pane.
3. Choose your snapshot.
4. For **Actions**, choose **Copy Snapshot**, and then choose **Enable encryption**.
5. For **AWS KMS key**, choose the new encryption key that you want to use.
6. Restore the copied snapshot. To do so, follow the procedure in Restoring from a DB cluster snapshot. The new DB instance uses your new encryption key.
7. (Optional) Delete the old DB cluster if you no longer need it. To do so, follow the procedure in Deleting a DB cluster snapshot. Before you do, confirm that your new cluster has all necessary data and that your application can access it successfully.

Checking if a cluster that you own is shared with other AWS accounts

You can check if other users have permission to share a cluster. Doing so can help you understand whether the cluster is approaching the limit for the maximum number of cross-account clones.

For the procedures to share resources using the AWS RAM console, see Sharing resources owned by you in the *AWS RAM User Guide*.

AWS CLI

To find out if a cluster that you own is shared with other AWS accounts

- Call the AWS RAM CLI command `list-principals`, using your account ID as the resource owner and the ARN of your cluster as the resource ARN. You can see all shares with the following command. The results indicate which AWS accounts are allowed to clone the cluster.

```
aws ram list-principals \
    --resource-arns your_cluster_arn \
    --principals your_aws_id
```

AWS RAM API

To find out if a cluster that you own is shared with other AWS accounts

- Call the AWS RAM API operation ListPrincipals. Use your account ID as the resource owner and the ARN of your cluster as the resource ARN.

Cloning a cluster that is owned by another AWS account

To clone a cluster that's owned by another AWS account, use AWS RAM to get permission to make the clone. After you have the required permission, use the standard procedure for cloning an Aurora cluster.

You can also check whether a cluster that you own is a clone of a cluster owned by a different AWS account.

For the procedures to work with resources owned by others in the AWS RAM console, see Accessing resources shared with you in the *AWS RAM User Guide.*

Topics

- Viewing invitations to clone clusters that are owned by other AWS accounts
- Accepting invitations to share clusters owned by other AWS accounts
- Cloning an Aurora cluster that is owned by another AWS account
- Checking if a DB cluster is a cross-account clone

Viewing invitations to clone clusters that are owned by other AWS accounts

To work with invitations to clone clusters owned by AWS accounts in other AWS organizations, use the AWS CLI, the AWS RAM console, or the AWS RAM API. Currently, you can't perform this procedure using the Amazon RDS console.

For the procedures to work with invitations in the AWS RAM console, see Accessing resources shared with you in the *AWS RAM User Guide*.

AWS CLI

To see invitations to clone clusters that are owned by other AWS accounts

1. Run the AWS RAM CLI command `get-resource-share-invitations`.

   ```
   aws ram get-resource-share-invitations --region region_name
   ```

   The results from the preceding command show all invitations to clone clusters, including any that you already accepted or rejected.

2. (Optional) Filter the list so you see only the invitations that require action from you. To do so, add the parameter `--query 'resourceShareInvitations[?status==`PENDING`]'`.

AWS RAM API

To see invitations to clone clusters that are owned by other AWS accounts

1. Call the AWS RAM API operation `GetResourceShareInvitations`. This operation returns all such invitations, including any that you already accepted or rejected.
2. (Optional) Find only the invitations that require action from you by checking the `resourceShareAssociations` return field for a `status` value of `PENDING`.

Accepting invitations to share clusters owned by other AWS accounts

You can accept invitations to share clusters owned by other AWS accounts that are in different AWS organizations. To work with these invitations, use the AWS CLI, the AWS RAM and RDS APIs, or the AWS RAM console. Currently, you can't perform this procedure using the RDS console.

For the procedures to work with invitations in the AWS RAM console, see Accessing resources shared with you in the *AWS RAM User Guide*.

AWS CLI

To accept an invitation to share a cluster from another AWS account

1. Find the invitation ARN by running the AWS RAM CLI command `get-resource-share-invitations`, as shown preceding.
2. Accept the invitation by calling the AWS RAM CLI command `accept-resource-share-invitation`, as shown following.

   For Linux, macOS, or Unix:

   ```
   aws ram accept-resource-share-invitation \
     --resource-share-invitation-arn invitation_arn \
     --region region
   ```

   For Windows:

   ```
   aws ram accept-resource-share-invitation ^
     --resource-share-invitation-arn invitation_arn ^
     --region region
   ```

AWS RAM and RDS API

To accept invitations to share somebody's cluster

1. Find the invitation ARN by calling the AWS RAM API operation `GetResourceShareInvitations`, as shown preceding.
2. Pass that ARN as the `resourceShareInvitationArn` parameter to the RDS API operation AcceptResourceShareInvitation.

Cloning an Aurora cluster that is owned by another AWS account

After you accept the invitation from the AWS account that owns the DB cluster, as shown preceding, you can clone the cluster.

Console

To clone an Aurora cluster that is owned by another AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

   At the top of the database list, you should see one or more items with a **Role** value of `Shared from account #`*`account_id`*. For security reasons, you can only see limited information about the original clusters. The properties that you can see are the ones such as database engine and version that must be the same in your cloned cluster.
3. Choose the cluster that you intend to clone.
4. For **Actions**, choose **Create clone**.
5. Follow the procedure in Console to finish setting up the cloned cluster.
6. As needed, enable encryption for the cloned cluster. If the cluster that you are cloning is encrypted, you must enable encryption for the cloned cluster. The AWS account that shared the cluster with you must also share the KMS key that was used to encrypt the cluster. You can use the same KMS key to encrypt the clone, or your own KMS key. You can't create a cross-account clone for a cluster that is encrypted with the default KMS key.

   The account that owns the encryption key must grant permission to use the key to the destination account by using a key policy. This process is similar to how encrypted snapshots are shared, by using a key policy that grants permission to the destination account to use the key.

AWS CLI

To clone an Aurora cluster owned by another AWS account

1. Accept the invitation from the AWS account that owns the DB cluster, as shown preceding.

2. Clone the cluster by specifying the full ARN of the source cluster in the `source-db-cluster-identifier` parameter of the RDS CLI command [`restore-db-cluster-to-point-in-time`](), as shown following.

   If the ARN passed as the `source-db-cluster-identifier` hasn't been shared, the same error is returned as if the specified cluster doesn't exist.

   For Linux, macOS, or Unix:

   ```
   aws rds restore-db-cluster-to-point-in-time \
     --source-db-cluster-identifier=arn:aws:rds:arn_details \
     --db-cluster-identifier=new_cluster_id \
     --restore-type=copy-on-write \
     --use-latest-restorable-time
   ```

   For Windows:

   ```
   aws rds restore-db-cluster-to-point-in-time ^
     --source-db-cluster-identifier=arn:aws:rds:arn_details ^
     --db-cluster-identifier=new_cluster_id ^
     --restore-type=copy-on-write ^
     --use-latest-restorable-time
   ```

3. If the cluster that you are cloning is encrypted, encrypt your cloned cluster by including a `kms-key-id` parameter. This `kms-key-id` value can be the same one used to encrypt the original DB cluster, or your own KMS key. Your account must have permission to use that encryption key.

   For Linux, macOS, or Unix:

   ```
   aws rds restore-db-cluster-to-point-in-time \
     --source-db-cluster-identifier=arn:aws:rds:arn_details \
     --db-cluster-identifier=new_cluster_id \
     --restore-type=copy-on-write \
     --use-latest-restorable-time \
     --kms-key-id=arn:aws:kms:arn_details
   ```

   For Windows:

   ```
   aws rds restore-db-cluster-to-point-in-time ^
   ```

```
  --source-db-cluster-identifier=arn:aws:rds:arn_details ^
  --db-cluster-identifier=new_cluster_id ^
  --restore-type=copy-on-write ^
  --use-latest-restorable-time ^
  --kms-key-id=arn:aws:kms:arn_details
```

The account that owns the encryption key must grant permission to use the key to the destination account by using a key policy. This process is similar to how encrypted snapshots are shared, by using a key policy that grants permission to the destination account to use the key. An example of a key policy follows.

```
{
  "Id": "key-policy-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Allow use of the key",
      "Effect": "Allow",
      "Principal": {"AWS": [
        "arn:aws:iam::account_id:user/KeyUser",
        "arn:aws:iam::account_id:root"
      ]},
      "Action": [
        "kms:CreateGrant",
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey"
      ],
      "Resource": "*"
    },
    {
      "Sid": "Allow attachment of persistent resources",
      "Effect": "Allow",
      "Principal": {"AWS": [
        "arn:aws:iam::account_id:user/KeyUser",
        "arn:aws:iam::account_id:root"
      ]},
      "Action": [
        "kms:CreateGrant",
        "kms:ListGrants",
        "kms:RevokeGrant"
      ],
      "Resource": "*",
      "Condition": {"Bool": {"kms:GrantIsForAWSResource": true}}
    }
  ]
}
```

> **Note**
>
> The restore-db-cluster-to-point-in-time AWS CLI command restores only the DB cluster, not the DB instances for that DB cluster. To create DB instances for the restored DB cluster, invoke the create-db-instance command. Specify the identifier of the restored DB cluster in `--db-cluster-identifier`.
> You can create DB instances only after the `restore-db-cluster-to-point-in-time` command has completed and the DB cluster is available.

RDS API

To clone an Aurora cluster owned by another AWS account

1. Accept the invitation from the AWS account that owns the DB cluster, as shown preceding.
2. Clone the cluster by specifying the full ARN of the source cluster in the `SourceDBClusterIdentifier` parameter of the RDS API operation `RestoreDBClusterToPointInTime`.

   If the ARN passed as the `SourceDBClusterIdentifier` hasn't been shared, then the same error is returned as if the specified cluster doesn't exist.
3. If the cluster that you are cloning is encrypted, include a `KmsKeyId` parameter to encrypt your cloned cluster. This `kms-key-id` value can be the same one used to encrypt the original DB cluster, or your own KMS key. Your account must have permission to use that encryption key.

   When you clone a volume, the destination account must have permission to use the encryption key used to encrypt the source cluster. Aurora encrypts the new cloned cluster with the encryption key specified in `KmsKeyId`.

The account that owns the encryption key must grant permission to use the key to the destination account by using a key policy. This process is similar to how encrypted snapshots are shared, by using a key policy that grants permission to the destination account to use the key. An example of a key policy follows.

```
{
  "Id": "key-policy-1",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Allow use of the key",
      "Effect": "Allow",
      "Principal": {"AWS": [
        "arn:aws:iam::account_id:user/KeyUser",
        "arn:aws:iam::account_id:root"
      ]},
      "Action": [
        "kms:CreateGrant",
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey"
      ],
      "Resource": "*"
    },
    {
      "Sid": "Allow attachment of persistent resources",
      "Effect": "Allow",
      "Principal": {"AWS": [
        "arn:aws:iam::account_id:user/KeyUser",
        "arn:aws:iam::account_id:root"
      ]},
      "Action": [
        "kms:CreateGrant",
        "kms:ListGrants",
        "kms:RevokeGrant"
      ],
      "Resource": "*",
      "Condition": {"Bool": {"kms:GrantIsForAWSResource": true}}
    }
  ]
}
```

Note

The RestoreDBClusterToPointInTime RDS API operation restores only the DB cluster, not the DB instances for that DB cluster. To create DB instances for the restored DB cluster, invoke the CreateDBInstance RDS API operation. Specify the identifier of the restored DB cluster in `DBClusterIdentifier`. You can create DB instances only after the `RestoreDBClusterToPointInTime` operation has completed and the DB cluster is available.

Checking if a DB cluster is a cross-account clone

The `DBClusters` object identifies whether each cluster is a cross-account clone. You can see the clusters that you have permission to clone by using the `include-shared` option when you run the RDS CLI command `describe-db-clusters`. However, you can't see most of the configuration details for such clusters.

AWS CLI

To check if a DB cluster is a cross-account clone

- Call the RDS CLI command `describe-db-clusters`.

  The following example shows how actual or potential cross-account clone DB clusters appear in `describe-db-clusters` output. For existing clusters owned by your AWS account, the `CrossAccountClone` field indicates whether the cluster is a clone of a DB cluster that is owned by another AWS account.

  In some cases, an entry might have a different AWS account number than yours in the `DBClusterArn` field. In this case, that entry represents a cluster that is owned by a different AWS account and that you can clone. Such entries have few fields other than `DBClusterArn`. When creating the cloned cluster, specify the same `StorageEncrypted`, `Engine`, and `EngineVersion` values as in the original cluster.

  ```
  $aws rds describe-db-clusters --include-shared --region us-east-1
  {
    "DBClusters": [
  ```

```
      {
          "EarliestRestorableTime": "2023-02-01T21:17:54.106Z",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0",
          "CrossAccountClone": false,
...
      },
      {
          "EarliestRestorableTime": "2023-02-09T16:01:07.398Z",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0",
          "CrossAccountClone": true,
...
      },
      {
          "StorageEncrypted": false,
          "DBClusterArn": "arn:aws:rds:us-east-1:12345678:cluster:cluster-
abcdefgh",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0
  ]
}
```

RDS API

To check if a DB cluster is a cross-account clone

- Call the RDS API operation DescribeDBClusters.

  For existing clusters owned by your AWS account, the `CrossAccountClone` field indicates whether the cluster is a clone of a DB cluster owned by another AWS account. Entries with a different AWS account number in the `DBClusterArn` field represent clusters that you can clone and that are owned by other AWS accounts. These entries have few fields other than `DBClusterArn`. When creating the cloned cluster, specify the same `StorageEncrypted`, `Engine`, and `EngineVersion` values as in the original cluster.

  The following example shows a return value that demonstrates both actual and potential cloned clusters.

  ```
  {
    "DBClusters": [
  ```

```
      {
          "EarliestRestorableTime": "2023-02-01T21:17:54.106Z",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0",
          "CrossAccountClone": false,
...
      },
      {
          "EarliestRestorableTime": "2023-02-09T16:01:07.398Z",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0",
          "CrossAccountClone": true,
...
      },
      {
          "StorageEncrypted": false,
          "DBClusterArn": "arn:aws:rds:us-east-1:12345678:cluster:cluster-
abcdefgh",
          "Engine": "aurora-mysql",
          "EngineVersion": "8.0.mysql_aurora.3.02.0"
      }
  ]
}
```

Integrating Aurora with other AWS services

Integrate Amazon Aurora with other AWS services so that you can extend your Aurora DB cluster to use additional capabilities in the AWS Cloud.

Topics

Integrating AWS services with Amazon Aurora MySQL

Amazon Aurora MySQL integrates with other AWS services so that you can extend your Aurora MySQL DB cluster to use additional capabilities in the AWS Cloud. Your Aurora MySQL DB cluster can use AWS services to do the following:

- Synchronously or asynchronously invoke an AWS Lambda function using the native functions `lambda_sync` or `lambda_async`. Or, asynchronously invoke an AWS Lambda function using the `mysql.lambda_async` procedure.
- Load data from text or XML files stored in an Amazon S3 bucket into your DB cluster using the `LOAD DATA FROM S3` or `LOAD XML FROM S3` command.
- Save data to text files stored in an Amazon S3 bucket from your DB cluster using the `SELECT INTO OUTFILE S3` command.
- Automatically add or remove Aurora Replicas with Application Auto Scaling. For more information, see Using Amazon Aurora Auto Scaling with Aurora Replicas.

For more information about integrating Aurora MySQL with other AWS services, see Integrating Amazon Aurora MySQL with other AWS services.

Integrating AWS services with Amazon Aurora PostgreSQL

Amazon Aurora PostgreSQL integrates with other AWS services so that you can extend your Aurora PostgreSQL DB cluster to use additional capabilities in the AWS Cloud. Your Aurora PostgreSQL DB cluster can use AWS services to do the following:

- Quickly collect, view, and assess performance on your relational database workloads with Performance Insights.
- Automatically add or remove Aurora Replicas with Aurora Auto Scaling. For more information, see Using Amazon Aurora Auto Scaling with Aurora Replicas.

For more information about integrating Aurora PostgreSQL with other AWS services, see Integrating Amazon Aurora PostgreSQL with other AWS services.

Using Amazon Aurora Auto Scaling with Aurora Replicas

To meet your connectivity and workload requirements, Aurora Auto Scaling dynamically adjusts the number of Aurora Replicas (reader DB instances) provisioned for an Aurora DB cluster. Aurora Auto Scaling is available for both Aurora MySQL and Aurora PostgreSQL. Aurora Auto Scaling enables your Aurora DB cluster to handle sudden increases in connectivity or workload. When the connectivity or workload decreases, Aurora Auto Scaling removes unnecessary Aurora Replicas so that you don't pay for unused provisioned DB instances.

You define and apply a scaling policy to an Aurora DB cluster. The *scaling policy* defines the minimum and maximum number of Aurora Replicas that Aurora Auto Scaling can manage. Based on the policy, Aurora Auto Scaling adjusts the number of Aurora Replicas up or down in response to actual workloads, determined by using Amazon CloudWatch metrics and target values.

You can use the AWS Management Console to apply a scaling policy based on a predefined metric. Alternatively, you can use either the AWS CLI or Aurora Auto Scaling API to apply a scaling policy based on a predefined or custom metric.

Topics

Before you begin

Before you can use Aurora Auto Scaling with an Aurora DB cluster, you must first create an Aurora DB cluster with a primary (writer) DB instance. For more information about creating an Aurora DB cluster, see Creating an Amazon Aurora DB cluster.

Aurora Auto Scaling only scales a DB cluster if the DB cluster is in the available state.

When Aurora Auto Scaling adds a new Aurora Replica, the new Aurora Replica is the same DB instance class as the one used by the primary instance. For more information about DB instance classes, see Aurora DB instance classes. Also, the promotion tier for new Aurora Replicas is set to the last priority, which is 15 by default. This means that during a failover, a replica with a better priority, such as one created manually, would be promoted first. For more information, see Fault tolerance for an Aurora DB cluster.

Aurora Auto Scaling only removes Aurora Replicas that it created.

To benefit from Aurora Auto Scaling, your applications must support connections to new Aurora Replicas. To do so, we recommend using the Aurora reader endpoint. You can use a driver such as the AWS JDBC Driver. For more information, see Connecting to an Amazon Aurora DB cluster.

> **Note**
>
> Aurora global databases currently don't support Aurora Auto Scaling for secondary DB clusters.

Aurora Auto Scaling policies

Aurora Auto Scaling uses a scaling policy to adjust the number of Aurora Replicas in an Aurora DB cluster. Aurora Auto Scaling has the following components:

- A service-linked role
- A target metric
- Minimum and maximum capacity
- A cooldown period

Topics

- Service linked role
- Target metric
- Minimum and maximum capacity
- Cooldown period
- Enable or disable scale-in activities

Service linked role

Aurora Auto Scaling uses the `AWSServiceRoleForApplicationAutoScaling_RDSCluster` service-linked role. For more information, see Service-linked roles for Application Auto Scaling in the *Application Auto Scaling User Guide*.

Target metric

In this type of policy, a predefined or custom metric and a target value for the metric is specified in a target-tracking scaling policy configuration. Aurora Auto Scaling creates and manages CloudWatch alarms that trigger the scaling policy and calculates the scaling adjustment based on the metric and target value. The scaling policy adds or removes Aurora Replicas as required to keep the metric at, or close to, the specified target value. In addition to keeping the metric close to the target value, a target-tracking scaling policy also adjusts to fluctuations in the metric due to a changing workload. Such a policy also minimizes rapid fluctuations in the number of available Aurora Replicas for your DB cluster.

For example, take a scaling policy that uses the predefined average CPU utilization metric. Such a policy can keep CPU utilization at, or close to, a specified percentage of utilization, such as 40 percent.

Note

For each Aurora DB cluster, you can create only one Auto Scaling policy for each target metric.

Minimum and maximum capacity

You can specify the maximum number of Aurora Replicas to be managed by Application Auto Scaling. This value must be set to 0–15, and must be equal to or greater than the value specified for the minimum number of Aurora Replicas.

You can also specify the minimum number of Aurora Replicas to be managed by Application Auto Scaling. This value must be set to 0–15, and must be equal to or less than the value specified for the maximum number of Aurora Replicas.

> **Note**
>
> The minimum and maximum capacity are set for an Aurora DB cluster. The specified values apply to all of the policies associated with that Aurora DB cluster.

Cooldown period

You can tune the responsiveness of a target-tracking scaling policy by adding cooldown periods that affect scaling your Aurora DB cluster in and out. A cooldown period blocks subsequent scale-in or scale-out requests until the period expires. These blocks slow the deletions of Aurora Replicas in your Aurora DB cluster for scale-in requests, and the creation of Aurora Replicas for scale-out requests.

You can specify the following cooldown periods:

- A scale-in activity reduces the number of Aurora Replicas in your Aurora DB cluster. A scale-in cooldown period specifies the amount of time, in seconds, after a scale-in activity completes before another scale-in activity can start.
- A scale-out activity increases the number of Aurora Replicas in your Aurora DB cluster. A scale-out cooldown period specifies the amount of time, in seconds, after a scale-out activity completes before another scale-out activity can start.

  > **Note**
  >
  > A scale-out cooldown period is ignored if a subsequent scale-out request is for a larger number of Aurora Replicas than the first request.

If you don't set the scale-in or scale-out cooldown period, the default for each is 300 seconds.

Enable or disable scale-in activities

You can enable or disable scale-in activities for a policy. Enabling scale-in activities allows the scaling policy to delete Aurora Replicas. When scale-in activities are enabled, the scale-in cooldown period in the scaling policy applies to scale-in activities. Disabling scale-in activities prevents the scaling policy from deleting Aurora Replicas.

> **Note**
>
> Scale-out activities are always enabled so that the scaling policy can create Aurora Replicas as needed.

Adding a scaling policy to an Aurora DB cluster

You can add a scaling policy using the AWS Management Console, the AWS CLI, or the Application Auto Scaling API.

> **Note**
>
> For an example that adds a scaling policy using AWS CloudFormation, see Declaring a scaling policy for an Aurora DB cluster in the *AWS CloudFormation User Guide.*

Console

You can add a scaling policy to an Aurora DB cluster by using the AWS Management Console.

To add an auto scaling policy to an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora DB cluster that you want to add a policy for.
4. Choose the **Logs & events** tab.
5. In the **Auto scaling policies** section, choose **Add**.

   The **Add Auto Scaling policy** dialog box appears.
6. For **Policy Name**, type the policy name.
7. For the target metric, choose one of the following:

   - **Average CPU utilization of Aurora Replicas** to create a policy based on the average CPU utilization.

- **Average connections of Aurora Replicas** to create a policy based on the average number of connections to Aurora Replicas.

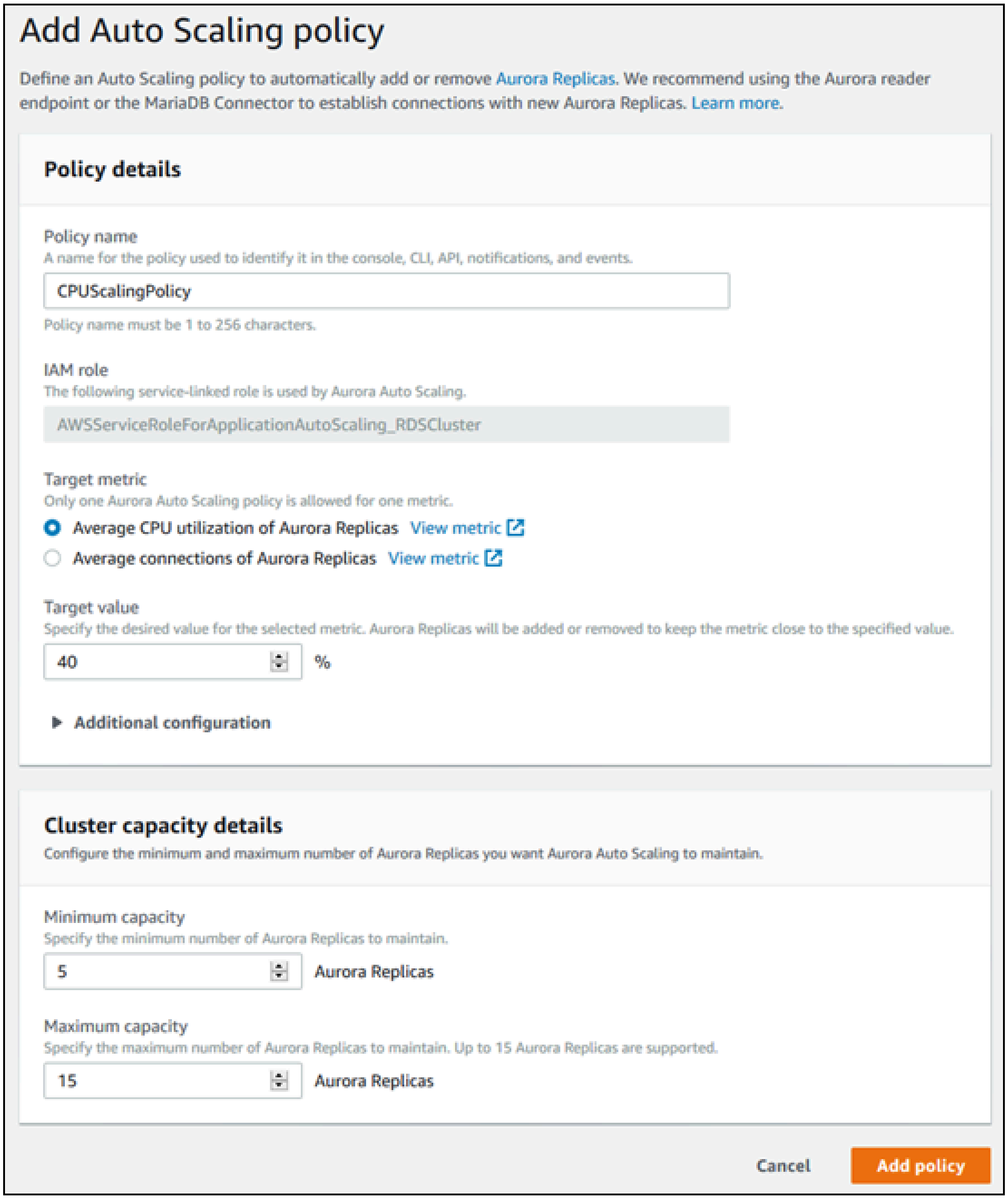
8. For the target value, type one of the following:

   - If you chose **Average CPU utilization of Aurora Replicas** in the previous step, type the percentage of CPU utilization that you want to maintain on Aurora Replicas.
   - If you chose **Average connections of Aurora Replicas** in the previous step, type the number of connections that you want to maintain.

   Aurora Replicas are added or removed to keep the metric close to the specified value.

9. (Optional) Expand **Additional Configuration** to create a scale-in or scale-out cooldown period.
10. For **Minimum capacity**, type the minimum number of Aurora Replicas that the Aurora Auto Scaling policy is required to maintain.
11. For **Maximum capacity**, type the maximum number of Aurora Replicas the Aurora Auto Scaling policy is required to maintain.
12. Choose **Add policy**.

The following dialog box creates an Auto Scaling policy based an average CPU utilization of 40 percent. The policy specifies a minimum of 5 Aurora Replicas and a maximum of 15 Aurora Replicas.



Add Auto Scaling policy

Define an Auto Scaling policy to automatically add or remove Aurora Replicas. We recommend using the Aurora reader endpoint or the MariaDB Connector to establish connections with new Aurora Replicas. Learn more.

Policy details

Policy name
A name for the policy used to identify it in the console, CLI, API, notifications, and events.
CPUScalingPolicy
Policy name must be 1 to 256 characters.

IAM role
The following service-linked role is used by Aurora Auto Scaling.
AWSServiceRoleForApplicationAutoScaling_RDSCluster

Target metric
Only one Aurora Auto Scaling policy is allowed for one metric.
● Average CPU utilization of Aurora Replicas View metric
○ Average connections of Aurora Replicas View metric

Target value
Specify the desired value for the selected metric. Aurora Replicas will be added or removed to keep the metric close to the specified value.
40 %

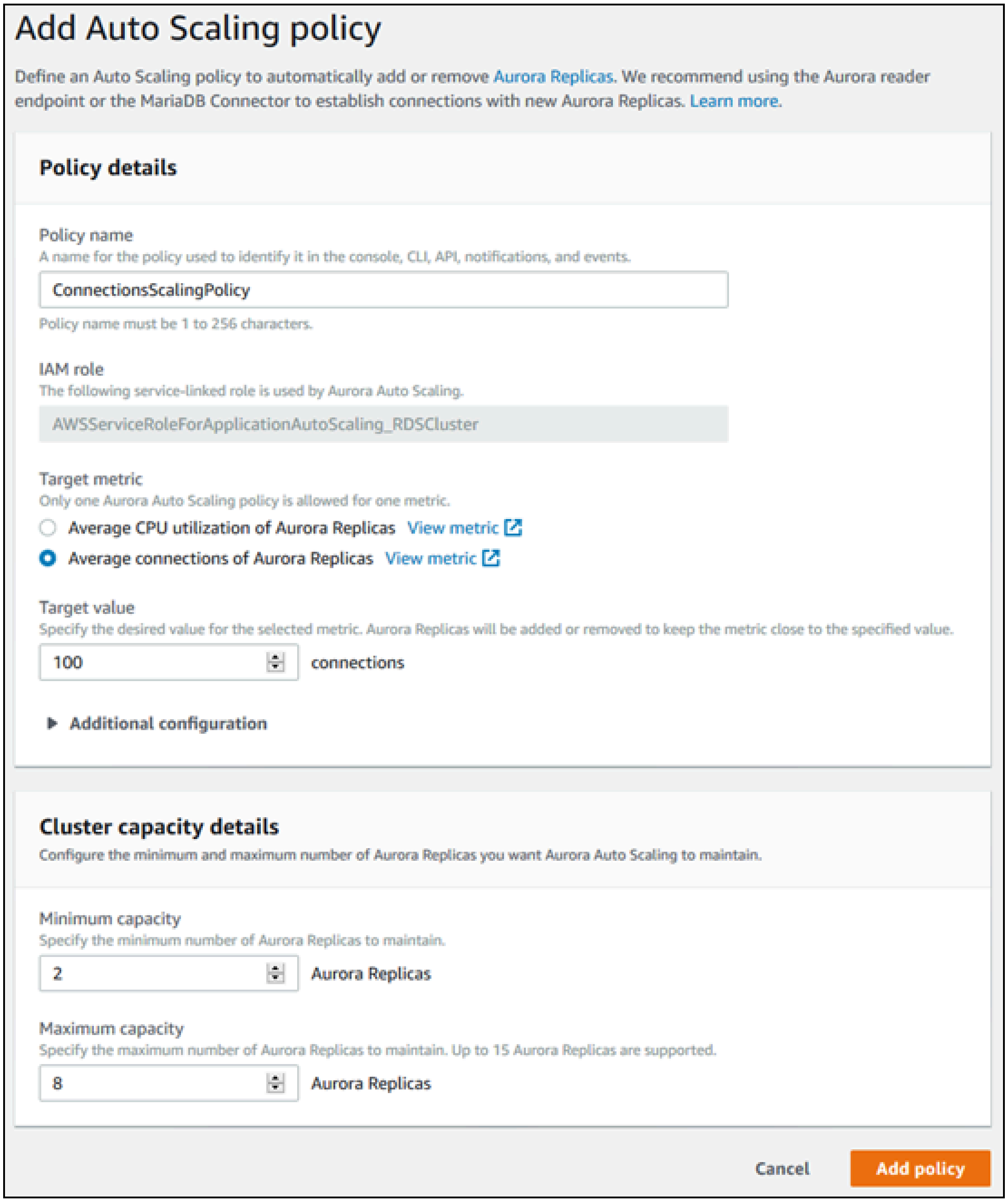
▶ Additional configuration

Cluster capacity details
Configure the minimum and maximum number of Aurora Replicas you want Aurora Auto Scaling to maintain.

Minimum capacity
Specify the minimum number of Aurora Replicas to maintain.
5 Aurora Replicas

Maximum capacity
Specify the maximum number of Aurora Replicas to maintain. Up to 15 Aurora Replicas are supported.
15 Aurora Replicas

The following dialog box creates an auto scaling policy based an average number of connections of 100. The policy specifies a minimum of two Aurora Replicas and a maximum of eight Aurora Replicas.



Add Auto Scaling policy

Define an Auto Scaling policy to automatically add or remove Aurora Replicas. We recommend using the Aurora reader endpoint or the MariaDB Connector to establish connections with new Aurora Replicas. Learn more.

Policy details

Policy name
A name for the policy used to identify it in the console, CLI, API, notifications, and events.
ConnectionsScalingPolicy
Policy name must be 1 to 256 characters.

IAM role
The following service-linked role is used by Aurora Auto Scaling.
AWSServiceRoleForApplicationAutoScaling_RDSCluster

Target metric
Only one Aurora Auto Scaling policy is allowed for one metric.
○ Average CPU utilization of Aurora Replicas View metric
● Average connections of Aurora Replicas View metric

Target value
Specify the desired value for the selected metric. Aurora Replicas will be added or removed to keep the metric close to the specified value.
100 connections

▶ Additional configuration

Cluster capacity details
Configure the minimum and maximum number of Aurora Replicas you want Aurora Auto Scaling to maintain.

Minimum capacity
Specify the minimum number of Aurora Replicas to maintain.
2 Aurora Replicas

Maximum capacity
Specify the maximum number of Aurora Replicas to maintain. Up to 15 Aurora Replicas are supported.
8 Aurora Replicas

AWS CLI or Application Auto Scaling API

You can apply a scaling policy based on either a predefined or custom metric. To do so, you can use the AWS CLI or the Application Auto Scaling API. The first step is to register your Aurora DB cluster with Application Auto Scaling.

Registering an Aurora DB cluster

Before you can use Aurora Auto Scaling with an Aurora DB cluster, you register your Aurora DB cluster with Application Auto Scaling. You do so to define the scaling dimension and limits to be applied to that cluster. Application Auto Scaling dynamically scales the Aurora DB cluster along the `rds:cluster:ReadReplicaCount` scalable dimension, which represents the number of Aurora Replicas.

To register your Aurora DB cluster, you can use either the AWS CLI or the Application Auto Scaling API.

AWS CLI

To register your Aurora DB cluster, use the `register-scalable-target` AWS CLI command with the following parameters:

- `--service-namespace` – Set this value to `rds`.
- `--resource-id` – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is `cluster` and the unique identifier is the name of the Aurora DB cluster, for example `cluster:myscalablecluster`.
- `--scalable-dimension` – Set this value to `rds:cluster:ReadReplicaCount`.
- `--min-capacity` – The minimum number of reader DB instances to be managed by Application Auto Scaling. For information about the relationship between `--min-capacity`, `--max-capacity`, and the number of DB instances in your cluster, see Minimum and maximum capacity.
- `--max-capacity` – The maximum number of reader DB instances to be managed by Application Auto Scaling. For information about the relationship between `--min-capacity`, `--max-capacity`, and the number of DB instances in your cluster, see Minimum and maximum capacity.

Example

In the following example, you register an Aurora DB cluster named `myscalablecluster`. The registration indicates that the DB cluster should be dynamically scaled to have from one to eight Aurora Replicas.

For Linux, macOS, or Unix:

```
aws application-autoscaling register-scalable-target \
    --service-namespace rds \
    --resource-id cluster:myscalablecluster \
    --scalable-dimension rds:cluster:ReadReplicaCount \
    --min-capacity 1 \
    --max-capacity 8 \
```

For Windows:

```
aws application-autoscaling register-scalable-target ^
    --service-namespace rds ^
    --resource-id cluster:myscalablecluster ^
    --scalable-dimension rds:cluster:ReadReplicaCount ^
    --min-capacity 1 ^
    --max-capacity 8 ^
```

Application Auto Scaling API

To register your Aurora DB cluster with Application Auto Scaling, use the `RegisterScalableTarget` Application Auto Scaling API operation with the following parameters:

- `ServiceNamespace` – Set this value to `rds`.
- `ResourceID` – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is `cluster` and the unique identifier is the name of the Aurora DB cluster, for example `cluster:myscalablecluster`.
- `ScalableDimension` – Set this value to `rds:cluster:ReadReplicaCount`.
- `MinCapacity` – The minimum number of reader DB instances to be managed by Application Auto Scaling. For information about the relationship between `MinCapacity`, `MaxCapacity`, and the number of DB instances in your cluster, see Minimum and maximum capacity.
- `MaxCapacity` – The maximum number of reader DB instances to be managed by Application Auto Scaling. For information about the relationship between `MinCapacity`, `MaxCapacity`, and the number of DB instances in your cluster, see Minimum and maximum capacity.

Example

In the following example, you register an Aurora DB cluster named `myscalablecluster` with the Application Auto Scaling API. This registration indicates that the DB cluster should be dynamically scaled to have from one to eight Aurora Replicas.

```
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encoding: identity
Content-Length: 219
X-Amz-Target: AnyScaleFrontendService.RegisterScalableTarget
X-Amz-Date: 20160506T182145Z
User-Agent: aws-cli/1.10.23 Python/2.7.11 Darwin/15.4.0 botocore/1.4.8
Content-Type: application/x-amz-json-1.1
Authorization: AUTHPARAMS

{
    "ServiceNamespace": "rds",
    "ResourceId": "cluster:myscalablecluster",
    "ScalableDimension": "rds:cluster:ReadReplicaCount",
    "MinCapacity": 1,
    "MaxCapacity": 8
}
```

Defining a scaling policy for an Aurora DB cluster

A target-tracking scaling policy configuration is represented by a JSON block that the metrics and target values are defined in. You can save a scaling policy configuration as a JSON block in a text file. You use that text file when invoking the AWS CLI or the Application Auto Scaling API. For more information about policy configuration syntax, see `TargetTrackingScalingPolicyConfiguration` in the *Application Auto Scaling API Reference*.

The following options are available for defining a target-tracking scaling policy configuration.

Topics

Using a predefined metric

By using predefined metrics, you can quickly define a target-tracking scaling policy for an Aurora DB cluster that works well with both target tracking and dynamic scaling in Aurora Auto Scaling.

Currently, Aurora supports the following predefined metrics in Aurora Auto Scaling:

- **RDSReaderAverageCPUUtilization –** The average value of the `CPUUtilization` metric in CloudWatch across all Aurora Replicas in the Aurora DB cluster.
- **RDSReaderAverageDatabaseConnections –** The average value of the `DatabaseConnections` metric in CloudWatch across all Aurora Replicas in the Aurora DB cluster.

For more information about the `CPUUtilization` and `DatabaseConnections` metrics, see Amazon CloudWatch metrics for Amazon Aurora.

To use a predefined metric in your scaling policy, you create a target tracking configuration for your scaling policy. This configuration must include a `PredefinedMetricSpecification` for the predefined metric and a `TargetValue` for the target value of that metric.

Example

The following example describes a typical policy configuration for target-tracking scaling for an Aurora DB cluster. In this configuration, the `RDSReaderAverageCPUUtilization` predefined metric is used to adjust the Aurora DB cluster based on an average CPU utilization of 40 percent across all Aurora Replicas.

```
{
    "TargetValue": 40.0,
    "PredefinedMetricSpecification":
    {
        "PredefinedMetricType": "RDSReaderAverageCPUUtilization"
    }
}
```

Using a custom metric

By using custom metrics, you can define a target-tracking scaling policy that meets your custom requirements. You can define a custom metric based on any Aurora metric that changes in proportion to scaling.

Not all Aurora metrics work for target tracking. The metric must be a valid utilization metric and describe how busy an instance is. The value of the metric must increase or decrease in proportion to the number of Aurora Replicas in the Aurora DB cluster. This proportional increase or decrease is necessary to use the metric data to proportionally scale out or in the number of Aurora Replicas.

Example

The following example describes a target-tracking configuration for a scaling policy. In this configuration, a custom metric adjusts an Aurora DB cluster based on an average CPU utilization of 50 percent across all Aurora Replicas in an Aurora DB cluster named `my-db-cluster`.

```
{
    "TargetValue": 50,
    "CustomizedMetricSpecification":
    {
        "MetricName": "CPUUtilization",
        "Namespace": "AWS/RDS",
        "Dimensions": [
            {"Name": "DBClusterIdentifier","Value": "my-db-cluster"},
            {"Name": "Role","Value": "READER"}
        ],
        "Statistic": "Average",
        "Unit": "Percent"
    }
}
```

Using cooldown periods

You can specify a value, in seconds, for `ScaleOutCooldown` to add a cooldown period for scaling out your Aurora DB cluster. Similarly, you can add a value, in seconds, for `ScaleInCooldown` to add a cooldown period for scaling in your Aurora DB cluster. For more information about `ScaleInCooldown` and `ScaleOutCooldown`, see `TargetTrackingScalingPolicyConfiguration` in the *Application Auto Scaling API Reference*.

Example

The following example describes a target-tracking configuration for a scaling policy. In this configuration, the `RDSReaderAverageCPUUtilization` predefined metric is used to adjust an Aurora DB cluster based on an average CPU utilization of 40 percent across all Aurora Replicas in that Aurora DB cluster. The configuration provides a scale-in cooldown period of 10 minutes and a scale-out cooldown period of 5 minutes.

```
{
    "TargetValue": 40.0,
    "PredefinedMetricSpecification":
    {
        "PredefinedMetricType": "RDSReaderAverageCPUUtilization"
    },
    "ScaleInCooldown": 600,
    "ScaleOutCooldown": 300
}
```

Disabling scale-in activity

You can prevent the target-tracking scaling policy configuration from scaling in your Aurora DB cluster by disabling scale-in activity. Disabling scale-in activity prevents the scaling policy from deleting Aurora Replicas, while still allowing the scaling policy to create them as needed.

You can specify a Boolean value for `DisableScaleIn` to enable or disable scale in activity for your Aurora DB cluster. For more information about `DisableScaleIn`, see [`TargetTrackingScalingPolicyConfiguration`] in the *Application Auto Scaling API Reference*.

Example

The following example describes a target-tracking configuration for a scaling policy. In this configuration, the `RDSReaderAverageCPUUtilization` predefined metric adjusts an Aurora DB cluster based on an average CPU utilization of 40 percent across all Aurora Replicas in that Aurora DB cluster. The configuration disables scale-in activity for the scaling policy.

```
{
    "TargetValue": 40.0,
    "PredefinedMetricSpecification":
    {
        "PredefinedMetricType": "RDSReaderAverageCPUUtilization"
    },
    "DisableScaleIn": true
}
```

Applying a scaling policy to an Aurora DB cluster

After registering your Aurora DB cluster with Application Auto Scaling and defining a scaling policy, you apply the scaling policy to the registered Aurora DB cluster. To apply a scaling policy to an Aurora DB cluster, you can use the AWS CLI or the Application Auto Scaling API.

AWS CLI

To apply a scaling policy to your Aurora DB cluster, use the `put-scaling-policy` AWS CLI command with the following parameters:

- `--policy-name` – The name of the scaling policy.
- `--policy-type` – Set this value to `TargetTrackingScaling`.
- `--resource-id` – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is `cluster` and the unique identifier is the name of the Aurora DB cluster, for example `cluster:myscalablecluster`.
- `--service-namespace` – Set this value to `rds`.
- `--scalable-dimension` – Set this value to `rds:cluster:ReadReplicaCount`.
- `--target-tracking-scaling-policy-configuration` – The target-tracking scaling policy configuration to use for the Aurora DB cluster.

Example

In the following example, you apply a target-tracking scaling policy named `myscalablepolicy` to an Aurora DB cluster named `myscalablecluster` with Application Auto Scaling. To do so, you use a policy configuration saved in a file named `config.json`.

For Linux, macOS, or Unix:

```
aws application-autoscaling put-scaling-policy \
    --policy-name myscalablepolicy \
    --policy-type TargetTrackingScaling \
    --resource-id cluster:myscalablecluster \
    --service-namespace rds \
    --scalable-dimension rds:cluster:ReadReplicaCount \
    --target-tracking-scaling-policy-configuration file://config.json
```

For Windows:

```
aws application-autoscaling put-scaling-policy ^
    --policy-name myscalablepolicy ^
    --policy-type TargetTrackingScaling ^
    --resource-id cluster:myscalablecluster ^
    --service-namespace rds ^
    --scalable-dimension rds:cluster:ReadReplicaCount ^
    --target-tracking-scaling-policy-configuration file://config.json
```

Application Auto Scaling API

To apply a scaling policy to your Aurora DB cluster with the Application Auto Scaling API, use the `PutScalingPolicy` Application Auto Scaling API operation with the following parameters:

- `PolicyName` – The name of the scaling policy.
- `ServiceNamespace` – Set this value to `rds`.
- `ResourceID` – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is `cluster` and the unique identifier is the name of the Aurora DB cluster, for example `cluster:myscalablecluster`.
- `ScalableDimension` – Set this value to `rds:cluster:ReadReplicaCount`.
- `PolicyType` – Set this value to `TargetTrackingScaling`.
- `TargetTrackingScalingPolicyConfiguration` – The target-tracking scaling policy configuration to use for the Aurora DB cluster.

Example

In the following example, you apply a target-tracking scaling policy named `myscalablepolicy` to an Aurora DB cluster named `myscalablecluster` with Application Auto Scaling. You use a policy configuration based on the `RDSReaderAverageCPUUtilization` predefined metric.

```
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encoding: identity
Content-Length: 219
X-Amz-Target: AnyScaleFrontendService.PutScalingPolicy
X-Amz-Date: 20160506T182145Z
User-Agent: aws-cli/1.10.23 Python/2.7.11 Darwin/15.4.0 botocore/1.4.8
Content-Type: application/x-amz-json-1.1
Authorization: AUTHPARAMS

{
    "PolicyName": "myscalablepolicy",
    "ServiceNamespace": "rds",
    "ResourceId": "cluster:myscalablecluster",
    "ScalableDimension": "rds:cluster:ReadReplicaCount",
    "PolicyType": "TargetTrackingScaling",
    "TargetTrackingScalingPolicyConfiguration": {
        "TargetValue": 40.0,
        "PredefinedMetricSpecification":
        {
            "PredefinedMetricType": "RDSReaderAverageCPUUtilization"
        }
    }
}
```

Editing a scaling policy

You can edit a scaling policy using the AWS Management Console, the AWS CLI, or the Application Auto Scaling API.

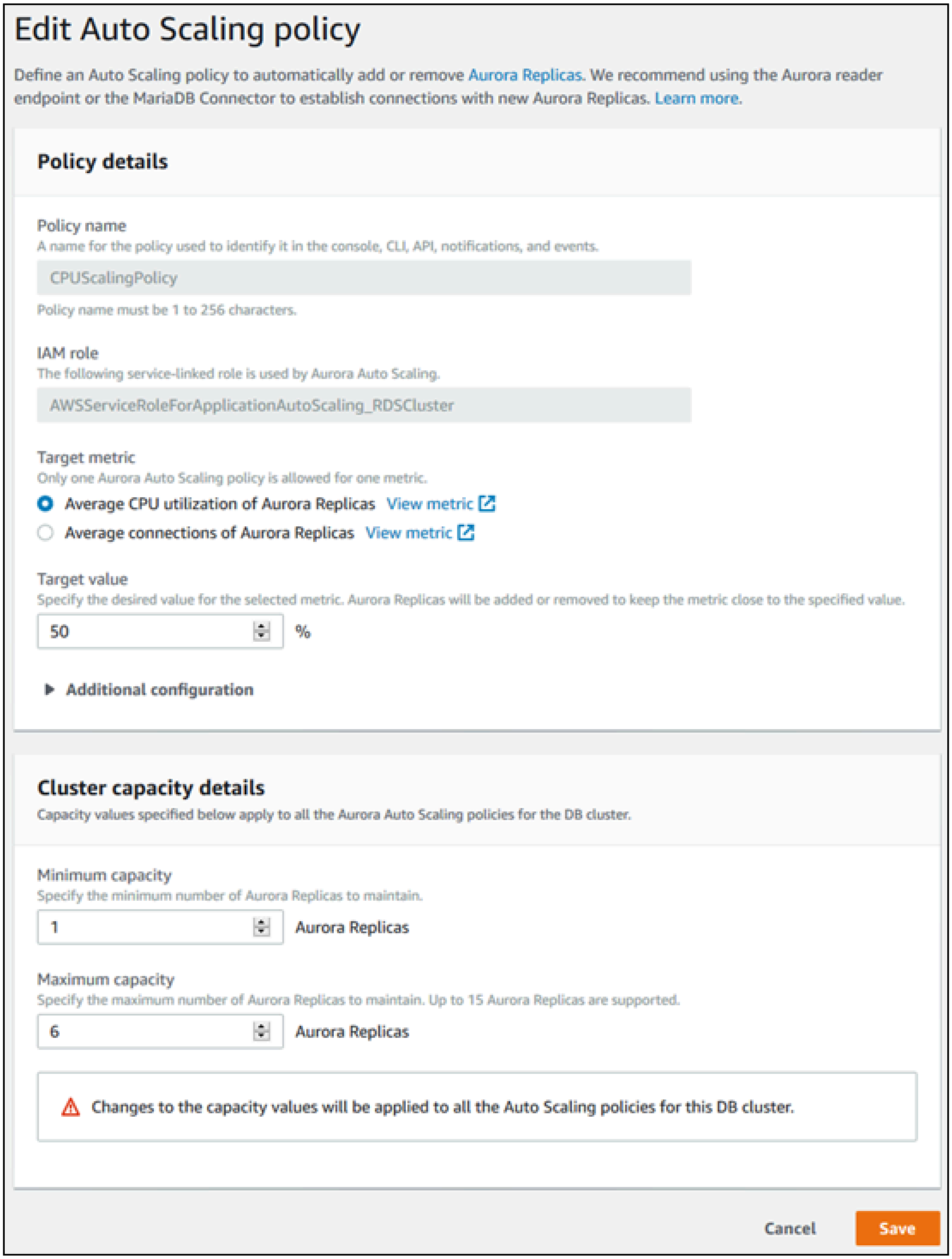
Console

You can edit a scaling policy by using the AWS Management Console.

To edit an auto scaling policy for an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora DB cluster whose auto scaling policy you want to edit.
4. Choose the **Logs & events** tab.
5. In the **Auto scaling policies** section, choose the auto scaling policy, and then choose **Edit**.
6. Make changes to the policy.
7. Choose **Save**.

The following is a sample **Edit Auto Scaling policy** dialog box.



Edit Auto Scaling policy

Define an Auto Scaling policy to automatically add or remove Aurora Replicas. We recommend using the Aurora reader endpoint or the MariaDB Connector to establish connections with new Aurora Replicas. Learn more.

Policy details

Policy name
A name for the policy used to identify it in the console, CLI, API, notifications, and events.
CPUScalingPolicy
Policy name must be 1 to 256 characters.

IAM role
The following service-linked role is used by Aurora Auto Scaling.
AWSServiceRoleForApplicationAutoScaling_RDSCluster

Target metric
Only one Aurora Auto Scaling policy is allowed for one metric.
● Average CPU utilization of Aurora Replicas View metric
○ Average connections of Aurora Replicas View metric

Target value
Specify the desired value for the selected metric. Aurora Replicas will be added or removed to keep the metric close to the specified value.
50 %

▶ Additional configuration

Cluster capacity details
Capacity values specified below apply to all the Aurora Auto Scaling policies for the DB cluster.

Minimum capacity
Specify the minimum number of Aurora Replicas to maintain.
1 Aurora Replicas

Maximum capacity
Specify the maximum number of Aurora Replicas to maintain. Up to 15 Aurora Replicas are supported.
6 Aurora Replicas

⚠ Changes to the capacity values will be applied to all the Auto Scaling policies for this DB cluster.

AWS CLI or Application Auto Scaling API

You can use the AWS CLI or the Application Auto Scaling API to edit a scaling policy in the same way that you apply a scaling policy:

- When using the AWS CLI, specify the name of the policy you want to edit in the `--policy-name` parameter. Specify new values for the parameters you want to change.
- When using the Application Auto Scaling API, specify the name of the policy you want to edit in the `PolicyName` parameter. Specify new values for the parameters you want to change.

For more information, see Applying a scaling policy to an Aurora DB cluster.

Deleting a scaling policy

You can delete a scaling policy using the AWS Management Console, the AWS CLI, or the Application Auto Scaling API.

Console

You can delete a scaling policy by using the AWS Management Console.

To delete an auto scaling policy for an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora DB cluster whose auto scaling policy you want to delete.
4. Choose the **Logs & events** tab.
5. In the **Auto scaling policies** section, choose the auto scaling policy, and then choose **Delete**.

AWS CLI

To delete a scaling policy from your Aurora DB cluster, use the `delete-scaling-policy` AWS CLI command with the following parameters:

- `--policy-name` – The name of the scaling policy.
- `--resource-id` – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is `cluster` and the unique identifier is the name of the Aurora DB cluster, for example `cluster:myscalablecluster`.
- `--service-namespace` – Set this value to `rds`.
- `--scalable-dimension` – Set this value to `rds:cluster:ReadReplicaCount`.

Example

In the following example, you delete a target-tracking scaling policy named myscalablepolicy from an Aurora DB cluster named myscalablecluster.

For Linux, macOS, or Unix:

```
aws application-autoscaling delete-scaling-policy \
    --policy-name myscalablepolicy \
    --resource-id cluster:myscalablecluster \
    --service-namespace rds \
    --scalable-dimension rds:cluster:ReadReplicaCount \
```

For Windows:

```
aws application-autoscaling delete-scaling-policy ^
    --policy-name myscalablepolicy ^
    --resource-id cluster:myscalablecluster ^
    --service-namespace rds ^
    --scalable-dimension rds:cluster:ReadReplicaCount ^
```

Application Auto Scaling API

To delete a scaling policy from your Aurora DB cluster, use the DeleteScalingPolicy the Application Auto Scaling API operation with the following parameters:

- PolicyName – The name of the scaling policy.
- ServiceNamespace – Set this value to rds.
- ResourceID – The resource identifier for the Aurora DB cluster. For this parameter, the resource type is cluster and the unique identifier is the name of the Aurora DB cluster, for example cluster:myscalablecluster.
- ScalableDimension – Set this value to rds:cluster:ReadReplicaCount.

Example

In the following example, you delete a target-tracking scaling policy named `myscalablepolicy` from an Aurora DB cluster named `myscalablecluster` with the Application Auto Scaling API.

```
POST / HTTP/1.1
Host: autoscaling.us-east-2.amazonaws.com
Accept-Encoding: identity
Content-Length: 219
X-Amz-Target: AnyScaleFrontendService.DeleteScalingPolicy
X-Amz-Date: 20160506T182145Z
User-Agent: aws-cli/1.10.23 Python/2.7.11 Darwin/15.4.0 botocore/1.4.8
Content-Type: application/x-amz-json-1.1
Authorization: AUTHPARAMS

{
    "PolicyName": "myscalablepolicy",
    "ServiceNamespace": "rds",
    "ResourceId": "cluster:myscalablecluster",
    "ScalableDimension": "rds:cluster:ReadReplicaCount"
}
```

DB instance IDs and tagging

When a replica is added by Aurora Auto Scaling, its DB instance ID is prefixed by `application-autoscaling-`, for example, `application-autoscaling-61aabbcc-4e2f-4c65-b620-ab7421abc123`.

The following tag is automatically added to the DB instance. You can view it on the **Tags** tab of the DB instance detail page.

Tag	Value
application-autoscaling:resourceId	cluster:mynewcluster-cluster

For more information on Amazon RDS resource tags, see Tagging Amazon RDS resources.

Aurora Auto Scaling and Performance Insights

You can use Performance Insights to monitor replicas that have been added by Aurora Auto Scaling, the same as with any Aurora reader DB instance.

You can't turn on Performance Insights for an Aurora DB cluster. You can manually turn on Performance Insights for each DB instance in the DB cluster.

When you turn on Performance Insights for the writer DB instance in your Aurora DB cluster, Performance Insights isn't turned on automatically for reader DB instances. You have to turn on Performance Insights manually for the existing reader DB instances and new replicas added by Aurora Auto Scaling.

For more information on using Performance Insights to monitor Aurora DB clusters, see Monitoring DB load with Performance Insights on Amazon Aurora.

Maintaining an Amazon Aurora DB cluster

Periodically, Amazon RDS performs maintenance on Amazon RDS resources. Maintenance most often involves updates to the following resources in your DB cluster:

- Underlying hardware
- Underlying operating system (OS)
- Database engine version

Updates to the operating system most often occur for security issues. You should do them as soon as possible.

Some maintenance items require that Amazon RDS take your DB cluster offline for a short time. Maintenance items that require a resource to be offline include required operating system or database patching. Required patching is automatically scheduled only for patches that are related to security and instance reliability. Such patching occurs infrequently, typically once every few months. It seldom requires more than a fraction of your maintenance window.

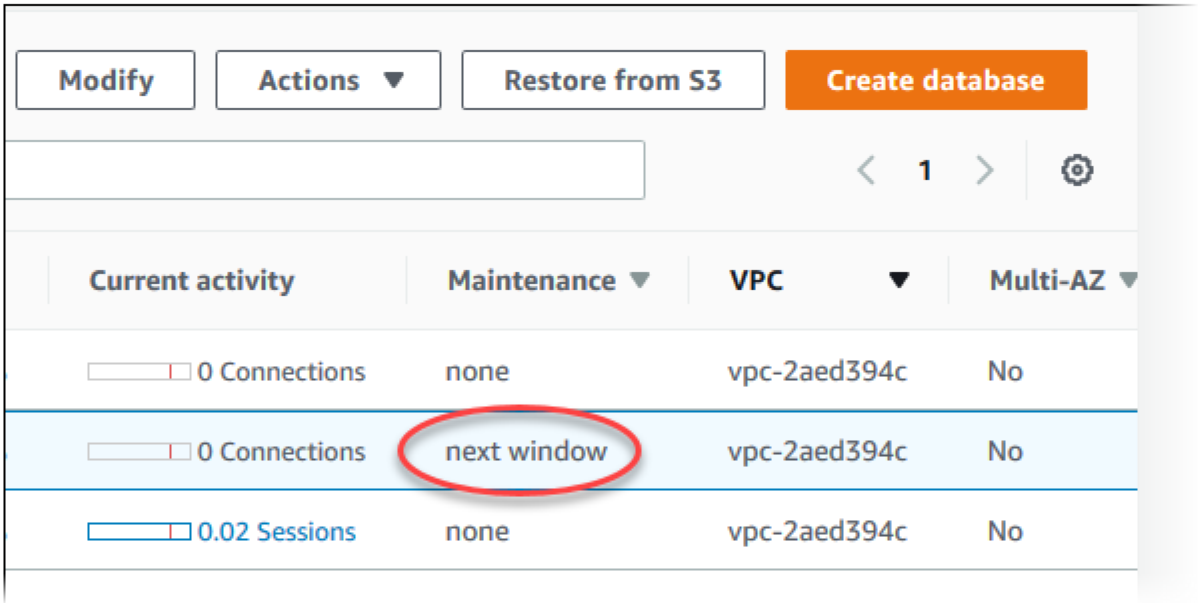
Deferred DB cluster and instance modifications that you have chosen not to apply immediately are also applied during the maintenance window. For example, you might choose to change DB instance classes or cluster or DB parameter groups during the maintenance window. Such modifications that you specify using the **pending reboot** setting don't show up in the **Pending maintenance** list. For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

To see the modifications that are pending for the next maintenance window, use the describe-db-clusters AWS CLI command and check the `PendingModifiedValues` field.

Topics

Viewing pending maintenance

View whether a maintenance update is available for your DB cluster by using the RDS console, the AWS CLI, or the RDS API. If an update is available, it is indicated in the **Maintenance** column for the DB cluster on the Amazon RDS console, as shown following.



Current activity	Maintenance	VPC	Multi-AZ
0 Connections	none	vpc-2aed394c	No
0 Connections	next window	vpc-2aed394c	No
0.02 Sessions	none	vpc-2aed394c	No

If no maintenance update is available for a DB cluster, the column value is **none** for it.

If a maintenance update is available for a DB cluster, the following column values are possible:

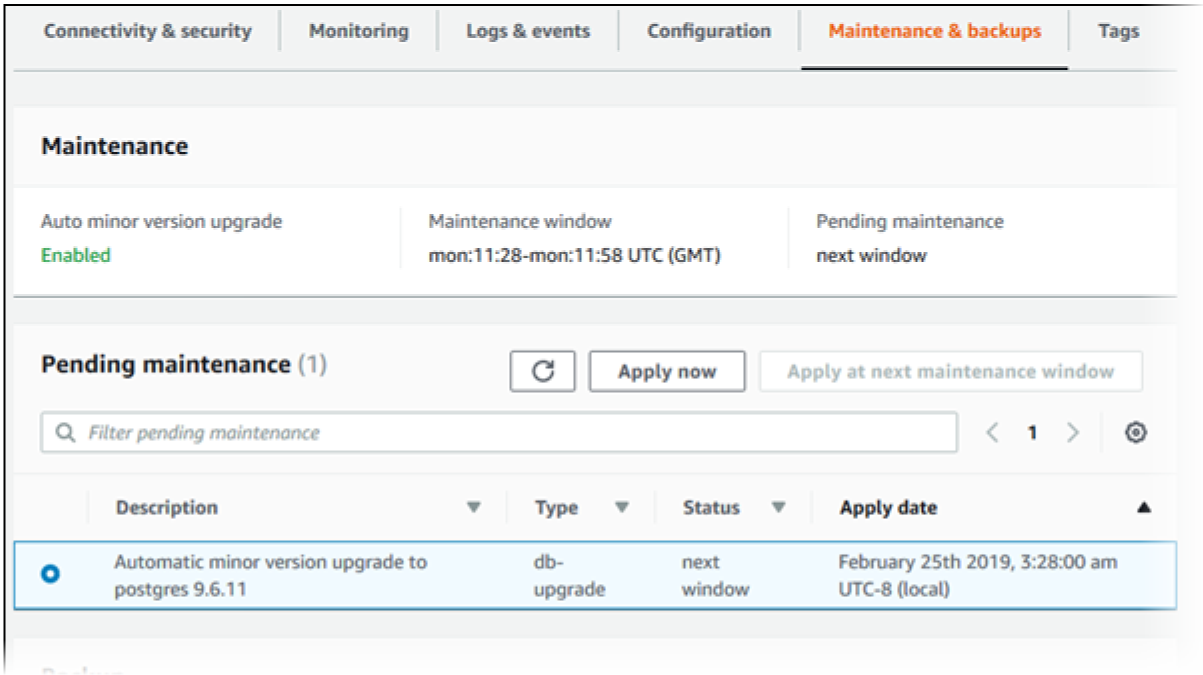
- **required** – The maintenance action will be applied to the resource and can't be deferred indefinitely.
- **available** – The maintenance action is available, but it will not be applied to the resource automatically. You can apply it manually.
- **next window** – The maintenance action will be applied to the resource during the next maintenance window.
- **In progress** – The maintenance action is in the process of being applied to the resource.

If an update is available, you can take one of the actions:

- If the maintenance value is **next window**, defer the maintenance items by choosing **Defer upgrade** from **Actions**. You can't defer a maintenance action if it has already started.
- Apply the maintenance items immediately.
- Schedule the maintenance items to start during your next maintenance window.

- Take no action.

To take an action, choose the DB cluster to show its details, then choose **Maintenance & backups**. The pending maintenance items appear.



Maintenance

Auto minor version upgrade: Enabled

Maintenance window: mon:11:28-mon:11:58 UTC (GMT)

Pending maintenance: next window

Pending maintenance (1)

	Description	Type	Status	Apply date
○	Automatic minor version upgrade to postgres 9.6.11	db-upgrade	next window	February 25th 2019, 3:28:00 am UTC-8 (local)

The maintenance window determines when pending operations start, but doesn't limit the total run time of these operations. Maintenance operations aren't guaranteed to finish before the maintenance window ends, and can continue beyond the specified end time. For more information, see The Amazon RDS maintenance window.

For information about updates to Amazon Aurora engines and instructions for upgrading and patching them, see Database engine updates for Amazon Aurora MySQL and Amazon Aurora PostgreSQL updates.

You can also view whether a maintenance update is available for your DB cluster by running the `describe-pending-maintenance-actions` AWS CLI command.

Applying updates for a DB cluster

With Amazon RDS, you can choose when to apply maintenance operations. You can decide when Amazon RDS applies updates by using the RDS console, AWS Command Line Interface (AWS CLI), or RDS API.

Console

To manage an update for a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster that has a required update.
4. For **Actions**, choose one of the following:

   - **Upgrade now**
   - **Upgrade at next window**

   > **Note**
   >
   > If you choose **Upgrade at next window** and later want to delay the update, you can choose **Defer upgrade**. You can't defer a maintenance action if it has already started. To cancel a maintenance action, modify the DB instance and disable **Auto minor version upgrade**.

AWS CLI

To apply a pending update to a DB cluster, use the apply-pending-maintenance-action AWS CLI command.

Example

For Linux, macOS, or Unix:

```
aws rds apply-pending-maintenance-action \
    --resource-identifier arn:aws:rds:us-west-2:001234567890:db:mysql-db \
    --apply-action system-update \
    --opt-in-type immediate
```

For Windows:

```
aws rds apply-pending-maintenance-action ^
    --resource-identifier arn:aws:rds:us-west-2:001234567890:db:mysql-db ^
    --apply-action system-update ^
    --opt-in-type immediate
```

Note

To defer a maintenance action, specify `undo-opt-in` for `--opt-in-type`. You can't specify `undo-opt-in` for `--opt-in-type` if the maintenance action has already started. To cancel a maintenance action, run the modify-db-instance AWS CLI command and specify `--no-auto-minor-version-upgrade`.

To return a list of resources that have at least one pending update, use the describe-pending-maintenance-actions AWS CLI command.

Example

For Linux, macOS, or Unix:

```
aws rds describe-pending-maintenance-actions \
    --resource-identifier arn:aws:rds:us-west-2:001234567890:db:mysql-db
```

For Windows:

```
aws rds describe-pending-maintenance-actions ^
    --resource-identifier arn:aws:rds:us-west-2:001234567890:db:mysql-db
```

You can also return a list of resources for a DB cluster by specifying the `--filters` parameter of the `describe-pending-maintenance-actions` AWS CLI command. The format for the `--filters` command is `Name=`*`filter-name`*`,Value=`*`resource-id`*`,...`.

The following are the accepted values for the `Name` parameter of a filter:

- `db-instance-id` – Accepts a list of DB instance identifiers or Amazon Resource Names (ARNs). The returned list only includes pending maintenance actions for the DB instances identified by these identifiers or ARNs.
- `db-cluster-id` – Accepts a list of DB cluster identifiers or ARNs for Amazon Aurora. The returned list only includes pending maintenance actions for the DB clusters identified by these identifiers or ARNs.

For example, the following example returns the pending maintenance actions for the `sample-cluster1` and `sample-cluster2` DB clusters.

Example

For Linux, macOS, or Unix:

```
aws rds describe-pending-maintenance-actions \
 --filters Name=db-cluster-id,Values=sample-cluster1,sample-cluster2
```

For Windows:

```
aws rds describe-pending-maintenance-actions ^
 --filters Name=db-cluster-id,Values=sample-cluster1,sample-cluster2
```

RDS API

To apply an update to a DB cluster, call the Amazon RDS API `ApplyPendingMaintenanceAction` operation.

To return a list of resources that have at least one pending update, call the Amazon RDS API `DescribePendingMaintenanceActions` operation.

The Amazon RDS maintenance window

The *maintenance windows* is a weekly time interval during which any system changes are applied. Every DB cluster has a weekly maintenance window. The maintenance window as an opportunity to control when modifications and software patching occur.

RDS consumes some of the resources on your DB cluster while maintenance is being applied. You might observe a minimal effect on performance. For a DB instance, on rare occasions, a Multi-AZ failover might be required for a maintenance update to complete.

If a maintenance event is scheduled for a given week, it's initiated during the 30-minute maintenance window you identify. Most maintenance events also complete during the 30-minute maintenance window, although larger maintenance events may take more than 30 minutes to complete. The maintenance window is paused when the DB cluster is stopped.

The 30-minute maintenance window is selected at random from an 8-hour block of time per region. If you don't specify a maintenance window when you create the DB cluster, RDS assigns a 30-minute maintenance window on a randomly selected day of the week.

Following, you can find the time blocks for each region from which default maintenance windows are assigned.

Region Name	Region	Time Block
US East (Ohio)	us-east-2	03:00–11:00 UTC
US East (N. Virginia)	us-east-1	03:00–11:00 UTC
US West (N. California)	us-west-1	06:00–14:00 UTC
US West (Oregon)	us-west-2	06:00–14:00 UTC
Africa (Cape Town)	af-south-1	03:00–11:00 UTC
Asia Pacific (Hong Kong)	ap-east-1	06:00–14:00 UTC
Asia Pacific (Hyderabad)	ap-south-2	06:30–14:30 UTC
Asia Pacific (Jakarta)	ap-southeast-3	08:00–16:00 UTC
Asia Pacific (Melbourne)	ap-southeast-4	11:00–19:00 UTC
Asia Pacific (Mumbai)	ap-south-1	06:00–14:00 UTC
Asia Pacific (Osaka)	ap-northeast-3	22:00–23:59 UTC
Asia Pacific (Seoul)	ap-northeast-2	13:00–21:00 UTC

Region Name	Region	Time Block
Asia Pacific (Singapore)	ap-southeast-1	14:00–22:00 UTC
Asia Pacific (Sydney)	ap-southeast-2	12:00–20:00 UTC
Asia Pacific (Tokyo)	ap-northeast-1	13:00–21:00 UTC
Canada (Central)	ca-central-1	03:00–11:00 UTC
Canada West (Calgary)	ca-west-1	18:00–02:00 UTC
China (Beijing)	cn-north-1	06:00–14:00 UTC
China (Ningxia)	cn-northwest-1	06:00–14:00 UTC
Europe (Frankfurt)	eu-central-1	21:00–05:00 UTC
Europe (Ireland)	eu-west-1	22:00–06:00 UTC
Europe (London)	eu-west-2	22:00–06:00 UTC
Europe (Milan)	eu-south-1	02:00–10:00 UTC
Europe (Paris)	eu-west-3	23:59–07:29 UTC
Europe (Spain)	eu-south-2	02:00–10:00 UTC
Europe (Stockholm)	eu-north-1	23:00–07:00 UTC
Europe (Zurich)	eu-central-2	02:00–10:00 UTC
Israel (Tel Aviv)	il-central-1	03:00–11:00 UTC
Middle East (Bahrain)	me-south-1	06:00–14:00 UTC
Middle East (UAE)	me-central-1	05:00–13:00 UTC
South America (São Paulo)	sa-east-1	00:00–08:00 UTC

Region Name	Region	Time Block
AWS GovCloud (US-East)	us-gov-east-1	17:00–01:00 UTC
AWS GovCloud (US-West)	us-gov-west-1	06:00–14:00 UTC

Adjusting the preferred DB cluster maintenance window

The Aurora DB cluster maintenance window should fall at the time of lowest usage and thus might need modification from time to time. Your DB cluster is unavailable during this time only if the updates that are being applied require an outage. The outage is for the minimum amount of time required to make the necessary updates.

Console

To adjust the preferred DB cluster maintenance window

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster for which you want to change the maintenance window.
4. Choose **Modify**.
5. In the **Maintenance** section, update the maintenance window.
6. Choose **Continue**.

   On the confirmation page, review your changes.
7. To apply the changes to the maintenance window immediately, choose **Immediately** in the **Schedule of modifications** section.
8. Choose **Modify cluster** to save your changes.

   Alternatively, choose **Back** to edit your changes, or choose **Cancel** to cancel your changes.

AWS CLI

To adjust the preferred DB cluster maintenance window, use the AWS CLI `modify-db-cluster` command with the following parameters:

- `--db-cluster-identifier`
- `--preferred-maintenance-window`

Example

The following code example sets the maintenance window to Tuesdays from 4:00–4:30 AM UTC.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
--db-cluster-identifier my-cluster \
--preferred-maintenance-window Tue:04:00-Tue:04:30
```

For Windows:

```
aws rds modify-db-cluster ^
--db-cluster-identifier my-cluster ^
--preferred-maintenance-window Tue:04:00-Tue:04:30
```

RDS API

To adjust the preferred DB cluster maintenance window, use the Amazon RDS `ModifyDBCluster` API operation with the following parameters:

- `DBClusterIdentifier`
- `PreferredMaintenanceWindow`

Automatic minor version upgrades for Aurora DB clusters

The **Auto minor version upgrade** setting specifies whether Aurora automatically applies upgrades to your DB cluster. These upgrades include new minor versions containing additional features and patches containing bug fixes.

This setting is turned on by default. For each new DB cluster, choose the appropriate value for this setting. This value is based on its importance, expected lifetime, and the amount of verification testing that you do after each upgrade.

For instructions on turning the **Auto minor version upgrade** setting on or off, see the following:

- Enabling automatic minor version upgrades for an Aurora DB cluster
- Enabling automatic minor version upgrades for individual DB instances in an Aurora DB cluster

> ⚠ **Important**
>
> We strongly recommend that for new and existing DB clusters, you apply this setting to the DB cluster and not to the DB instances in the cluster individually. If any DB instance in your cluster has this setting turned off, the DB cluster isn't automatically upgraded.

The following table shows how the **Auto minor version upgrade** setting works when applied at the cluster and instance levels.

Action	Cluster setting	Instance settings	Cluster upgraded automatically?
You set it to True on the DB cluster.	True	True for all new and existing instances	Yes
You set it to False on the DB cluster.	False	False for all new and existing instances	No
It was set previousl y to True on the DB cluster. You set it to False on at least one DB instance.	Changes to False	False for one or more instances	No

Action	Cluster setting	Instance settings	Cluster upgraded automatically?
It was set previousl y to False on the DB cluster. You set it to True on at least one DB instance, but not all instances.	False	True for one or more instances, but not all instances	No
It was set previousl y to False on the DB cluster. You set it to True on all DB instances.	Changes to True	True for all instances	Yes

Automatic minor version upgrades are communicated in advance through an Amazon RDS DB cluster event with a category of `maintenance` and ID of `RDS-EVENT-0156`. For more information, see Amazon RDS event categories and event messages for Aurora.

Automatic upgrades occur during the maintenance window. If the individual DB instances in the DB cluster have different maintenance windows from the cluster maintenance window, then the cluster maintenance window takes precedence.

For more information about engine updates for Aurora PostgreSQL, see Amazon Aurora PostgreSQL updates.

For more information about the **Auto minor version upgrade** setting for Aurora MySQL, see Enabling automatic upgrades between minor Aurora MySQL versions. For general information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.

Enabling automatic minor version upgrades for an Aurora DB cluster

Follow the general procedure in Modifying the DB cluster by using the console, CLI, and API.

Console

On the **Modify DB cluster** page, in the **Maintenance** section, select the **Enable auto minor version upgrade** check box.

AWS CLI

Call the modify-db-cluster AWS CLI command. Specify the name of your DB cluster for the `--db-cluster-identifier` option and `true` for the `--auto-minor-version-upgrade` option. Optionally, specify the `--apply-immediately` option to immediately enable this setting for your DB cluster.

RDS API

Call the ModifyDBCluster API operation and specify the name of your DB cluster for the `DBClusterIdentifier` parameter and `true` for the `AutoMinorVersionUpgrade` parameter. Optionally, set the `ApplyImmediately` parameter to `true` to immediately enable this setting for your DB cluster.

Enabling automatic minor version upgrades for individual DB instances in an Aurora DB cluster

Follow the general procedure in Modifying a DB instance in a DB cluster.

Console

On the **Modify DB instance** page, in the **Maintenance** section, select the **Enable auto minor version upgrade** check box.

AWS CLI

Call the modify-db-instance AWS CLI command. Specify the name of your DB instance for the `--db-instance-identifier` option and `true` for the `--auto-minor-version-upgrade` option. Optionally, specify the `--apply-immediately` option to immediately enable this setting for your DB instance. Run a separate `modify-db-instance` command for each DB instance in the cluster.

RDS API

Call the ModifyDBInstance API operation and specify the name of your DB cluster for the `DBInstanceIdentifier` parameter and `true` for the `AutoMinorVersionUpgrade` parameter. Optionally, set the `ApplyImmediately` parameter to `true` to immediately enable this setting for your DB instance. Call a separate `ModifyDBInstance` operation for each DB instance in the cluster.

You can use a CLI command such as the following to check the status of the `AutoMinorVersionUpgrade` setting for all of the DB instances in your Aurora MySQL clusters.

```
aws rds describe-db-instances \
  --query '*[].
{DBClusterIdentifier:DBClusterIdentifier,DBInstanceIdentifier:DBInstanceIdentifier,AutoMinorVer
```

That command produces output similar to the following:

```
[
  {
      "DBInstanceIdentifier": "db-writer-instance",
      "DBClusterIdentifier": "my-db-cluster-57",
      "AutoMinorVersionUpgrade": true
  },
  {
      "DBInstanceIdentifier": "db-reader-instance1",
      "DBClusterIdentifier": "my-db-cluster-57",
      "AutoMinorVersionUpgrade": false
  },
  {
      "DBInstanceIdentifier": "db-writer-instance2",
      "DBClusterIdentifier": "my-db-cluster-80",
      "AutoMinorVersionUpgrade": true
  },
... output omitted ...
```

In this example, **Enable auto minor version upgrade** is turned off for the DB cluster `my-db-cluster-57`, because it's turned off for one of the DB instances in the cluster.

Choosing the frequency of Aurora MySQL maintenance updates

You can control whether Aurora MySQL upgrades happen frequently or rarely for each DB cluster. The best choice depends on your usage of Aurora MySQL and the priorities for your applications that run on Aurora. For information about the Aurora MySQL long-term stability (LTS) releases that require less frequent upgrades, see Aurora MySQL long-term support (LTS) releases.

You might choose to upgrade an Aurora MySQL cluster rarely if some or all of the following conditions apply:

- Your testing cycle for your application takes a long time for each update to the Aurora MySQL database engine.
- You have many DB clusters or many applications all running on the same Aurora MySQL version. You prefer to upgrade all of your DB clusters and associated applications at the same time.
- You use both Aurora MySQL and RDS for MySQL. You prefer to keep the Aurora MySQL clusters and RDS for MySQL DB instances compatible with the same level of MySQL.
- Your Aurora MySQL application is in production or is otherwise business-critical. You can't afford downtime for upgrades outside of rare occurrences for critical patches.
- Your Aurora MySQL application isn't limited by performance issues or feature gaps that are addressed in subsequent Aurora MySQL versions.

If the preceding factors apply to your situation, you can limit the number of forced upgrades for an Aurora MySQL DB cluster. You do so by choosing a specific Aurora MySQL version known as the "Long-Term Support" (LTS) version when you create or upgrade that DB cluster. Doing so minimizes the number of upgrade cycles, testing cycles, and upgrade-related outages for that DB cluster.

You might choose to upgrade an Aurora MySQL cluster frequently if some or all of the following conditions apply:

- The testing cycle for your application is straightforward and brief.
- Your application is still in the development stage.
- Your database environment uses a variety of Aurora MySQL versions, or Aurora MySQL and RDS for MySQL versions. Each Aurora MySQL cluster has its own upgrade cycle.
- You are waiting for specific performance or feature improvements before you increase your usage of Aurora MySQL.

If the preceding factors apply to your situation, you can enable Aurora to apply important upgrades more frequently. To do so, upgrade an Aurora MySQL DB cluster to a more recent Aurora MySQL version than the LTS version. Doing so makes the latest performance enhancements, bug fixes, and features available to you more quickly.

Working with operating system updates

DB instances in Aurora MySQL and Aurora PostgreSQL DB clusters occasionally require operating system updates. Amazon RDS upgrades the operating system to a newer version to improve database performance and customers' overall security posture. Typically, the updates take about 10 minutes. Operating system updates don't change the DB engine version or DB instance class of a DB instance.

We recommend that you update the reader DB instances in a DB cluster first, then the writer DB instance. We don't recommend updating reader and writer instances at the same time, because you might incur downtime in the event of a failover.

We recommend that you use the AWS drivers to achieve faster database failover. For more information, see Connecting to Aurora DB clusters with the AWS drivers.

There are two types of operating system updates, differentiated by the description visible in the pending maintenance action on the DB instance:

- **Operating system distribution upgrade** – Used to migrate to the latest supported major version of Amazon Linux. Its description in the pending maintenance action is `New Operating System upgrade is available`.
- **Operating system patch** – Used to apply various security fixes and sometimes to improve database performance. Its description in the pending maintenance action is `New Operating System patch is available`.

Operating system updates can be either optional or mandatory:

- An **optional update** can be applied at any time. While these updates are optional, we recommend that you apply them periodically to keep your RDS fleet up to date. RDS *does not* apply these updates automatically.

  To be notified when a new, optional operating system patch becomes available, you can subscribe to RDS-EVENT-0230 in the security patching event category. For information about subscribing to RDS events, see Subscribing to Amazon RDS event notification.

  > **Note**
  >
  > `RDS-EVENT-0230` doesn't apply to operating system distribution upgrades.

- A **mandatory update** is required, and we send a notification before the mandatory update. The notification might contain a due date. Plan to schedule your update before this due date. After the specified due date, Amazon RDS automatically upgrades the operating system for your DB instance to the latest version during one of your assigned maintenance windows.

  Operating system distribution upgrades are mandatory.

> **Note**
>
> Staying current on all optional and mandatory updates might be required to meet various compliance obligations. We recommend that you apply all updates made available by RDS routinely during your maintenance windows.

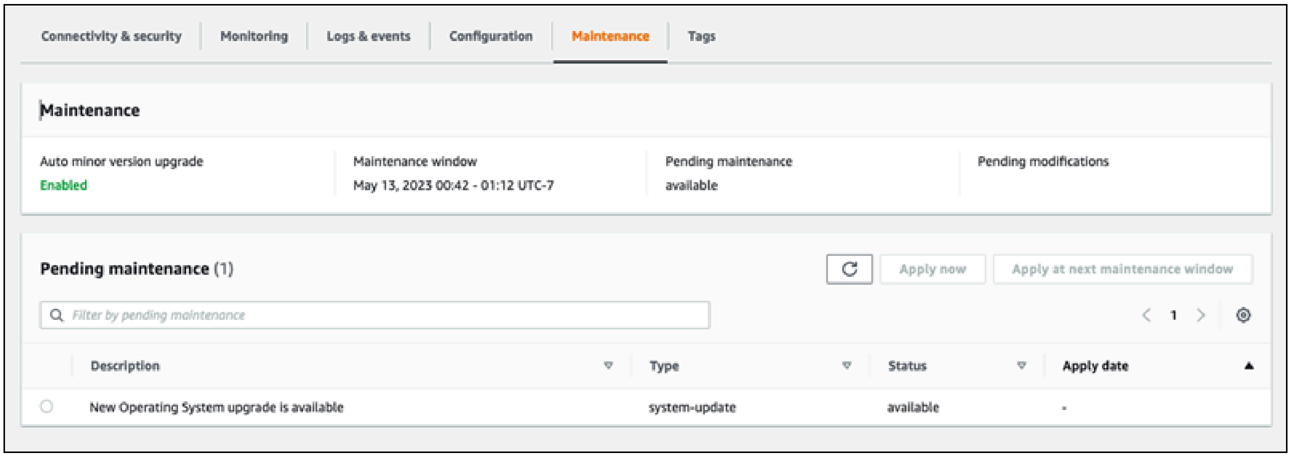
You can use the AWS Management Console or the AWS CLI to get information about the type of operating system upgrade.

Console

To get update information using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then select the DB instance.
3. Choose **Maintenance**.
4. In the **Pending maintenance** section, find the operating system update, and check the **Description** value.

In the AWS Management Console, an operating system distribution upgrade has its **Description** set to **New Operating System upgrade is available**, as shown in the following image. This upgrade is mandatory.



Maintenance

Auto minor version upgrade
Enabled

Maintenance window
May 13, 2023 00:42 - 01:12 UTC-7

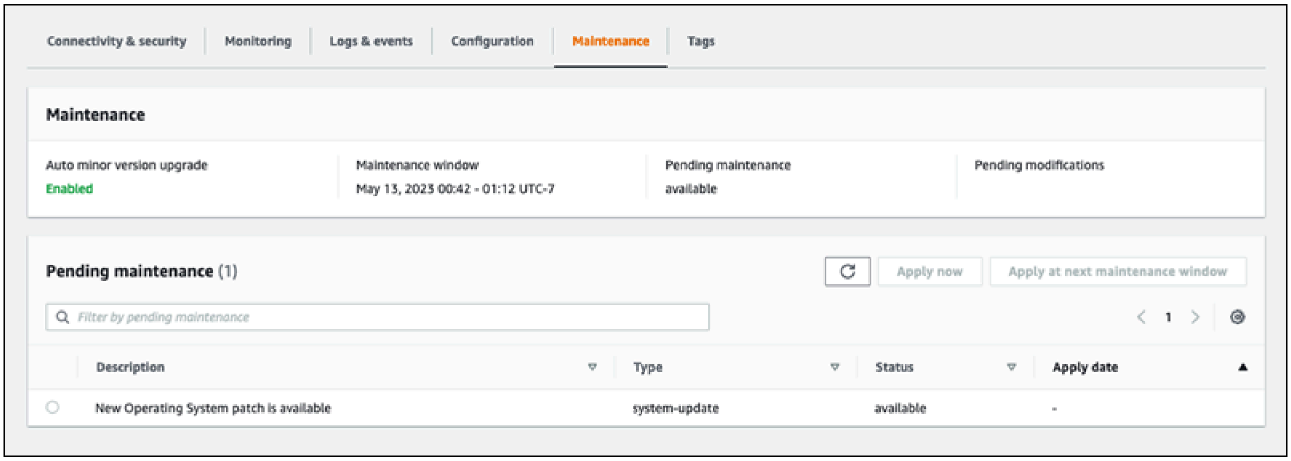
Pending maintenance
available

Pending modifications

Pending maintenance (1)

	Description	Type	Status	Apply date
○	New Operating System upgrade is available	system-update	available	-

An operating system patch has its **Description** set to **New Operating System patch is available**, as shown in the following image.



Maintenance

Auto minor version upgrade
Enabled

Maintenance window
May 13, 2023 00:42 - 01:12 UTC-7

Pending maintenance
available

Pending modifications

Pending maintenance (1)

	Description	Type	Status	Apply date
○	New Operating System patch is available	system-update	available	-

AWS CLI

To get update information from the AWS CLI, use the describe-pending-maintenance-actions command.

```
aws rds describe-pending-maintenance-actions
```

The following output shows an operating system distribution upgrade.

```
{
  "ResourceIdentifier": "arn:aws:rds:us-east-1:123456789012:db:mydb1",
  "PendingMaintenanceActionDetails": [
    {
      "Action": "system-update",
      "Description": "New Operating System upgrade is available"
    }
  ]
}
```

The following output shows an operating system patch.

```
{
  "ResourceIdentifier": "arn:aws:rds:us-east-1:123456789012:db:mydb2",
  "PendingMaintenanceActionDetails": [
    {
      "Action": "system-update",
      "Description": "New Operating System patch is available"
    }
  ]
}
```

Availability of operating system updates

Operating system updates are specific to DB engine version and DB instance class. Therefore, DB instances receive or require updates at different times. When an operating system update is available for your DB instance based on its engine version and instance class, the update appears in the console. It can also be viewed by running AWS CLI describe-pending-maintenance-actions command or by calling the RDS DescribePendingMaintenanceActions API operation. If an update is available for your instance, you can update your operating system by following the instructions in Applying updates for a DB cluster.

Rebooting an Amazon Aurora DB cluster or Amazon Aurora DB instance

You might need to reboot your DB cluster or some instances within the cluster, usually for maintenance reasons. For example, suppose that you modify the parameters within a parameter group or associate a different parameter group with your cluster. In these cases, you must reboot the cluster for the changes to take effect. Similarly, you might reboot one or more reader DB instances within the cluster. You can arrange the reboot operations for individual instances to minimize downtime for the entire cluster.

The time required to reboot each DB instance in your cluster depends on the database activity at the time of reboot. It also depends on the recovery process of your specific DB engine. If it's practical, reduce database activity on that particular instance before starting the reboot process. Doing so can reduce the time needed to restart the database.

You can only reboot each DB instance in your cluster when it's in the available state. A DB instance can be unavailable for several reasons. These include the cluster being stopped state, a modification being applied to the instance, and a maintenance-window action such as a version upgrade.

Rebooting a DB instance restarts the database engine process. Rebooting a DB instance results in a momentary outage, during which the DB instance status is set to *rebooting*.

> **Note**
>
> If a DB instance isn't using the latest changes to its associated DB parameter group, the AWS Management Console shows the DB parameter group with a status of **pending-reboot**. The **pending-reboot** parameter groups status doesn't result in an automatic reboot during the next maintenance window. To apply the latest parameter changes to that DB instance, manually reboot the DB instance. For more information about parameter groups, see Working with parameter groups.

Topics

- Checking uptime for Aurora clusters and instances
- Examples of Aurora reboot operations

Rebooting a DB instance within an Aurora cluster

This procedure is the most important operation that you take when performing reboots with Aurora. Many of the maintenance procedures involve rebooting one or more Aurora DB instances in a particular order.

Console

To reboot a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB instance that you want to reboot.
3. For **Actions**, choose **Reboot**.

   The **Reboot DB Instance** page appears.
4. Choose **Reboot** to reboot your DB instance.

   Or choose **Cancel**.

AWS CLI

To reboot a DB instance by using the AWS CLI, call the `reboot-db-instance` command.

Example

For Linux, macOS, or Unix:

```
aws rds reboot-db-instance \
    --db-instance-identifier mydbinstance
```

For Windows:

```
aws rds reboot-db-instance ^
```

```
    --db-instance-identifier mydbinstance
```

RDS API

To reboot a DB instance by using the Amazon RDS API, call the `RebootDBInstance` operation.

Rebooting an Aurora cluster with read availability

With the read availability feature, you can reboot the writer instance of your Aurora cluster without rebooting the reader instances in the primary or secondary DB cluster. Doing so can help maintain high availability of the cluster for read operations while you reboot the writer instance. You can reboot the reader instances later, on a schedule that's convenient for you. For example, in a production cluster you might reboot the reader instances one at a time, starting only after the reboot of the primary instance is finished. For each DB instance that you reboot, follow the procedure in Rebooting a DB instance within an Aurora cluster.

The read availability feature for primary DB clusters is available in Aurora MySQL version 2.10 and higher. Read availability for secondary DB clusters is available in Aurora MySQL version 3.06 and higher.

For Aurora PostgreSQL this feature is available by default in the following versions:

- 15.2 and higher 15 versions
- 14.7 and higher 14 versions
- 13.10 and higher 13 versions
- 12.14 and higher 12 versions

For more information on the read availability feature in Aurora PostgreSQL, see Improving the read availability of Aurora Replicas.

Before this feature, rebooting the primary instance caused a reboot for each reader instance at the same time. If your Aurora cluster is running an older version, use the reboot procedure in Rebooting an Aurora cluster without read availability instead.

> **Note**
>
> The change to reboot behavior in Aurora DB clusters with read availability is different for Aurora global databases in Aurora MySQL versions lower than 3.06. If you reboot the writer

instance for the primary cluster in an Aurora global database, the reader instances in the primary cluster remain available. However, the DB instances in any secondary clusters reboot at the same time.
A limited version of the improved read availability feature is supported by Aurora global databases for Aurora PostgreSQL versions 12.16, 13.12, 14.9, 15.4, and higher.

You frequently reboot the cluster after making changes to cluster parameter groups. You make parameter changes by following the procedures in Working with parameter groups. Suppose that you reboot the writer DB instance in an Aurora cluster to apply changes to cluster parameters. Some or all of the reader DB instances might continue using the old parameter settings. However, the different parameter settings don't affect the data integrity of the cluster. Any cluster parameters that affect the organization of data files are only used by the writer DB instance.

For example, in an Aurora MySQL cluster, you can update cluster parameters such as `binlog_format` and `innodb_purge_threads` on the writer instance before the reader instances. Only the writer instance is writing binary logs and purging undo records. For parameters that change how queries interpret SQL statements or query output, you might need to take care to reboot the reader instances immediately. You do this to avoid unexpected application behavior during queries. For example, suppose that you change the `lower_case_table_names` parameter and reboot the writer instance. In this case, the reader instances might not be able to access a newly created table until they are all rebooted.

For a list of all the Aurora MySQL cluster parameters, see Cluster-level parameters.

For a list of all the Aurora PostgreSQL cluster parameters, see Aurora PostgreSQL cluster-level parameters.

Tip

Aurora MySQL might still reboot some of the reader instances along with the writer instance if your cluster is processing a workload with high throughput.
The reduction in the number of reboots applies during failover operations also. Aurora MySQL only restarts the writer DB instance and the failover target during a failover. Other reader DB instances in the cluster remain available to continue processing queries through connections to the reader endpoint. Thus, you can improve availability during a failover by having more than one reader DB instance in a cluster.

Rebooting an Aurora cluster without read availability

Without the read availability feature, you reboot an entire Aurora DB cluster by rebooting the writer DB instance of that cluster. To do so, follow the procedure in Rebooting a DB instance within an Aurora cluster.

Rebooting the writer DB instance also initiates a reboot for each reader DB instance in the cluster. That way, any cluster-wide parameter changes are applied to all DB instances at the same time. However, the reboot of all DB instances causes a brief outage for the cluster. The reader DB instances remain unavailable until the writer DB instance finishes rebooting and becomes available.

This reboot behavior applies to all DB clusters created in Aurora MySQL version 2.09 and lower.

For Aurora PostgreSQL this behavior applies to the following versions:

- 14.6 and lower 14 versions
- 13.9 and lower 13 versions
- 12.13 and lower 12 versions
- All PostgreSQL 11 versions

In the RDS console, the writer DB instance has the value **Writer** under the **Role** column on the **Databases** page. In the RDS CLI, the output of the `describe-db-clusters` command includes a section `DBClusterMembers`. The `DBClusterMembers` element representing the writer DB instance has a value of `true` for the `IsClusterWriter` field.

⚠ **Important**

With the read availability feature, the reboot behavior is different in Aurora MySQL and Aurora PostgreSQL: the reader DB instances typically remain available while you reboot the writer instance. Then you can reboot the reader instances at a convenient time. You can reboot the reader instances on a staggered schedule if you want some reader instances to always be available. For more information, see Rebooting an Aurora cluster with read availability.

Checking uptime for Aurora clusters and instances

You can check and monitor the length of time since the last reboot for each DB instance in your Aurora cluster. The Amazon CloudWatch metric `EngineUptime` reports the number of seconds since the last time a DB instance was started. You can examine this metric at a point in time to find out the uptime for the DB instance. You can also monitor this metric over time to detect when the instance is rebooted.

You can also examine the `EngineUptime` metric at the cluster level. The `Minimum` and `Maximum` dimensions report the smallest and largest uptime values for all DB instances in the cluster. To check the most recent time when any reader instance in a cluster was rebooted, or restarted for another reason, monitor the cluster-level metric using the `Minimum` dimension. To check which instance in the cluster has gone the longest without a reboot, monitor the cluster-level metric using the `Maximum` dimension. For example, you might want to confirm that all DB instances in the cluster were rebooted after a configuration change.

Tip

For long-term monitoring, we recommend monitoring the `EngineUptime` metric for individual instances instead of at the cluster level. The cluster-level `EngineUptime` metric is set to zero when a new DB instance is added to the cluster. Such cluster changes can happen as part of maintenance and scaling operations such as those performed by Auto Scaling.

The following CLI examples show how to examine the `EngineUptime` metric for the writer and reader instances in a cluster. The examples use a cluster named `tpch100g`. This cluster has a writer DB instance `instance-1234`. It also has two reader DB instances, `instance-7448` and `instance-6305`.

First, the `reboot-db-instance` command reboots one of the reader instances. The `wait` command waits until the instance is finished rebooting.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-6305
{
    "DBInstance": {
        "DBInstanceIdentifier": "instance-6305",
        "DBInstanceStatus": "rebooting",
...
```

```
$ aws rds wait db-instance-available --db-instance-id instance-6305
```

The CloudWatch `get-metric-statistics` command examines the `EngineUptime` metric over the last five minutes at one-minute intervals. The uptime for the `instance-6305` instance is reset to zero and begins counting upwards again. This AWS CLI example for Linux uses `$()` variable substitution to insert the appropriate timestamps into the CLI commands. It also uses the Linux `sort` command to order the output by the time the metric was collected. That timestamp value is the third field in each line of output.

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" \
  --period 60 --namespace "AWS/RDS" --statistics Maximum \
  --dimensions Name=DBInstanceIdentifier,Value=instance-6305 --output text \
  | sort -k 3
EngineUptime
DATAPOINTS 231.0 2021-03-16T18:19:00+00:00 Seconds
DATAPOINTS 291.0 2021-03-16T18:20:00+00:00 Seconds
DATAPOINTS 351.0 2021-03-16T18:21:00+00:00 Seconds
DATAPOINTS 411.0 2021-03-16T18:22:00+00:00 Seconds
DATAPOINTS 471.0 2021-03-16T18:23:00+00:00 Seconds
```

The minimum uptime for the cluster is reset to zero because one of the instances in the cluster was rebooted. The maximum uptime for the cluster isn't reset because at least one of the DB instances in the cluster remained available.

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" \
  --period 60 --namespace "AWS/RDS" --statistics Minimum \
  --dimensions Name=DBClusterIdentifier,Value=tpch100g --output text \
  | sort -k 3
EngineUptime
DATAPOINTS 63099.0 2021-03-16T18:12:00+00:00 Seconds
DATAPOINTS 63159.0 2021-03-16T18:13:00+00:00 Seconds
DATAPOINTS 63219.0 2021-03-16T18:14:00+00:00 Seconds
DATAPOINTS 63279.0 2021-03-16T18:15:00+00:00 Seconds
DATAPOINTS 51.0 2021-03-16T18:16:00+00:00 Seconds

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" \
  --period 60 --namespace "AWS/RDS" --statistics Maximum \
  --dimensions Name=DBClusterIdentifier,Value=tpch100g --output text \
  | sort -k 3
EngineUptime
DATAPOINTS 63389.0 2021-03-16T18:16:00+00:00 Seconds
DATAPOINTS 63449.0 2021-03-16T18:17:00+00:00 Seconds
DATAPOINTS 63509.0 2021-03-16T18:18:00+00:00 Seconds
DATAPOINTS 63569.0 2021-03-16T18:19:00+00:00 Seconds
DATAPOINTS 63629.0 2021-03-16T18:20:00+00:00 Seconds
```

Then another `reboot-db-instance` command reboots the writer instance of the cluster. Another `wait` command pauses until the writer instance is finished rebooting.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-1234
{
  "DBInstanceIdentifier": "instance-1234",
  "DBInstanceStatus": "rebooting",
...
$ aws rds wait db-instance-available --db-instance-id instance-1234
```

Now the `EngineUptime` metric for the writer instance shows that the instance `instance-1234` was rebooted recently. The reader instance `instance-6305` was also rebooted automatically along with the writer instance. This cluster is running Aurora MySQL 2.09, which doesn't keep the reader instances running as the writer instance reboots.

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" \
  --period 60 --namespace "AWS/RDS" --statistics Maximum \
  --dimensions Name=DBInstanceIdentifier,Value=instance-1234 --output text \
  | sort -k 3
EngineUptime
DATAPOINTS 63749.0 2021-03-16T18:22:00+00:00 Seconds
DATAPOINTS 63809.0 2021-03-16T18:23:00+00:00 Seconds
DATAPOINTS 63869.0 2021-03-16T18:24:00+00:00 Seconds
DATAPOINTS 41.0 2021-03-16T18:25:00+00:00 Seconds
DATAPOINTS 101.0 2021-03-16T18:26:00+00:00 Seconds

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" \
  --period 60 --namespace "AWS/RDS" --statistics Maximum \
  --dimensions Name=DBInstanceIdentifier,Value=instance-6305 --output text \
  | sort -k 3
EngineUptime
DATAPOINTS 411.0 2021-03-16T18:22:00+00:00 Seconds
DATAPOINTS 471.0 2021-03-16T18:23:00+00:00 Seconds
DATAPOINTS 531.0 2021-03-16T18:24:00+00:00 Seconds
DATAPOINTS 49.0 2021-03-16T18:26:00+00:00 Seconds
```

Examples of Aurora reboot operations

The following Aurora MySQL examples show different combinations of reboot operations for reader and writer DB instances in an Aurora DB cluster. After each reboot, SQL queries demonstrate the uptime for the instances in the cluster.

Topics

Finding the writer and reader instances for an Aurora cluster

In an Aurora MySQL cluster with multiple DB instances, it's important to know which one is the writer and which ones are the readers. The writer and reader instances also can switch roles when a failover operation happens. Thus, it's best to perform a check like the following before doing any operation that requires a writer or reader instance. In this case, the `False` values for `IsClusterWriter` identify the reader instances, `instance-6305` and `instance-7448`. The `True` value identifies the writer instance, `instance-1234`.

```
$ aws rds describe-db-clusters --db-cluster-id tpch100g \
  --query "*[].['Cluster:',DBClusterIdentifier,DBClusterMembers[*].
['Instance:',DBInstanceIdentifier,IsClusterWriter]]" \
  --output text
Cluster:     tpch100g
Instance:    instance-6305    False
Instance:    instance-7448    False
Instance:    instance-1234    True
```

Before we start the examples of rebooting, the writer instance has an uptime of approximately one week. The SQL query in this example shows a MySQL-specific way to check the uptime. You might use this technique in a database application. For another technique that uses the AWS CLI and works for both Aurora engines, see Checking uptime for Aurora clusters and instances.

```
$ mysql -h instance-7448.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status
    -> where variable_name='Uptime';
+----------------------------+---------+
| Last Startup               | Uptime  |
+----------------------------+---------+
| 2021-03-08 17:49:06.000000 | 174h 42m|
+----------------------------+---------+
```

Rebooting a single reader instance

This example reboots one of the reader DB instances. Perhaps this instance was overloaded by a huge query or many concurrent connections. Or perhaps it fell behind the writer instance because of a network issue. After starting the reboot operation, the example uses a `wait` command to pause until the instance becomes available. By that point, the instance has an uptime of a few minutes.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-6305
{
    "DBInstance": {
        "DBInstanceIdentifier": "instance-6305",
        "DBInstanceStatus": "rebooting",
...
    }
}
$ aws rds wait db-instance-available --db-instance-id instance-6305
$ mysql -h instance-6305.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status
    -> where variable_name='Uptime';
+----------------------------+---------+
| Last Startup               | Uptime  |
+----------------------------+---------+
| 2021-03-16 00:35:02.000000 | 00h 03m |
+----------------------------+----------+
```

Rebooting the reader instance didn't affect the uptime of the writer instance. It still has an uptime of about one week.

```
$ mysql -h instance-7448.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status where variable_name='Uptime';
+----------------------------+----------+
| Last Startup               | Uptime   |
+----------------------------+----------+
| 2021-03-08 17:49:06.000000 | 174h 49m |
+----------------------------+----------+
```

Rebooting the writer instance

This example reboots the writer instance. This cluster is running Aurora MySQL version 2.09. Because the Aurora MySQL version is lower than 2.10, rebooting the writer instance also reboots any reader instances in the cluster.

A `wait` command pauses until the reboot is finished. Now the uptime for that instance is reset to zero. It's possible that a reboot operation might take substantially different times for writer and reader DB instances. The writer and reader DB instances perform different kinds of cleanup operations depending on their roles.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-1234
{
    "DBInstance": {
        "DBInstanceIdentifier": "instance-1234",
        "DBInstanceStatus": "rebooting",
...
    }
}
$ aws rds wait db-instance-available --db-instance-id instance-1234
$ mysql -h instance-1234.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status where variable_name='Uptime';
```

```
+----------------------------+---------+
| Last Startup               | Uptime  |
+----------------------------+---------+
| 2021-03-16 00:40:27.000000 | 00h 00m |
+----------------------------+---------+
```

After the reboot for the writer DB instance, both of the reader DB instances also have their uptime reset. Rebooting the writer instance caused the reader instances to reboot also. This behavior applies to Aurora PostgreSQL clusters and to Aurora MySQL clusters before version 2.10.

```
$ mysql -h instance-7448.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status where variable_name='Uptime';
+----------------------------+---------+
| Last Startup               | Uptime  |
+----------------------------+---------+
| 2021-03-16 00:40:35.000000 | 00h 00m |
+----------------------------+---------+

$ mysql -h instance-6305.a12345.us-east-1.rds.amazonaws.com -P 3306 -u my-user -p
...
mysql> select date_sub(now(), interval variable_value second) "Last Startup",
    -> time_format(sec_to_time(variable_value),'%Hh %im') as "Uptime"
    -> from performance_schema.global_status where variable_name='Uptime';
+----------------------------+---------+
| Last Startup               | Uptime  |
+----------------------------+---------+
| 2021-03-16 00:40:33.000000 | 00h 01m |
+----------------------------+---------+
```

Rebooting the writer and readers independently

These next examples show a cluster that runs Aurora MySQL version 2.10. In this Aurora MySQL version and higher, you can reboot the writer instance without causing reboots for all the reader instances. That way, your query-intensive applications don't experience any outage when you reboot the writer instance. You can reboot the reader instances later. You might do those reboots at a time of low query traffic. You might also reboot the reader instances one at a time. That way, at least one reader instance is always available for the query traffic of your application.

The following example uses a cluster named `cluster-2393`, running Aurora MySQL version `5.7.mysql_aurora.2.10.0`. This cluster has a writer instance named `instance-9404` and three reader instances named `instance-6772`, `instance-2470`, and `instance-5138`.

```
$ aws rds describe-db-clusters --db-cluster-id cluster-2393 \
  --query "*[].['Cluster:',DBClusterIdentifier,DBClusterMembers[*].
['Instance:',DBInstanceIdentifier,IsClusterWriter]]" \
  --output text
Cluster:        cluster-2393
Instance:       instance-5138        False
Instance:       instance-2470        False
Instance:       instance-6772        False
Instance:       instance-9404        True
```

Checking the `uptime` value of each database instance through the `mysql` command shows that each one has roughly the same uptime. For example, here is the uptime for `instance-5138`.

```
mysql> SHOW GLOBAL STATUS LIKE 'uptime';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Uptime        | 3866  |
+---------------+-------+
```

By using CloudWatch, we can get the corresponding uptime information without actually logging into the instances. That way, an administrator can monitor the database but can't view or change any table data. In this case, we specify a time period spanning five minutes, and check the uptime value every minute. The increasing uptime values demonstrate that the instances weren't restarted during that period.

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-9404 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 4648.0 2021-03-17T23:42:00+00:00 Seconds
DATAPOINTS 4708.0 2021-03-17T23:43:00+00:00 Seconds
DATAPOINTS 4768.0 2021-03-17T23:44:00+00:00 Seconds
DATAPOINTS 4828.0 2021-03-17T23:45:00+00:00 Seconds
DATAPOINTS 4888.0 2021-03-17T23:46:00+00:00 Seconds
```

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-6772 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 4315.0 2021-03-17T23:42:00+00:00 Seconds
DATAPOINTS 4375.0 2021-03-17T23:43:00+00:00 Seconds
DATAPOINTS 4435.0 2021-03-17T23:44:00+00:00 Seconds
DATAPOINTS 4495.0 2021-03-17T23:45:00+00:00 Seconds
DATAPOINTS 4555.0 2021-03-17T23:46:00+00:00 Seconds
```

Now we reboot one of the reader instances, `instance-5138`. We wait for the instance to become available again after the reboot. Now monitoring the uptime over a five-minute period shows that the uptime was reset to zero during that time. The most recent uptime value was measured five seconds after the reboot finished.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-5138
{
  "DBInstanceIdentifier": "instance-5138",
  "DBInstanceStatus": "rebooting"
}
$ aws rds wait db-instance-available --db-instance-id instance-5138

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-5138 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 4500.0 2021-03-17T23:46:00+00:00 Seconds
DATAPOINTS 4560.0 2021-03-17T23:47:00+00:00 Seconds
DATAPOINTS 4620.0 2021-03-17T23:48:00+00:00 Seconds
DATAPOINTS 4680.0 2021-03-17T23:49:00+00:00 Seconds
DATAPOINTS  5.0 2021-03-17T23:50:00+00:00 Seconds
```

Next, we perform a reboot for the writer instance, `instance-9404`. We compare the uptime values for the writer instance and one of the reader instances. By doing so, we can see that rebooting the writer didn't cause a reboot for the readers. In versions before Aurora MySQL 2.10, the uptime values for all the readers would be reset at the same time as the writer.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-9404
{
  "DBInstanceIdentifier": "instance-9404",
  "DBInstanceStatus": "rebooting"
}
$ aws rds wait db-instance-available --db-instance-id instance-9404

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-9404 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 371.0 2021-03-17T23:57:00+00:00 Seconds
DATAPOINTS 431.0 2021-03-17T23:58:00+00:00 Seconds
DATAPOINTS 491.0 2021-03-17T23:59:00+00:00 Seconds
DATAPOINTS 551.0 2021-03-18T00:00:00+00:00 Seconds
DATAPOINTS  37.0  2021-03-18T00:01:00+00:00 Seconds

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-6772 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 5215.0 2021-03-17T23:57:00+00:00 Seconds
DATAPOINTS 5275.0 2021-03-17T23:58:00+00:00 Seconds
DATAPOINTS 5335.0 2021-03-17T23:59:00+00:00 Seconds
DATAPOINTS 5395.0 2021-03-18T00:00:00+00:00 Seconds
DATAPOINTS 5455.0 2021-03-18T00:01:00+00:00 Seconds
```

To make sure that all the reader instances have all the same changes to configuration parameters as the writer instance, reboot all the reader instances after the writer. This example reboots all the readers and then waits until all of them are available before proceeding.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-6772
{
  "DBInstanceIdentifier": "instance-6772",
  "DBInstanceStatus": "rebooting"
}

$ aws rds reboot-db-instance --db-instance-identifier instance-2470
```

```
{
  "DBInstanceIdentifier": "instance-2470",
  "DBInstanceStatus": "rebooting"
}

$ aws rds reboot-db-instance --db-instance-identifier instance-5138
{
  "DBInstanceIdentifier": "instance-5138",
  "DBInstanceStatus": "rebooting"
}

$ aws rds wait db-instance-available --db-instance-id instance-6772
$ aws rds wait db-instance-available --db-instance-id instance-2470
$ aws rds wait db-instance-available --db-instance-id instance-5138
```

Now we can see that the writer DB instance has the highest uptime. This instance's uptime value increased steadily throughout the monitoring period. The reader DB instances were all rebooted after the reader. We can see the point within the monitoring period when each reader was rebooted and its uptime was reset to zero.

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-9404 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 457.0 2021-03-18T00:08:00+00:00 Seconds
DATAPOINTS 517.0 2021-03-18T00:09:00+00:00 Seconds
DATAPOINTS 577.0 2021-03-18T00:10:00+00:00 Seconds
DATAPOINTS 637.0 2021-03-18T00:11:00+00:00 Seconds
DATAPOINTS  697.0 2021-03-18T00:12:00+00:00 Seconds

$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-2470 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 5819.0 2021-03-18T00:08:00+00:00 Seconds
DATAPOINTS  35.0  2021-03-18T00:09:00+00:00 Seconds
DATAPOINTS 95.0 2021-03-18T00:10:00+00:00 Seconds
DATAPOINTS 155.0 2021-03-18T00:11:00+00:00 Seconds
DATAPOINTS 215.0 2021-03-18T00:12:00+00:00 Seconds
```

```
$ aws cloudwatch get-metric-statistics --metric-name "EngineUptime" \
  --start-time "$(date -d '5 minutes ago')" --end-time "$(date -d 'now')" --period 60 \
  --namespace "AWS/RDS" --statistics Minimum --dimensions
 Name=DBInstanceIdentifier,Value=instance-5138 \
  --output text | sort -k 3
EngineUptime
DATAPOINTS 1085.0 2021-03-18T00:08:00+00:00 Seconds
DATAPOINTS 1145.0 2021-03-18T00:09:00+00:00 Seconds
DATAPOINTS 1205.0 2021-03-18T00:10:00+00:00 Seconds
DATAPOINTS  49.0  2021-03-18T00:11:00+00:00 Seconds
DATAPOINTS 109.0 2021-03-18T00:12:00+00:00 Seconds
```

Applying a cluster parameter change to an Aurora MySQL version 2.10 cluster

The following example demonstrates how to apply a parameter change to all DB instances in your Aurora MySQL 2.10 cluster. With this Aurora MySQL version, you reboot the writer instance and all the reader instances independently.

The example uses the MySQL configuration parameter `lower_case_table_names` for illustration. When this parameter setting is different between the writer and reader DB instances, a query might not be able to access a table declared with an uppercase or mixed-case name. Or if two table names differ only in terms of uppercase and lowercase letters, a query might access the wrong table.

This example shows how to determine the writer and reader instances in the cluster by examining the `IsClusterWriter` attribute of each instance. The cluster is named `cluster-2393`. The cluster has a writer instance named `instance-9404`. The reader instances in the cluster are named `instance-5138` and `instance-2470`.

```
$ aws rds describe-db-clusters --db-cluster-id cluster-2393 \
  --query '*[].[DBClusterIdentifier,DBClusterMembers[*].
[DBInstanceIdentifier,IsClusterWriter]]' \
  --output text
cluster-2393
instance-5138        False
instance-2470        False
instance-9404        True
```

To demonstrate the effects of changing the `lower_case_table_names` parameter, we set up two DB cluster parameter groups. The `lower-case-table-names-0` parameter group has this parameter set to 0. The `lower-case-table-names-1` parameter group has this parameter group set to 1.

```
$ aws rds create-db-cluster-parameter-group --description 'lower-case-table-names-0' \
  --db-parameter-group-family aurora-mysql5.7 \
  --db-cluster-parameter-group-name lower-case-table-names-0
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupName": "lower-case-table-names-0",
        "DBParameterGroupFamily": "aurora-mysql5.7",
        "Description": "lower-case-table-names-0"
    }
}

$ aws rds create-db-cluster-parameter-group --description 'lower-case-table-names-1' \
  --db-parameter-group-family aurora-mysql5.7 \
  --db-cluster-parameter-group-name lower-case-table-names-1
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupName": "lower-case-table-names-1",
        "DBParameterGroupFamily": "aurora-mysql5.7",
        "Description": "lower-case-table-names-1"
    }
}

$ aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name lower-case-table-names-0 \
  --parameters
 ParameterName=lower_case_table_names,ParameterValue=0,ApplyMethod=pending-reboot
{
    "DBClusterParameterGroupName": "lower-case-table-names-0"
}

$ aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name lower-case-table-names-1 \
    --parameters
 ParameterName=lower_case_table_names,ParameterValue=1,ApplyMethod=pending-reboot
{
    "DBClusterParameterGroupName": "lower-case-table-names-1"
}
```

The default value of `lower_case_table_names` is 0. With this parameter setting, the table `foo` is distinct from the table `FOO`. This example verifies that the parameter is still at its default setting.

Then the example creates three tables that differ only in uppercase and lowercase letters in their names.

```
mysql> create database lctn;
Query OK, 1 row affected (0.07 sec)

mysql> use lctn;
Database changed
mysql> select @@lower_case_table_names;
+--------------------------+
| @@lower_case_table_names |
+--------------------------+
|                        0 |
+--------------------------+

mysql> create table foo (s varchar(128));
mysql> insert into foo values ('Lowercase table name foo');

mysql> create table Foo (s varchar(128));
mysql> insert into Foo values ('Mixed-case table name Foo');

mysql> create table FOO (s varchar(128));
mysql> insert into FOO values ('Uppercase table name FOO');

mysql> select * from foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from Foo;
+---------------------------+
| s                         |
+---------------------------+
| Mixed-case table name Foo |
+---------------------------+

mysql> select * from FOO;
+--------------------------+
| s                        |
+--------------------------+
| Uppercase table name FOO |
+--------------------------+
```

Next, we associate the DB parameter group with the cluster to set the `lower_case_table_names` parameter to 1. This change only takes effect after each DB instance is rebooted.

```
$ aws rds modify-db-cluster --db-cluster-identifier cluster-2393 \
  --db-cluster-parameter-group-name lower-case-table-names-1
{
  "DBClusterIdentifier": "cluster-2393",
  "DBClusterParameterGroup": "lower-case-table-names-1",
  "Engine": "aurora-mysql",
  "EngineVersion": "5.7.mysql_aurora.2.10.0"
}
```

The first reboot we do is for the writer DB instance. Then we wait for the instance to become available again. At that point, we connect to the writer endpoint and verify that the writer instance has the changed parameter value. The `SHOW TABLES` command confirms that the database contains the three different tables. However, queries that refer to tables named `foo`, `Foo`, or `FOO` all access the table whose name is all-lowercase, `foo`.

```
# Rebooting the writer instance
$ aws rds reboot-db-instance --db-instance-identifier instance-9404
$ aws rds wait db-instance-available --db-instance-id instance-9404
```

Now, queries using the cluster endpoint show the effects of the parameter change. Whether the table name in the query is uppercase, lowercase, or mixed case, the SQL statement accesses the table whose name is all lowercase.

```
mysql> select @@lower_case_table_names;
+--------------------------+
| @@lower_case_table_names |
+--------------------------+
|                        1 |
+--------------------------+

mysql> use lctn;
mysql> show tables;
+----------------+
| Tables_in_lctn |
+----------------+
| FOO            |
| Foo            |
| foo            |
+----------------+

mysql> select * from foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from Foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from FOO;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+
```

The next example shows the same queries as the previous one. In this case, the queries use the reader endpoint and run on one of the reader DB instances. Those instances haven't been rebooted yet. Thus, they still have the original setting for the `lower_case_table_names` parameter. That means that queries can access each of the `foo`, `Foo`, and `FOO` tables.

```
mysql> select @@lower_case_table_names;
+--------------------------+
| @@lower_case_table_names |
+--------------------------+
|                        0 |
+--------------------------+

mysql> use lctn;

mysql> select * from foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from Foo;
+---------------------------+
| s                         |
+---------------------------+
| Mixed-case table name Foo |
+---------------------------+

mysql> select * from FOO;
+--------------------------+
| s                        |
+--------------------------+
| Uppercase table name FOO |
+--------------------------+
```

Next, we reboot one of the reader instances and wait for it to become available again.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-2470
{
  "DBInstanceIdentifier": "instance-2470",
  "DBInstanceStatus": "rebooting"
}
$ aws rds wait db-instance-available --db-instance-id instance-2470
```

While connected to the instance endpoint for `instance-2470`, a query shows that the new parameter is in effect.

```
mysql> select @@lower_case_table_names;
+--------------------------+
| @@lower_case_table_names |
+--------------------------+
|                        1 |
+--------------------------+
```

At this point, the two reader instances in the cluster are running with different `lower_case_table_names` settings. Thus, any connection to the reader endpoint of the cluster uses a value for this setting that's unpredictable. It's important to immediately reboot the other reader instance so that they both have consistent settings.

```
$ aws rds reboot-db-instance --db-instance-identifier instance-5138
{
  "DBInstanceIdentifier": "instance-5138",
  "DBInstanceStatus": "rebooting"
}
$ aws rds wait db-instance-available --db-instance-id instance-5138
```

The following example confirms that all the reader instances have the same setting for the `lower_case_table_names` parameter. The commands check the `lower_case_table_names` setting value on each reader instance. Then the same command using the reader endpoint demonstrates that each connection to the reader endpoint uses one of the reader instances, but which one isn't predictable.

```
# Check lower_case_table_names setting on each reader instance.

$ mysql -h instance-5138.a12345.us-east-1.rds.amazonaws.com \
  -u my-user -p -e 'select @@aurora_server_id, @@lower_case_table_names'
+--------------------------+--------------------------+
| @@aurora_server_id       | @@lower_case_table_names |
+--------------------------+--------------------------+
| instance-5138            |                        1 |
+--------------------------+--------------------------+

$ mysql -h instance-2470.a12345.us-east-1.rds.amazonaws.com \
  -u my-user -p -e 'select @@aurora_server_id, @@lower_case_table_names'
+--------------------------+--------------------------+
| @@aurora_server_id       | @@lower_case_table_names |
+--------------------------+--------------------------+
| instance-2470            |                        1 |
+--------------------------+--------------------------+

# Check lower_case_table_names setting on the reader endpoint of the cluster.

$ mysql -h cluster-2393.cluster-ro-a12345.us-east-1.rds.amazonaws.com \
  -u my-user -p -e 'select @@aurora_server_id, @@lower_case_table_names'
+--------------------------+--------------------------+
| @@aurora_server_id       | @@lower_case_table_names |
+--------------------------+--------------------------+
| instance-5138            |                        1 |
+--------------------------+--------------------------+

# Run query on writer instance
```

```
$ mysql -h cluster-2393.cluster-a12345.us-east-1.rds.amazonaws.com \
  -u my-user -p -e 'select @@aurora_server_id, @@lower_case_table_names'
+--------------------------+--------------------------+
| @@aurora_server_id       | @@lower_case_table_names |
+--------------------------+--------------------------+
| instance-9404            |                        1 |
+--------------------------+--------------------------+
```

With the parameter change applied everywhere, we can see the effect of setting `lower_case_table_names=1`. Whether the table is referred to as `foo`, `Foo`, or `FOO` the query converts the name to `foo` and accesses the same table in each case.

```
mysql> use lctn;

mysql> select * from foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from Foo;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+

mysql> select * from FOO;
+--------------------------+
| s                        |
+--------------------------+
| Lowercase table name foo |
+--------------------------+
```

Deleting Aurora DB clusters and DB instances

You can delete an Aurora DB cluster when you no longer need it. Deleting the cluster removes the cluster volume containing all your data. Before deleting the cluster, you can save a snapshot of your data. You can restore the snapshot later to create a new cluster containing the same data.

You can also delete DB instances from a cluster while preserving the cluster itself and the data that it contains. Deleting DB instances can help you reduce your charges if the cluster isn't busy, or you don't need the computation capacity of multiple DB instances.

Topics

Deleting an Aurora DB cluster

Aurora doesn't provide a single-step method to delete a DB cluster. This design choice is intended to prevent you from accidentally losing data or taking your application offline. Aurora applications are typically mission-critical and require high availability. Thus, Aurora makes it easy to scale the capacity of the cluster up and down by adding and removing DB instances. Removing the DB cluster itself requires you to make a separate deletion.

Use the following general procedure to remove all the DB instances from a cluster and then delete the cluster itself.

1. Delete any reader instances in the cluster. Use the procedure in Deleting a DB instance from an Aurora DB cluster.

   If the cluster has any reader instances, deleting one of the instances only reduces the computation capacity of the cluster. Deleting the reader instances first ensures that the cluster remains available throughout the procedure and doesn't perform unnecessary failover operations.

2. Delete the writer instance from the cluster. Again, use the procedure in Deleting a DB instance from an Aurora DB cluster.

   When you delete the DB instances, the cluster and its associated cluster volume remain even after you delete all the DB instances.

3. Delete the DB cluster.

   - **AWS Management Console** – Choose the cluster, then choose **Delete** from the **Actions** menu. You can choose the following options to preserve the data from the cluster in case you need it later:

     - Create a final snapshot of the cluster volume. The default setting is to create a final snapshot.
     - Retain automated backups. The default setting is not to retain automated backups.

     > **Note**
     >
     > Automated backups for Aurora Serverless v1 DB clusters aren't retained.

     Aurora also requires you to confirm that you intend to delete the cluster.

   - **CLI and API** – Call the `delete-db-cluster` CLI command or `DeleteDBCluster` API operation. You can choose the following options to preserve the data from the cluster in case you need it later:

     - Create a final snapshot of the cluster volume.
     - Retain automated backups.

     > **Note**
     >
     > Automated backups for Aurora Serverless v1 DB clusters aren't retained.

Topics

Deleting an empty Aurora cluster

You can delete an empty DB cluster using the AWS Management Console, AWS CLI, or Amazon RDS API.

> **Tip**
>
> You can keep a cluster with no DB instances to preserve your data without incurring CPU charges for the cluster. You can quickly start using the cluster again by creating one or more new DB instances for the cluster. You can perform Aurora-specific administrative operations on the cluster while it doesn't have any associated DB instances. You just can't access the data or perform any operations that require connecting to a DB instance.

Console

To delete a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to delete.
3. For **Actions**, choose **Delete**.
4. To create a final DB snapshot for the DB cluster, choose **Create final snapshot?**. This is the default setting.
5. If you chose to create a final snapshot, enter the **Final snapshot name**.
6. To retain automated backups, choose **Retain automated backups**. This is not the default setting.
7. Enter **`delete me`** in the box.
8. Choose **Delete**.

CLI

To delete an empty Aurora DB cluster by using the AWS CLI, call the delete-db-cluster command.

Suppose that the empty cluster `deleteme-zero-instances` was only used for development and testing and doesn't contain any important data. In that case, you don't need to preserve a snapshot of the cluster volume when you delete the cluster. The following example demonstrates that a cluster doesn't contain any DB instances, and then deletes the empty cluster without creating a final snapshot or retaining automated backups.

```
$ aws rds describe-db-clusters --db-cluster-identifier deleteme-zero-instances --output
 text \
  --query '*[].["Cluster:",DBClusterIdentifier,DBClusterMembers[*].
["Instance:",DBInstanceIdentifier,IsClusterWriter]]
Cluster:        deleteme-zero-instances

$ aws rds delete-db-cluster --db-cluster-identifier deleteme-zero-instances \
  --skip-final-snapshot \
  --delete-automated-backups
{
  "DBClusterIdentifier": "deleteme-zero-instances",
  "Status": "available",
  "Engine": "aurora-mysql"
}
```

RDS API

To delete an empty Aurora DB cluster by using the Amazon RDS API, call the DeleteDBCluster operation.

Deleting an Aurora cluster with a single DB instance

You can delete the last DB instance, even if the DB cluster has deletion protection enabled. In this case, the DB cluster itself still exists and your data is preserved. You can access the data again by attaching a new DB instance to the cluster.

The following example shows how the `delete-db-cluster` command doesn't work when the cluster still has any associated DB instances. This cluster has a single writer DB instance. When we examine the DB instances in the cluster, we check the `IsClusterWriter` attribute of each one. The cluster could have zero or one writer DB instance. A value of `true` signifies a writer DB instance. A value of `false` signifies a reader DB instance. The cluster could have zero, one, or many reader DB instances. In this case, we delete the writer DB instance using the `delete-db-instance` command. As soon as the DB instance goes into `deleting` state, we can delete the cluster also. For this example also, suppose that the cluster doesn't contain any data worth preserving. Therefore, we don't create a snapshot of the cluster volume or retain automated backups.

```
$ aws rds delete-db-cluster --db-cluster-identifier deleteme-writer-only --skip-final-
snapshot
An error occurred (InvalidDBClusterStateFault) when calling the DeleteDBCluster
 operation:
  Cluster cannot be deleted, it still contains DB instances in non-deleting state.

$ aws rds describe-db-clusters --db-cluster-identifier deleteme-writer-only \
  --query '*[].[DBClusterIdentifier,Status,DBClusterMembers[*].
[DBInstanceIdentifier,IsClusterWriter]]'
[
    [
        "deleteme-writer-only",
        "available",
        [
            [
                "instance-2130",
                true
            ]
        ]
    ]
]

$ aws rds delete-db-instance --db-instance-identifier instance-2130
{
  "DBInstanceIdentifier": "instance-2130",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-cluster --db-cluster-identifier deleteme-writer-only \
  --skip-final-snapshot \
  --delete-automated-backups
{
  "DBClusterIdentifier": "deleteme-writer-only",
  "Status": "available",
  "Engine": "aurora-mysql"
}
```

Deleting an Aurora cluster with multiple DB instances

If your cluster contains multiple DB instances, typically there is a single writer instance and one or more reader instances. The reader instances help with high availability, by being on standby to take over if the writer instance encounters a problem. You can also use reader instances to scale the cluster up to handle a read-intensive workload without adding overhead to the writer instance.

To delete a cluster with multiple reader DB instances, you delete the reader instances first and then the writer instance. Deleting the writer instance leaves the cluster and its data in place. You delete the cluster through a separate action.

- For the procedure to delete an Aurora DB instance, see Deleting a DB instance from an Aurora DB cluster.
- For the procedure to delete the writer DB instance in an Aurora cluster, see Deleting an Aurora cluster with a single DB instance.
- For the procedure to delete an empty Aurora cluster, see Deleting an empty Aurora cluster.

This CLI example shows how to delete a cluster containing both a writer DB instance and a single reader DB instance. The `describe-db-clusters` output shows that `instance-7384` is the writer instance and `instance-1039` is the reader instance. The example deletes the reader instance first, because deleting the writer instance while a reader instance still existed would cause a failover operation. It doesn't make sense to promote the reader instance to a writer if you plan to delete that instance also. Again, suppose that these `db.t2.small` instances are only used for development and testing, and so the delete operation skips the final snapshot and doesn't retain automated backups..

```
$ aws rds delete-db-cluster --db-cluster-identifier deleteme-writer-and-reader --skip-
final-snapshot

An error occurred (InvalidDBClusterStateFault) when calling the DeleteDBCluster
 operation:
  Cluster cannot be deleted, it still contains DB instances in non-deleting state.

$ aws rds describe-db-clusters --db-cluster-identifier deleteme-writer-and-reader --
output text \
  --query '*[].["Cluster:",DBClusterIdentifier,DBClusterMembers[*].
["Instance:",DBInstanceIdentifier,IsClusterWriter]]
Cluster:        deleteme-writer-and-reader
Instance:       instance-1039  False
Instance:       instance-7384   True

$ aws rds delete-db-instance --db-instance-identifier instance-1039
{
  "DBInstanceIdentifier": "instance-1039",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-instance --db-instance-identifier instance-7384
{
  "DBInstanceIdentifier": "instance-7384",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-cluster --db-cluster-identifier deleteme-writer-and-reader \
  --skip-final-snapshot \
  --delete-automated-backups
{
  "DBClusterIdentifier": "deleteme-writer-and-reader",
  "Status": "available",
  "Engine": "aurora-mysql"
}
```

The following example shows how to delete a DB cluster containing a writer DB instance and multiple reader DB instances. It uses concise output from the `describe-db-clusters` command to get a report of the writer and reader instances. Again, we delete all reader DB instances before deleting the writer DB instance. It doesn't matter what order we delete the reader DB instances in.

Suppose that this cluster with several DB instances does contain data worth preserving. Thus, the `delete-db-cluster` command in this example includes the `--no-skip-final-snapshot` and `--final-db-snapshot-identifier` parameters to specify the details of the snapshot to create. It also includes the `--no-delete-automated-backups` parameter to retain automated backups.

```
$ aws rds describe-db-clusters --db-cluster-identifier deleteme-multiple-readers --
output text \
  --query '*[].["Cluster:",DBClusterIdentifier,DBClusterMembers[*].
["Instance:",DBInstanceIdentifier,IsClusterWriter]]
Cluster:        deleteme-multiple-readers
Instance:       instance-1010   False
Instance:       instance-5410   False
Instance:       instance-9948   False
Instance:       instance-8451   True

$ aws rds delete-db-instance --db-instance-identifier instance-1010
```

```
{
  "DBInstanceIdentifier": "instance-1010",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-instance --db-instance-identifier instance-5410
{
  "DBInstanceIdentifier": "instance-5410",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-instance --db-instance-identifier instance-9948
{
  "DBInstanceIdentifier": "instance-9948",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-instance --db-instance-identifier instance-8451
{
  "DBInstanceIdentifier": "instance-8451",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql"
}

$ aws rds delete-db-cluster --db-cluster-identifier deleteme-multiple-readers \
  --no-delete-automated-backups \
  --no-skip-final-snapshot \
  --final-db-snapshot-identifier deleteme-multiple-readers-final-snapshot
{
  "DBClusterIdentifier": "deleteme-multiple-readers",
  "Status": "available",
  "Engine": "aurora-mysql"
}
```

The following example shows how to confirm that Aurora created the requested snapshot. You can request details for the specific snapshot by specifying its identifier `deleteme-multiple-readers-final-snapshot`. You can also get a report of all snapshots for the cluster that was deleted by specifying the cluster identifier `deleteme-multiple-readers`. Both of those commands return information about the same snapshot.

```
$ aws rds describe-db-cluster-snapshots \
  --db-cluster-snapshot-identifier deleteme-multiple-readers-final-snapshot
{
    "DBClusterSnapshots": [
        {
            "AvailabilityZones": [],
            "DBClusterSnapshotIdentifier": "deleteme-multiple-readers-final-snapshot",
            "DBClusterIdentifier": "deleteme-multiple-readers",
            "SnapshotCreateTime": "11T01:40:07.354000+00:00",
            "Engine": "aurora-mysql",
...

$ aws rds describe-db-cluster-snapshots --db-cluster-identifier deleteme-multiple-
readers
{
    "DBClusterSnapshots": [
        {
            "AvailabilityZones": [],
            "DBClusterSnapshotIdentifier": "deleteme-multiple-readers-final-snapshot",
            "DBClusterIdentifier": "deleteme-multiple-readers",
            "SnapshotCreateTime": "11T01:40:07.354000+00:00",
            "Engine": "aurora-mysql",
...
```

Deletion protection for Aurora clusters

You can't delete clusters that have deletion protection enabled. You can delete DB instances within the cluster, but not the cluster itself. That way, the cluster volume containing all your data is safe from accidental deletion. Aurora enforces deletion protection for a DB cluster whether you try to delete the cluster using the console, the AWS CLI, or the RDS API.

Deletion protection is enabled by default when you create a production DB cluster using the AWS Management Console. However, deletion protection is disabled by default if you create a cluster using the AWS CLI or API. Enabling or disabling deletion protection doesn't cause an outage. To be able to delete the cluster, modify the cluster and disable deletion protection. For more information about turning deletion protection on and off, see Modifying the DB cluster by using the console, CLI, and API.

> **Tip**
>
> Even if all the DB instances are deleted, you can access the data by creating a new DB instance in the cluster.

Deleting a stopped Aurora cluster

You can't delete a cluster if it's in the `stopped` state. In this case, start the cluster before deleting it. For more information, see Starting an Aurora DB cluster.

Deleting Aurora MySQL clusters that are read replicas

For Aurora MySQL, you can't delete a DB instance in a DB cluster if both of the following conditions are true:

- The DB cluster is a read replica of another Aurora DB cluster.
- The DB instance is the only instance in the DB cluster.

To delete a DB instance in this case, first promote the DB cluster so that it's no longer a read replica. After the promotion completes, you can delete the final DB instance in the DB cluster. For more information, see Replicating Amazon Aurora MySQL DB clusters across AWS Regions.

The final snapshot when deleting a cluster

Throughout this section, the examples show how you can choose whether to take a final snapshot when you delete an Aurora cluster. If you choose to take a final snapshot but the name you specify matches an existing snapshot, the operation stops with an error. In this case, examine the snapshot details to confirm if it represents your current detail or if it is an older snapshot. If the existing snapshot doesn't have the latest data that you want to preserve, rename the snapshot and try again, or specify a different name for the **final snapshot** parameter.

Deleting a DB instance from an Aurora DB cluster

You can delete a DB instance from an Aurora DB cluster as part of the process of deleting the entire cluster. If your cluster contains a certain number of DB instances, then deleting the cluster requires deleting each of those DB instances. You can also delete one or more reader instances from a cluster while leaving the cluster running. You might do so to reduce compute capacity and associated charges if your cluster isn't busy.

To delete a DB instance, you specify the name of the instance.

You can delete a DB instance using the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> When an Aurora Replica is deleted, its instance endpoint is removed immediately, and the Aurora Replica is removed from the reader endpoint. If there are statements running on the Aurora Replica that is being deleted, there is a three-minute grace period. Existing statements can finish during the grace period. When the grace period ends, the Aurora Replica is shut down and deleted.

For Aurora DB clusters, deleting a DB instance doesn't necessarily delete the entire cluster. You can delete a DB instance in an Aurora cluster to reduce compute capacity and associated charges when the cluster isn't busy. For information about the special circumstances for Aurora clusters that have one DB instance or zero DB instances, see Deleting an Aurora cluster with a single DB instance and Deleting an empty Aurora cluster.

> **Note**
>
> You can't delete a DB cluster when deletion protection is enabled for it. For more information, see Deletion protection for Aurora clusters.
> You can disable deletion protection by modifying the DB cluster. For more information, see Modifying an Amazon Aurora DB cluster.

Console

To delete a DB instance in a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB instance that you want to delete.
3. For **Actions**, choose **Delete**.

4. Enter **delete me** in the box.
5. Choose **Delete**.

AWS CLI

To delete a DB instance by using the AWS CLI, call the delete-db-instance command and specify the `--db-instance-identifier` value.

Example

For Linux, macOS, or Unix:

```
aws rds delete-db-instance \
    --db-instance-identifier mydbinstance
```

For Windows:

```
aws rds delete-db-instance ^
    --db-instance-identifier mydbinstance
```

RDS API

To delete a DB instance by using the Amazon RDS API, call the DeleteDBInstance operation and specify the `DBInstanceIdentifier` parameter.

Note

When the status for a DB instance is `deleting`, its CA certificate value doesn't appear in the RDS console or in output for AWS CLI commands or RDS API operations. For more information about CA certificates, see Using SSL/TLS to encrypt a connection to a DB cluster.

Tagging Amazon RDS resources

An Amazon RDS *tag* is a name-value pair that you define and associate with an Amazon RDS resource such as a DB instance or DB snapshot. The name is referred to as the *key*. Optionally, you can supply a value for the key.

You can use the AWS Management Console, the AWS CLI, or the Amazon RDS API to add, list, and delete tags on Amazon RDS resources. When using the CLI or API, make sure to provide the Amazon Resource Name (ARN) for the RDS resource to work with. For more information about constructing an ARN, see Constructing an ARN for Amazon RDS.

Topics

Why use Amazon RDS resource tags?

You can use tags to do the following:

- Categorize your RDS resources by application, project, department, environment, and so on. For example, you could use a tag key to define a category, where the tag value is an item in this category. You might create the tag `environment=prod`. Or you might define a tag key of `project` and a tag value of `Salix`, which indicates that an Amazon RDS resource is assigned to the Salix project.
- Automate resource management tasks. For example, you could create a maintenance window for instances tagged `environment=prod` that differs from the window for instances tagged `environment=test`. You could also configure automatic DB snapshots for instances tagged `environment=prod`.
- Control access to RDS resources within an IAM policy. You can do this by using the global `aws:ResourceTag/tag-key` condition key. For example, a policy might allow only users in the `DBAdmin` group to modify DB instances tagged with `environment=prod`. For information about

managing access to tagged resources with IAM policies, see Identity and access management for Amazon Aurora and Controlling access to AWS resources in the *AWS Identity and Access Management User Guide*.

- Monitor resources based on a tag. For example, you can create an Amazon CloudWatch dashboard for DB instances tagged with `environment=prod`.
- Track costs by grouping expenses for similarly tagged resources. For example, if you tag RDS resources associated with the Salix project with `project=Salix`, you can generate cost reports for and allocate expenses to this project. For more information, see How AWS billing works with tags in Amazon RDS.

How Amazon RDS resource tags work

AWS doesn't apply any semantic meaning to your tags. Tags are interpreted strictly as character strings.

Topics

Tag sets in Amazon RDS

Every Amazon RDS resource has a container called a *tag set*. The container includes all the tags that are assigned to the resource. A resource has exactly one tag set.

A tag set contains 0—50 tags. If you add a tag to an RDS resource with the same key as an existing resource tag, the new value overwrites the old.

Tag structure in Amazon RDS

The structure of an RDS tag is as follows:

Tag key

The key is the required name of the tag. The string value must be 1—128 Unicode characters in length and cannot be prefixed with `aws:` or `rds:`. The string can contain only the set of

Unicode letters, digits, whitespace, `_`, `.`, `:`, `/`, =, +, `-`, and `@`. The Java regex is `"^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-@]*)$"`. Tag keys are case-sensitive. Thus, the keys `project` and `Project` are distinct.

A key is unique to a tag set. For example, you cannot have a key-pair in a tag set with the key the same but with different values, such as `project=Trinity` and `project=Xanadu`.

Tag value

The value is an optional string value of the tag. The string value must be 1—256 Unicode characters in length. The string can contain only the set of Unicode letters, digits, whitespace, `_`, `.`, `:`, `/`, =, +, `-`, and `@`. The Java regex is `"^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-@]*)$"`. Tag values are case-sensitive. Thus, the values `prod` and `Prod` are distinct.

Values don't need to be unique in a tag set and can be null. For example, you can have a key-value pair in a tag set of `project=Trinity` and `cost-center=Trinity`.

Amazon RDS resources eligible for tagging

You can tag the following Amazon RDS resources:

- DB instances
- DB clusters
- DB cluster endpoints
- Read replicas
- DB snapshots
- DB cluster snapshots
- Reserved DB instances
- Event subscriptions
- DB option groups
- DB parameter groups
- DB cluster parameter groups
- DB subnet groups
- RDS Proxies
- RDS Proxy endpoints

> **Note**
>
> Currently, you can't tag RDS Proxies and RDS Proxy endpoints by using the AWS Management Console.

- Blue/green deployments
- Zero-ETL integrations (preview)

How AWS billing works with tags in Amazon RDS

Use tags to organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill with tag key values included. Then, to see the cost of combined resources, organize your billing information according to resources with the same tag key values. For example, you can tag several resources with a specific application name, and then organize your billing information to see the total cost of that application across several services. For more information, see Using Cost Allocation Tags in the *AWS Billing User Guide*.

How cost allocation tags work with DB cluster snapshots

You can add a tag to a DB cluster snapshot. However, your bill won't reflect this grouping. For cost allocation tags to apply to DB cluster snapshots, the following conditions must be met:

- The tags must be attached to the parent DB instance.
- The parent DB instance must exist in the same AWS account as the DB cluster snapshot.
- The parent DB instance must exist in the same AWS Region as the DB cluster snapshot.

DB cluster snapshots are considered orphaned if they don't exist in the same Region as the parent DB instance, or if the parent DB instance is deleted. Orphaned DB snapshots don't support cost allocation tags. Costs for orphaned snapshots are aggregated in a single untagged line item. Cross-account DB cluster snapshots aren't considered orphaned when the following conditions are met:

- They exist in the same Region as the parent DB instance.
- The parent DB instance is owned by the source account.

Note

If the parent DB instance is owned by a different account, cost allocation tags don't apply to cross-account snapshots in the destination account.

Best practices for tagging Amazon RDS resources

When you use tags, we recommend that you adhere to the following best practices:

- Document conventions for tag use that are followed by all teams in your organization. In particular, ensure the names are both descriptive and consistent. For example, standardize on the format `environment:prod` rather than tagging some resources with `env:production`.

  Important

  Do not store personally identifiable information (PII) or other confidential or sensitive information in tags.

- Automate tagging to ensure consistency. For example, you can use the following techniques:
  - Include tags in an AWS CloudFormation template. When you create resources with the template, the resources are tagged automatically.
  - Define and apply tags using AWS Lambda functions.
  - Create an SSM document that includes steps to add tags to your RDS resources.
- Use tags only when necessary. You can add up to 50 tags for a single RDS resource, but a best practice is to avoid unnecessary tag proliferation and complexity.
- Review tags periodically for relevance and accuracy. Remove or modify outdated tags as needed.
- Consider creating tags with the AWS Tag Editor in the AWS Management Console. You can use the Tag Editor to add tags to multiple supported AWS resources, including RDS resources, at the same time. For more information, see Tag Editor in the *AWS Resource Groups User Guide*.

Managing tags in Amazon RDS

You can do the following:

- Create tags when you create a resource, for example, when you run the AWS CLI command `create-db-instance`.
- Add tags to an existing resource using the command `add-tags-to-resource`.
- List tags associated with a specific resource using the command `list-tags-for-resource`.
- Update tags using the command `add-tags-to-resource`.
- Remove tags from a resource using the command `remove-tags-from-resource`.

The following procedures show how you can perform typical tagging operations on resources related to DB instances and Aurora DB clusters. Note that tags are cached for authorization purposes. For this reason, when you add or update tags on Amazon RDS resources, several minutes can pass before the modifications are available.

Console

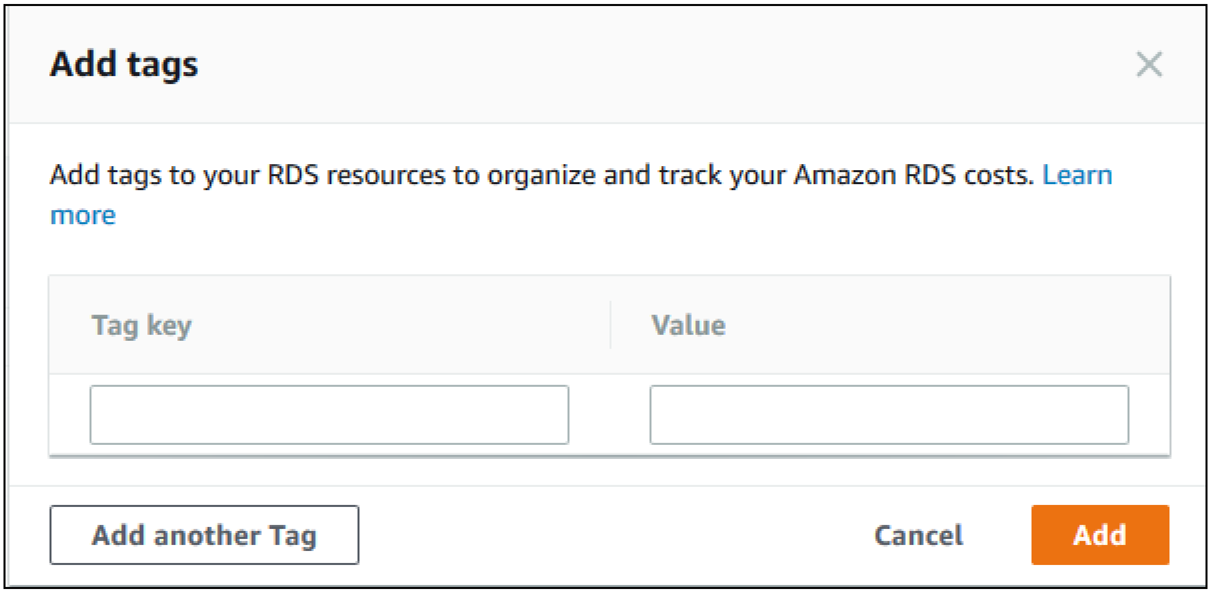
The process to tag an Amazon RDS resource is similar for all resources. The following procedure shows how to tag an Amazon RDS DB instance.

To add a tag to a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

> **Note**
>
> To filter the list of DB instances in the **Databases** pane, enter a text string for **Filter databases**. Only DB instances that contain the string appear.

3. Choose the name of the DB instance that you want to tag to show its details.
4. In the details section, scroll down to the **Tags** section.
5. Choose **Add**. The **Add tags** window appears.



Add tags

Add tags to your RDS resources to organize and track your Amazon RDS costs. Learn more

Tag key	Value

6. Enter a value for **Tag key** and **Value**.

7. To add another tag, you can choose **Add another Tag** and enter a value for its **Tag key** and **Value**.

   Repeat this step as many times as necessary.

8. Choose **Add**.

To delete a tag from a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

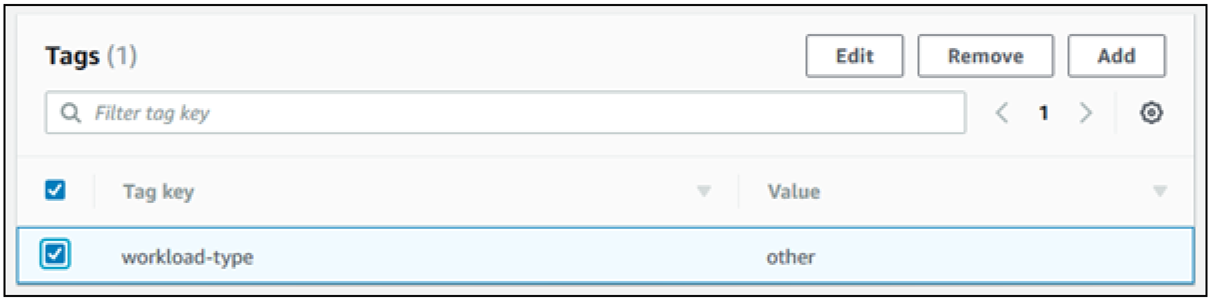
2. In the navigation pane, choose **Databases**.

> **Note**
>
> To filter the list of DB instances in the **Databases** pane, enter a text string in the **Filter databases** box. Only DB instances that contain the string appear.

3. Choose the name of the DB instance to show its details.

4. In the details section, scroll down to the **Tags** section.

5. Choose the tag you want to delete.


Tags (1)

☑	Tag key	Value
☑	workload-type	other

6. Choose **Delete**, and then choose **Delete** in the **Delete tags** window.

AWS CLI

You can add, list, or remove tags for a DB instance using the AWS CLI.

- To add one or more tags to an Amazon RDS resource, use the AWS CLI command `add-tags-to-resource`.
- To list the tags on an Amazon RDS resource, use the AWS CLI command `list-tags-for-resource`.
- To remove one or more tags from an Amazon RDS resource, use the AWS CLI command `remove-tags-from-resource`.

To learn more about how to construct the required ARN, see Constructing an ARN for Amazon RDS.

RDS API

You can add, list, or remove tags for a DB instance using the Amazon RDS API.

- To add a tag to an Amazon RDS resource, use the `AddTagsToResource` operation.
- To list tags that are assigned to an Amazon RDS resource, use the `ListTagsForResource`.
- To remove tags from an Amazon RDS resource, use the `RemoveTagsFromResource` operation.

To learn more about how to construct the required ARN, see Constructing an ARN for Amazon RDS.

When working with XML using the Amazon RDS API, tags use the following schema:

```
<Tagging>
    <TagSet>
        <Tag>
```

```
            <Key>Project</Key>
            <Value>Trinity</Value>
        </Tag>
        <Tag>
            <Key>User</Key>
            <Value>Jones</Value>
        </Tag>
    </TagSet>
</Tagging>
```

The following table provides a list of the allowed XML tags and their characteristics. Values for `Key` and `Value` are case-sensitive. For example, `project=Trinity` and `PROJECT=Trinity` are distinct tags.

Tagging element	Description
TagSet	A tag set is a container for all tags assigned to an Amazon RDS resource. There can be only one tag set per resource. You work with a TagSet only through the Amazon RDS API.
Tag	A tag is a user-defined key-value pair. There can be from 1 to 50 tags in a tag set.
Key	A key is the required name of the tag. For restrictions, see Tag structure in Amazon RDS. The string value can be from 1 to 128 Unicode characters in length and cannot be prefixed with `aws:` or `rds:`. The string can only contain only the set of Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$"). Keys must be unique to a tag set. For example, you cannot have a key-pair in a tag set with the key the same but with different values, such as project/Trinity and project/Xanadu.
Value	A value is the optional value of the tag. For restrictions, see Tag structure in Amazon RDS. The string value can be from 1 to 256 Unicode characters in length and cannot be prefixed with `aws:` or `rds:`. The string can only contain only the set of

Tagging element	Description
	Unicode letters, digits, white space, '_', '.', '/', '=', '+', '-' (Java regex: "^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$"). Values do not have to be unique in a tag set and can be null. For example, you can have a key-value pair in a tag set of project/Trinity and cost-center/Trinity.

Copying tags to DB cluster snapshots

When you create or restore a DB cluster, you can specify that the tags from the cluster are copied to snapshots of the DB cluster. Copying tags ensures that the metadata for the DB snapshots matches that of the source DB cluster. It also ensures any access policies for the DB snapshot also match those of the source DB cluster. Tags are not copied by default.

You can specify that tags are copied to DB snapshots for the following actions:

- Creating a DB cluster
- Restoring a DB cluster
- Creating a read replica
- Copying a DB cluster snapshot

Note

In some cases, you might include a value for the `--tags` parameter of the create-db-snapshot AWS CLI command. Or you might supply at least one tag to the CreateDBSnapshot API operation. In these cases, RDS doesn't copy tags from the source DB instance to the new DB snapshot. This functionality applies even if the source DB instance has the `--copy-tags-to-snapshot` (`CopyTagsToSnapshot`) option turned on.
If you take this approach, you can create a copy of a DB instance from a DB snapshot. This approach avoids adding tags that don't apply to the new DB instance. You create your DB snapshot using the AWS CLI `create-db-snapshot` command (or the `CreateDBSnapshot` RDS API operation). After you create your DB snapshot, you can add tags as described later in this topic.

Tutorial: Use tags to specify which Aurora DB clusters to stop

Suppose that you're creating a number of Aurora DB clusters in a development or test environment. You need to keep all of these clusters for several days. Some of the clusters run tests overnight. Other clusters can be stopped overnight and started again the next day. The following example shows how to assign a tag to those clusters that are suitable to stop overnight. Then the example shows how a script can detect which clusters have that tag and then stop those clusters. In this example, the value portion of the key-value pair doesn't matter. The presence of the `stoppable` tag signifies that the cluster has this user-defined property.

To specify which Aurora DB clusters to stop

1. Determine the ARN of a cluster that you want to designate as stoppable.

   The commands and APIs for tagging work with ARNs. That way, they can work seamlessly across AWS Regions, AWS accounts, and different types of resources that might have identical short names. You can specify the ARN instead of the cluster ID in CLI commands that operate on clusters. Substitute the name of your own cluster for *`dev-test-cluster`*. In subsequent commands that use ARN parameters, substitute the ARN of your own cluster. The ARN includes your own AWS account ID and the name of the AWS Region where your cluster is located.

   ```
   $ aws rds describe-db-clusters --db-cluster-identifier dev-test-cluster \
     --query "*[].{DBClusterArn:DBClusterArn}" --output text
   arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster
   ```

2. Add the tag `stoppable` to this cluster.

   You choose the name for this tag. This approach means that you can avoid devising a naming convention that encodes all relevant information in names. In such a convention, you might encode information in the DB instance name or names of other resources. Because this example treats the tag as an attribute that is either present or absent, it omits the `Value=` part of the `--tags` parameter.

   ```
   $ aws rds add-tags-to-resource \
     --resource-name arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster \
     --tags Key=stoppable
   ```

3. Confirm that the tag is present in the cluster.

These commands retrieve the tag information for the cluster in JSON format and in plain tab-separated text.

```
$ aws rds list-tags-for-resource \
  --resource-name arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster
{
    "TagList": [
        {
            "Key": "stoppable",
            "Value": ""

        }
    ]
}
$ aws rds list-tags-for-resource \
  --resource-name arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster --output
 text
TAGLIST stoppable
```

4. To stop all the clusters that are designated as `stoppable`, prepare a list of all your clusters. Loop through the list and check if each cluster is tagged with the relevant attribute.

   This Linux example uses shell scripting to save the list of cluster ARNs to a temporary file and then perform CLI commands for each cluster.

```
$ aws rds describe-db-clusters --query "*[].[DBClusterArn]" --output text >/tmp/
cluster_arns.lst
$ for arn in $(cat /tmp/cluster_arns.lst)
do
  match="$(aws rds list-tags-for-resource --resource-name $arn --output text | grep
 'TAGLIST\tstoppable')"
  if [[ ! -z "$match" ]]
  then
      echo "Cluster $arn is tagged as stoppable. Stopping it now."
# Note that you can specify the full ARN value as the parameter instead of the
 short ID 'dev-test-cluster'.
      aws rds stop-db-cluster --db-cluster-identifier $arn
  fi
done

Cluster arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster is tagged as
 stoppable. Stopping it now.
```

```
{
    "DBCluster": {
        "AllocatedStorage": 1,
        "AvailabilityZones": [
            "us-east-1e",
            "us-east-1c",
            "us-east-1d"
        ],
        "BackupRetentionPeriod": 1,
        "DBClusterIdentifier": "dev-test-cluster",
        ...
```

You can run a script like this at the end of each day to make sure that nonessential clusters are stopped. You might also schedule a job using a utility such as `cron` to perform such a check each night. For example, you might do this in case some clusters were left running by mistake. Here, you might fine-tune the command that prepares the list of clusters to check.

The following command produces a list of your clusters, but only the ones in `available` state. The script can ignore clusters that are already stopped, because they will have different status values such as `stopped` or `stopping`.

```
$ aws rds describe-db-clusters \
  --query '*[].{DBClusterArn:DBClusterArn,Status:Status}|[?Status == `available`]|[].
{DBClusterArn:DBClusterArn}' \
  --output text
arn:aws:rds:us-east-1:123456789:cluster:cluster-2447
arn:aws:rds:us-east-1:123456789:cluster:cluster-3395
arn:aws:rds:us-east-1:123456789:cluster:dev-test-cluster
arn:aws:rds:us-east-1:123456789:cluster:pg2-cluster
```

Tip

You can use assigning tags and finding clusters that have those tags to reduce costs in other ways. For example, take the scenario with Aurora DB clusters used for development and testing. Here, you might designate some clusters to be deleted at the end of each day, or to have only their reader DB instances deleted. Or you might designate some to have their DB instances changed to small DB instance classes during times of expected low usage.

Working with Amazon Resource Names (ARNs) in Amazon RDS

Resources created in Amazon Web Services are each uniquely identified with an Amazon Resource Name (ARN). For certain Amazon RDS operations, you must uniquely identify an Amazon RDS resource by specifying its ARN. For example, when you create an RDS DB instance read replica, you must supply the ARN for the source DB instance.

Constructing an ARN for Amazon RDS

Resources created in Amazon Web Services are each uniquely identified with an Amazon Resource Name (ARN). You can construct an ARN for an Amazon RDS resource using the following syntax.

```
arn:aws:rds:<region>:<account number>:<resourcetype>:<name>
```

Region Name	Region	Endpoint	Protocol	
US East (Ohio)	us-east-2	rds.us-east-2.amazonaws.com	HTTPS	
		rds-fips.us-east-2.api.aws	HTTPS	
		rds.us-east-2.api.aws	HTTPS	
		rds-fips.us-east-2.amazonaws.com	HTTPS	
US East (N. Virginia)	us-east-1	rds.us-east-1.amazonaws.com	HTTPS	
		rds-fips.us-east-1.api.aws	HTTPS	
		rds-fips.us-east-1.amazonaws.com	HTTPS	
		rds.us-east-1.api.aws	HTTPS	
US West (N. California)	us-west-1	rds.us-west-1.amazonaws.com	HTTPS	
		rds.us-west-1.api.aws	HTTPS	
		rds-fips.us-west-1.amazonaws.com	HTTPS	
		rds-fips.us-west-1.api.aws	HTTPS	

Region Name	Region	Endpoint	Protocol
US West (Oregon)	us-west-2	rds.us-west-2.amazonaws.com rds-fips.us-west-2.amazonaws.com rds.us-west-2.api.aws rds-fips.us-west-2.api.aws	HTTPS HTTPS HTTPS HTTPS
Africa (Cape Town)	af-south-1	rds.af-south-1.amazonaws.com rds.af-south-1.api.aws	HTTPS HTTPS
Asia Pacific (Hong Kong)	ap-east-1	rds.ap-east-1.amazonaws.com rds.ap-east-1.api.aws	HTTPS HTTPS
Asia Pacific (Hyderabad)	ap-south-2	rds.ap-south-2.amazonaws.com rds.ap-south-2.api.aws	HTTPS HTTPS
Asia Pacific (Jakarta)	ap-southeast-3	rds.ap-southeast-3.amazonaws.com rds.ap-southeast-3.api.aws	HTTPS HTTPS
Asia Pacific (Melbourne)	ap-southeast-4	rds.ap-southeast-4.amazonaws.com rds.ap-southeast-4.api.aws	HTTPS HTTPS
Asia Pacific (Mumbai)	ap-south-1	rds.ap-south-1.amazonaws.com rds.ap-south-1.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Asia Pacific (Osaka)	ap-northeast-3	rds.ap-northeast-3.amazonaws.com rds.ap-northeast-3.api.aws	HTTPS HTTPS
Asia Pacific (Seoul)	ap-northeast-2	rds.ap-northeast-2.amazonaws.com rds.ap-northeast-2.api.aws	HTTPS HTTPS
Asia Pacific (Singapore)	ap-southeast-1	rds.ap-southeast-1.amazonaws.com rds.ap-southeast-1.api.aws	HTTPS HTTPS
Asia Pacific (Sydney)	ap-southeast-2	rds.ap-southeast-2.amazonaws.com rds.ap-southeast-2.api.aws	HTTPS HTTPS
Asia Pacific (Tokyo)	ap-northeast-1	rds.ap-northeast-1.amazonaws.com rds.ap-northeast-1.api.aws	HTTPS HTTPS
Canada (Central)	ca-central-1	rds.ca-central-1.amazonaws.com rds.ca-central-1.api.aws rds-fips.ca-central-1.api.aws rds-fips.ca-central-1.amazonaws.com	HTTPS HTTPS HTTPS HTTPS
Canada West (Calgary)	ca-west-1	rds.ca-west-1.amazonaws.com rds-fips.ca-west-1.amazonaws.com	HTTPS HTTPS
Europe (Frankfurt)	eu-central-1	rds.eu-central-1.amazonaws.com rds.eu-central-1.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Europe (Ireland)	eu-west-1	rds.eu-west-1.amazonaws.com rds.eu-west-1.api.aws	HTTPS HTTPS
Europe (London)	eu-west-2	rds.eu-west-2.amazonaws.com rds.eu-west-2.api.aws	HTTPS HTTPS
Europe (Milan)	eu-south-1	rds.eu-south-1.amazonaws.com rds.eu-south-1.api.aws	HTTPS HTTPS
Europe (Paris)	eu-west-3	rds.eu-west-3.amazonaws.com rds.eu-west-3.api.aws	HTTPS HTTPS
Europe (Spain)	eu-south-2	rds.eu-south-2.amazonaws.com rds.eu-south-2.api.aws	HTTPS HTTPS
Europe (Stockholm)	eu-north-1	rds.eu-north-1.amazonaws.com rds.eu-north-1.api.aws	HTTPS HTTPS
Europe (Zurich)	eu-central-2	rds.eu-central-2.amazonaws.com rds.eu-central-2.api.aws	HTTPS HTTPS
Israel (Tel Aviv)	il-central-1	rds.il-central-1.amazonaws.com rds.il-central-1.api.aws	HTTPS HTTPS
Middle East (Bahrain)	me-south-1	rds.me-south-1.amazonaws.com rds.me-south-1.api.aws	HTTPS HTTPS

Region Name	Region	Endpoint	Protocol
Middle East (UAE)	me-central-1	rds.me-central-1.amazonaws.com rds.me-central-1.api.aws	HTTPS HTTPS
South America (São Paulo)	sa-east-1	rds.sa-east-1.amazonaws.com rds.sa-east-1.api.aws	HTTPS HTTPS
AWS GovCloud (US-East)	us-gov-east-1	rds.us-gov-east-1.amazonaws.com rds.us-gov-east-1.api.aws	HTTPS HTTPS
AWS GovCloud (US-West)	us-gov-west-1	rds.us-gov-west-1.amazonaws.com rds.us-gov-west-1.api.aws	HTTPS HTTPS

The following table shows the format that you should use when constructing an ARN for a particular Amazon RDS resource type.

Resource type	ARN format
DB instance	arn:aws:rds:*<region>*:*<account>* :db:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :db:my-mysql-instance-1`
DB cluster	arn:aws:rds:*<region>*:*<account>* :cluster:*<name>* For example:

Resource type	ARN format
	`arn:aws:rds: us-east-2 :123456789012 :cluster: my-aurora-cluster-1`
Event subscription	arn:aws:rds:<region>:<account> :es:<name> For example: `arn:aws:rds: us-east-2 :123456789012 :es:my-subscription`
DB parameter group	arn:aws:rds:<region>:<account> :pg:<name> For example: `arn:aws:rds: us-east-2 :123456789012 :pg:my-param-enable-logs`
DB cluster parameter group	arn:aws:rds:<region>:<account> :cluster-pg:<name> For example: `arn:aws:rds: us-east-2 :123456789012 :cluster-pg: my-cluster-param-timezone`
Reserved DB instance	arn:aws:rds:<region>:<account> :ri:<name> For example: `arn:aws:rds: us-east-2 :123456789012 :ri:my-reserved-postgresql`
DB security group	arn:aws:rds:<region>:<account> :secgrp:<name> For example: `arn:aws:rds: us-east-2 :123456789012 :secgrp:my-public`

Resource type	ARN format
Automated DB snapshot	arn:aws:rds:*<region>*:*<account>* :snapshot:rds:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :snapshot:rds: my-mysql-db-2019-07-22-07-23`
Automated DB cluster snapshot	arn:aws:rds:*<region>*:*<account>* :cluster-snapshot:rds:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :cluster-snapshot:rds:  my-aurora-cluster-2019-07-22-16-16`
Manual DB snapshot	arn:aws:rds:*<region>*:*<account>* :snapshot:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :snapshot: my-mysql-db-snap`
Manual DB cluster snapshot	arn:aws:rds:*<region>*:*<account>* :cluster-snapshot:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :cluster-snapshot:  my-aurora-cluster-snap`
DB subnet group	arn:aws:rds:*<region>*:*<account>* :subgrp:*<name>* For example: `arn:aws:rds: us-east-2 :123456789012 :subgrp:my-subnet-10`

Getting an existing ARN

You can get the ARN of an RDS resource by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or RDS API.

Console

To get an ARN from the AWS Management Console, navigate to the resource you want an ARN for, and view the details for that resource.

For example, you can get the ARN for a DB cluster from the **Configuration** tab of the DB cluster details.

AWS CLI

To get an ARN from the AWS CLI for a particular RDS resource, you use the `describe` command for that resource. The following table shows each AWS CLI command, and the ARN property used with the command to get an ARN.

AWS CLI command	ARN property
describe-event-subscriptions	EventSubscriptionArn
describe-certificates	CertificateArn
describe-db-parameter-groups	DBParameterGroupArn
describe-db-cluster-parameter-groups	DBClusterParameterGroupArn
describe-db-instances	DBInstanceArn
describe-db-security-groups	DBSecurityGroupArn
describe-db-snapshots	DBSnapshotArn
describe-events	SourceArn
describe-reserved-db-instances	ReservedDBInstanceArn
describe-db-subnet-groups	DBSubnetGroupArn

AWS CLI command	ARN property
describe-db-clusters	DBClusterArn
describe-db-cluster-snapshots	DBClusterSnapshotArn

For example, the following AWS CLI command gets the ARN for a DB instance.

Example

For Linux, macOS, or Unix:

```
aws rds describe-db-instances \
--db-instance-identifier DBInstanceIdentifier \
--region us-west-2 \
--query "*[].{DBInstanceIdentifier:DBInstanceIdentifier,DBInstanceArn:DBInstanceArn}"
```

For Windows:

```
aws rds describe-db-instances ^
--db-instance-identifier DBInstanceIdentifier ^
--region us-west-2 ^
--query "*[].{DBInstanceIdentifier:DBInstanceIdentifier,DBInstanceArn:DBInstanceArn}"
```

The output of that command is like the following:

```

[
    {
        "DBInstanceArn": "arn:aws:rds:us-west-2:account_id:db:instance_id",
        "DBInstanceIdentifier": "instance_id"
    }
]
```

RDS API

To get an ARN for a particular RDS resource, you can call the following RDS API operations and use the ARN properties shown following.

RDS API operation	ARN property
DescribeEventSubscriptions	EventSubscriptionArn
DescribeCertificates	CertificateArn
DescribeDBParameterGroups	DBParameterGroupArn
DescribeDBClusterParameterGroups	DBClusterParameterGroupArn
DescribeDBInstances	DBInstanceArn
DescribeDBSecurityGroups	DBSecurityGroupArn
DescribeDBSnapshots	DBSnapshotArn
DescribeEvents	SourceArn
DescribeReservedDBInstances	ReservedDBInstanceArn
DescribeDBSubnetGroups	DBSubnetGroupArn
DescribeDBClusters	DBClusterArn
DescribeDBClusterSnapshots	DBClusterSnapshotArn

Amazon Aurora updates

Amazon Aurora releases updates regularly. Updates are applied to Amazon Aurora DB clusters during system maintenance windows. The timing when updates are applied depends on the region and maintenance window setting for the DB cluster, and also the type of update. Updates require a database restart, so you typically experience 20–30 seconds of downtime. After this downtime, you can resume using your DB cluster or clusters. You can view or change your maintenance window settings from the AWS Management Console.

> **Note**
>
> The time required to reboot your DB instance depends on the crash recovery process, database activity at the time of reboot, and the behavior of your specific DB engine. To improve the reboot time, we recommend that you reduce database activity as much as possible during the reboot process. Reducing database activity reduces rollback activity for in-transit transactions.

For information on operating system updates for Amazon Aurora, see Working with operating system updates.

Some updates are specific to a database engine supported by Aurora. For more information about database engine updates, see the following table.

Database engine	Updates
Amazon Aurora MySQL	See Database engine updates for Amazon Aurora MySQL
Amazon Aurora PostgreSQL	See Amazon Aurora PostgreSQL updates

Identifying your Amazon Aurora version

Amazon Aurora includes certain features that are general to Aurora and available to all Aurora DB clusters. Aurora includes other features that are specific to a particular database engine that Aurora supports. These features are available only to those Aurora DB clusters that use that database engine, such as Aurora PostgreSQL.

An Aurora DB instance provides two version numbers, the Aurora version number and the Aurora database engine version number. Aurora version numbers use the following format.

```
<major version>.<minor version>.<patch version>
```

To get the Aurora version number from an Aurora DB instance using a particular database engine, use one of the following queries.

Database engine	Queries

Amazon Aurora MySQL	`SELECT AURORA_VERSION();`
	`SHOW @@aurora_version;`

Amazon Aurora PostgreSQL	`SELECT AURORA_VERSION();`

Using Amazon RDS Extended Support

With Amazon RDS Extended Support, you can continue running your database on a major engine version past the Aurora end of standard support date for an additional cost. On the Aurora end of standard support date, Amazon Aurora automatically enrolls your databases in RDS Extended Support. Automatic enrollment into RDS Extended Support doesn't change the database engine and doesn't impact the uptime or performance of your DB instance.

This paid offering gives you more time to upgrade to a supported major engine version.

For example, the Aurora end of standard support date for Aurora MySQL version 2 is October 31, 2024. However, you aren't ready to manually upgrade to Aurora MySQL version 3 before that date. In this case, Amazon Aurora automatically enrolls your cluster in RDS Extended Support on October 31, 2024, and you can continue to run Aurora MySQL version 2. Starting December 1, 2024, Amazon Aurora automatically charges you for RDS Extended Support.

RDS Extended Support is available for up to 3 years past the Aurora end of standard support date for a major engine version (3 years and 4 months for Aurora MySQL version 2). After this time, if you haven't upgraded your major engine version to a supported version, then Amazon Aurora will automatically upgrade your major engine version. We recommend that you upgrade to a supported major engine version as soon as possible.

Topics

Overview of Amazon RDS Extended Support

After the Aurora end of standard support date, Amazon Aurora will automatically enroll your databases in RDS Extended Support. Aurora automatically upgrades your DB instance to the last minor version released before the Aurora end of standard support date, if you aren't already running that version. Amazon Aurora won't upgrade your minor version until *after* the Aurora end of standard support date for your major engine version.

You can create new databases with major engine versions that have reached the Aurora end of standard support date. Aurora automatically enrolls these new databases in RDS Extended Support and charges you for this offering.

If you upgrade to an engine that's still under Aurora standard support *before* the Aurora end of standard support date, Amazon Aurora won't enroll your engine in RDS Extended Support.

If you attempt to restore a snapshot of a database compatible with an engine that's past the Aurora end of standard support date but isn't enrolled in RDS Extended Support, then Amazon Aurora will attempt to upgrade the snapshot to be compatible with the latest engine version that is still under Aurora standard support. If the restore fails, then Amazon Aurora will automatically enroll your engine in RDS Extended Support with a version that's compatible with the snapshot.

You can end enrollment in RDS Extended Support at any time. To end enrollment, upgrade each enrolled engine to a newer engine version that's still under Aurora standard support. The end of RDS Extended Support enrollment will be effective the day that you complete an upgrade to a newer engine version that's still under Aurora standard support.

Topics

Amazon RDS Extended Support charges

You will incur charges for all engines enrolled in RDS Extended Support beginning the day after the Aurora end of standard support date. For the Aurora end of standard support date, see Amazon Aurora major versions.

The additional charge for RDS Extended Support automatically stops when you take one of the following actions:

- Upgrade to an engine version that's covered under standard support.
- Delete the database that's running a major version past the Aurora end of standard support date.

The charges will restart if your target engine version enters RDS Extended Support in the future.

For example, Aurora PostgreSQL 11 enters Extended Support on March 1, 2024, but charges don't start until April 1, 2024. You upgrade your Aurora PostgreSQL 11 database to Aurora PostgreSQL 12 on April 30, 2024. You will only be charged for 30 days of Extended Support on Aurora PostgreSQL 11. You continue running Aurora PostgreSQL 12 on this DB instance past the RDS end of standard support date of February 28, 2025. Your database will again incur RDS Extended Support charges starting on March 1, 2025.

For more information, see Amazon Aurora pricing.

Avoiding charges for Amazon RDS Extended Support

You can avoid being charged for RDS Extended Support by preventing Aurora from creating or restoring an Aurora DB cluster or a global cluster past the Aurora end of standard support date. To do this, use the AWS CLI or the RDS API.

In the AWS CLI, specify `open-source-rds-extended-support-disabled` for the `--engine-lifecycle-support` option. In the RDS API, specify `open-source-rds-extended-support-disabled` for the `LifeCycleSupport` parameter. For more information, see Creating an Aurora DB cluster or a global cluster or Restoring an Aurora DB cluster or a global cluster.

Versions with Amazon RDS Extended Support

RDS Extended Support is available for Aurora MySQL versions 2 and 3, and for Aurora PostgreSQL version 11 and higher. For more information, see Amazon Aurora major versions.

RDS Extended Support is only available on certain minor versions. For more information, see Amazon Aurora minor versions.

RDS Extended Support is only available on Aurora Serverless v2. It isn't available on Aurora Serverless v1.

Amazon Aurora and customer responsibilities with Amazon RDS Extended Support

The following content describes the responsibilities of Amazon Aurora and your responsibilities with RDS Extended Support.

Topics

- Amazon Aurora responsibilities

- Your responsibilities

Amazon Aurora responsibilities

After the Aurora end of standard support date, Amazon Aurora will supply patches, bug fixes, and upgrades for engines that are enrolled in RDS Extended Support. This will occur for up to 3 years, or until you stop using the engines, whichever happens first.

The patches will be for Critical and High CVEs as defined by the National Vulnerability Database (NVD) CVSS severity ratings. For more information, see Vulnerability Metrics.

Your responsibilities

You're responsible for applying the patches, bug fixes, and upgrades given for Aurora DB clusters or global clusters enrolled in RDS Extended Support. Amazon Aurora reserves the right to change, replace, or withdraw such patches, bug fixes, and upgrades at any time. If a patch is necessary to address security or critical stability issues, Amazon Aurora reserves the right to update your Aurora DB clusters or global clusters with the patch, or to require that you install the patch.

You're also responsible for upgrading your engine to a newer engine version *before* the RDS end of Extended Support date. The RDS end of Extended Support date is typically 3 years after the community end of life. . For the RDS end of Extended Support date for your database major engine version, see Amazon Aurora major versions.

If you don't upgrade your engine, then after the RDS end of Extended Support date, Amazon Aurora will attempt to upgrade your engine to the latest engine version that's supported under Aurora standard support. If the upgrade fails, then Amazon Aurora reserves the right to delete the Aurora DB cluster or global cluster that's running the engine past the Aurora end of standard support date. However, before doing so, Amazon Aurora will preserve your data from that engine.

Creating an Aurora DB cluster or a global cluster with Amazon RDS Extended Support

When you create an Aurora DB cluster or a global cluster, select **Enable RDS Extended Support** in the console, or use the Extended Support option in the AWS CLI or the parameter in the RDS API.

Note

If you don't specify the RDS Extended Support setting, Aurora defaults to RDS Extended Support. This default behavior maintains the availability of your database past the Aurora end of standard support date.

Topics

- Considerations for RDS Extended Support
- Create an Aurora DB cluster or a global cluster with RDS Extended Support

Considerations for RDS Extended Support

Before creating an Aurora DB cluster or a global cluster, consider the following items:

- *After* the Aurora end of standard support date has passed, you can prevent the creation of a new Aurora DB cluster or a new global cluster and avoid RDS Extended Support charges. To do this, use the AWS CLI or the RDS API. In the AWS CLI, specify `open-source-rds-extended-support-disabled` for the `--engine-lifecycle-support` option. In the RDS API, specify `open-source-rds-extended-support-disabled` for the `LifeCycleSupport` parameter. If you specify `open-source-rds-extended-support-disabled` and the Aurora end of standard support date has passed, creating an Aurora DB cluster or a global cluster will always fail.
- RDS Extended Support is set at the cluster level. Members of a cluster will always have the same setting for RDS Extended Support in the RDS console, `--engine-lifecycle-support` in the AWS CLI, and `EngineLifecycleSupport` in the RDS API.

For more information, see Amazon Aurora versions.

Create an Aurora DB cluster or a global cluster with RDS Extended Support

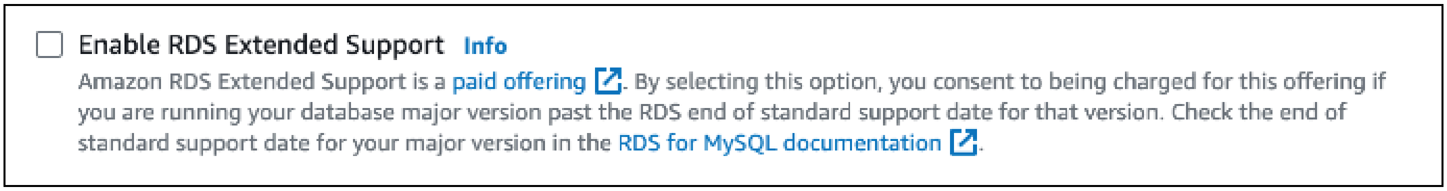
You can create an Aurora DB cluster or a global cluster with an RDS Extended Support version using the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> The AWS CLI `--engine-lifecycle-support` option and the RDS API `EngineLifeCycle` parameter are currently only available for Aurora PostgreSQL. They will become available for Aurora MySQL closer to the Aurora end of standard support date.

Console

When you create an Aurora DB cluster or a global cluster, in the **Engine options** section, select **Enable RDS Extended Support**.

The following image shows the **Enable RDS Extended Support** setting:



□ Enable RDS Extended Support Info

Amazon RDS Extended Support is a paid offering. By selecting this option, you consent to being charged for this offering if you are running your database major version past the RDS end of standard support date for that version. Check the end of standard support date for your major version in the RDS for MySQL documentation.

AWS CLI

When you run the create-db-cluster or create-global-cluster AWS CLI command, select RDS Extended Support by specifying `open-source-rds-extended-support` for the `--engine-lifecycle-support` option. By default, this option is set to `open-source-rds-extended-support`.

To prevent the creation of a new Aurora DB cluster or a global cluster after the Aurora end of standard support date, specify `open-source-rds-extended-support-disabled` for the `--engine-lifecycle-support` option. By doing so, you will avoid any associated RDS Extended Support charges.

RDS API

When you use the CreateDBCluster or CreateGlobalCluster Amazon RDS API operation, select RDS Extended Support by setting the `EngineLifecycleSupport` parameter to `open-source-rds-extended-support`. By default, this parameter is set to `open-source-rds-extended-support`.

To prevent the creation of a new Aurora DB cluster or a global cluster after the Aurora end of standard support date, specify `open-source-rds-extended-support-disabled` for the

`EngineLifecycleSupport` parameter. By doing so, you will avoid any associated RDS Extended Support charges.

For more information, see the following topics:

- To create an Aurora DB cluster, follow the instructions for your DB engine in Creating an Amazon Aurora DB cluster.
- To create a global cluster, follow the instructions for your DB engine in Creating an Amazon Aurora global database.

Viewing the enrollment of your Aurora DB clusters or global clusters in Amazon RDS Extended Support

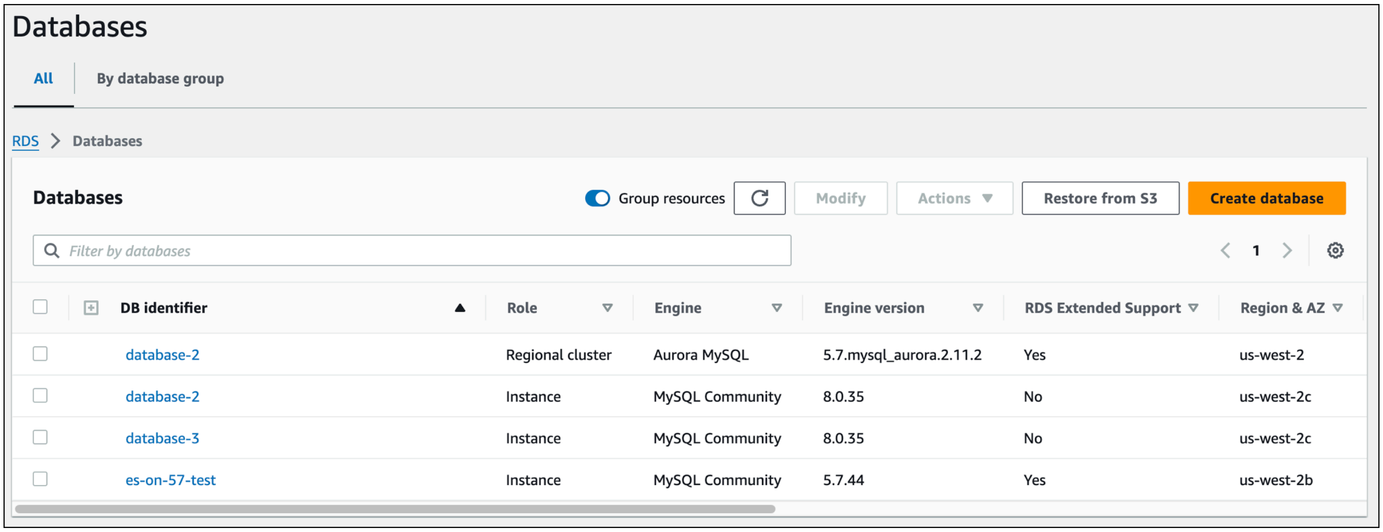
You can view the enrollment of your Aurora DB clusters or global clusters in RDS Extended Support using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To view the enrollment of your Aurora DB clusters or global clusters in RDS Extended Support

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**. The value under **RDS Extended Support** indicates if an Aurora DB cluster or global cluster is enrolled in RDS Extended Support. If no value appears, then RDS Extended Support isn't available for your database.

> **Tip**
>
> If the **RDS Extended Support** column doesn't appear, choose the **Preferences** icon, and then turn on **RDS Extended Support**.



Databases

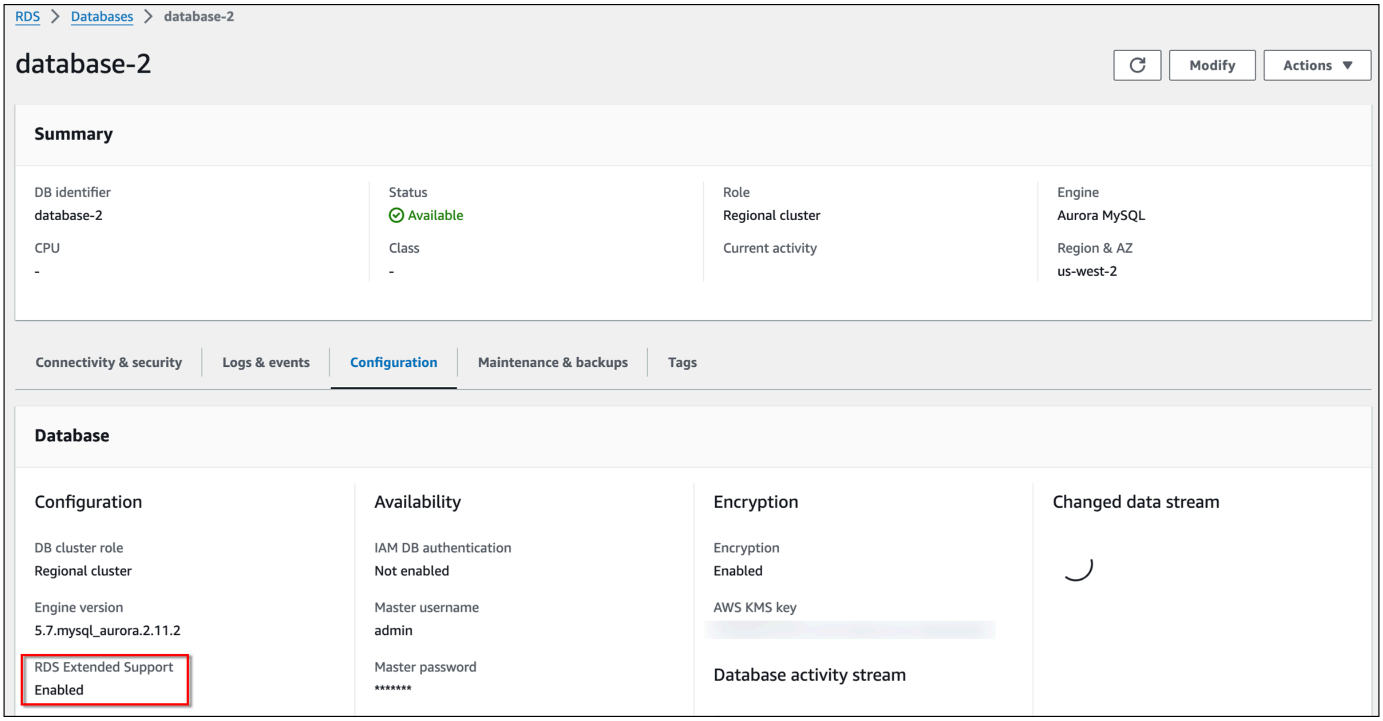
All | By database group

RDS > Databases

Databases

□	DB identifier	Role	Engine	Engine version	RDS Extended Support	Region & AZ
□	database-2	Regional cluster	Aurora MySQL	5.7.mysql_aurora.2.11.2	Yes	us-west-2
□	database-2	Instance	MySQL Community	8.0.35	No	us-west-2c
□	database-3	Instance	MySQL Community	8.0.35	No	us-west-2c
□	es-on-57-test	Instance	MySQL Community	5.7.44	Yes	us-west-2b

3. You can also view the enrollment on the **Configuration** tab for each database. Choose a database under **DB identifier**. On the **Configuration** tab, look under **Extended Support** to see if the database is enrolled or not.



RDS > Databases > database-2

database-2

Summary

DB identifier
database-2

Status
Available

Role
Regional cluster

Engine
Aurora MySQL

CPU
-

Class
-

Current activity

Region & AZ
us-west-2

Database

Configuration

DB cluster role
Regional cluster

Engine version
5.7.mysql_aurora.2.11.2

RDS Extended Support
Enabled

Availability

IAM DB authentication
Not enabled

Master username
admin

Master password
****

Encryption

Encryption
Enabled

AWS KMS key
[redacted]

Database activity stream

Changed data stream

AWS CLI

To view the enrollment of your databases in RDS Extended Support by using the AWS CLI, run the describe-db-clusters or describe-global-clusters command.

If RDS Extended Support is available for a database, then the response includes the parameter EngineLifecycleSupport. The value open-source-rds-extended-support indicates that an Aurora DB cluster or global cluster is enrolled in RDS Extended Support. The value open-source-rds-extended-support-disabled indicates that enrollment of the Aurora DB cluster or global cluster in RDS Extended Support was disabled.

Example

The following command returns information for all of your Aurora DB clusters:

```
aws rds describe-db-clusters
```

The following response shows that an Aurora PostgreSQL engine running on the Aurora DB cluster database-1 is enrolled in RDS Extended Support:

```
{
    "DBClusterIdentifier": "database-1",
    ...
    "Engine": "aurora-postgresql",
    ...
    "EngineLifecycleSupport": "open-source-rds-extended-support"
}
```

RDS API

To view the enrollment of your databases in RDS Extended Support by using the Amazon RDS API, use the DescribeDBClusters or DescribeGlobalClusters operation.

If RDS Extended Support is available for a database, then the response includes the parameter EngineLifecycleSupport. The value open-source-rds-extended-support indicates that an Aurora DB cluster or global cluster is enrolled in RDS Extended Support. The value open-source-rds-extended-support-disabled indicates that enrollment of the Aurora DB cluster or global cluster in RDS Extended Support was disabled.

Restoring an Aurora DB cluster or a global cluster with Amazon RDS Extended Support

When you restore an Aurora DB cluster or a global cluster, select **Enable RDS Extended Support** in the console, or use the Extended Support option in the AWS CLI or the parameter in the RDS API.

> **Note**
>
> If you don't specify the RDS Extended Support setting, Aurora defaults to RDS Extended Support. This default behavior maintains the availability of your database past the Aurora end of standard support date.

Topics

- Considerations for RDS Extended Support
- Restore an Aurora DB cluster DB cluster or a global cluster with RDS Extended Support

Considerations for RDS Extended Support

Before restoring an Aurora DB cluster or a global cluster, consider the following items:

- *After* the Aurora end of standard support date has passed, if you want to restore an Aurora DB cluster or a global cluster from Amazon S3, you can only do so by using the AWS CLI or the RDS API. Use the `--engine-lifecycle-support` option in the restore-db-cluster-from-s3 AWS CLI command or the `EngineLifecycleSupport` parameter in the RestoreDBClusterFromS3 RDS API operation.
- If you want to prevent Aurora from restoring your databases to RDS Extended Support versions, specify `open-source-rds-extended-support-disabled` in the AWS CLI or the RDS API. By doing so, you will avoid any associated RDS Extended Support charges.

  If you specify this setting, Amazon Aurora will automatically upgrade your restored database to a newer, supported major version. If the upgrade fails pre-upgrade checks, Amazon Aurora will safely roll back to the RDS Extended Support engine version. This database will remain in RDS Extended Support mode, and Amazon Aurora will charge you for RDS Extended Support until you manually upgrade your database.
- RDS Extended Support is set at the cluster level. Members of a cluster will always have the same setting for RDS Extended Support in the RDS console, `--engine-lifecycle-support` in the AWS CLI, and `EngineLifecycleSupport` in the RDS API.

For more information, see Amazon Aurora versions.

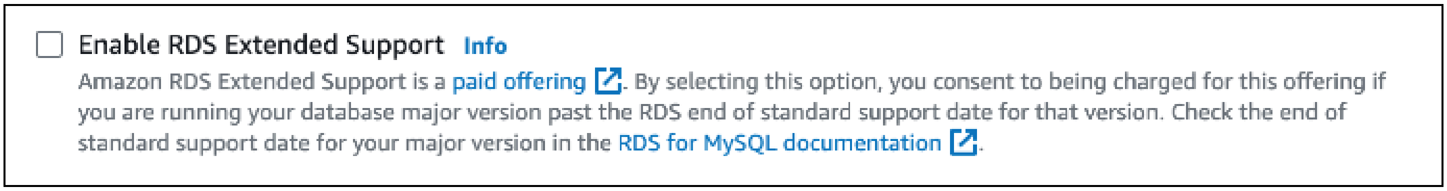
Restore an Aurora DB cluster DB cluster or a global cluster with RDS Extended Support

You can restore an Aurora DB cluster or a global cluster with an RDS Extended Support version using the AWS Management Console, the AWS CLI, or the RDS API.

Console

When you restore an Aurora DB cluster or a global cluster, select **Enable RDS Extended Support** in the **Engine options** section.

The following image shows the **Enable RDS Extended Support** setting:



□ Enable RDS Extended Support Info

Amazon RDS Extended Support is a paid offering. By selecting this option, you consent to being charged for this offering if you are running your database major version past the RDS end of standard support date for that version. Check the end of standard support date for your major version in the RDS for MySQL documentation.

AWS CLI

When you run the restore-db-cluster-from-snapshot AWS CLI command, select RDS Extended Support by specifying `open-source-rds-extended-support` for the `--engine-lifecycle-support` option.

If you want to avoid charges associated with RDS Extended Support, set the `--engine-lifecycle-support` option to `open-source-rds-extended-support-disabled`. By default, this option is set to `open-source-rds-extended-support`.

You can also specify this value using the following AWS CLI commands:

- restore-db-cluster-from-s3
- restore-db-cluster-to-point-in-time

RDS API

When you use the RestoreDBClusterFromSnapshot Amazon RDS API operation, select RDS Extended Support by setting the `EngineLifecycleSupport` parameter to `open-source-rds-extended-support`.

If you want to avoid charges associated with RDS Extended Support, set the `EngineLifecycleSupport` parameter to `open-source-rds-extended-support-disabled`. By default, this parameter is set to `open-source-rds-extended-support`.

You can also specify this value using the following RDS API operations:

- RestoreDBClusterFromS3
- RestoreDBClusterToPointInTime

For more information about restoring an Aurora DB cluster, follow the instructions for your DB engine in Backing up and restoring an Amazon Aurora DB cluster.

Using Amazon RDS Blue/Green Deployments for database updates

A blue/green deployment copies a production database environment to a separate, synchronized staging environment. By using Amazon RDS Blue/Green Deployments, you can make changes to the database in the staging environment without affecting the production environment. For example, you can upgrade the major or minor DB engine version, change database parameters, or make schema changes in the staging environment. When you're ready, you can promote the staging environment to be the new production database environment, with downtime typically under one minute.

Amazon Aurora creates the staging environment by *cloning* the underlying Aurora storage volume in the production environment. The cluster volume in the staging environment only stores incremental changes made to that environment.

> **Note**
>
> Currently, Blue/Green Deployments are supported for Aurora MySQL and Aurora PostgreSQL. For Amazon RDS engine availability, see Using Amazon RDS Blue/Green Deployments for database updates in the *Amazon RDS User Guide*.

Topics

Overview of Amazon RDS Blue/Green Deployments for Aurora

By using Amazon RDS Blue/Green Deployments, you can make and test database changes before implementing them in a production environment. A *blue/green deployment* creates a staging environment that copies the production environment. In a blue/green deployment, the *blue environment* is the current production environment. The *green environment* is the staging environment. The staging environment stays in sync with the current production environment using logical replication.

You can make changes to the Aurora DB cluster in the green environment without affecting production workloads. For example, you can upgrade the major or minor DB engine version or change database parameters in the staging environment. You can thoroughly test changes in the green environment. When ready, you can *switch over* the environments to promote the green environment to be the new production environment. The switchover typically takes under a minute with no data loss and no need for application changes.

Because the green environment is a copy of the topology of the production environment, the DB cluster and all of its DB instances are copied in the deployment. The green environment also includes the features used by the DB cluster, such as DB cluster snapshots, Performance Insights, Enhanced Monitoring, and Aurora Serverless v2.

> **Note**
>
> Blue/Green Deployments are supported for Aurora MySQL and Aurora PostgreSQL. For Amazon RDS availability, see Using Amazon RDS Blue/Green Deployments for database updates in the *Amazon RDS User Guide*.

Topics

- Region and version availability
- Benefits of using Amazon RDS Blue/Green Deployments
- Workflow of a blue/green deployment
- Authorizing access to blue/green deployment operations
- Considerations for blue/green deployments
- Best practices for blue/green deployments
- Limitations for blue/green deployments

Region and version availability

Feature availability and support varies across specific versions of each database engine, and across AWS Regions. For more information, see the section called “Blue/Green Deployments”.

Benefits of using Amazon RDS Blue/Green Deployments

By using Amazon RDS Blue/Green Deployments, you can stay current on security patches, improve database performance, and adopt newer database features with short, predictable downtime. Blue/green deployments reduce the risks and downtime for database updates, such as major or minor engine version upgrades.

Blue/green deployments provide the following benefits:

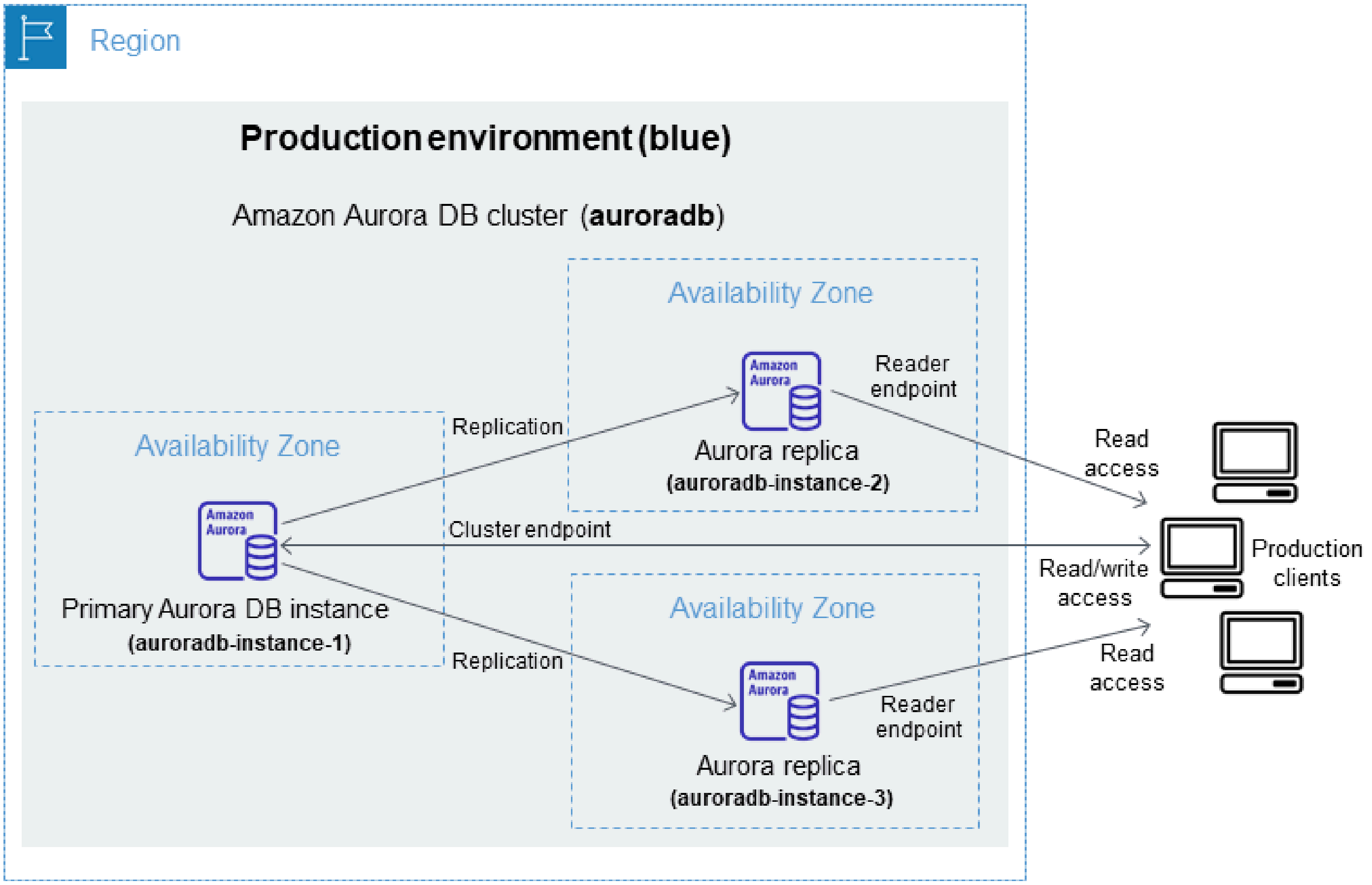
- Easily create a production-ready staging environment.
- Automatically replicate database changes from the production environment to the staging environment.
- Test database changes in a safe staging environment without affecting the production environment.
- Stay current with database patches and system updates.
- Implement and test newer database features.
- Switch over your staging environment to be the new production environment without changes to your application.
- Safely switch over through the use of built-in switchover guardrails.
- Eliminate data loss during switchover.
- Switch over quickly, typically under a minute depending on your workload.

Workflow of a blue/green deployment

Complete the following major steps when you use a blue/green deployment for Aurora DB cluster updates.

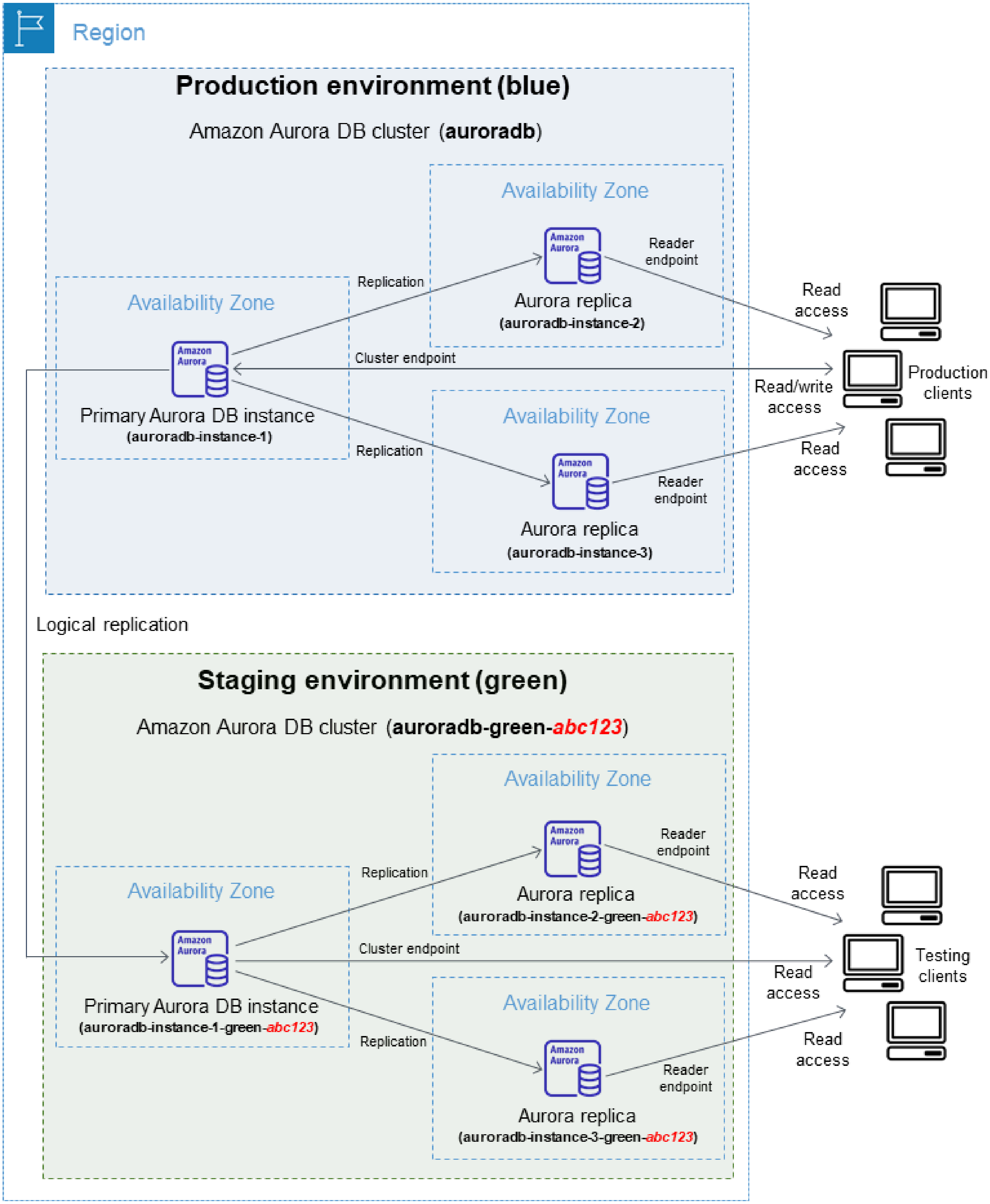
1. Identify a production DB cluster that requires updates.

   The following image shows an example of a production DB cluster.



2. Create the blue/green deployment. For instructions, see [Creating a blue/green deployment](#).

   The following image shows an example of a blue/green deployment of the production environment from step 1. While creating the blue/green deployment, RDS copies the complete topology and configuration of the Aurora DB cluster to create the green environment. The names of the copied DB cluster and DB instances are appended with `-green-`*`random-characters`*. The staging environment in the image contains the DB cluster (auroradb-green-*abc123*). It also contains the three DB instances in the DB cluster (auroradb-instance1-green-*abc123*, auroradb-instance2-green-*abc123*, and auroradb-instance3-green-*abc123*).


Region
Production environment (blue)
Amazon Aurora DB cluster (auroradb)
Availability Zone
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-2)
Replication
Availability Zone
Amazon Aurora
Primary Aurora DB instance
(auroradb-instance-1)
Cluster endpoint
Read access
Read/write access
Production clients
Read access
Availability Zone
Replication
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-3)
Logical replication
Staging environment (green)
Amazon Aurora DB cluster (auroradb-green-abc123)
Availability Zone
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-2-green-abc123)
Replication
Availability Zone
Amazon Aurora
Primary Aurora DB instance
(auroradb-instance-1-green-abc123)
Cluster endpoint
Read access
Read access
Testing clients
Read access
Availability Zone
Replication
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-3-green-abc123)

When you create the blue/green deployment, you can specify a higher DB engine version and a different DB cluster parameter group for the DB cluster in the green environment. You can also specify a different DB parameter group for the DB instances in the DB cluster.

RDS also configures replication from the primary DB instance in the blue environment to the primary DB instance in the green environment.

> ⚠ **Important**
>
> For Aurora MySQL version 3, after you create the blue/green deployment, the DB cluster in the green environment allows write operations by default. We recommend that you make the DB cluster read-only by setting the `read_only` parameter to 1 and rebooting the cluster.

3. Make changes to the staging environment.

   For example, you might make schema changes to your database or change the DB instance class used by one or more DB instances in the green environment.

   For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

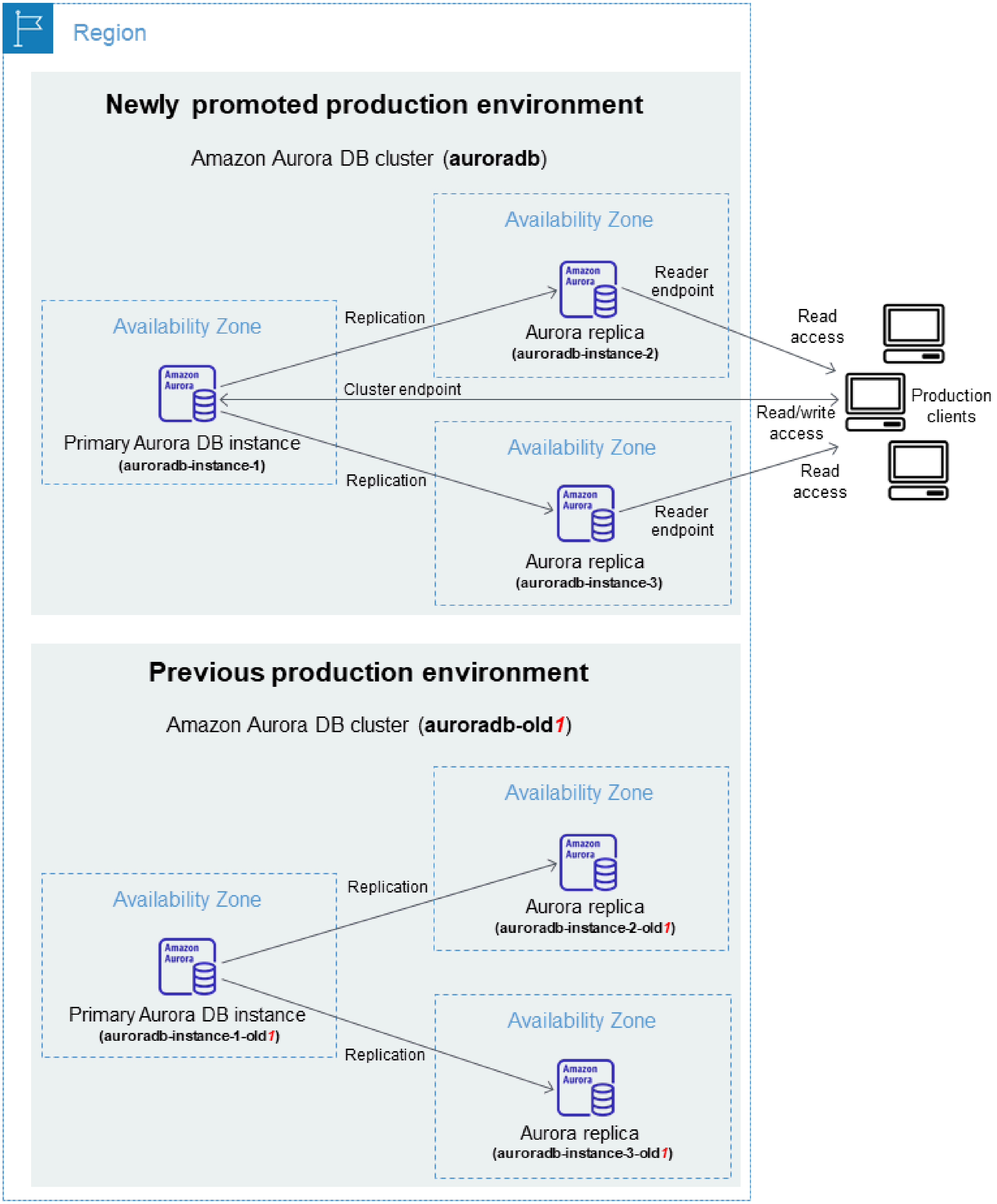
4. Test your staging environment.

   During testing, we recommend that you keep your databases in the green environment read only. Enable write operations on the green environment with caution because they can result in replication conflicts. They can also result in unintended data in the production databases after switchover. To enable write operations for Aurora MySQL, set the `read_only` parameter to `0`, then reboot the DB instance. For Aurora PostgreSQL, set the `default_transaction_read_only` parameter to `off` at the session level.

5. When ready, switch over to promote the staging environment to be the new production environment. For instructions, see Switching a blue/green deployment.

   The switchover results in downtime. The downtime is usually under one minute, but it can be longer depending on your workload.

   The following image shows the DB clusters after the switchover.


Region
Newly promoted production environment
Amazon Aurora DB cluster (auroradb)
Availability Zone
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-2)
Availability Zone
Replication
Amazon Aurora
Primary Aurora DB instance
(auroradb-instance-1)
Cluster endpoint
Read access
Read/write access
Production clients
Read access
Availability Zone
Replication
Amazon Aurora
Reader endpoint
Aurora replica
(auroradb-instance-3)
Previous production environment
Amazon Aurora DB cluster (auroradb-old1)
Availability Zone
Amazon Aurora
Aurora replica
(auroradb-instance-2-old1)
Availability Zone
Replication
Amazon Aurora
Primary Aurora DB instance
(auroradb-instance-1-old1)
Replication
Availability Zone
Amazon Aurora
Aurora replica
(auroradb-instance-3-old1)

After the switchover, the Aurora DB cluster in the green environment becomes the new production DB cluster. The names and endpoints in the current production environment are assigned to the newly promoted production environment, requiring no changes to your application. As a result, your production traffic now flows to the new production environment. The DB cluster and DB instances in the blue environment are renamed by appending `-old`*n* to the current name, where *n* is a number. For example, assume the name of the DB instance in the blue environment is `auroradb-instance-1`. After switchover, the DB instance name might be `auroradb-instance-1-old1`.

In the example in the image, the following changes occur during switchover:

- The green environment DB cluster `auroradb-green-abc123` becomes the production DB cluster named `auroradb`.
- The green environment DB instance named `auroradb-instance1-green-abc123` becomes the production DB instance `auroradb-instance1`.
- The green environment DB instance named `auroradb-instance2-green-abc123` becomes the production DB instance `auroradb-instance2`.
- The green environment DB instance named `auroradb-instance3-green-abc123` becomes the production DB instance `auroradb-instance3`.
- The blue environment DB cluster named `auroradb` becomes `auroradb-old1`.
- The blue environment DB instance named `auroradb-instance1` becomes `auroradb-instance1-old1`.
- The blue environment DB instance named `auroradb-instance2` becomes `auroradb-instance2-old1`.
- The blue environment DB instance named `auroradb-instance3` becomes `auroradb-instance3-old1`.

6. If you no longer need a blue/green deployment, you can delete it. For instructions, see Deleting a blue/green deployment.

   After switchover, the previous production environment isn't deleted so that you can use it for regression testing, if necessary.

Authorizing access to blue/green deployment operations

Users must have the required permissions to perform operations related to blue/green deployments. You can create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. You can then attach those policies to the IAM permission sets or roles that require those permissions. For more information, see Identity and access management for Amazon Aurora.

The user who creates a blue/green deployment must have permissions to perform the following RDS operations:

- `rds:AddTagsToResource`
- `rds:CreateDBCluster`
- `rds:CreateDBInstance`
- `rds:CreateDBClusterEndpoint`

The user who switches over a blue/green deployment must have permissions to perform the following RDS operations:

- `rds:ModifyDBCluster`
- `rds:PromoteReadReplicaDBCluster`

The user who deletes a blue/green deployment must have permissions to perform the following RDS operations:

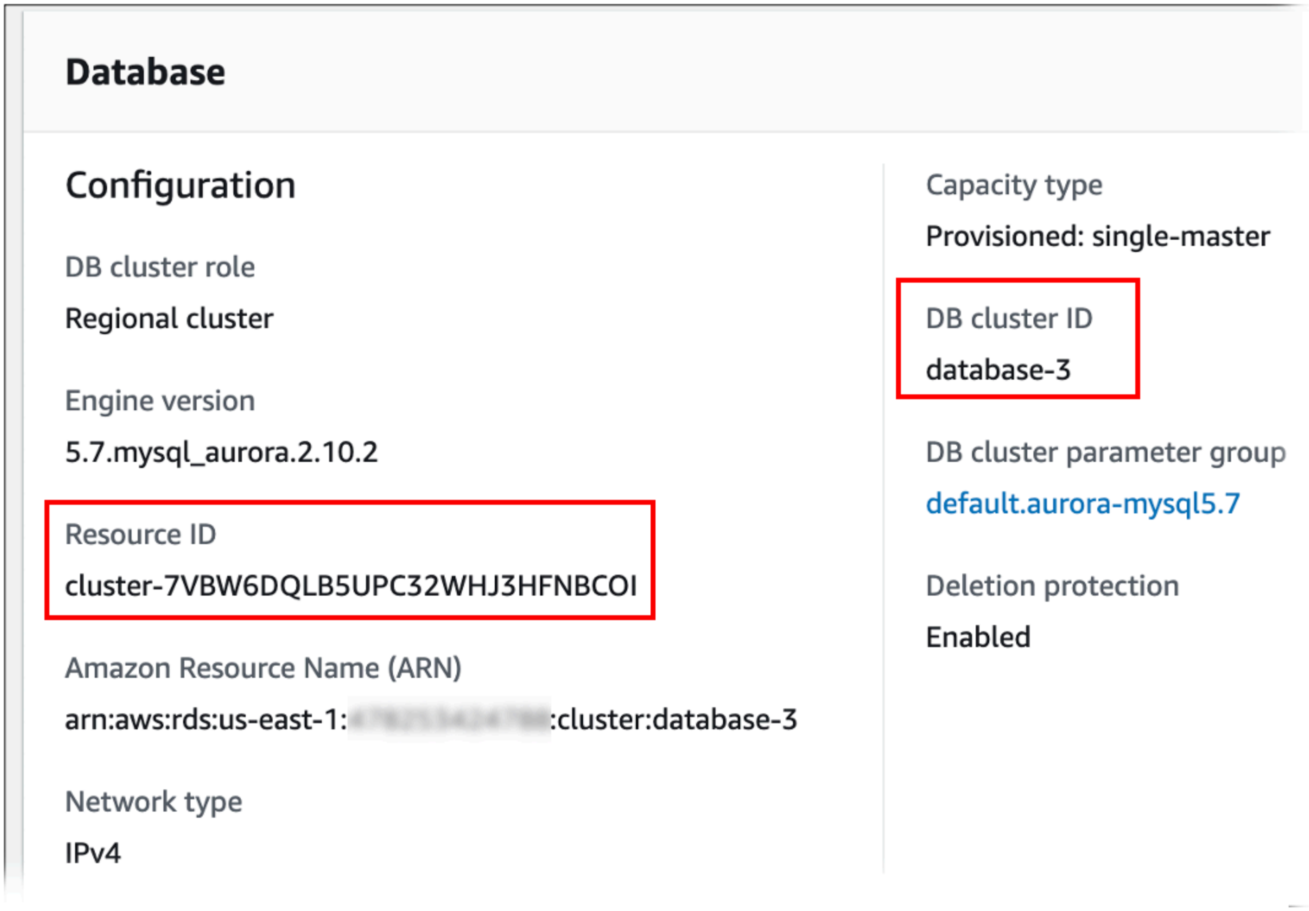
- `rds:DeleteDBCluster`
- `rds:DeleteDBInstance`
- `rds:DeleteDBClusterEndpoint`

Aurora provisions and modifies resources in the staging environment on your behalf. These resources include DB instances that use an internally defined naming convention. Therefore, attached IAM policies can't contain partial resource name patterns such as `my-db-prefix-*`. Only wildcards (*) are supported. In general, we recommend using resource tags and other supported attributes to control access to these resources, rather than wildcards. For more information, see Actions, resources, and condition keys for Amazon RDS.

Considerations for blue/green deployments

Amazon RDS tracks resources in blue/green deployments with the `DbiResourceId` and `DbClusterResourceId` of each resource. This resource ID is an AWS Region-unique, immutable identifier for the resource.

The *resource* ID is separate from the DB *cluster* ID:



Database

Configuration

DB cluster role
Regional cluster

Engine version
5.7.mysql_aurora.2.10.2

Resource ID
cluster-7VBW6DQLB5UPC32WHJ3HFNBCOI

Amazon Resource Name (ARN)
arn:aws:rds:us-east-1:[redacted]:cluster:database-3

Network type
IPv4

Capacity type
Provisioned: single-master

DB cluster ID
database-3

DB cluster parameter group
default.aurora-mysql5.7

Deletion protection
Enabled

The name (cluster ID) of a resource changes when you switch over a blue/green deployment, but each resource keeps the same resource ID. For example, a DB cluster identifier might have been `mycluster` in the blue environment. After switchover, the same DB cluster might be renamed to `mycluster-old1`. However, the resource ID of the DB cluster doesn't change during switchover. So, when the green resources are promoted to be the new production resources, their resource IDs don't match the blue resource IDs that were previously in production.

After switching over a blue/green deployment, consider updating the resource IDs to those of the newly promoted production resources for integrated features and services that you used with the production resources. Specifically, consider the following updates:

- If you perform filtering using the RDS API and resource IDs, adjust the resource IDs used in filtering after switchover.
- If you use CloudTrail for auditing resources, adjust the consumers of the CloudTrail to track the new resource IDs after switchover. For more information, see Monitoring Amazon Aurora API calls in AWS CloudTrail.
- If you use Database Activity Streams for resources in the blue environment, adjust your application to monitor database events for the new stream after switchover. For more information, see Supported Regions and Aurora DB engines for database activity streams.
- If you use the Performance Insights API, adjust the resource IDs in calls to the API after switchover. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.

  You can monitor a database with the same name after switchover, but it doesn't contain the data from before the switchover.
- If you use resource IDs in IAM policies, make sure you add the resource IDs of the newly promoted resources when necessary. For more information, see Identity and access management for Amazon Aurora.
- If you have IAM roles associated with your DB cluster, make sure to reassociate them after switchover. Attached roles aren't automatically copied to the green environment.
- If you authenticate to your DB cluster using IAM database authentication, make sure that the IAM policy used for database access has both the blue and the green databases listed under the `Resource` element of the policy. This is required in order to connect to the green database after switchover. For more information, see the section called “Creating and using an IAM policy for IAM database access”.
- If you want to restore a manual DB cluster snapshot for a DB cluster that was part of a blue/green deployment, make sure you restore the correct DB cluster snapshot by examining the time when the snapshot was taken. For more information, see Restoring from a DB cluster snapshot.
- Amazon Aurora creates the green environment by *cloning* the underlying Aurora storage volume in the blue environment. The green cluster volume only stores incremental changes made to the green environment. If you delete the DB cluster in the blue environment, the size of the underlying Aurora storage volume in the green environment grows to the full size. For more information, see the section called “Cloning a volume for an Aurora DB cluster”.
- When you add a DB instance to the DB cluster in the green environment of a blue/green deployment, the new DB instance won't replace a DB instance in the blue environment when

you switch over. However, the new DB instance is retained in the DB cluster and becomes a DB instance in the new production environment.

- When you delete a DB instance in the DB cluster in the green environment of a blue/green deployment, you can't create a new DB instance to replace it in the blue/green deployment.

  If you create a new DB instance with the same name and ARN as the deleted DB instance, it has a different `DbiResourceId`, so it isn't part of the green environment.

  The following behavior results if you delete a DB instance in the DB cluster in the green environment:

  - If the DB instance in the blue environment with the same name exists, it won't be switched over to the DB instance in the green environment. This DB instance won't be renamed by appending `-old`*`n`* to the DB instance name.
  - Any application that points to the DB instance in the blue environment continues to use the same DB instance after switchover.

Best practices for blue/green deployments

The following are best practices for blue/green deployments:

General best practices

- Thoroughly test the Aurora DB cluster in the green environment before switching over.
- Keep your databases in the green environment read only. We recommend that you enable write operations on the green environment with caution because they can result in replication conflicts. They can also result in unintended data in the production databases after switchover.
- When using a blue/green deployment to implement schema changes, make only replication-compatible changes.

  For example, you can add new columns at the end of a table without disrupting replication from the blue deployment to the green deployment. However, schema changes, such as renaming columns or renaming tables, break replication to the green deployment.

  For more information about replication-compatible changes, see Replication with Differing Table Definitions on Source and Replica in the MySQL documentation and Restrictions in the PostgreSQL logical replication documentation.

- Use the cluster endpoint, reader endpoint, or custom endpoint for all connections in both environments. Don't use instance endpoints or custom endpoints with static or exclusion lists.
- When you switch over a blue/green deployment, follow the switchover best practices. For more information, see the section called "Switchover best practices".

Aurora PostgreSQL best practices

- Monitor the Aurora PostgreSQL logical replication write-through cache and make adjustments to the cache buffer if necessary. For more information, see the section called "Monitoring the logical replication write-through cache".
- If your database has sufficient freeable memory, increase the value of the `logical_decoding_work_mem` DB parameter in the blue environment. Doing so allows for less decoding on disk and instead uses memory. You can monitor freeable memory with the `FreeableMemory` CloudWatch metric. For more information, see the section called "CloudWatch metrics for Aurora".
- Update all of your PostgreSQL extensions to the latest version before you create a blue/green deployment. For more information, see the section called "Upgrading PostgreSQL extensions".
- If you're using the `aws_s3` extension, make sure to give the green DB cluster access to Amazon S3 through an IAM role after the green environment is created. This allows the import and export commands to continue functioning after switchover. For instructions, see the section called "Setting up access to an Amazon S3 bucket".
- If you specify a higher engine version for the green environment, run the `ANALYZE` operation on all databases to refresh the `pg_statistic` table. Optimizer statistics aren't transferred during a major version upgrade, so you must regenerate all statistics to avoid performance issues. For additional best practices during major version upgrades, see the section called "How to perform a major version upgrade".
- Avoid configuring triggers as `ENABLE REPLICA` or `ENABLE ALWAYS` if the trigger is used on the source to manipulate data. Otherwise, the replication system propagates changes and executes the trigger, which leads to duplication.
- Long-running transactions can cause significant replica lag. To reduce replica lag, consider doing the following:
  - Reduce long-running transactions that can be delayed until after the green environment catches up to the blue environment.

- Initiate a manual vacuum freeze operation on busy tables prior to creating the blue/green deployment.
- For PostgreSQL version 12 and higher, disable the `index_cleanup` parameter on large or busy tables to increase the rate of normal maintenance on blue databases.

- Slow replication can cause senders and receivers to restart often, which delays synchronization. To ensure that they remain active, disable timeouts by setting the `wal_sender_timeout` parameter to `0` in the blue environment, and the `wal_receiver_timeout` parameter to `0` in the green environment.

Limitations for blue/green deployments

The following limitations apply to blue/green deployments.

Topics

General limitations for blue/green deployments

The following general limitations apply to blue/green deployments:

- Aurora MySQL versions 2.08 and 2.09 aren't supported as upgrade source or target versions.
- You can't stop and start a cluster that is part of a blue/green deployment.
- Blue/green deployments don't support managing master user passwords with AWS Secrets Manager.
- If you create a blue/green deployment from an Aurora MySQL source DB cluster that has backtrack enabled, the green DB cluster is created without backtracking support. This is because backtracking doesn't work with binary log (binlog) replication, which is required for blue/green deployments. For more information, see the section called “Backtracking a DB cluster”.

  If you attempt to force a backtrack on the blue DB cluster, the blue/green deployment breaks and switchover is blocked.

- For Aurora MySQL, the source DB cluster can't contain any databases named `tmp`. Databases with this name will not be copied to the green environment.
- For Aurora PostgreSQL, unlogged tables aren't replicated to the green environment unless the `rds.logically_replicate_unlogged_tables` parameter is set to 1 on the blue DB cluster. We recommend that you don't modify this parameter value after you create a blue/green deployment to avoid possible replication errors on unlogged tables.
- For Aurora PostgreSQL, the blue environment DB cluster can't be a self-managed logical source (publisher) or replica (subscriber). For Aurora MySQL, the blue environment DB cluster can't be an external binlog replica.
- During switchover, the blue and green environments can't have zero-ETL integrations with Amazon Redshift. You must delete the integration first and switch over, then recreate the integration.
- The Event Scheduler (`event_scheduler` parameter) must be disabled on the green environment when you create a blue/green deployment. This prevents events from being generated in the green environment and causing inconsistencies.
- Any Aurora Auto Scaling policies that are defined on the blue DB cluster aren't copied to the green environment.
- Blue/green deployments don't support the AWS JDBC Driver for MySQL. For more information, see Known Limitations on GitHub.
- Blue/green deployments aren't supported for the following features:
  - Amazon RDS Proxy
  - Cross-Region read replicas
  - Aurora Serverless v1 DB clusters
  - DB clusters that are part of an Aurora global database
  - Babelfish for Aurora PostgreSQL
  - AWS CloudFormation

PostgreSQL extension limitations for blue/green deployments

The following limitations apply to PostgreSQL extensions:

- The `pg_partman` extension must be disabled on the blue environment when you create a blue/green deployment. The extension performs DDL operations such as `CREATE TABLE`, which break logical replication from the blue environment to the green environment.

- The pg_cron extension must remain disabled on all green databases after the blue/green deployment is created. The extension has background workers that run as superuser and bypass the read-only setting of the green environment, which might cause replication conflicts.
- The apg_plan_mgmt extension must have the apg_plan_mgmt.capture_plan_baselines parameter set to off on all green databases to avoid primary key conflicts if an identical plan is captured in the blue environment. For more information, see the section called "Overview of Aurora PostgreSQL query plan management".

  If you want to capture execution plans in Aurora Replicas, you must provide the blue DB cluster endpoint when calling the apg_plan_mgmt.create_replica_plan_capture function. This ensures that plan captures continue to work after switchover. For more information, see the section called "Capturing Aurora PostgreSQL execution plans in Replicas".
- If the blue DB cluster is configured as the foreign server of a foreign data wrapper (FDW) extension, you must use the cluster endpoint name instead of IP addresses. This allows the configuration to remain functional after switchover.
- The pglogical and pg_active extensions must be disabled on the blue environment when you create a blue/green deployment. After you promote the green environment to be the new production environment, you can enable the extensions again. In addition, the blue database can't be a logical subscriber of an external instance.
- If you're using the pgAudit extension, it must remain in the shared libraries (shared_preload_libraries) on the custom DB parameter groups for both the blue and the green DB instances. For more information, see the section called "Setting up the pgAudit extension".

Limitations for changes in blue/green deployments

The following are limitations for changes in a blue/green deployment:

- You can't change an unencrypted DB cluster into an encrypted DB cluster.
- You can't change an encrypted DB cluster into an unencrypted DB cluster.
- You can't change a blue environment DB cluster to a higher engine version than its corresponding green environment DB cluster.
- The resources in the blue environment and green environment must be in the same AWS account.

- If the blue environment contains any Aurora Auto Scaling policies, these policies aren't copied over to the green environment. You must manually re-add the policies to the green environment.

PostgreSQL logical replication limitations for blue/green deployments

Blue/green deployments use logical replication to keep the staging environment in sync with the production environment. PostgreSQL has certain restrictions related to logical replication, which translate to limitations when creating blue/green deployments for Aurora PostgreSQL DB clusters.

The following table describes logical replication limitations that apply to blue/green deployments for Aurora PostgreSQL.

Limitation	Explanation
Data definition language (DDL) statements, such as `CREATE TABLE` and `CREATE SCHEMA`, aren't replicated from the blue environment to the green environment.	If Aurora detects a DDL change in the blue environment, your green databases enter a state of **Replication degraded**. You receive an event notifying you that DDL changes in the blue environment can't be replicated to the green environment. You must delete the blue/green deployment and all green databases, then recreate it. Otherwise, you won't be able to switch over the blue/green deployment.
`NEXTVAL` operations on sequence objects aren't synchronized between the blue environment and the green environment.	During switchover, Aurora increments sequence values in the green environment to match those in the blue environment. If you have thousands of sequences, this can delay switchover.

Limitation	Explanation
Creation or modification of large objects in the blue environment aren't replicated to the green environment.	If Aurora detects the creation or modification of large objects in the blue environment that are stored in the `pg_largeobject` system table, your green databases enter a state of **Replication degraded**. Aurora generates an event notifying you that large object changes in the blue environment can't be replicated to the green environment. You must delete the blue/green deployment and all green databases, then recreate it. Otherwise, you won't be able to switch over the blue/green deployment.
Materialized views aren't automatically refreshed on the green environment.	Refreshing materialized views in the blue environment doesn't refresh them in the green environment. After switchover, you can schedule a refresh of materialized views.
UPDATE and DELETE operations aren't permitted on tables that don't have a primary key.	Before you create a blue/green deployment, make sure that all tables in the DB cluster have a primary key.

For more information, see Restrictions in the PostgreSQL logical replication documentation.

Creating a blue/green deployment

When you create a blue/green deployment, you specify the DB cluster to copy in the deployment. The DB cluster you choose is the production DB cluster, and it becomes the DB cluster in the blue environment. RDS copies the blue environment's topology to a staging area, along with its configured features. The DB cluster is copied to the green environment, and RDS configures replication from the DB cluster in the blue environment to the DB cluster in the green environment. RDS also copies all of the DB instances in the DB cluster.

Topics

Preparing for a blue/green deployment

There are certain steps you must take before you create a blue/green deployment, depending on the engine that your Aurora DB cluster is running.

Topics

Preparing an Aurora MySQL DB cluster for a blue/green deployment

Before you create a blue/green deployment for an Aurora MySQL DB cluster, the cluster must be associated with a custom DB cluster parameter group with binary logging (`binlog_format`) turned on. Binary logging is required for replication from the blue environment to the green environment. While any binlog format works, we recommend `ROW` to reduce the risk of replication inconsistencies. For information about creating a custom DB cluster parameter group and setting parameters, see the section called "Working with DB cluster parameter groups".

> **Note**
>
> Enabling binary logging increases the number of write disk I/O operations to the DB cluster. You can monitor IOPS usage with the `VolumeWriteIOPs` CloudWatch metric.

After you enable binary logging, make sure to reboot the DB cluster so that your changes take effect. Blue/green deployments *require* that the writer instance be in sync with the DB cluster parameter group, otherwise creation fails. For more information, see Rebooting a DB instance within an Aurora cluster.

In addition, we recommend changing the binary log retention period to a value other than NULL to prevent binary log files from being purged. For more information, see the section called “Configuring”.

Preparing an Aurora PostgreSQL DB cluster for a blue/green deployment

Before you create a blue/green deployment for an Aurora PostgreSQL DB cluster, make sure to do the following:

- Associate the cluster with a custom DB cluster parameter group that has logical replication (`rds.logical_replication`) enabled. Logical replication is required for replication from the blue environment to the green environment.

  When you enable logical replication, you also need to tune certain cluster parameters, such as `max_replication_slots`, `max_logical_replication_workers`, and `max_worker_processes`. For instructions to enable logical replication and tune these parameters, see the section called “Setting up logical replication”.

  In addition, make sure that the `synchronous_commit` parameter is set to `on`.

  After you configure the required parameters, make sure to reboot the DB cluster so that your changes take effect. Blue/green deployments *require* that the writer instance be in sync with the DB cluster parameter group, otherwise creation fails. For more information, see Rebooting a DB instance within an Aurora cluster.
- Make sure that your DB cluster is running a version of Aurora PostgreSQL that's compatible with Blue/Green Deployments. For a list of compatible versions, see the section called “Blue/Green Deployments with Aurora PostgreSQL”.
- Make sure that all tables in the DB cluster have a primary key. PostgreSQL logical replication doesn't allow UPDATE or DELETE operations on tables that don't have a primary key.
- If you're using triggers, make sure they don't interfere with the creating, updating, and dropping of `pg_catalog.pg_publication`, `pg_catalog.pg_subscription`, and `pg_catalog.pg_replication_slots` objects whose names start with 'rds'.

Specifying changes when creating a blue/green deployment

You can make the following changes to the DB cluster in the green environment when you create the blue/green deployment.

You can make other modifications to the DB cluster and its DB instances in the green environment after it is deployed. For example, you might make schema changes to your database.

For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

Specify a higher engine version

You can specify a higher engine version if you want to test a DB engine upgrade. Upon switchover, the database is upgraded to the major or minor DB engine version that you specify.

Specify a different DB parameter group

Specify a DB cluster parameter group that is different from the one used by the DB cluster. You can test how parameter changes affect the DB cluster in the green environment or specify a parameter group for a new major DB engine version in the case of an upgrade.

If you specify a different DB cluster parameter group, the specified parameter group is associated with the DB cluster in the green environment. If you don't specify a different DB cluster parameter group, the DB cluster in the green environment is associated with the same parameter group as the blue DB cluster.

Creating a blue/green deployment

You can create a blue/green deployment using the AWS Management Console, the AWS CLI, or the RDS API.

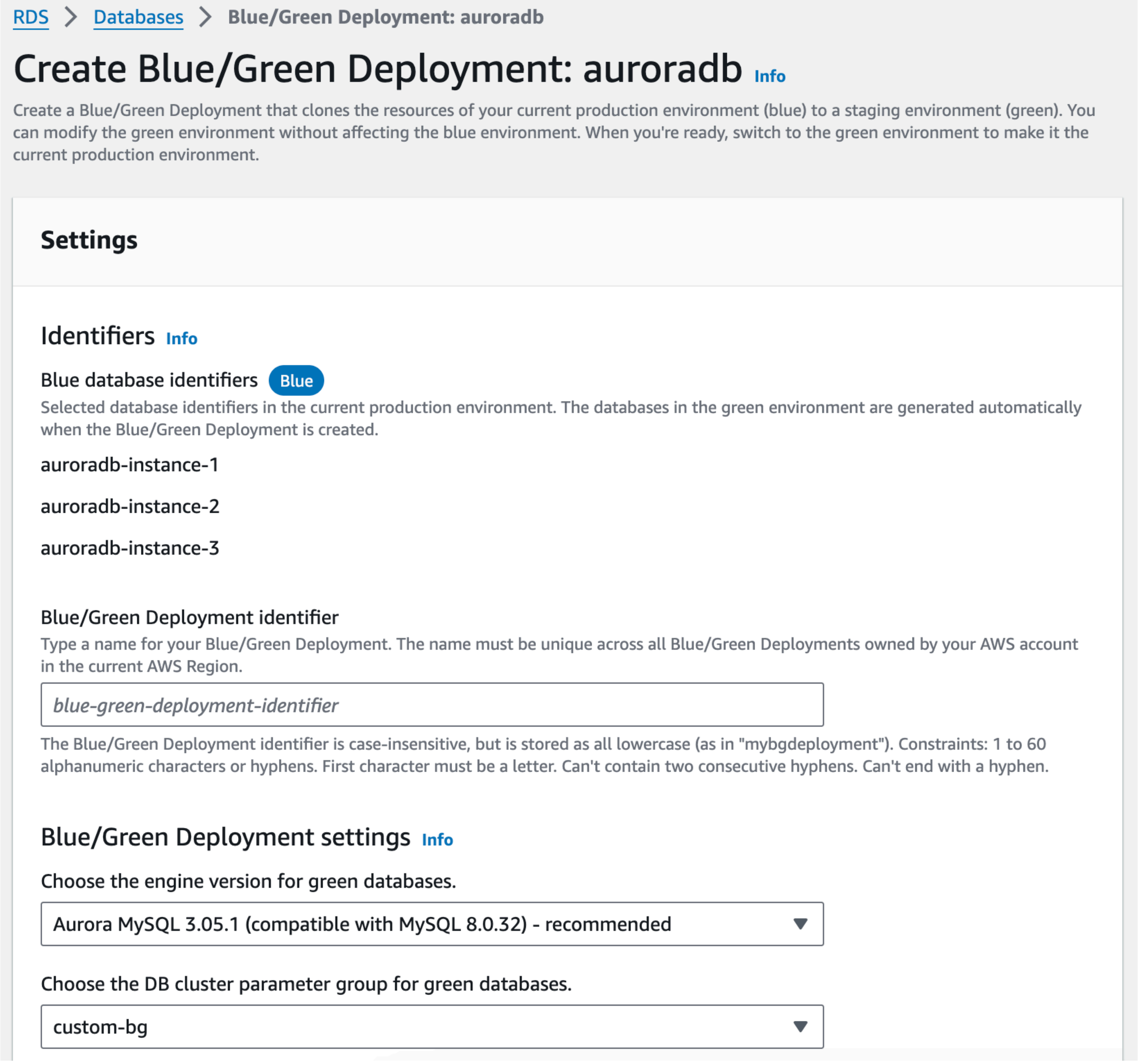
Console

To create a blue/green deployment

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to copy to a green environment.
3. Choose **Actions**, **Create Blue/Green Deployment**.

   If you choose an Aurora PostgreSQL DB cluster, review and acknowledge the logical replication limitations. For more information, see the section called “PostgreSQL logical replication limitations”.

   The **Create Blue/Green Deployment** page appears.



RDS > Databases > Blue/Green Deployment: auroradb

Create Blue/Green Deployment: auroradb Info

Create a Blue/Green Deployment that clones the resources of your current production environment (blue) to a staging environment (green). You can modify the green environment without affecting the blue environment. When you're ready, switch to the green environment to make it the current production environment.

Settings

Identifiers Info

Blue database identifiers Blue

Selected database identifiers in the current production environment. The databases in the green environment are generated automatically when the Blue/Green Deployment is created.

auroradb-instance-1

auroradb-instance-2

auroradb-instance-3

Blue/Green Deployment identifier

Type a name for your Blue/Green Deployment. The name must be unique across all Blue/Green Deployments owned by your AWS account in the current AWS Region.

blue-green-deployment-identifier

The Blue/Green Deployment identifier is case-insensitive, but is stored as all lowercase (as in "mybgdeployment"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

Blue/Green Deployment settings Info

Choose the engine version for green databases.

Aurora MySQL 3.05.1 (compatible with MySQL 8.0.32) - recommended ▼

Choose the DB cluster parameter group for green databases.

custom-bg ▼

4. Review the blue database identifiers. Make sure that they match the DB instances that you expect in the blue environment. If they don't, choose **Cancel**.

5. For **Blue/Green Deployment identifier**, enter a name for your blue/green deployment.

6. In the remaining sections, specify the settings for the green environment. For information about each setting, see the section called “Available settings”.

   You can make other modifications to the databases in the green environment after it is deployed.

7. Choose **Create staging environment**.

AWS CLI

To create a blue/green deployment using the AWS CLI, use the create-blue-green-deployment command. For information about each option, see the section called “Available settings”.

Example

For Linux, macOS, or Unix:

```
aws rds create-blue-green-deployment \
    --blue-green-deployment-name aurora-blue-green-deployment \
    --source arn:aws:rds:us-east-2:123456789012:cluster:auroradb \
    --target-engine-version 8.0 \
    --target-db-cluster-parameter-group-name mydbclusterparametergroup
```

For Windows:

```
aws rds create-blue-green-deployment ^
    --blue-green-deployment-name aurora-blue-green-deployment ^
    --source arn:aws:rds:us-east-2:123456789012:cluster:auroradb ^
    --target-engine-version 8.0 ^
    --target-db-cluster-parameter-group-name mydbclusterparametergroup
```

RDS API

To create a blue/green deployment by using the Amazon RDS API, use the `CreateBlueGreenDeployment` operation. For information about each option, see the section called “Available settings”.

Settings for creating blue/green deployments

The following table explains the settings that you can choose when you create a blue/green deployment. For more information about the AWS CLI options, see create-blue-green-deployment. For more information about the RDS API parameters, see CreateBlueGreenDeployment.

Console setting	Setting description	CLI option and RDS API parameter
Blue/Green Deployment identifier	A name for the blue/green deployment.	**CLI option:** `--blue-green-deployment-name`

Console setting	Setting description	CLI option and RDS API parameter
		API parameter: `BlueGreenDeploymentName`

Blue database identifier	The identifier of the cluster that you want to copy to the green environment. When using the CLI or API, specify the cluster Amazon Resource Name (ARN).	**CLI option:**
		`--source`
		API parameter:
		`Source`

DB cluster parameter group for green databases	A parameter group to associate with the databases in the green environment.	**CLI option:**
		`--target-db-cluster-parameter-group-name`
		API parameter:
		`TargetDBClusterParameterGroupName`

Engine version for green databases	Upgrade the cluster in the green environment to the specified DB engine version.	**CLI option:**
		`--target-engine-version`
		RDS API parameter:
		`TargetEngineVersion`

Viewing a blue/green deployment

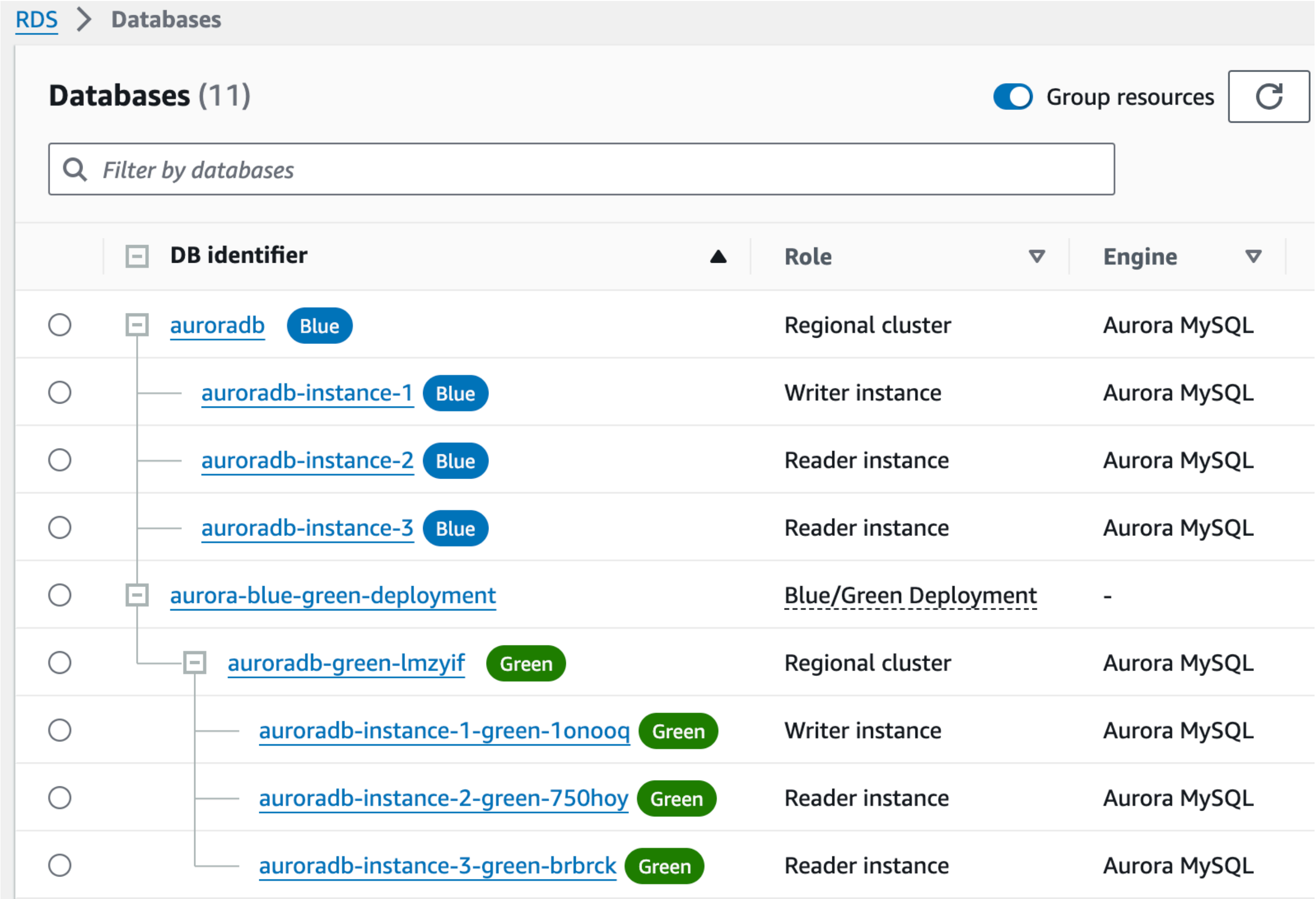
You can view the details about a blue/green deployment using the AWS Management Console, the AWS CLI, or the RDS API.

You can also view and subscribe to events for information about a blue/green deployment. For more information, see Blue/green deployment events.

Console

To view the details about a blue/green deployment

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then find the blue/green deployment in the list.



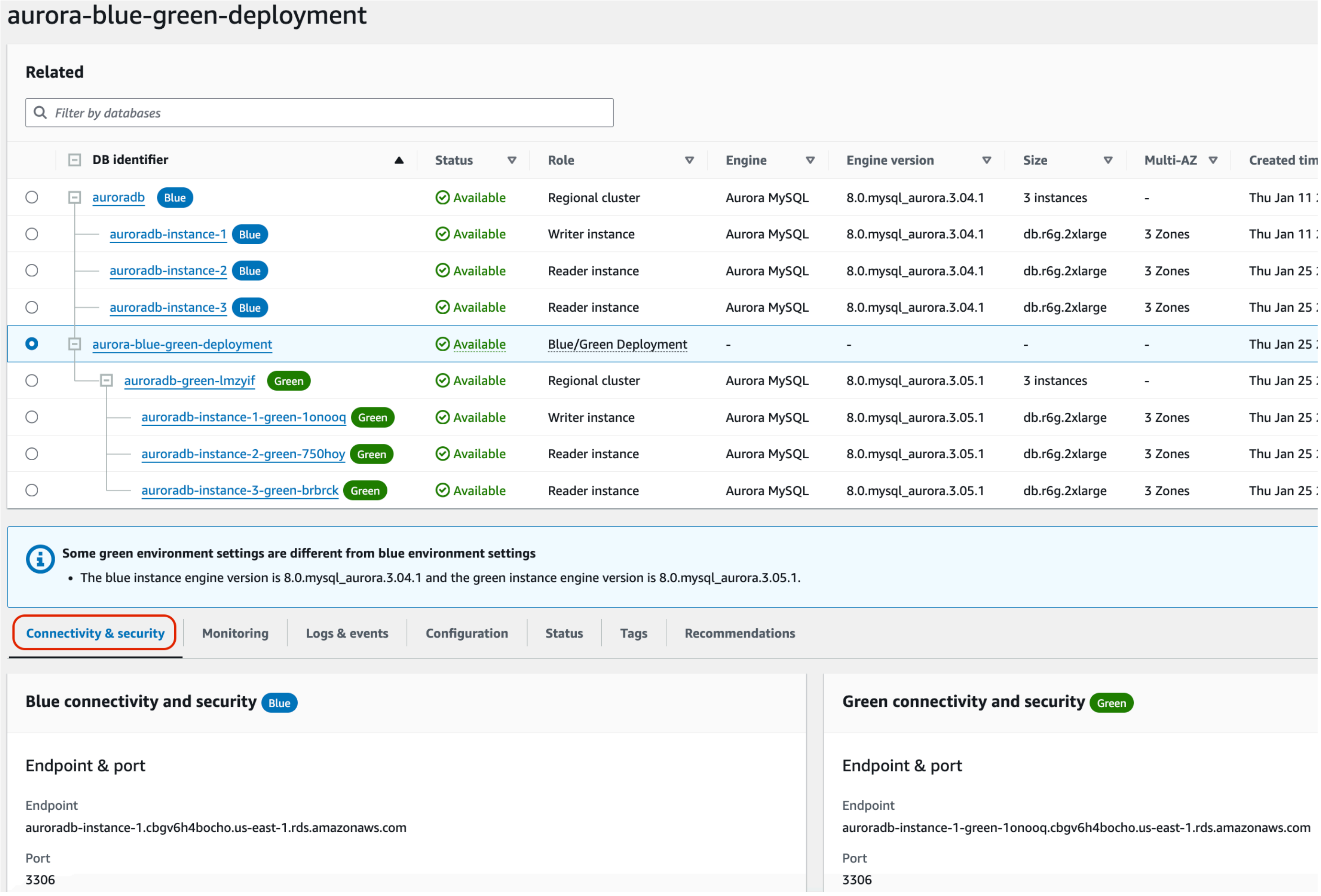
	DB identifier	Role	Engine
○	auroradb Blue	Regional cluster	Aurora MySQL
○	auroradb-instance-1 Blue	Writer instance	Aurora MySQL
○	auroradb-instance-2 Blue	Reader instance	Aurora MySQL
○	auroradb-instance-3 Blue	Reader instance	Aurora MySQL
○	aurora-blue-green-deployment	Blue/Green Deployment	-
○	auroradb-green-lmzyif Green	Regional cluster	Aurora MySQL
○	auroradb-instance-1-green-1onooq Green	Writer instance	Aurora MySQL
○	auroradb-instance-2-green-750hoy Green	Reader instance	Aurora MySQL
○	auroradb-instance-3-green-brbrck Green	Reader instance	Aurora MySQL

The **Role** value for the blue/green deployment is **Blue/Green Deployment**.

3. Choose the name of blue/green deployment that you want to view to display its details.

   Each tab has a section for the blue deployment and a section for the green deployment. For example, on the **Configuration** tab, the DB engine version might be different in the blue environment and in the green environment if you're upgrading the DB engine version in the green environment.

The following image shows an example of the **Connectivity & security** tab:



aurora-blue-green-deployment

Related

Filter by databases

	DB identifier	Status	Role	Engine	Engine version	Size	Multi-AZ	Created tim
○	auroradb Blue	✓ Available	Regional cluster	Aurora MySQL	8.0.mysql_aurora.3.04.1	3 instances	-	Thu Jan 11
○	auroradb-instance-1 Blue	✓ Available	Writer instance	Aurora MySQL	8.0.mysql_aurora.3.04.1	db.r6g.2xlarge	3 Zones	Thu Jan 11
○	auroradb-instance-2 Blue	✓ Available	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.04.1	db.r6g.2xlarge	3 Zones	Thu Jan 25
○	auroradb-instance-3 Blue	✓ Available	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.04.1	db.r6g.2xlarge	3 Zones	Thu Jan 25
●	aurora-blue-green-deployment	✓ Available	Blue/Green Deployment	-	-	-	-	Thu Jan 25
○	auroradb-green-lmzyif Green	✓ Available	Regional cluster	Aurora MySQL	8.0.mysql_aurora.3.05.1	3 instances	-	Thu Jan 25
○	auroradb-instance-1-green-1onooq Green	✓ Available	Writer instance	Aurora MySQL	8.0.mysql_aurora.3.05.1	db.r6g.2xlarge	3 Zones	Thu Jan 25
○	auroradb-instance-2-green-750hoy Green	✓ Available	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.05.1	db.r6g.2xlarge	3 Zones	Thu Jan 25
○	auroradb-instance-3-green-brbrck Green	✓ Available	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.05.1	db.r6g.2xlarge	3 Zones	Thu Jan 25

Some green environment settings are different from blue environment settings

- The blue instance engine version is 8.0.mysql_aurora.3.04.1 and the green instance engine version is 8.0.mysql_aurora.3.05.1.

Blue connectivity and security Blue

Endpoint & port

Endpoint
auroradb-instance-1.cbgv6h4bocho.us-east-1.rds.amazonaws.com

Port
3306

Green connectivity and security Green

Endpoint & port

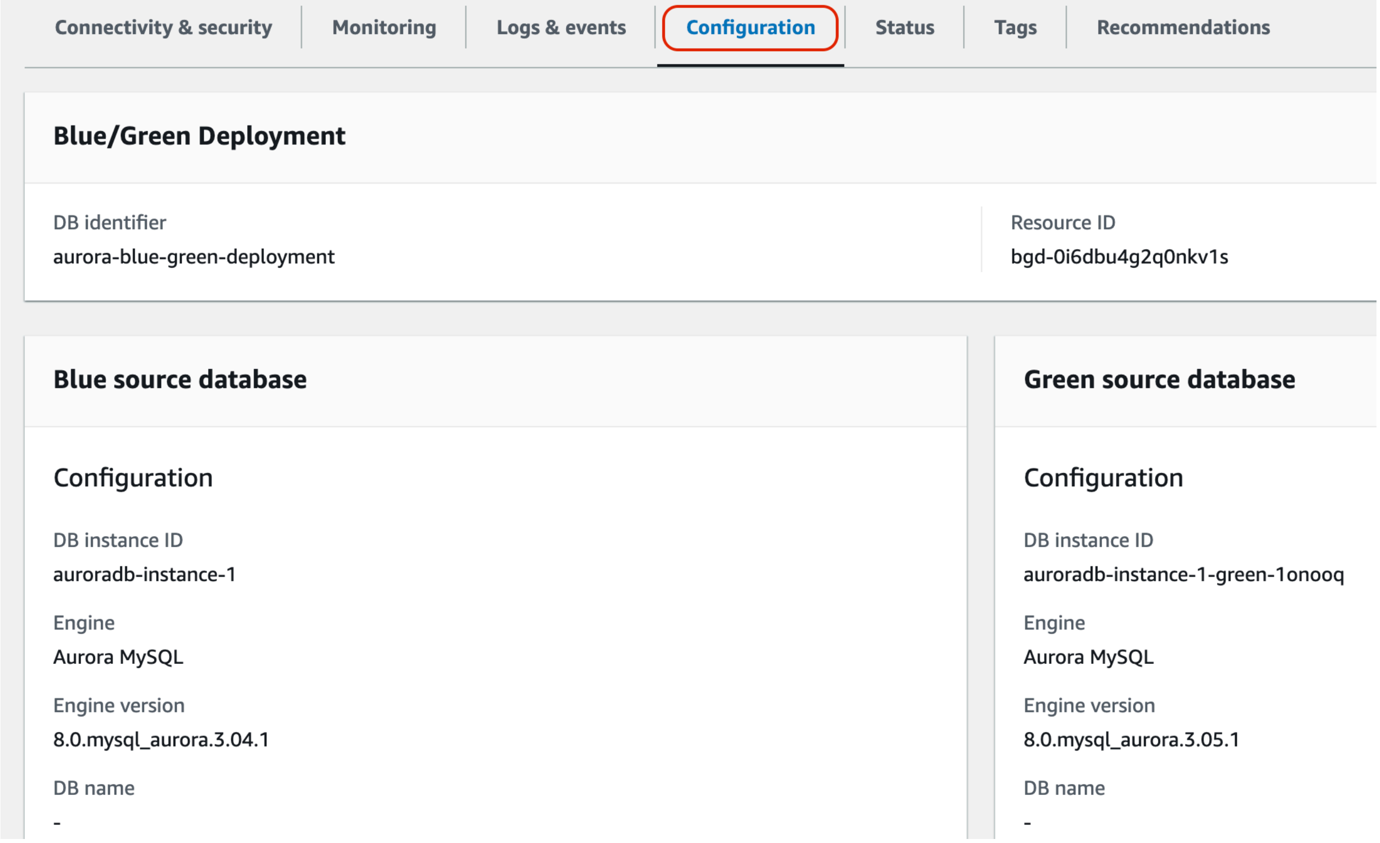
Endpoint
auroradb-instance-1-green-1onooq.cbgv6h4bocho.us-east-1.rds.amazonaws.com

Port
3306

The **Connectivity & security** tab also includes a section called **Replication**, which shows the current state of logical replication and replica lag between the blue and green environments. If the replication state is `Replicating`, the blue/green deployment is replicating successfully.

For Aurora PostgreSQL blue/green deployments, the replication state can change to `Replication degraded` if you make unsupported DDL or large object changes in the blue environment. For more information, see the section called "PostgreSQL logical replication limitations".

The following image shows an example of the **Configuration** tab:



Blue/Green Deployment

DB identifier
aurora-blue-green-deployment

Resource ID
bgd-0i6dbu4g2q0nkv1s

Blue source database

Configuration

DB instance ID
auroradb-instance-1

Engine
Aurora MySQL

Engine version
8.0.mysql_aurora.3.04.1

DB name
-

Green source database

Configuration

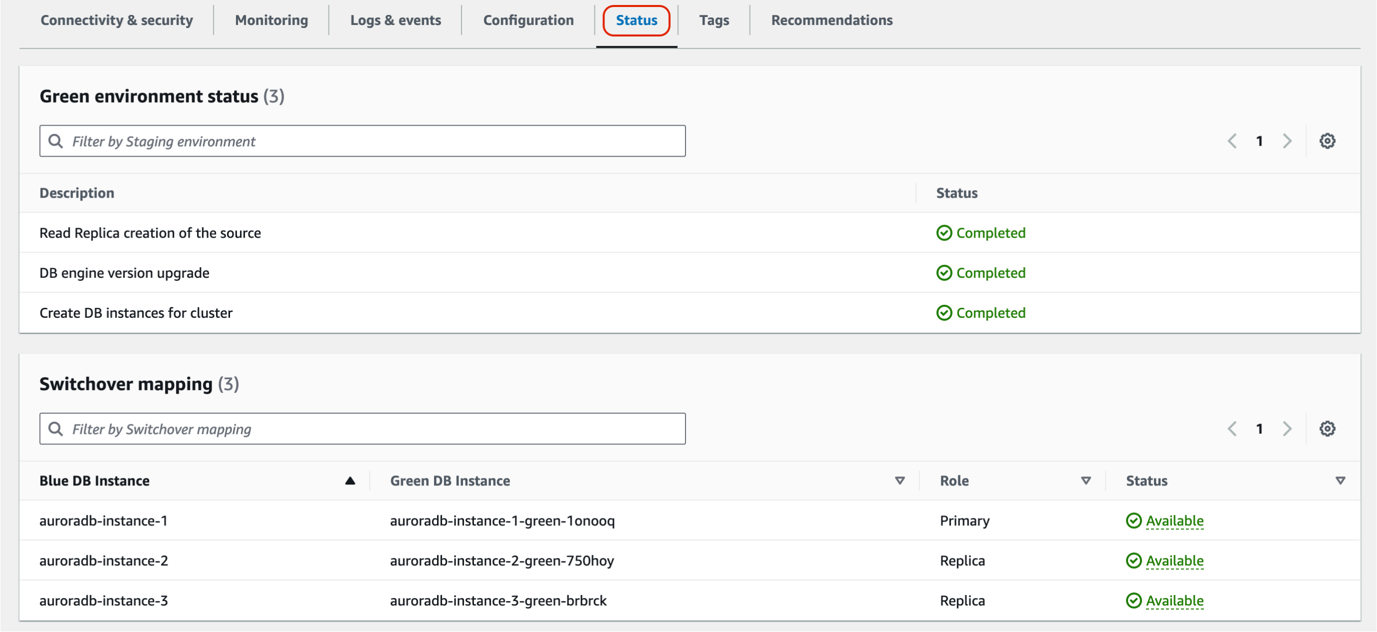
DB instance ID
auroradb-instance-1-green-1onooq

Engine
Aurora MySQL

Engine version
8.0.mysql_aurora.3.05.1

DB name
-

The following image shows an example of the **Status** tab:



Green environment status (3)

Description	Status
Read Replica creation of the source	✓ Completed
DB engine version upgrade	✓ Completed
Create DB instances for cluster	✓ Completed

Switchover mapping (3)

Blue DB Instance ▲	Green DB Instance ▽	Role ▽	Status ▽
auroradb-instance-1	auroradb-instance-1-green-1onooq	Primary	✓ Available
auroradb-instance-2	auroradb-instance-2-green-750hoy	Replica	✓ Available
auroradb-instance-3	auroradb-instance-3-green-brbrck	Replica	✓ Available

AWS CLI

To view the details about a blue/green deployment by using the AWS CLI, use the describe-blue-green-deployments command.

Example View the details about a blue/green deployment by filtering on its name

When you use the describe-blue-green-deployments command, you can filter on the `--blue-green-deployment-name`. The following example shows the details for a blue/green deployment named *`my-blue-green-deployment`*.

```
aws rds describe-blue-green-deployments --filters Name=blue-green-deployment-
name,Values=my-blue-green-deployment
```

Example View the details about a blue/green deployment by specifying its identifier

When you use the describe-blue-green-deployments command, you can specify the `--blue-green-deployment-identifier`. The following example shows the details for a blue/green deployment with the identifier *`bgd-1234567890abcdef`*.

```
aws rds describe-blue-green-deployments --blue-green-deployment-
identifier bgd-1234567890abcdef
```

RDS API

To view the details about a blue/green deployment by using the Amazon RDS API, use the `DescribeBlueGreenDeployments` operation and specify the `BlueGreenDeploymentIdentifier`.

Switching a blue/green deployment

A *switchover* promotes the DB cluster, including its DB instances, in the green environment to be the production DB cluster. Before you switch over, production traffic is routed to the cluster in the blue environment. After you switch over, production traffic is routed to the DB cluster in the green environment.

Topics

- Switchover timeout
- Switchover guardrails

- Switchover actions
- Switchover best practices
- Verifying CloudWatch metrics before switchover
- Monitoring replica lag prior to switchover
- Switching over a blue/green deployment
- After switchover

Switchover timeout

You can specify a switchover timeout period between 30 seconds and 3,600 seconds (one hour). If the switchover takes longer than the specified duration, then any changes are rolled back and no changes are made to either environment. The default timeout period is 300 seconds (five minutes).

Switchover guardrails

When you start a switchover, Amazon RDS runs some basic checks to test the readiness of the blue and green environments for switchover. These checks are known as *switchover guardrails*. These switchover guardrails prevent a switchover if the environments aren't ready for it. Therefore, they avoid longer than expected downtime and prevent the loss of data between the blue and green environments that might result if the switchover started.

Amazon RDS runs the following guardrail checks on the green environment:

- **Replication health** – Checks if green DB cluster replication status is healthy. The green DB cluster is a replica of the blue DB cluster.
- **Replication lag** – Checks if the replica lag of the green DB cluster is within allowable limits for switchover. The allowable limits are based on the specified timeout period. Replica lag indicates how far the green DB cluster is lagging behind its blue DB cluster. For more information, see the section called “Diagnosing and resolving lag between read replicas” for Aurora MySQL and the section called “Monitoring replication” for Aurora PostgreSQL.
- **Active writes** – Makes sure there are no active writes on the green DB cluster.

Amazon RDS runs the following guardrail checks on the blue environment:

- **External replication** – For Aurora PostgreSQL, makes sure that the blue environment isn't a self-managed logical source (publisher) or replica (subscriber). If it is, we recommend that you drop the self-managed replication slots and subscriptions across all databases in the blue environment, proceed with switchover, then recreate them to resume replication. For Aurora MySQL, checks whether the blue database isn't an external binlog replica. If it is, make sure that it is not actively replicating.

- **Long-running active writes** – Makes sure there are no long-running active writes on the blue DB cluster because they can increase replica lag.
- **Long-running DDL statements** – Makes sure there are no long-running DDL statements on the blue DB cluster because they can increase replica lag.
- **Unsupported PostgreSQL changes** – For Aurora PostgreSQL DB clusters, makes sure that no DDL changes and no additions or modifications of large objects have been performed on the blue environment. For more information, see the section called “PostgreSQL logical replication limitations”.

  If Amazon RDS detects unsupported PostgreSQL changes, it changes the replication state to `Replication degraded` and notifies you that switchover is not available for the blue/green deployment. To proceed with switchover, we recommend that you delete and recreate the blue/green deployment and all green databases. To do so, choose **Actions**, **Delete with green databases**.

Switchover actions

When you switch over a blue/green deployment, RDS performs the following actions:

1. Runs guardrail checks to verify if the blue and green environments are ready for switchover.
2. Stops new write operations on the DB cluster in both environments.
3. Drops connections to the DB instances in both environments and doesn't allow new connections.
4. Waits for replication to catch up in the green environment so that the green environment is in sync with the blue environment.
5. Renames the DB cluster and DB instances in the both environments.

   RDS renames the DB cluster and DB instances in the green environment to match the corresponding DB cluster and DB instances in the blue environment. For example, assume the name of a DB instance in the blue environment is `mydb`. Also assume the name of the corresponding DB instance in the green environment is `mydb-green-abc123`. During switchover, the name of the DB instance in the green environment is changed to `mydb`.

RDS renames the DB cluster and DB instances in the blue environment by appending `-old`*n* to the current name, where *n* is a number. For example, assume the name of a DB instance in the blue environment is `mydb`. After switchover, the DB instance name might be `mydb-old1`.

RDS also renames the endpoints in the green environment to match the corresponding endpoints in the blue environment so that application changes aren't required.

6. Allows connections to databases in both environments.
7. Allows write operations on the DB cluster in the new production environment.

   After switchover, the previous production DB cluster only allows read operations. Even if you disable the `read_only` parameter on the DB cluster, it remains read-only until you delete the blue/green deployment.

You can monitor the status of a switchover using Amazon EventBridge. For more information, see the section called “Blue/green deployment events”.

If you have tags configured in the blue environment, these tags are moved to the new production environment during switchover. The previous production environment also retains these tags. For more information about tags, see Tagging Amazon RDS resources.

If the switchover starts and then stops before finishing for any reason, then any changes are rolled back, and no changes are made to either environment.

Switchover best practices

Before you switch over, we strongly recommend that you adhere to best practices by completing the following tasks:

- Thoroughly test the resources in the green environment. Make sure they function properly and efficiently.
- Monitor relevant Amazon CloudWatch metrics. For more information, see the section called “Verifying CloudWatch metrics before switchover”.
- Identify the best time for the switchover.

  During the switchover, writes are cut off from databases in both environments. Identify a time when traffic is lowest on your production environment. Long-running transactions, such as active DDLs, can increase your switchover time, resulting in longer downtime for your production workloads.

If there's a large number of connections on your DB cluster and DB instances, consider manually reducing them to the minimum amount necessary for your application before you switch over the blue/green deployment. One way to achieve this is to create a script that monitors the status of the blue/green deployment and starts cleaning up connections when it detects that the status has changed to `SWITCHOVER_IN_PROGRESS`.

- Make sure the DB cluster and DB instances in both environments are in `Available` state.
- Make sure the DB cluster in the green environment is healthy and replicating.
- Make sure that your network and client configurations don't increase the DNS cache Time-To-Live (TTL) beyond five seconds, which is the default for Aurora DNS zones.
  Otherwise, applications will continue to send write traffic to the blue environment after switchover.
- For Aurora PostgreSQL DB clusters, do the following:
  - Review the logical replication limitations and take any required actions prior to switchover. For more information, see the section called "PostgreSQL logical replication limitations".
  - Run the `ANALYZE` operation to refresh the `pg_statistics` table. This reduces the risk of performance issues after switchover.

> **Note**
>
> During a switchover, you can't modify any DB cluster included in the switchover.

Verifying CloudWatch metrics before switchover

Before you switch over a blue/green deployment, we recommend that you check the values of the following metrics within Amazon CloudWatch.

- `DatabaseConnections` – Use this metric to estimate the level of activity on the blue/green deployment, and make sure that the value is at an acceptable level for your deployment before you switch over. If Performance Insights is turned on, `DBLoad` is a more accurate metric.
- `ActiveTransactions` – If `innodb_monitor_enable` is set to `all` in the DB parameter group for any of your DB instances, use this metric to see if there's a high number of active transactions that might block switchover.

For more information about these metrics, see the section called “CloudWatch metrics for Aurora”.

Monitoring replica lag prior to switchover

Before you switch over a blue/green deployment, make sure that replica lag on the green database is close to zero in order to reduce downtime.

- For Aurora MySQL, use the `AuroraBinlogReplicaLag` CloudWatch metric to identify the current replication lag on the green environment.
- For Aurora PostgreSQL, use the following SQL query:

```
SELECT slot_name,
       confirmed_flush_lsn as flushed,
       pg_current_wal_lsn(),
       (pg_current_wal_lsn() - confirmed_flush_lsn) AS lsn_distance
FROM pg_catalog.pg_replication_slots
WHERE slot_type = 'logical';

slot_name        |    flushed    | pg_current_wal_lsn | lsn_distance
-----------------+---------------+--------------------+------------
logical_replica1 | 47D97/CF32980 | 47D97/CF3BAC8      | 37192
```

The `confirmed_flush_lsn` represents the last log sequence number (LSN) that was sent to the replica. The `pg_current_wal_lsn` represents where the database is now. An `lsn_distance` of 0 means that the replica is caught up.

Switching over a blue/green deployment

You can switch over a blue/green deployment using the AWS Management Console, the AWS CLI, or the RDS API.

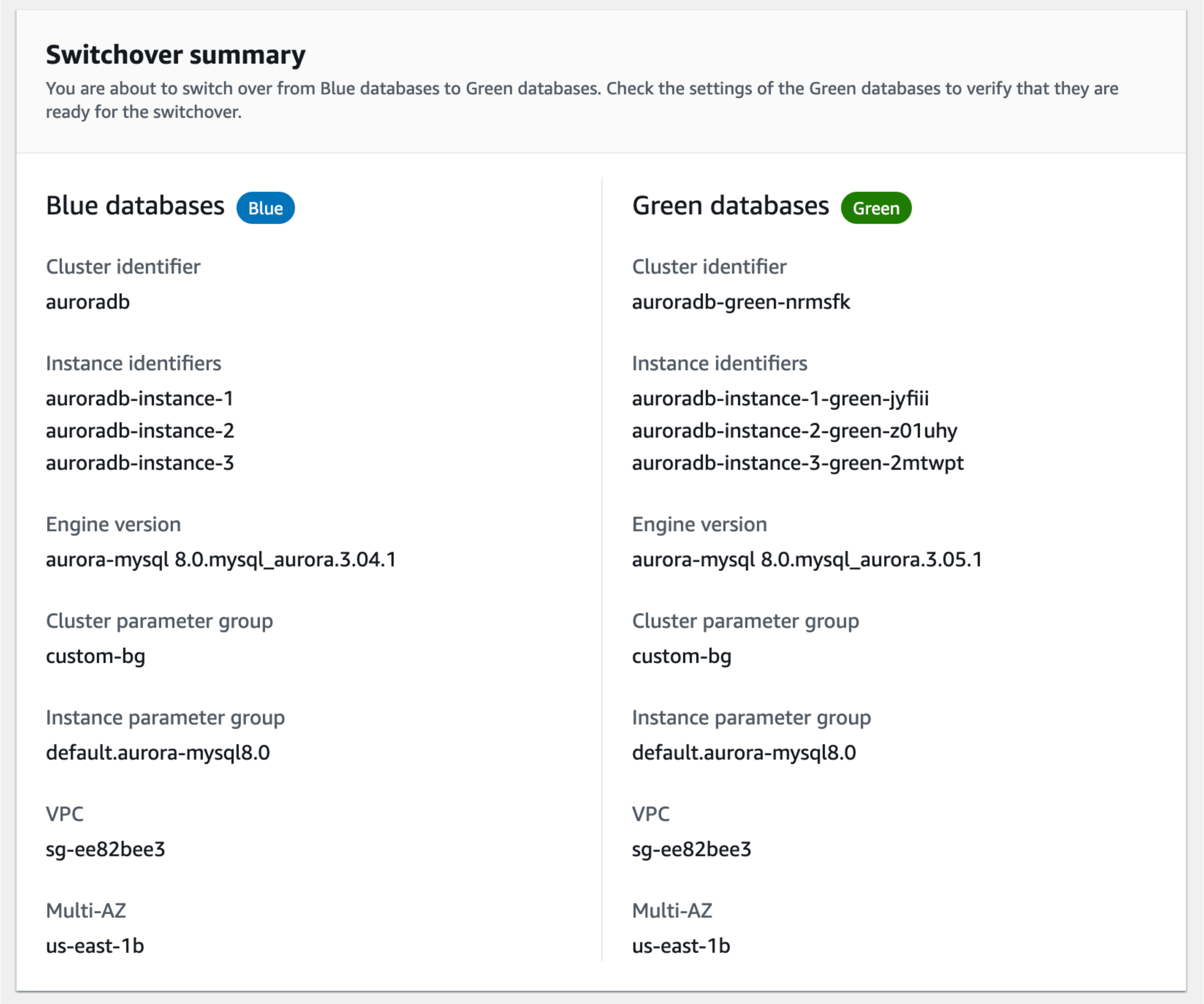
Console

To switch over a blue/green deployment

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the blue/green deployment that you want to switch over.

3. For **Actions**, choose **Switch over**.

   The **Switch over** page appears.



Switchover summary

You are about to switch over from Blue databases to Green databases. Check the settings of the Green databases to verify that they are ready for the switchover.

Blue databases Blue

Cluster identifier
auroradb

Instance identifiers
auroradb-instance-1
auroradb-instance-2
auroradb-instance-3

Engine version
aurora-mysql 8.0.mysql_aurora.3.04.1

Cluster parameter group
custom-bg

Instance parameter group
default.aurora-mysql8.0

VPC
sg-ee82bee3

Multi-AZ
us-east-1b

Green databases Green

Cluster identifier
auroradb-green-nrmsfk

Instance identifiers
auroradb-instance-1-green-jyfiii
auroradb-instance-2-green-z01uhy
auroradb-instance-3-green-2mtwpt

Engine version
aurora-mysql 8.0.mysql_aurora.3.05.1

Cluster parameter group
custom-bg

Instance parameter group
default.aurora-mysql8.0

VPC
sg-ee82bee3

Multi-AZ
us-east-1b

4. On the **Switch over** page, review the switchover summary. Make sure the resources in both environments match what you expect. If they don't, choose **Cancel**.
5. For **Timeout settings**, enter the time limit for switchover.
6. If your cluster is running Aurora PostgreSQL, review and acknowledge the pre-switchover recommendations. For more information, see the section called “PostgreSQL logical replication limitations”.
7. Choose **Switch over**.

AWS CLI

To switch over a blue/green deployment by using the AWS CLI, use the switchover-blue-green-deployment command with the following options:

- `--blue-green-deployment-identifier` – Specify the resource ID of the blue/green deployment.
- `--switchover-timeout` – Specify the time limit for the switchover, in seconds. The default is 300.

Example Switch over a blue/green deployment

For Linux, macOS, or Unix:

```
aws rds switchover-blue-green-deployment \
    --blue-green-deployment-identifier bgd-1234567890abcdef \
    --switchover-timeout 600
```

For Windows:

```
aws rds switchover-blue-green-deployment ^
    --blue-green-deployment-identifier bgd-1234567890abcdef ^
    --switchover-timeout 600
```

RDS API

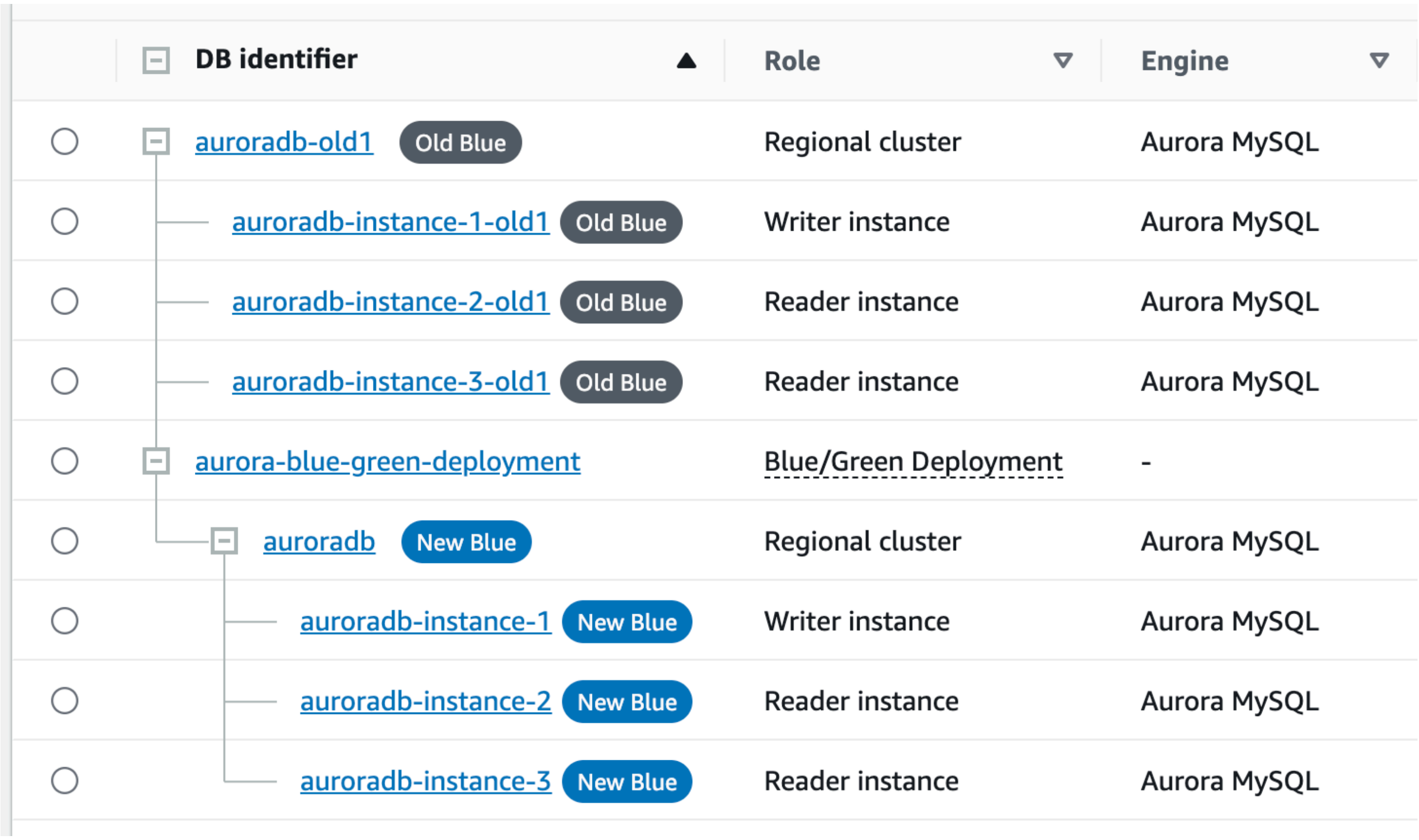
To switch over a blue/green deployment by using the Amazon RDS API, use the `SwitchoverBlueGreenDeployment` operation with the following parameters:

- `BlueGreenDeploymentIdentifier` – Specify the resource ID of the blue/green deployment.
- `SwitchoverTimeout` – Specify the time limit for the switchover, in seconds. The default is 300.

After switchover

After a switchover, the DB cluster and DB instances in the previous blue environment are retained. Standard costs apply to these resources. Replication and binary logging between the blue and green environments stops.

RDS renames the DB cluster and DB instances in the blue environment by appending `-old`*n* to the current resource name, where *n* is a number. The DB cluster is forced into a read-only state. Even if you disable the `read_only` parameter on the DB cluster, it remains read-only until you delete the blue/green deployment.



	DB identifier ▲	Role ▽	Engine ▽
○	⊟ auroradb-old1 Old Blue	Regional cluster	Aurora MySQL
○	auroradb-instance-1-old1 Old Blue	Writer instance	Aurora MySQL
○	auroradb-instance-2-old1 Old Blue	Reader instance	Aurora MySQL
○	auroradb-instance-3-old1 Old Blue	Reader instance	Aurora MySQL
○	⊟ aurora-blue-green-deployment	Blue/Green Deployment	-
○	⊟ auroradb New Blue	Regional cluster	Aurora MySQL
○	auroradb-instance-1 New Blue	Writer instance	Aurora MySQL
○	auroradb-instance-2 New Blue	Reader instance	Aurora MySQL
○	auroradb-instance-3 New Blue	Reader instance	Aurora MySQL

Updating the parent node for consumers

After you switch over an Aurora MySQL blue/green deployment, if the blue DB cluster had any external replicas or binary log consumers prior to switchover, you must update their parent node after switchover in order to maintain replication continuity.

After switchover, the writer DB instance that was previously in the green environment emits an event that contains the master log file name and master log position. For example:

```
aws rds describe-events --output json --source-type db-instance --source-identifier db-
instance-identifier

{
    "Events": [
...
        {
            "SourceIdentifier": "db-instance-identifier",
            "SourceType": "db-instance",
            "Message": "Binary log coordinates in green environment after switchover:
             file mysql-bin-changelog.000003 and position 804",
            "EventCategories": [],
            "Date": "2023-11-10T01:33:41.911Z",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:db:db-instance-identifier"
        }
    ]
}
```

First, make sure that the consumer or replica has applied all binary logs from the old blue environment. Then, use the provided binary log coordinates to resume application on the consumers. For example, if you're running a MySQL replica on EC2, you can use the `CHANGE MASTER TO` command:

```
CHANGE MASTER TO MASTER_HOST='{new-writer-endpoint}', MASTER_LOG_FILE='mysql-bin-
changelog.000003', MASTER_LOG_POS=804;
```

Deleting a blue/green deployment

You can delete a blue/green deployment before or after you switch it over.

When you delete a blue/green deployment before switching it over, Amazon RDS optionally deletes the DB cluster in the green environment:

- If you choose to delete the DB cluster in the green environment (`--delete-target`), it must have deletion protection turned off.
- If you don't delete the DB cluster in the green environment (`--no-delete-target`), the cluster is retained, but it's no longer part of a blue/green deployment. Replication continues between the environments.

The option to delete the green databases isn't available in the console after switchover. When you delete blue/green deployments using the AWS CLI, you can't specify the `--delete-target` option if the deployment status is `SWITCHOVER_COMPLETED`.

Important

Deleting a blue/green deployment doesn't affect the blue environment.

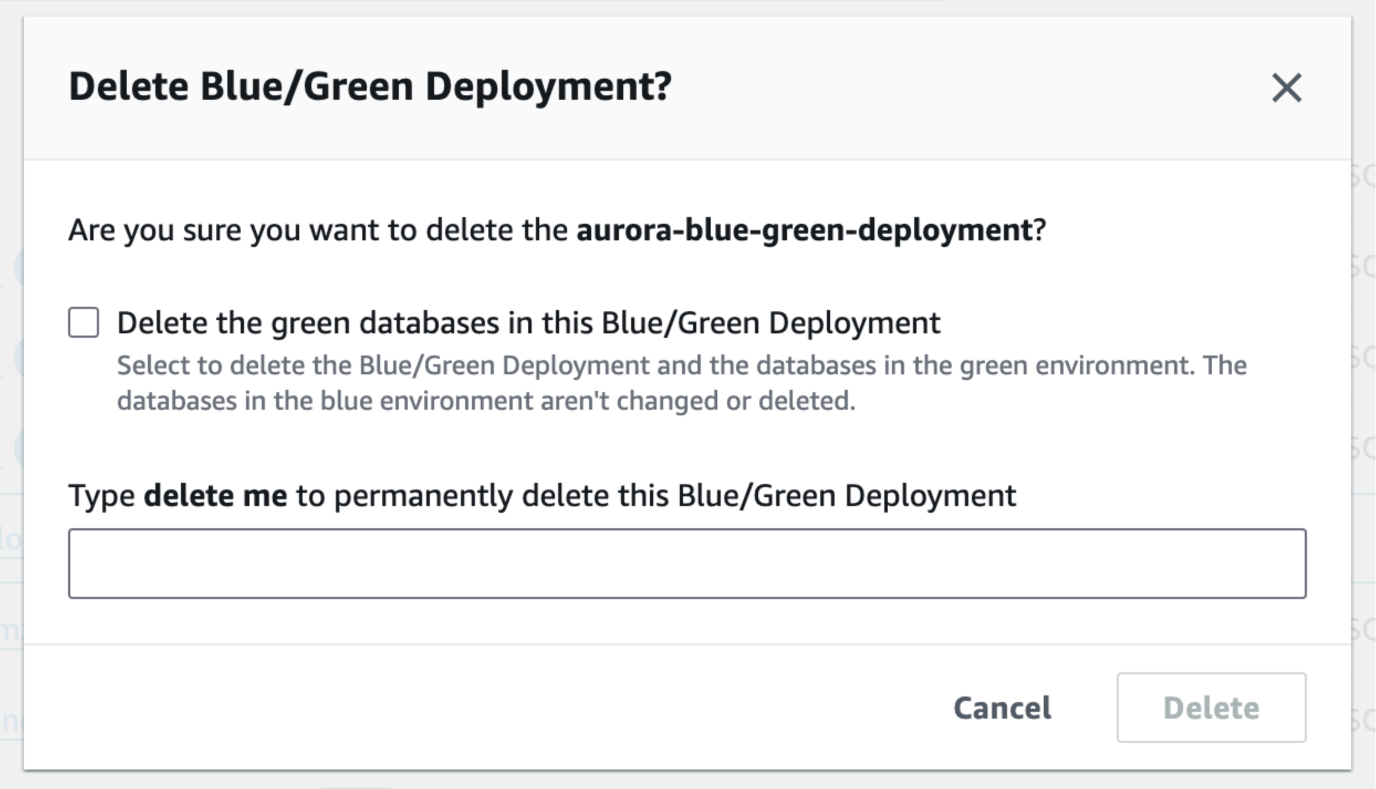
You can delete a blue/green deployment using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To delete a blue/green deployment

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the blue/green deployment that you want to delete.
3. For **Actions**, choose **Delete**.

   The **Delete Blue/Green Deployment?** window appears.



Delete Blue/Green Deployment?

Are you sure you want to delete the **aurora-blue-green-deployment**?

□ Delete the green databases in this Blue/Green Deployment

Select to delete the Blue/Green Deployment and the databases in the green environment. The databases in the blue environment aren't changed or deleted.

Type **delete me** to permanently delete this Blue/Green Deployment

To delete the green databases, select **Delete the green databases in this Blue/Green Deployment**.

4. Enter **`delete me`** in the box.
5. Choose **Delete**.

AWS CLI

To delete a blue/green deployment by using the AWS CLI, use the delete-blue-green-deployment command with the following options:

- `--blue-green-deployment-identifier` – The resource ID of the blue/green deployment to be deleted.
- `--delete-target` – Specifies that the DB cluster in the green environment is deleted. You can't specify this option if the blue/green deployment has a status of `SWITCHOVER_COMPLETED`.
- `--no-delete-target` – Specifies that the DB cluster in the green environment is retained.

Example Delete a blue/green deployment and the DB cluster in the green environment

For Linux, macOS, or Unix:

```
aws rds delete-blue-green-deployment \
    --blue-green-deployment-identifier bgd-1234567890abcdef \
    --delete-target
```

For Windows:

```
aws rds delete-blue-green-deployment ^
    --blue-green-deployment-identifier bgd-1234567890abcdef ^
    --delete-target
```

Example Delete a blue/green deployment but retain the DB cluster in the green environment

For Linux, macOS, or Unix:

```
aws rds delete-blue-green-deployment \
    --blue-green-deployment-identifier bgd-1234567890abcdef \
    --no-delete-target
```

For Windows:

```
aws rds delete-blue-green-deployment ^
    --blue-green-deployment-identifier bgd-1234567890abcdef ^
    --no-delete-target
```

RDS API

To delete a blue/green deployment by using the Amazon RDS API, use the DeleteBlueGreenDeployment operation with the following parameters:

- BlueGreenDeploymentIdentifier – The resource ID of the blue/green deployment to be deleted.
- DeleteTarget – Specify TRUE to delete the DB cluster in the green environment or FALSE to retain it. Cannot be TRUE if the blue/green deployment has a status of SWITCHOVER_COMPLETED.

Backing up and restoring an Amazon Aurora DB cluster

These topics provide information about backing up and restoring Amazon Aurora DB clusters.

> **Tip**
>
> The Aurora high availability features and automatic backup capabilities help to keep your data safe without requiring extensive setup from you. Before you implement a backup strategy, learn about the ways that Aurora maintains multiple copies of your data and helps you to access them across multiple DB instances and AWS Regions. For details, see High availability for Amazon Aurora.

Topics

Overview of backing up and restoring an Aurora DB cluster

The following topics describe Aurora backups and how to restore your Aurora DB cluster.

Contents

Backups

Aurora backs up your cluster volume automatically and retains restore data for the length of the *backup retention period*. Aurora automated backups are continuous and incremental, so you can quickly restore to any point within the backup retention period. No performance impact or interruption of database service occurs as backup data is being written. You can specify a backup retention period from 1–35 days when you create or modify a DB cluster. Aurora automated backups are stored in Amazon S3.

If you want to retain data beyond the backup retention period, you can take a snapshot of the data in your cluster volume. Aurora DB cluster snapshots don't expire. You can create a new DB cluster from the snapshot. For more information, see Creating a DB cluster snapshot.

Note

- For Amazon Aurora DB clusters, the default backup retention period is one day regardless of how the DB cluster is created.

- You can't disable automated backups on Aurora. The backup retention period for Aurora is managed by the DB cluster.

Your costs for backup storage depend upon the amount of Aurora backup and snapshot data you keep and how long you keep it. For information about the storage associated with Aurora backups and snapshots, see Understanding Amazon Aurora backup storage usage. For pricing information about Aurora backup storage, see Amazon RDS for Aurora pricing. After the Aurora cluster associated with a snapshot is deleted, storing that snapshot incurs the standard backup storage charges for Aurora.

Using AWS Backup

You can use AWS Backup to manage backups of Amazon Aurora DB clusters.

Snapshots managed by AWS Backup are considered manual DB cluster snapshots, but don't count toward the DB cluster snapshot quota for Aurora. Snapshots that were created with AWS Backup have names with `awsbackup:job-`*`AWS-Backup-job-number`*. For more information about AWS Backup, see the *AWS Backup Developer Guide*.

You can also use AWS Backup to manage automated backups of Amazon Aurora DB clusters. If your DB cluster is associated with a backup plan in AWS Backup, you can use that backup plan for point-in-time recovery. Automated (continuous) backups that are managed by AWS Backup have names with `continuous:cluster-`*`AWS-Backup-job-number`*. For more information, see Restoring a DB cluster to a specified time using AWS Backup.

Backup window

Automated backups occur daily during the preferred backup window. If the backup requires more time than allotted to the backup window, the backup continues after the window ends, until it finishes. The backup window can't overlap with the weekly maintenance window for the DB cluster.

Aurora automated backups are continuous and incremental, but the backup window is used to create a daily system backup that is preserved within the backup retention period. You can copy the backup to preserve it outside of the retention period.

Note

When you create a DB cluster using the AWS Management Console, you can't specify a backup window. However, you can specify a backup window when you create a DB cluster using the AWS CLI or RDS API.

If you don't specify a preferred backup window when you create the DB cluster, Aurora assigns a default 30-minute backup window. This window is selected at random from an 8-hour block of time for each AWS Region. The following table lists the time blocks for each AWS Region from which the default backup windows are assigned.

Region Name	Region	Time Block
US East (Ohio)	us-east-2	03:00–11:00 UTC
US East (N. Virginia)	us-east-1	03:00–11:00 UTC
US West (N. California)	us-west-1	06:00–14:00 UTC
US West (Oregon)	us-west-2	06:00–14:00 UTC
Africa (Cape Town)	af-south-1	03:00–11:00 UTC
Asia Pacific (Hong Kong)	ap-east-1	06:00–14:00 UTC
Asia Pacific (Hyderabad)	ap-south-2	06:30–14:30 UTC
Asia Pacific (Jakarta)	ap-southeast-3	08:00–16:00 UTC
Asia Pacific (Melbourne)	ap-southeast-4	11:00–19:00 UTC
Asia Pacific (Mumbai)	ap-south-1	16:30–00:30 UTC
Asia Pacific (Osaka)	ap-northeast-3	00:00–08:00 UTC

Region Name	Region	Time Block
Asia Pacific (Seoul)	ap-northeast-2	13:00–21:00 UTC
Asia Pacific (Singapore)	ap-southeast-1	14:00–22:00 UTC
Asia Pacific (Sydney)	ap-southeast-2	12:00–20:00 UTC
Asia Pacific (Tokyo)	ap-northeast-1	13:00–21:00 UTC
Canada (Central)	ca-central-1	03:00–11:00 UTC
Canada West (Calgary)	ca-west-1	18:00–02:00 UTC
China (Beijing)	cn-north-1	06:00–14:00 UTC
China (Ningxia)	cn-northwest-1	06:00–14:00 UTC
Europe (Frankfurt)	eu-central-1	20:00–04:00 UTC
Europe (Ireland)	eu-west-1	22:00–06:00 UTC
Europe (London)	eu-west-2	22:00–06:00 UTC
Europe (Milan)	eu-south-1	02:00–10:00 UTC
Europe (Paris)	eu-west-3	07:29–14:29 UTC
Europe (Spain)	eu-south-2	02:00–10:00 UTC
Europe (Stockholm)	eu-north-1	23:00–07:00 UTC
Europe (Zurich)	eu-central-2	02:00–10:00 UTC
Israel (Tel Aviv)	il-central-1	03:00–11:00 UTC
Middle East (Bahrain)	me-south-1	06:00–14:00 UTC
Middle East (UAE)	me-central-1	05:00–13:00 UTC

Region Name	Region	Time Block
South America (São Paulo)	sa-east-1	23:00–07:00 UTC
AWS GovCloud (US-East)	us-gov-east-1	17:00–01:00 UTC
AWS GovCloud (US-West)	us-gov-west-1	06:00–14:00 UTC

Retaining automated backups

When you delete a provisioned or Aurora Serverless v2 DB cluster, you can retain automated backups. This allows you to restore a DB cluster to a specific point in time within the backup retention period, even after the cluster is deleted.

Retained automated backups contain system snapshots and transaction logs from a DB cluster. They also include DB cluster properties, such as DB instance class, which are required to restore it to an active cluster.

You can restore or remove retained automated backups using the AWS Management Console, RDS API, and AWS CLI.

> **Note**
>
> You can't retain automated backups for Aurora Serverless v1 DB clusters.

Topics

- Retention period
- Viewing retained backups
- Retention costs
- Limitations
- Deleting retained automated backups

Retention period

The system snapshots and transaction logs in a retained automated backup expire the same way that they expire for the source DB cluster. The settings for the retention period of the source cluster also apply to the automated backups. Because no new snapshots or logs are created for this cluster, the retained automated backups eventually expire completely. After the retention period is over, you continue to retain manual DB cluster snapshots, but all of the automated backups expire.

You can remove retained automated backups using the console, AWS CLI or RDS API. For more information, see Deleting retained automated backups.

Unlike a retained automated backup, a final snapshot doesn't expire. We strongly suggest that you take a final snapshot even if you retain automated backups, because the retained automated backups eventually expire.

Viewing retained backups

To view your retained automated backups in the RDS console, choose **Automated backups** in the navigation pane, then choose **Retained**. To view individual snapshots associated with a retained automated backup, choose **Snapshots** in the navigation pane. Alternatively, you can describe individual snapshots associated with a retained automated backup. From there, you can restore a DB instance directly from one of those snapshots.

To describe your retained automated backups using the AWS CLI, use the following command:

```
aws rds describe-db-cluster-automated-backups --db-cluster-resource-
id DB_cluster_resource_ID
```

To describe your retained automated backups using the RDS API, call the `DescribeDBClusterAutomatedBackups` action with the `DbClusterResourceId` parameter.

Retention costs

There is no additional charge for backup storage of up to 100% of your total Aurora database storage for each Aurora DB cluster. There is also no additional charge up to one day when you retain automated backups after deleting a DB cluster. Backups that you retain for more than one day are charged.

There is no additional charge for transaction logs or instance metadata. All other pricing rules for backups apply to restorable clusters. For more information, see the Amazon Aurora pricing page.

Limitations

The following limitations apply to retained automated backups:

- The maximum number of retained automated backups in one AWS Region is 40. It's not included in the quota for DB clusters. You can have up to 40 running DB clusters, 40 running DB instances, and 40 retained automated backups for DB clusters at the same time.

  For more information, see Quotas in Amazon Aurora.
- Retained automated backups don't contain information about parameters or option groups.
- You can restore a deleted cluster to a point in time that is within the retention period at the time of deletion.
- You can't modify a retained automated backup because it consists of system backups, transaction logs, and the DB cluster properties that existed at the time that you deleted the source cluster.

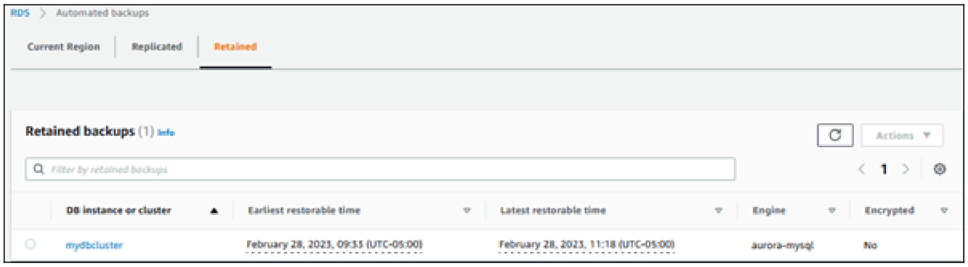
Deleting retained automated backups

You can delete retained automated backups when they are no longer needed.

Console

To delete a retained automated backup

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Automated backups**.
3. Choose the **Retained** tab.



	DB instance or cluster ▲	Earliest restorable time	Latest restorable time	Engine	Encrypted
○	mydbcluster	February 28, 2023, 09:33 (UTC-05:00)	February 28, 2023, 11:18 (UTC-05:00)	aurora-mysql	No

4. Choose the retained automated backup that you want to delete.
5. For **Actions**, choose **Delete**.
6. On the confirmation page, enter **`delete me`** and choose **Delete**.

AWS CLI

You can delete a retained automated backup by using the AWS CLI command delete-db-cluster-automated-backup with the following option:

- `--db-cluster-resource-id` – The resource identifier for the source DB cluster.

  You can find the resource identifier for the source DB cluster of a retained automated backup by running the AWS CLI command describe-db-cluster-automated-backups.

Example

This example deletes the retained automated backup for the source DB cluster that has the resource ID `cluster-123ABCEXAMPLE`.

For Linux, macOS, or Unix:

```
aws rds delete-db-cluster-automated-backup \
    --db-cluster-resource-id cluster-123ABCEXAMPLE
```

For Windows:

```
aws rds delete-db-cluster-automated-backup ^
    --db-cluster-resource-id cluster-123ABCEXAMPLE
```

RDS API

You can delete a retained automated backup by using the Amazon RDS API operation DeleteDBClusterAutomatedBackup with the following parameter:

- `DbClusterResourceId` – The resource identifier for the source DB cluster.

  You can find the resource identifier for the source DB instance of a retained automated backup using the Amazon RDS API operation DescribeDBClusterAutomatedBackups.

Restoring data

You can recover your data by creating a new Aurora DB cluster from the backup data that Aurora retains, from a DB cluster snapshot that you have saved, or from a retained automated backup.

You can quickly restore a new copy of a DB cluster created from backup data to any point in time during your backup retention period. Because Aurora backups are continuous and incremental during the backup retention period, you don't need to take frequent snapshots of your data to improve restore times.

The *latest restorable time* for a DB cluster is the most recent point to which you can restore your DB cluster. This is typically within 5 minutes of the current time for an active DB cluster, or 5 minutes of the cluster deletion time for a retained automated backup.

The *earliest restorable time* specifies how far back within the backup retention period that you can restore your cluster volume.

To determine the latest or earliest restorable time for a DB cluster, look for the `Latest restorable time` or `Earliest restorable time` values on the RDS console. For information about viewing these values, see [Viewing retained backups]().

You can determine when the restore of a DB cluster is complete by checking the `Latest restorable time` and `Earliest restorable time` values. These values return NULL until the restore operation is complete. You can't request a backup or restore operation if either `Latest restorable time` or `Earliest restorable time` returns NULL.

For information about restoring a DB cluster to a specified time, see [Restoring a DB cluster to a specified time]().

Database cloning for Aurora

You can also use database cloning to clone the databases of your Aurora DB cluster to a new DB cluster, instead of restoring a DB cluster snapshot. The clone databases use only minimal additional space when first created. Data is copied only as data changes, either on the source databases or on the clone databases. You can make multiple clones from the same DB cluster, or create additional clones even from other clones. For more information, see [Cloning a volume for an Amazon Aurora DB cluster]().

Backtrack

Aurora MySQL now supports "rewinding" a DB cluster to a specific time, without restoring data from a backup. For more information, see [Backtracking an Aurora DB cluster]().

Understanding Amazon Aurora backup storage usage

Amazon Aurora maintains two types of backup: automated (continuous) backups and snapshots.

Automated backup storage

The automated (continuous) backup for a cluster incrementally stores all database changes within a specified retention period to be able to restore to any point in time within that retention period. Retention periods can range from 1–35 days. Automated backups are incremental and charged based on the amount of storage that’s required to restore to any time within the retention period.

Aurora also provides a free amount of backup usage. This free amount of usage is equal to the latest cluster volume size (as represented by the `VolumeBytesUsed` Amazon CloudWatch metric). This amount is subtracted from the calculated automated backup usage. There is also no charge for an automated backup whose retention period is just 1 day.

For example, your automated backup has a retention period of 7 days, and you want to restore your cluster to its state from four days ago. Aurora uses the incremental data stored in the automated backup to re-create the state of the cluster at that exact time four days ago.

The automated backup stores all the required information to be able to restore the cluster at any point in time in the retention window. That means that it stores all changes during the retention window, including writes of new information or deletion of existing information. For databases where many changes occur, the size of the automated backup grows over time. After a database stops experiencing changes, you can expect the size of the automated backup to decrease, as the previously stored changes exit the retention window.

The total billed usage for the automated backup never exceeds the cumulative cluster volume size over the retention period. For example, if your retention period is 7 days, and your cluster volume was 100 GB every day, then the billed automated backup usage never exceeds 700 GB (100 GB * 7).

Snapshot storage

DB cluster snapshots are always full backups whose size is that of the cluster volume at the time the snapshot is taken. Snapshots, either taken manually by the user or automatically by an AWS Backups plan, are treated as manual snapshots. Aurora provides unlimited free storage for all snapshots that lie within the automated backup retention period. After a manual snapshot is outside the retention period, it's billed per GB-month. Any automated system snapshot is never charged unless copied and retained past the retention period.

For general information about Aurora backups, see Backups. For pricing information about Aurora backup storage, see the Amazon Aurora pricing page.

Amazon CloudWatch metrics for Aurora backup storage

You can monitor your Aurora clusters and create reports using Amazon CloudWatch metrics through the CloudWatch console. You can use the following CloudWatch metrics to review and monitor the amount of storage used by your Aurora backups. These metrics are computed independently for each Aurora DB cluster.

- `BackupRetentionPeriodStorageUsed` – Represents the amount of backup storage used, in bytes, for storing automated backups at the current time.
  - The value depends on the size of the cluster volume and the number of changes (writes and updates) that are made to the DB cluster during the retention period. This is because the automated backup must store all incremental changes made to the cluster to be able to restore to any point in time.
  - This metric doesn't subtract the free tier of backup usage that Aurora provides.
  - This metric emits a single daily data point for the automated backup usage recorded on that day.
- `SnapshotStorageUsed` – Represents the amount of backup storage used, in bytes, for storing manual snapshots beyond the automated backup's retention period.
  - The value depends on the number of snapshots you keep beyond the automated backup’s retention period and the size of each snapshot.
  - The size of each snapshot is the size of the cluster volume at the time you take the snapshot.
  - Snapshots are full backups, not incremental.
  - This metric emits one daily data point for each snapshot being charged. To retrieve your daily total snapshot usage, take the sum of this metric over a period of 1 day.
- `TotalBackupStorageBilled` – Represents the metrics for all billed backup usage, in bytes, for the given cluster:

  ```
  BackupRetentionPeriodStorageUsed + SnapshotStorageUsed - free tier
  ```

  - This metric emits one daily data point for the `BackupRetentionPeriodStorageUsed` value *minus* the free tier of backup usage that Aurora provides. This free tier is equal to the latest recorded size of the DB cluster volume. This data point represents the actual billed usage for the automated backup.

- This metric emits individual daily data points for all of the `SnapshotStorageUsed` values.
- To retrieve your total daily billed backup usage, take the sum of this metric over a period of 1 day. This sums all of the billed snapshot usage with the billed automated backup usage, to give your total billed backup usage.

For more information about how to use CloudWatch metrics, see Availability of Aurora metrics in the Amazon RDS console.

Calculating backup storage usage

The usage for an automated backup is calculated by looking at all of the incremental records that must be stored, to be able to restore to any point in time within the retention period of the backup.

For example, you have an automated backup with retention period of 7 days. Your cluster volume size just before the retention period was 100 GB, so that's the least amount that Aurora needs to store. Then you have the following activity for the next 7 days, where the incremental record size is the amount of storage needed to store the change records coming from your database's writes and updates.

Day	Incremental record size (GB)
1	10
2	15
3	25
4	20
5	10
6	25
7	30
Total	135

This data means that the calculated automated backup usage for your backup is the following:

```
100 GB (volume size before retention period) + 135 GB (size of incremental records) =
 235 GB total backup usage
```

The billed usage then subtracts the free tier of usage. Assume that the latest size of your volume is 200 GB:

```
235 GB total backup usage - 200 GB (latest volume size) = 35 GB billed backup usage
```

FAQs

When am I billed for snapshots?

You're billed for manual snapshots that are outside (older than) the retention period of the automated backup.

What's a manual snapshot?

A manual snapshot is a snapshot to which one of the following conditions applies:

- Manually requested by you
- Taken by an automated backup service such as AWS Backup
- Copied from an automated system snapshot to preserve it outside the retention period

What happens to my manual snapshots if I delete my DB cluster?

Manual snapshots don't expire until you delete them.

When you delete your DB cluster, the manual snapshots that you previously took continue to exist. If these snapshots previously weren't being billed because they were within the automated backup retention period, now they're not covered anymore and all start to be billed at their full size for their usage.

How can I reduce my backup storage costs?

There are a few ways to reduce backup usage related costs:

- Delete manual snapshots that lie outside your automated backup’s retention period. This includes the snapshots you’ve taken, as well as the snapshots that your AWS Backup plan might have taken. Make sure to check your AWS Backup plan to make sure it isn't keeping snapshots outside the retention period that you don't expect.
- Evaluate your writes and updates to your database to see if you can reduce the number of changes you're making. Because our automated backup stores all incremental changes within

the retention period, reducing the number of updates that you're making also reduces your automated backup charges.

- Evaluate whether reducing your automated backup’s retention period would make sense. Reducing the retention period means that the backup stores fewer days of incremental data, which could reduce the overall backup cost. However, reducing this retention period could also cause some snapshots to start being billed because they're now outside the retention period. Make sure to check all the extra snapshot costs that you might incur before deciding whether this is the right course of action for you.

How is backup storage billed?

Backup storage is billed by the GB-month.

This means that the backup storage usage is charged as the weighted average of the usage over the given month. Here are a few examples for a 30-day month:

- Billed backup usage is 100 GB for all 30 days of the month. Your charge is the following:

```
(100 GB * 30) / 30 = 100 GB-month
```

- Billed backup usage is 100 GB for the first 15 days of the month, then 0 GB for the last 15. Your charge is the following:

```
(100 GB * 15 + 0 GB * 15) / 30 = 50 GB-month
```

- Billed backup usage is 50 GB for the first 10 days of the month, 100 GB for the next 10 days, then 150 GB for the final 10. Your charge is the following:

```
(50 GB * 10 + 100 GB * 10 + 150 GB * 10) / 30 = 100 GB-month
```

How does the backtrack setting for my DB cluster affect backup storage usage?

The backtrack setting for an Aurora DB cluster doesn't affect the volume of backup data for that cluster. Amazon bills the storage for backtracking data separately. For pricing information about Aurora backtracking, see the Amazon Aurora pricing page.

How do storage costs apply to shared snapshots?

If you share a snapshot with another user, you're still the owner of that snapshot. The storage costs apply to the snapshot owner. If you delete a shared snapshot that you own, nobody can access it.

To keep access to a shared snapshot owned by someone else, you can copy that snapshot. Doing so makes you the owner of the new snapshot. Any storage costs for the copied snapshot apply to your account.

For more information on sharing snapshots, see Sharing a DB cluster snapshot. For more information on copying snapshots, see Copying a DB cluster snapshot.

Creating a DB cluster snapshot

Amazon RDS creates a storage volume snapshot of your DB cluster, backing up the entire DB cluster and not just individual databases. When you create a DB cluster snapshot, you need to identify which DB cluster you are going to back up, and then give your DB cluster snapshot a name so you can restore from it later. The amount of time it takes to create a DB cluster snapshot varies with the size of your databases. Because the snapshot includes the entire storage volume, the size of files, such as temporary files, also affects the amount of time it takes to create the snapshot.

Note

Your DB cluster must be in the `available` state to take a DB cluster snapshot.

Unlike automated backups, manual snapshots aren't subject to the backup retention period. Snapshots don't expire.

For very long-term backups, we recommend exporting snapshot data to Amazon S3. If the major version of your DB engine is no longer supported, you can't restore to that version from a snapshot. For more information, see Exporting DB cluster snapshot data to Amazon S3.

You can create a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To create a DB cluster snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.

   The **Manual snapshots** list appears.
3. Choose **Take snapshot**.

   The **Take DB snapshot** window appears.
4. For **Snapshot type**, select **DB cluster**.
5. Choose the **DB cluster** for which you want to take a snapshot.

6. Enter the **Snapshot name**.

7. Choose **Take snapshot**.

   The **Manual snapshots** list appears, with the new DB cluster snapshot's status shown as `Creating`. After its status is `Available`, you can see its creation time.

AWS CLI

When you create a DB cluster snapshot using the AWS CLI, you need to identify which DB cluster you are going to back up, and then give your DB cluster snapshot a name so you can restore from it later. You can do this by using the AWS CLI `create-db-cluster-snapshot` command with the following parameters:

- `--db-cluster-identifier`
- `--db-cluster-snapshot-identifier`

In this example, you create a DB cluster snapshot named *`mydbclustersnapshot`* for a DB cluster called *`mydbcluster`*.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-cluster-snapshot \
    --db-cluster-identifier mydbcluster \
    --db-cluster-snapshot-identifier mydbclustersnapshot
```

For Windows:

```
aws rds create-db-cluster-snapshot ^
    --db-cluster-identifier mydbcluster ^
    --db-cluster-snapshot-identifier mydbclustersnapshot
```

RDS API

When you create a DB cluster snapshot using the Amazon RDS API, you need to identify which DB cluster you are going to back up, and then give your DB cluster snapshot a name so you can restore from it later. You can do this by using the Amazon RDS API `CreateDBClusterSnapshot` command with the following parameters:

- DBClusterIdentifier
- DBClusterSnapshotIdentifier

Determining whether the DB cluster snapshot is available

You can check that the DB cluster snapshot is available by looking under **Snapshots** on the **Maintenance & backups** tab on the detail page for the cluster in the AWS Management Console, by using the `describe-db-cluster-snapshots` CLI command, or by using the `DescribeDBClusterSnapshots` API action.

You can also use the `wait db-cluster-snapshot-available` CLI command to poll the API every 30 seconds until the snapshot is available.

Restoring from a DB cluster snapshot

Amazon RDS creates a storage volume snapshot of your DB cluster, backing up the entire DB cluster and not just individual databases. You can create a new DB cluster by restoring from a DB snapshot. You provide the name of the DB cluster snapshot to restore from, and then provide a name for the new DB cluster that is created from the restore. You can't restore from a DB cluster snapshot to an existing DB cluster; a new DB cluster is created when you restore.

> ⚠ **Important**
>
> If you attempt to restore a snapshot to a deprecated DB engine version, an immediate upgrade to the latest engine version will occur. Additionally, Extended Support charges might apply if the version is on Extended Support or has reached the end of standard support. For more information, see Using Amazon RDS Extended Support.

You can use the restored DB cluster as soon as its status is `available`.

You can use AWS CloudFormation to restore a DB cluster from a DB cluster snapshot. For more information, see AWS::RDS::DBCluster in the *AWS CloudFormation User Guide*.

> ⓘ **Note**
>
> Sharing a manual DB cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to directly restore a DB cluster from the snapshot instead of taking a copy of it and restoring from that. For more information, see Sharing a DB cluster snapshot.

For information about restoring an Aurora DB cluster or a global cluster with an RDS Extended Support version, see Restoring an Aurora DB cluster or a global cluster with Amazon RDS Extended Support.

Parameter group considerations

We recommend that you retain the DB parameter group and DB cluster parameter group for any DB cluster snapshots you create, so that you can associate your restored DB cluster with the correct parameter groups.

The default DB parameter group and DB cluster parameter group are associated with the restored cluster, unless you choose different ones. No custom parameter settings are available in the default parameter groups.

You can specify the parameter groups when you restore the DB cluster.

For more information about DB parameter groups and DB cluster parameter groups, see Working with parameter groups.

Security group considerations

When you restore a DB cluster, the default virtual private cloud (VPC), DB subnet group, and VPC security group are associated with the restored instance, unless you choose different ones.

- If you're using the Amazon RDS console, you can specify a custom VPC security group to associate with the cluster or create a new VPC security group.
- If you're using the AWS CLI, you can specify a custom VPC security group to associate with the cluster by including the `--vpc-security-group-ids` option in the `restore-db-cluster-from-snapshot` command.
- If you're using the Amazon RDS API, you can include the `VpcSecurityGroupIds.VpcSecurityGroupId.N` parameter in the `RestoreDBClusterFromSnapshot` action.

As soon as the restore is complete and your new DB cluster is available, you can also change the VPC settings by modifying the DB cluster. For more information, see Modifying an Amazon Aurora DB cluster.

Amazon Aurora considerations

With Aurora, you restore a DB cluster snapshot to a DB cluster.

With both Aurora MySQL and Aurora PostgreSQL, you can also restore a DB cluster snapshot to an Aurora Serverless DB cluster. For more information, see Restoring an Aurora Serverless v1 DB cluster.

With Aurora MySQL, you can restore a DB cluster snapshot from a cluster without parallel query to a cluster with parallel query. Because parallel query is typically used with very large tables, the snapshot mechanism is the fastest way to ingest large volumes of data to an Aurora MySQL parallel query-enabled cluster. For more information, see Working with parallel query for Amazon Aurora MySQL.

Restoring from a snapshot

You can restore a DB cluster from a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To restore a DB cluster from a DB cluster snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Choose the DB cluster snapshot that you want to restore from.
4. For **Actions**, choose **Restore snapshot**.

   The **Restore snapshot** page displays.
5. Choose the DB engine version to which you want to restore the DB cluster.

   By default, the snapshot is restored to the same DB engine version as the source DB cluster, if that version is available.
6. For **DB instance identifier**, enter the name for your restored DB cluster.
7. Specify other settings, such as the DB cluster storage configuration.

   For information about each setting, see Settings for Aurora DB clusters.
8. Choose **Restore DB cluster**.

AWS CLI

To restore a DB cluster from a DB cluster snapshot, use the AWS CLI command restore-db-cluster-from-snapshot.

In this example, you restore from a previously created DB cluster snapshot named `mydbclustersnapshot`. You restore to a new DB cluster named `mynewdbcluster`.

You can specify other settings, such as the DB engine version. If you don't specify an engine version, the DB cluster is restored to the default engine version.

For information about each setting, see Settings for Aurora DB clusters.

Example

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier mynewdbcluster \
    --snapshot-identifier mydbclustersnapshot \
    --engine aurora-mysql|aurora-postgresql
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-cluster-identifier mynewdbcluster ^
    --snapshot-identifier mydbclustersnapshot ^
    --engine aurora-mysql|aurora-postgresql
```

After the DB cluster has been restored, you must add the DB cluster to the security group used by the DB cluster used to create the DB cluster snapshot if you want the same functionality as that of the previous DB cluster.

> ⚠ **Important**
>
> If you use the console to restore a DB cluster, then Amazon RDS automatically creates the primary DB instance (writer) for your DB cluster. If you use the AWS CLI to restore a DB cluster, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster. If you don't create the primary DB instance, the DB cluster endpoints remain in the `creating` status.
> Call the create-db-instance AWS CLI command to create the primary instance for your DB cluster. Include the name of the DB cluster as the `--db-cluster-identifier` option value.

RDS API

To restore a DB cluster from a DB cluster snapshot, call the RDS API operation RestoreDBClusterFromSnapshot with the following parameters:

- `DBClusterIdentifier`
- `SnapshotIdentifier`

⚠ **Important**

If you use the console to restore a DB cluster, then Amazon RDS automatically creates the primary DB instance (writer) for your DB cluster. If you use the RDS API to restore a DB cluster, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster. If you don't create the primary DB instance, the DB cluster endpoints remain in the `creating` status.

Call the RDS API operation CreateDBInstance to create the primary instance for your DB cluster. Include the name of the DB cluster as the `DBClusterIdentifier` parameter value.

Copying a DB cluster snapshot

With Amazon Aurora, you can copy automated backups or manual DB cluster snapshots. After you copy a snapshot, the copy is a manual snapshot. You can make multiple copies of an automated backup or manual snapshot, but each copy must have a unique identifier.

You can copy a snapshot within the same AWS Region, you can copy a snapshot across AWS Regions, and you can copy shared snapshots.

You can't copy a DB cluster snapshot across Regions and accounts in a single step. Perform one step for each of these copy actions. As an alternative to copying, you can also share manual snapshots with other AWS accounts. For more information, see Sharing a DB cluster snapshot.

> **Note**
>
> Amazon bills you based upon the amount of Amazon Aurora backup and snapshot data you keep and the period of time that you keep it. For information about the storage associated with Aurora backups and snapshots, see Understanding Amazon Aurora backup storage usage. For pricing information about Aurora storage, see Amazon RDS for Aurora pricing.

Topics

Limitations

The following are some limitations when you copy snapshots:

- You can't copy a snapshot to or from the following AWS Regions:

  - China (Beijing)
  - China (Ningxia)
- You can copy a snapshot between AWS GovCloud (US-East) and AWS GovCloud (US-West). However, you can't copy a snapshot between these AWS GovCloud (US) Regions and commercial AWS Regions.
- If you delete a source snapshot before the target snapshot becomes available, the snapshot copy might fail. Verify that the target snapshot has a status of `AVAILABLE` before you delete a source snapshot.
- You can have up to five snapshot copy requests in progress to a single destination Region per account.
- When you request multiple snapshot copies for the same source DB instance, they're queued internally. The copies requested later won't start until the previous snapshot copies are completed. For more information, see Why is my EC2 AMI or EBS snapshot creation slow? in the AWS Knowledge Center.
- Depending on the AWS Regions involved and the amount of data to be copied, a cross-Region snapshot copy can take hours to complete. In some cases, there might be a large number of cross-Region snapshot copy requests from a given source Region. In such cases, Amazon RDS might put new cross-Region copy requests from that source Region into a queue until some in-progress copies complete. No progress information is displayed about copy requests while they are in the queue. Progress information is displayed when the copy starts.

Snapshot retention

Amazon RDS deletes automated backups in several situations:

- At the end of their retention period.
- When you disable automated backups for a DB cluster.
- When you delete a DB cluster.

If you want to keep an automated backup for a longer period, copy it to create a manual snapshot, which is retained until you delete it. Amazon RDS storage costs might apply to manual snapshots if they exceed your default storage space.

For more information about backup storage costs, see Amazon RDS pricing.

Copying shared snapshots

You can copy snapshots shared to you by other AWS accounts. In some cases, you might copy an encrypted snapshot that has been shared from another AWS account. In these cases, you must have access to the AWS KMS key that was used to encrypt the snapshot.

You can only copy a shared DB cluster snapshot, whether encrypted or not, in the same AWS Region. For more information, see Sharing encrypted snapshots.

Handling encryption

You can copy a snapshot that has been encrypted using a KMS key. If you copy an encrypted snapshot, the copy of the snapshot must also be encrypted. If you copy an encrypted snapshot within the same AWS Region, you can encrypt the copy with the same KMS key as the original snapshot. Or you can specify a different KMS key.

If you copy an encrypted snapshot across Regions, you must specify a KMS key valid in the destination AWS Region. It can be a Region-specific KMS key, or a multi-Region key. For more information on multi-Region KMS keys, see Using multi-Region keys in AWS KMS.

The source snapshot remains encrypted throughout the copy process. For more information, see Limitations of Amazon Aurora encrypted DB clusters.

> **Note**
>
> For Amazon Aurora DB cluster snapshots, you can't encrypt an unencrypted DB cluster snapshot when you copy the snapshot.

Incremental snapshot copying

Aurora doesn't support incremental snapshot copying. Aurora DB cluster snapshot copies are always full copies. A full snapshot copy contains all of the data and metadata required to restore the DB cluster.

Cross-Region snapshot copying

You can copy DB cluster snapshots across AWS Regions. However, there are certain constraints and considerations for cross-Region snapshot copying.

Depending on the AWS Regions involved and the amount of data to be copied, a cross-Region snapshot copy can take hours to complete.

In some cases, there might be a large number of cross-Region snapshot copy requests from a given source AWS Region. In such cases, Amazon RDS might put new cross-Region copy requests from that source AWS Region into a queue until some in-progress copies complete. No progress information is displayed about copy requests while they are in the queue. Progress information is displayed when the copying starts.

If you use AWS Backup for cross-Region snapshot copying, while the copies are full copies, the data transfer charges are incremental. For more information, see Creating backup copies across AWS Regions in the *AWS Backup Developer Guide*.

Parameter group considerations

When you copy a snapshot across Regions, the copy doesn't include the parameter group used by the original DB cluster. When you restore a snapshot to create a new DB cluster, that DB cluster gets the default parameter group for the AWS Region it is created in. To give the new DB cluster the same parameters as the original, do the following:

1. In the destination AWS Region, create a DB cluster parameter group with the same settings as the original DB cluster. If one already exists in the new AWS Region, you can use that one.
2. After you restore the snapshot in the destination AWS Region, modify the new DB cluster and add the new or existing parameter group from the previous step.

Copying a DB cluster snapshot

Use the procedures in this topic to copy a DB cluster snapshot. If your source database engine is Aurora, then your snapshot is a DB cluster snapshot.

For each AWS account, you can copy up to five DB cluster snapshots at a time from one AWS Region to another. Copying both encrypted and unencrypted DB cluster snapshots is supported. If you copy a DB cluster snapshot to another AWS Region, you create a manual DB cluster snapshot that is retained in that AWS Region. Copying a DB cluster snapshot out of the source AWS Region incurs Amazon RDS data transfer charges.

For more information about data transfer pricing, see Amazon RDS pricing.

After the DB cluster snapshot copy has been created in the new AWS Region, the DB cluster snapshot copy behaves the same as all other DB cluster snapshots in that AWS Region.

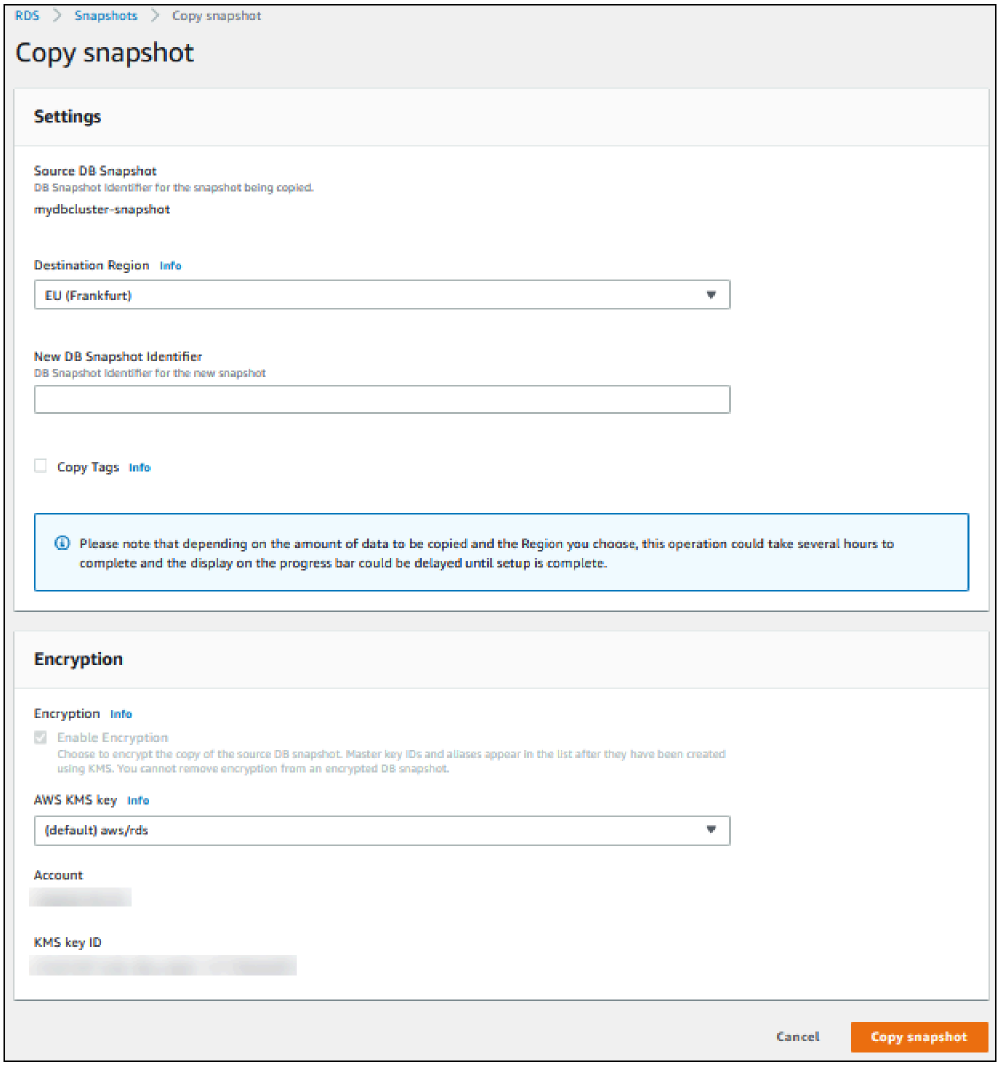
Console

This procedure works for copying encrypted or unencrypted DB cluster snapshots, in the same AWS Region or across Regions.

To cancel a copy operation once it is in progress, delete the target DB cluster snapshot while that DB cluster snapshot is in **copying** status.

To copy a DB cluster snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Select the DB cluster snapshot you want to copy.
4. For **Actions**, choose **Copy snapshot**. The **Copy snapshot** page appears.



RDS > Snapshots > Copy snapshot

Copy snapshot

Settings

Source DB Snapshot
DB Snapshot Identifier for the snapshot being copied.
mydbcluster-snapshot

Destination Region Info
EU (Frankfurt)

New DB Snapshot Identifier
DB Snapshot Identifier for the new snapshot

□ Copy Tags Info

Please note that depending on the amount of data to be copied and the Region you choose, this operation could take several hours to complete and the display on the progress bar could be delayed until setup is complete.

Encryption

Encryption Info
☑ Enable Encryption
Choose to encrypt the copy of the source DB snapshot. Master key IDs and aliases appear in the list after they have been created using KMS. You cannot remove encryption from an encrypted DB snapshot.

AWS KMS key Info
(default) aws/rds

Account
[redacted]

KMS key ID
[redacted]

5. (Optional) To copy the DB cluster snapshot to a different AWS Region, choose that AWS Region for **Destination Region**.

6. Enter the name of the DB cluster snapshot copy in **New DB Snapshot Identifier**.

7. To copy tags and values from the snapshot to the copy of the snapshot, choose **Copy Tags**.

8. Choose **Copy Snapshot**.

Copying an unencrypted DB cluster snapshot by using the AWS CLI or Amazon RDS API

Use the procedures in the following sections to copy an unencrypted DB cluster snapshot by using the AWS CLI or Amazon RDS API.

To cancel a copy operation once it is in progress, delete the target DB cluster snapshot identified by `--target-db-cluster-snapshot-identifier` or `TargetDBClusterSnapshotIdentifier` while that DB cluster snapshot is in **copying** status.

AWS CLI

To copy a DB cluster snapshot, use the AWS CLI copy-db-cluster-snapshot command. If you are copying the snapshot to another AWS Region, run the command in the AWS Region to which the snapshot will be copied.

The following options are used to copy an unencrypted DB cluster snapshot:

- `--source-db-cluster-snapshot-identifier` – The identifier for the DB cluster snapshot to be copied. If you are copying the snapshot to another AWS Region, this identifier must be in the ARN format for the source AWS Region.
- `--target-db-cluster-snapshot-identifier` – The identifier for the new copy of the DB cluster snapshot.

The following code creates a copy of DB cluster snapshot `arn:aws:rds:us-east-1:123456789012:cluster-snapshot:aurora-cluster1-snapshot-20130805` named `myclustersnapshotcopy` in the AWS Region in which the command is run. When the copy is made, all tags on the original snapshot are copied to the snapshot copy.

Example

For Linux, macOS, or Unix:

```
aws rds copy-db-cluster-snapshot \
  --source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:123456789012:cluster-
snapshot:aurora-cluster1-snapshot-20130805 \
  --target-db-cluster-snapshot-identifier myclustersnapshotcopy \
  --copy-tags
```

For Windows:

```
aws rds copy-db-cluster-snapshot ^
  --source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:123456789012:cluster-
snapshot:aurora-cluster1-snapshot-20130805 ^
```

```
  --target-db-cluster-snapshot-identifier myclustersnapshotcopy ^
  --copy-tags
```

RDS API

To copy a DB cluster snapshot, use the Amazon RDS API CopyDBClusterSnapshot operation. If you are copying the snapshot to another AWS Region, perform the action in the AWS Region to which the snapshot will be copied.

The following parameters are used to copy an unencrypted DB cluster snapshot:

- `SourceDBClusterSnapshotIdentifier` – The identifier for the DB cluster snapshot to be copied. If you are copying the snapshot to another AWS Region, this identifier must be in the ARN format for the source AWS Region.
- `TargetDBClusterSnapshotIdentifier` – The identifier for the new copy of the DB cluster snapshot.

The following code creates a copy of a snapshot `arn:aws:rds:us-east-1:123456789012:cluster-snapshot:aurora-cluster1-snapshot-20130805` named `myclustersnapshotcopy` in the US West (N. California) Region. When the copy is made, all tags on the original snapshot are copied to the snapshot copy.

Example

```
https://rds.us-west-1.amazonaws.com/
   ?Action=CopyDBClusterSnapshot
   &CopyTags=true
   &SignatureMethod=HmacSHA256
   &SignatureVersion=4
   &SourceDBSnapshotIdentifier=arn%3Aaws%3Ards%3Aus-east-1%3A123456789012%3Acluster-
snapshot%3Aaurora-cluster1-snapshot-20130805
   &TargetDBSnapshotIdentifier=myclustersnapshotcopy
   &Version=2013-09-09
   &X-Amz-Algorithm=AWS4-HMAC-SHA256
   &X-Amz-Credential=AKIADQKE4SARGYLE/20140429/us-west-1/rds/aws4_request
   &X-Amz-Date=20140429T175351Z
   &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
   &X-Amz-Signature=9164337efa99caf850e874a1cb7ef62f3cea29d0b448b9e0e7c53b288ddffed2
```

Copying an encrypted DB cluster snapshot by using the AWS CLI or Amazon RDS API

Use the procedures in the following sections to copy an encrypted DB cluster snapshot by using the AWS CLI or Amazon RDS API.

To cancel a copy operation once it is in progress, delete the target DB cluster snapshot identified by `--target-db-cluster-snapshot-identifier` or `TargetDBClusterSnapshotIdentifier` while that DB cluster snapshot is in **copying** status.

AWS CLI

To copy a DB cluster snapshot, use the AWS CLI copy-db-cluster-snapshot command. If you are copying the snapshot to another AWS Region, run the command in the AWS Region to which the snapshot will be copied.

The following options are used to copy an encrypted DB cluster snapshot:

- `--source-db-cluster-snapshot-identifier` – The identifier for the encrypted DB cluster snapshot to be copied. If you are copying the snapshot to another AWS Region, this identifier must be in the ARN format for the source AWS Region.
- `--target-db-cluster-snapshot-identifier` – The identifier for the new copy of the encrypted DB cluster snapshot.
- `--kms-key-id` – The KMS key identifier for the key to use to encrypt the copy of the DB cluster snapshot.

  You can optionally use this option if the DB cluster snapshot is encrypted, you copy the snapshot in the same AWS Region, and you want to specify a new KMS key to encrypt the copy. Otherwise, the copy of the DB cluster snapshot is encrypted with the same KMS key as the source DB cluster snapshot.

  You must use this option if the DB cluster snapshot is encrypted and you are copying the snapshot to another AWS Region. In that case, you must specify a KMS key for the destination AWS Region.

The following code example copies the encrypted DB cluster snapshot from the US West (Oregon) Region to the US East (N. Virginia) Region. The command is called in the US East (N. Virginia) Region.

Example

For Linux, macOS, or Unix:

```
aws rds copy-db-cluster-snapshot \
  --source-db-cluster-snapshot-identifier arn:aws:rds:us-west-2:123456789012:cluster-snapshot:aurora-cluster1-snapshot-20161115 \
  --target-db-cluster-snapshot-identifier myclustersnapshotcopy \
  --kms-key-id my-us-east-1-key
```

For Windows:

```
aws rds copy-db-cluster-snapshot ^
  --source-db-cluster-snapshot-identifier arn:aws:rds:us-west-2:123456789012:cluster-snapshot:aurora-cluster1-snapshot-20161115 ^
  --target-db-cluster-snapshot-identifier myclustersnapshotcopy ^
  --kms-key-id my-us-east-1-key
```

The `--source-region` parameter is required when you're copying an encrypted DB cluster snapshot between the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions. For `--source-region`, specify the AWS Region of the source DB instance. The AWS Region specified in `source-db-cluster-snapshot-identifier` must match the AWS Region specified for `--source-region`.

If `--source-region` isn't specified, specify a `--pre-signed-url` value. A *presigned URL* is a URL that contains a Signature Version 4 signed request for the `copy-db-cluster-snapshot` command that's called in the source AWS Region. To learn more about the `pre-signed-url` option, see copy-db-cluster-snapshot in the *AWS CLI Command Reference*.

RDS API

To copy a DB cluster snapshot, use the Amazon RDS API CopyDBClusterSnapshot operation. If you are copying the snapshot to another AWS Region, perform the action in the AWS Region to which the snapshot will be copied.

The following parameters are used to copy an encrypted DB cluster snapshot:

- `SourceDBClusterSnapshotIdentifier` – The identifier for the encrypted DB cluster snapshot to be copied. If you are copying the snapshot to another AWS Region, this identifier must be in the ARN format for the source AWS Region.

- `TargetDBClusterSnapshotIdentifier` – The identifier for the new copy of the encrypted DB cluster snapshot.
- `KmsKeyId` – The KMS key identifier for the key to use to encrypt the copy of the DB cluster snapshot.

  You can optionally use this parameter if the DB cluster snapshot is encrypted, you copy the snapshot in the same AWS Region, and you specify a new KMS key to use to encrypt the copy. Otherwise, the copy of the DB cluster snapshot is encrypted with the same KMS key as the source DB cluster snapshot.

  You must use this parameter if the DB cluster snapshot is encrypted and you are copying the snapshot to another AWS Region. In that case, you must specify a KMS key for the destination AWS Region.
- `PreSignedUrl` – If you are copying the snapshot to another AWS Region, you must specify the `PreSignedUrl` parameter. The `PreSignedUrl` value must be a URL that contains a Signature Version 4 signed request for the `CopyDBClusterSnapshot` action to be called in the source AWS Region where the DB cluster snapshot is copied from. To learn more about using a presigned URL, see CopyDBClusterSnapshot.

The following code example copies the encrypted DB cluster snapshot from the US West (Oregon) Region to the US East (N. Virginia) Region. The action is called in the US East (N. Virginia) Region.

Example

```
https://rds.us-east-1.amazonaws.com/
    ?Action=CopyDBClusterSnapshot
    &KmsKeyId=my-us-east-1-key
    &PreSignedUrl=https%253A%252F%252Frds.us-west-2.amazonaws.com%252F
         %253FAction%253DCopyDBClusterSnapshot
         %2526DestinationRegion%253Dus-east-1
         %2526KmsKeyId%253Dmy-us-east-1-key
         %2526SourceDBClusterSnapshotIdentifier%253Darn%25253Aaws%25253Ards
%25253Aus-west-2%25253A123456789012%25253Acluster-snapshot%25253Aaurora-cluster1-
snapshot-20161115
         %2526SignatureMethod%253DHmacSHA256
         %2526SignatureVersion%253D4
         %2526Version%253D2014-10-31
         %2526X-Amz-Algorithm%253DAWS4-HMAC-SHA256
         %2526X-Amz-Credential%253DAKIADQKE4SARGYLE%252F20161117%252Fus-west-2%252Frds
%252Faws4_request
```

```
         %2526X-Amz-Date%253D20161117T215409Z
         %2526X-Amz-Expires%253D3600
         %2526X-Amz-SignedHeaders%253Dcontent-type%253Bhost%253Buser-agent%253Bx-amz-
content-sha256%253Bx-amz-date
         %2526X-Amz-Signature
%253D255a0f17b4e717d3b67fad163c3ec26573b882c03a65523522cf890a67fca613
    &SignatureMethod=HmacSHA256
    &SignatureVersion=4
    &SourceDBClusterSnapshotIdentifier=arn%3Aaws%3Ards%3Aus-
west-2%3A123456789012%3Acluster-snapshot%3Aaurora-cluster1-snapshot-20161115
    &TargetDBClusterSnapshotIdentifier=myclustersnapshotcopy
    &Version=2014-10-31
    &X-Amz-Algorithm=AWS4-HMAC-SHA256
    &X-Amz-Credential=AKIADQKE4SARGYLE/20161117/us-east-1/rds/aws4_request
    &X-Amz-Date=20161117T221704Z
    &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
    &X-Amz-Signature=da4f2da66739d2e722c85fcfd225dc27bba7e2b8dbea8d8612434378e52adccf
```

The `PreSignedUrl` parameter is required when you are copying an encrypted DB cluster snapshot between the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions. The `PreSignedUrl` value must be a URL that contains a Signature Version 4 signed request for the `CopyDBClusterSnapshot` operation to be called in the source AWS Region where the DB cluster snapshot is copied from. To learn more about using a presigned URL, see CopyDBClusterSnapshot in the *Amazon RDS API Reference*.

To automatically rather than manually generate a presigned URL, use the AWS CLI copy-db-cluster-snapshot command with the `--source-region` option instead.

Copying a DB cluster snapshot across accounts

You can enable other AWS accounts to copy DB cluster snapshots that you specify by using the Amazon RDS API `ModifyDBClusterSnapshotAttribute` and `CopyDBClusterSnapshot` actions. You can only copy DB cluster snapshots across accounts in the same AWS Region. The cross-account copying process works as follows, where Account A is making the snapshot available to copy, and Account B is copying it.

1. Using Account A, call `ModifyDBClusterSnapshotAttribute`, specifying **`restore`** for the `AttributeName` parameter, and the ID for Account B for the `ValuesToAdd` parameter.
2. (If the snapshot is encrypted) Using Account A, update the key policy for the KMS key, first adding the ARN of Account B as a `Principal`, and then allow the `kms:CreateGrant` action.

3. (If the snapshot is encrypted) Using Account B, choose or create a user and attach an IAM policy to that user that allows it to copy an encrypted DB cluster snapshot using your KMS key.
4. Using Account B, call `CopyDBClusterSnapshot` and use the `SourceDBClusterSnapshotIdentifier` parameter to specify the ARN of the DB cluster snapshot to be copied, which must include the ID for Account A.

To list all of the AWS accounts permitted to restore a DB cluster snapshot, use the DescribeDBSnapshotAttributes or DescribeDBClusterSnapshotAttributes API operation.

To remove sharing permission for an AWS account, use the `ModifyDBSnapshotAttribute` or `ModifyDBClusterSnapshotAttribute` action with `AttributeName` set to `restore` and the ID of the account to remove in the `ValuesToRemove` parameter.

Copying an unencrypted DB cluster snapshot to another account

Use the following procedure to copy an unencrypted DB cluster snapshot to another account in the same AWS Region.

1. In the source account for the DB cluster snapshot, call `ModifyDBClusterSnapshotAttribute`, specifying **`restore`** for the `AttributeName` parameter, and the ID for the target account for the `ValuesToAdd` parameter.

   Running the following example using the account `987654321` permits two AWS account identifiers, `123451234512` and `123456789012`, to restore the DB cluster snapshot named `manual-snapshot1`.

   ```
   https://rds.us-west-2.amazonaws.com/
    ?Action=ModifyDBClusterSnapshotAttribute
    &AttributeName=restore
    &DBClusterSnapshotIdentifier=manual-snapshot1
    &SignatureMethod=HmacSHA256&SignatureVersion=4
    &ValuesToAdd.member.1=123451234512
    &ValuesToAdd.member.2=123456789012
    &Version=2014-10-31
    &X-Amz-Algorithm=AWS4-HMAC-SHA256
    &X-Amz-Credential=AKIADQKE4SARGYLE/20150922/us-west-2/rds/aws4_request
    &X-Amz-Date=20150922T220515Z
    &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
    &X-Amz-Signature=ef38f1ce3dab4e1dbf113d8d2a265c67d17ece1999ffd36be85714ed36dddbb3
   ```

2. In the target account, call `CopyDBClusterSnapshot` and use the `SourceDBClusterSnapshotIdentifier` parameter to specify the ARN of the DB cluster snapshot to be copied, which must include the ID for the source account.

   Running the following example using the account `123451234512` copies the DB cluster snapshot `aurora-cluster1-snapshot-20130805` from account `987654321` and creates a DB cluster snapshot named `dbclustersnapshot1`.

```
https://rds.us-west-2.amazonaws.com/
   ?Action=CopyDBClusterSnapshot
   &CopyTags=true
   &SignatureMethod=HmacSHA256
   &SignatureVersion=4
   &SourceDBClusterSnapshotIdentifier=arn:aws:rds:us-west-2:987654321:cluster-
snapshot:aurora-cluster1-snapshot-20130805
   &TargetDBClusterSnapshotIdentifier=dbclustersnapshot1
   &Version=2013-09-09
   &X-Amz-Algorithm=AWS4-HMAC-SHA256
   &X-Amz-Credential=AKIADQKE4SARGYLE/20150922/us-west-2/rds/aws4_request
   &X-Amz-Date=20140429T175351Z
   &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-
date
   &X-Amz-
Signature=9164337efa99caf850e874a1cb7ef62f3cea29d0b448b9e0e7c53b288ddffed2
```

Copying an encrypted DB cluster snapshot to another account

Use the following procedure to copy an encrypted DB cluster snapshot to another account in the same AWS Region.

1. In the source account for the DB cluster snapshot, call `ModifyDBClusterSnapshotAttribute`, specifying **`restore`** for the `AttributeName` parameter, and the ID for the target account for the `ValuesToAdd` parameter.

   Running the following example using the account `987654321` permits two AWS account identifiers, `123451234512` and `123456789012`, to restore the DB cluster snapshot named `manual-snapshot1`.

```
https://rds.us-west-2.amazonaws.com/
 ?Action=ModifyDBClusterSnapshotAttribute
```

```
&AttributeName=restore
&DBClusterSnapshotIdentifier=manual-snapshot1
&SignatureMethod=HmacSHA256&SignatureVersion=4
&ValuesToAdd.member.1=123451234512
&ValuesToAdd.member.2=123456789012
&Version=2014-10-31
&X-Amz-Algorithm=AWS4-HMAC-SHA256
&X-Amz-Credential=AKIADQKE4SARGYLE/20150922/us-west-2/rds/aws4_request
&X-Amz-Date=20150922T220515Z
&X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
&X-Amz-Signature=ef38f1ce3dab4e1dbf113d8d2a265c67d17ece1999ffd36be85714ed36dddbb3
```

2. In the source account for the DB cluster snapshot, create a custom KMS key in the same AWS Region as the encrypted DB cluster snapshot. While creating the customer managed key, you give access to it for the target AWS account. For more information, see Create a customer managed key and give access to it.

3. Copy and share the snapshot to the target AWS account. For more information, see Copy and share the snapshot from the source account.

4. In the target account, call `CopyDBClusterSnapshot` and use the `SourceDBClusterSnapshotIdentifier` parameter to specify the ARN of the DB cluster snapshot to be copied, which must include the ID for the source account.

   Running the following example using the account `123451234512` copies the DB cluster snapshot `aurora-cluster1-snapshot-20130805` from account `987654321` and creates a DB cluster snapshot named `dbclustersnapshot1`.

```
https://rds.us-west-2.amazonaws.com/
   ?Action=CopyDBClusterSnapshot
   &CopyTags=true
   &SignatureMethod=HmacSHA256
   &SignatureVersion=4
   &SourceDBClusterSnapshotIdentifier=arn:aws:rds:us-west-2:987654321:cluster-
snapshot:aurora-cluster1-snapshot-20130805
   &TargetDBClusterSnapshotIdentifier=dbclustersnapshot1
   &Version=2013-09-09
   &X-Amz-Algorithm=AWS4-HMAC-SHA256
   &X-Amz-Credential=AKIADQKE4SARGYLE/20150922/us-west-2/rds/aws4_request
   &X-Amz-Date=20140429T175351Z
   &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-
date
```

```
    &X-Amz-
Signature=9164337efa99caf850e874a1cb7ef62f3cea29d0b448b9e0e7c53b288ddffed2
```

Sharing a DB cluster snapshot

Using Amazon RDS, you can share a manual DB cluster snapshot in the following ways:

- Sharing a manual DB cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to copy the snapshot.
- Sharing a manual DB cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to directly restore a DB cluster from the snapshot instead of taking a copy of it and restoring from that.

> **Note**
>
> To share an automated DB cluster snapshot, create a manual DB cluster snapshot by copying the automated snapshot, and then share that copy. This process also applies to AWS Backup–generated resources.

For more information on copying a snapshot, see Copying a DB cluster snapshot. For more information on restoring a DB instance from a DB cluster snapshot, see Restoring from a DB cluster snapshot.

For more information on restoring a DB cluster from a DB cluster snapshot, see Overview of backing up and restoring an Aurora DB cluster.

You can share a manual snapshot with up to 20 other AWS accounts.

The following limitation applies when sharing manual snapshots with other AWS accounts:

- When you restore a DB cluster from a shared snapshot using the AWS Command Line Interface (AWS CLI) or Amazon RDS API, you must specify the Amazon Resource Name (ARN) of the shared snapshot as the snapshot identifier.

Contents

Sharing a snapshot

You can share a DB cluster snapshot using the AWS Management Console, the AWS CLI, or the RDS API.

Console

Using the Amazon RDS console, you can share a manual DB cluster snapshot with up to 20 AWS accounts. You can also use the console to stop sharing a manual snapshot with one or more accounts.

To share a manual DB cluster snapshot by using the Amazon RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Select the manual snapshot that you want to share.
4. For **Actions**, choose **Share snapshot**.
5. Choose one of the following options for **DB snapshot visibility**.

   - If the source is unencrypted, choose **Public** to permit all AWS accounts to restore a DB cluster from your manual DB cluster snapshot, or choose **Private** to permit only AWS accounts that you specify to restore a DB cluster from your manual DB cluster snapshot.

     > ⚠ **Warning**
     >
     > If you set **DB snapshot visibility** to **Public**, all AWS accounts can restore a DB cluster from your manual DB cluster snapshot and have access to your data. Do not share any manual DB cluster snapshots that contain private information as **Public**.

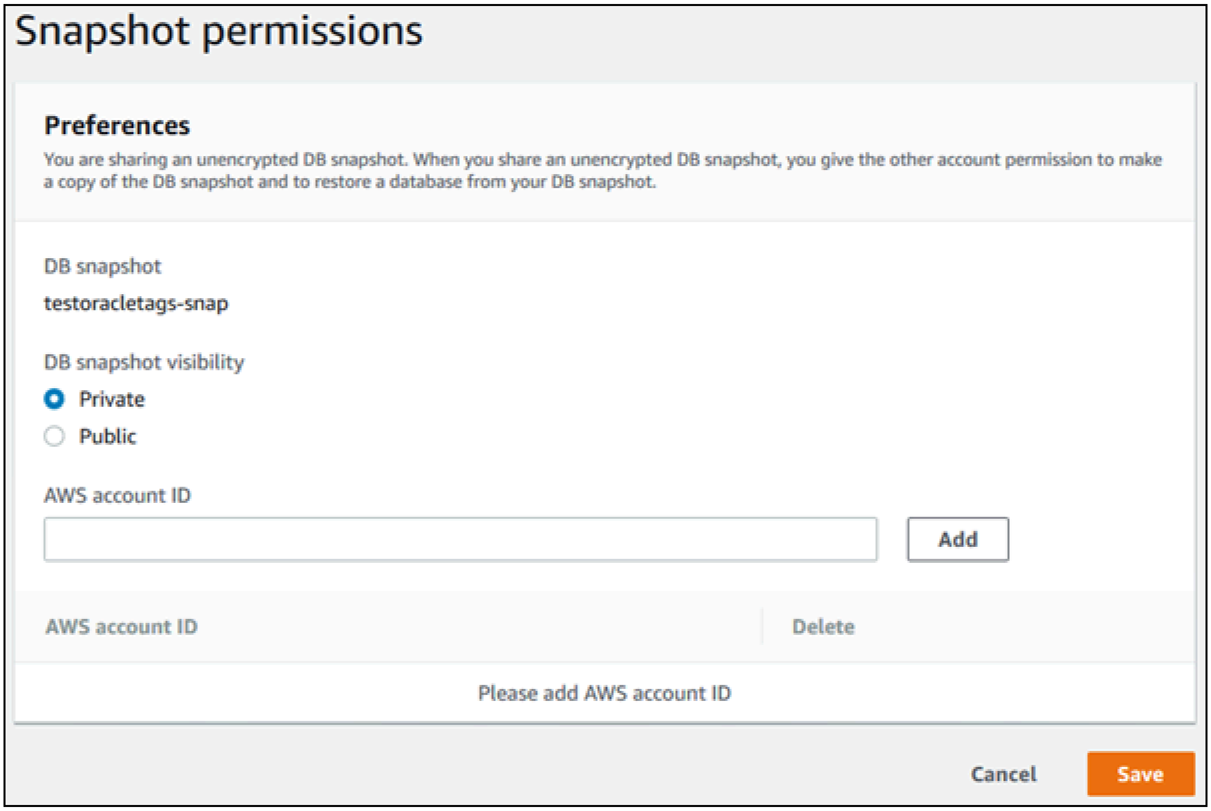
For more information, see Sharing public snapshots.

- If the source is encrypted, **DB snapshot visibility** is set as **Private** because encrypted snapshots can't be shared as public.

> **Note**
>
> Snapshots that have been encrypted with the default AWS KMS key can't be shared. For information on how to work around this issue, see Sharing encrypted snapshots.

6. For **AWS Account ID**, enter the AWS account identifier for an account that you want to permit to restore a DB cluster from your manual snapshot, and then choose **Add**. Repeat to include additional AWS account identifiers, up to 20 AWS accounts.

   If you make an error when adding an AWS account identifier to the list of permitted accounts, you can delete it from the list by choosing **Delete** at the right of the incorrect AWS account identifier.



Snapshot permissions

Preferences

You are sharing an unencrypted DB snapshot. When you share an unencrypted DB snapshot, you give the other account permission to make a copy of the DB snapshot and to restore a database from your DB snapshot.

DB snapshot

testoracletags-snap

DB snapshot visibility

● Private

○ Public

AWS account ID

AWS account ID	Delete
Please add AWS account ID	

7. After you have added identifiers for all of the AWS accounts that you want to permit to restore the manual snapshot, choose **Save** to save your changes.

AWS CLI

To share a DB cluster snapshot, use the `aws rds modify-db-cluster-snapshot-attribute` command. Use the `--values-to-add` parameter to add a list of the IDs for the AWS accounts that are authorized to restore the manual snapshot.

Example of sharing a snapshot with a single account

The following example enables AWS account identifier `123456789012` to restore the DB cluster snapshot named `cluster-3-snapshot`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-snapshot-attribute \
--db-cluster-snapshot-identifier cluster-3-snapshot \
--attribute-name restore \
--values-to-add 123456789012
```

For Windows:

```
aws rds modify-db-cluster-snapshot-attribute ^
--db-cluster-snapshot-identifier cluster-3-snapshot ^
--attribute-name restore ^
--values-to-add 123456789012
```

Example of sharing a snapshot with multiple accounts

The following example enables two AWS account identifiers, `111122223333` and `444455556666`, to restore the DB cluster snapshot named `manual-cluster-snapshot1`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-snapshot-attribute \
--db-cluster-snapshot-identifier manual-cluster-snapshot1 \
--attribute-name restore \
--values-to-add {"111122223333","444455556666"}
```

For Windows:

```
aws rds modify-db-cluster-snapshot-attribute ^
--db-cluster-snapshot-identifier manual-cluster-snapshot1 ^
--attribute-name restore ^
```

```
--values-to-add "[\"111122223333\",\"444455556666\"]"
```

> **Note**
>
> When using the Windows command prompt, you must escape double quotes (") in JSON code by prefixing them with a backslash (\).

To list the AWS accounts enabled to restore a snapshot, use the `describe-db-cluster-snapshot-attributes` AWS CLI command.

RDS API

You can also share a manual DB cluster snapshot with other AWS accounts by using the Amazon RDS API. To do so, call the `ModifyDBClusterSnapshotAttribute` operation. Specify `restore` for `AttributeName`, and use the `ValuesToAdd` parameter to add a list of the IDs for the AWS accounts that are authorized to restore the manual snapshot.

To make a manual snapshot public and restorable by all AWS accounts, use the value `all`. However, take care not to add the `all` value for any manual snapshots that contain private information that you don't want to be available to all AWS accounts. Also, don't specify `all` for encrypted snapshots, because making such snapshots public isn't supported.

To list all of the AWS accounts permitted to restore a snapshot, use the `DescribeDBClusterSnapshotAttributes` API operation.

Sharing public snapshots

You can share an unencrypted manual snapshot as public, which makes the snapshot available to all AWS accounts. Make sure when sharing a snapshot as public that none of your private information is included in the public snapshot.

When a snapshot is shared publicly, it gives all AWS accounts permission both to copy the snapshot and to create DB clusters from it.

You aren't billed for the backup storage of public snapshots owned by other accounts. You're billed only for snapshots that you own.

If you copy a public snapshot, you own the copy. You're billed for the backup storage of your snapshot copy. If you create a DB cluster from a public snapshot, you're billed for that DB cluster. For Amazon Aurora pricing information, see the Aurora pricing page.

You can delete only the public snapshots that you own. To delete a shared or public snapshot, make sure to log into the AWS account that owns the snapshot.

Viewing public snapshots owned by other AWS accounts

You can view public snapshots owned by other accounts in a particular AWS Region on the **Public** tab of the **Snapshots** page in the Amazon RDS console. Your snapshots (those owned by your account) don't appear on this tab.

To view public snapshots

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Choose the **Public** tab.

   The public snapshots appear. You can see which account owns a public snapshot in the **Owner** column.

   > **Note**
   >
   > You might have to modify the page preferences, by selecting the gear icon at the upper right of the **Public snapshots** list, to see this column.

Viewing your own public snapshots

You can use the following AWS CLI command (Unix only) to view the public snapshots owned by your AWS account in a particular AWS Region.

```
aws rds describe-db-cluster-snapshots --snapshot-type public --include-public |
 grep account_number
```

The output returned is similar to the following example if you have public snapshots.

```
"DBClusterSnapshotArn": "arn:aws:rds:us-west-2:123456789012:cluster-
snapshot:myclustersnapshot1",
"DBClusterSnapshotArn": "arn:aws:rds:us-west-2:123456789012:cluster-
snapshot:myclustersnapshot2",
```

Sharing public snapshots from deprecated DB engine versions

Restoring or copying public snapshots from deprecated DB engine versions isn't supported. To make your existing unsupported public snapshot available to restore or copy, perform the following steps:

1. Mark the snapshot as private.
2. Restore the snapshot.
3. Upgrade the restored DB cluster to a supported engine version.
4. Create a new snapshot.
5. Re-share the snapshot publicly.

Sharing encrypted snapshots

You can share DB cluster snapshots that have been encrypted "at rest" using the AES-256 encryption algorithm, as described in Encrypting Amazon Aurora resources.

The following restrictions apply to sharing encrypted snapshots:

- You can't share encrypted snapshots as public.
- You can't share a snapshot that has been encrypted using the default KMS key of the AWS account that shared the snapshot.

To work around the default KMS key issue, perform the following tasks:

1. Create a customer managed key and give access to it.
2. Copy and share the snapshot from the source account.
3. Copy the shared snapshot in the target account.

Create a customer managed key and give access to it

First you create a custom KMS key in the same AWS Region as the encrypted DB cluster snapshot. While creating the customer managed key, you give access to it for another AWS account.

To create a customer managed key and give access to it

1. Sign in to the AWS Management Console from the source AWS account.

2. Open the AWS KMS console at https://console.aws.amazon.com/kms.
3. To change the AWS Region, use the Region selector in the upper-right corner of the page.
4. In the navigation pane, choose **Customer managed keys**.
5. Choose **Create key**.
6. On the **Configure key** page:
    a. For **Key type**, select **Symmetric**.
    b. For **Key usage**, select **Encrypt and decrypt**.
    c. Expand **Advanced options**.
    d. For **Key material origin**, select **KMS**.
    e. For **Regionality**, select **Single-Region key**.
    f. Choose **Next**.
7. On the **Add labels** page:
    a. For **Alias**. enter a display name for your KMS key, for example **`share-snapshot`**.
    b. (Optional) Enter a description for your KMS key.
    c. (Optional) Add tags to your KMS key.
    d. Choose **Next**.
8. On the **Define key administrative permissions** page, choose **Next.**
9. On the **Define key usage permissions** page:
    a. For **Other AWS accounts**, choose **Add another AWS account**.
    b. Enter the ID of the AWS account to which you want to give access.

       You can give access to multiple AWS accounts.
    c. Choose **Next**.
10. Review your KMS key, then choose **Finish**.

Copy and share the snapshot from the source account

Next you copy the source DB cluster snapshot to a new snapshot using the customer managed key. Then you share it with the target AWS account.

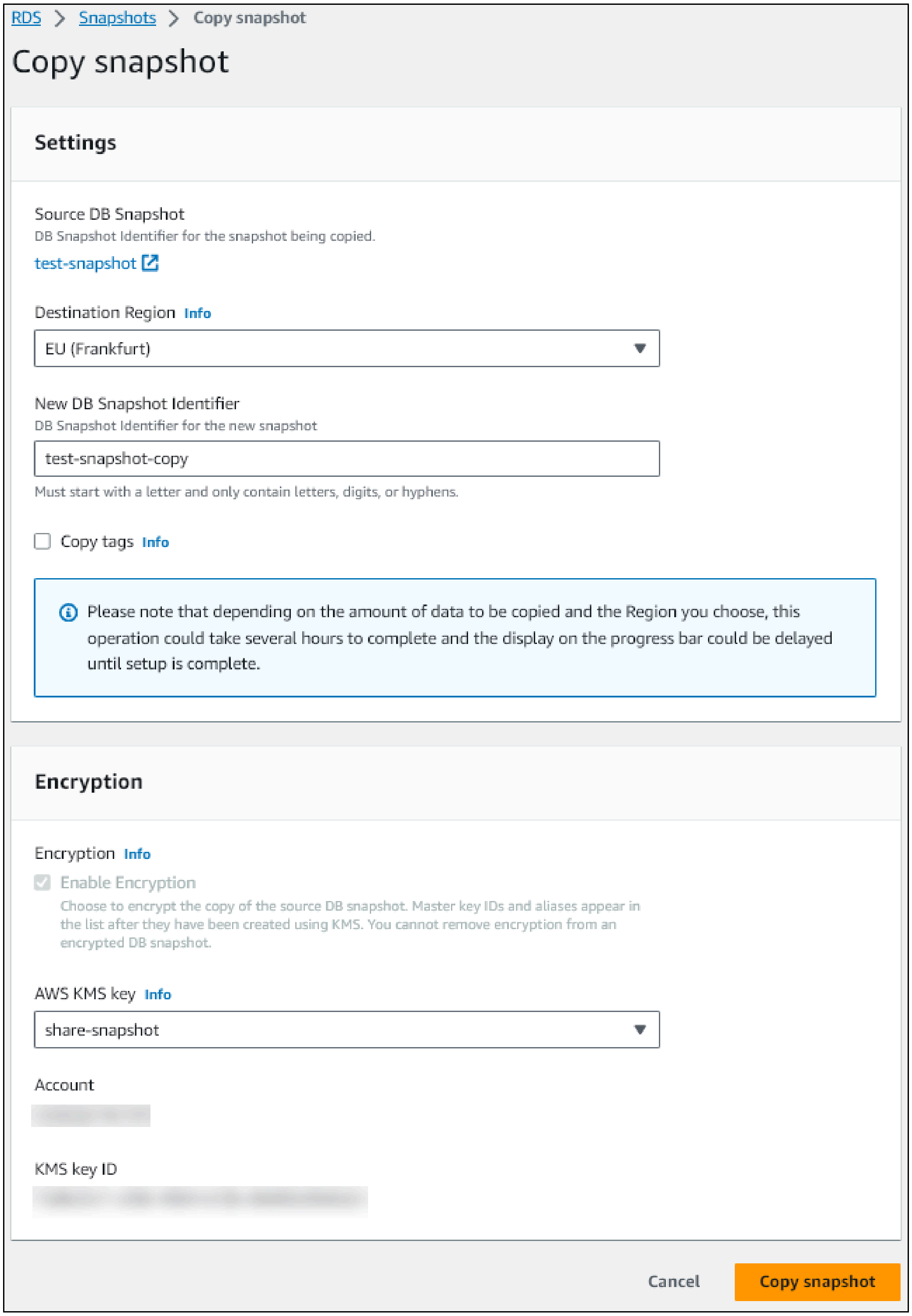
To copy and share the snapshot

1. Sign in to the AWS Management Console from the source AWS account.
2. Open the Amazon RDS console at https://console.aws.amazon.com/rds/
3. In the navigation pane, choose **Snapshots**.
4. Select the DB cluster snapshot you want to copy.
5. For **Actions**, choose **Copy snapshot**.
6. On the **Copy snapshot** page:

   a. For **Destination Region**, choose the AWS Region where you created the customer managed key in the previous procedure.

   b. Enter the name of the DB cluster snapshot copy in **New DB Snapshot Identifier**.

   c. For **AWS KMS key**, choose the customer managed key that you created.



RDS > Snapshots > Copy snapshot

Copy snapshot

Settings

Source DB Snapshot
DB Snapshot Identifier for the snapshot being copied.
test-snapshot

Destination Region Info
EU (Frankfurt)

New DB Snapshot Identifier
DB Snapshot Identifier for the new snapshot
test-snapshot-copy
Must start with a letter and only contain letters, digits, or hyphens.

□ Copy tags Info

Please note that depending on the amount of data to be copied and the Region you choose, this operation could take several hours to complete and the display on the progress bar could be delayed until setup is complete.

Encryption

Encryption Info
☑ Enable Encryption
Choose to encrypt the copy of the source DB snapshot. Master key IDs and aliases appear in the list after they have been created using KMS. You cannot remove encryption from an encrypted DB snapshot.

AWS KMS key Info
share-snapshot

Account
[redacted]

KMS key ID
[redacted]

Cancel | Copy snapshot

d. Choose **Copy snapshot**.

7. When the snapshot copy is available, select it.

8. For **Actions**, choose **Share snapshot**.

9. On the **Snapshot permissions** page:

   a. Enter the **AWS account ID** with which you're sharing the snapshot copy, then choose **Add**.
   b. Choose **Save**.

   The snapshot is shared.

Copy the shared snapshot in the target account

Now you can copy the shared snapshot in the target AWS account.

To copy the shared snapshot

1. Sign in to the AWS Management Console from the target AWS account.
2. Open the Amazon RDS console at https://console.aws.amazon.com/rds/
3. In the navigation pane, choose **Snapshots**.
4. Choose the **Shared with me** tab.
5. Select the shared snapshot.
6. For **Actions**, choose **Copy snapshot**.
7. Choose your settings for copying the snapshot as in the previous procedure, but use an AWS KMS key that belongs to the target account.

   Choose **Copy snapshot**.

Stopping snapshot sharing

To stop sharing a DB cluster snapshot, you remove permission from the target AWS account.

Console

To stop sharing a manual DB cluster snapshot with an AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Select the manual snapshot that you want to stop sharing.
4. Choose **Actions**, and then choose **Share snapshot**.

5. To remove permission for an AWS account, choose **Delete** for the AWS account identifier for that account from the list of authorized accounts.
6. Choose **Save** to save your changes.

CLI

To remove an AWS account identifier from the list, use the `--values-to-remove` parameter.

Example of stopping snapshot sharing

The following example prevents AWS account ID 444455556666 from restoring the snapshot.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-snapshot-attribute \
--db-cluster-snapshot-identifier manual-cluster-snapshot1 \
--attribute-name restore \
--values-to-remove 444455556666
```

For Windows:

```
aws rds modify-db-cluster-snapshot-attribute ^
--db-cluster-snapshot-identifier manual-cluster-snapshot1 ^
--attribute-name restore ^
--values-to-remove 444455556666
```

RDS API

To remove sharing permission for an AWS account, use the `ModifyDBClusterSnapshotAttribute` operation with `AttributeName` set to `restore` and the `ValuesToRemove` parameter. To mark a manual snapshot as private, remove the value `all` from the values list for the `restore` attribute.

Exporting DB cluster data to Amazon S3

You can export data from a live Amazon Aurora DB cluster to an Amazon S3 bucket. The export process runs in the background and doesn't affect the performance of your active DB cluster.

By default, all data in the DB cluster is exported. However, you can choose to export specific sets of databases, schemas, or tables.

Amazon Aurora clones the DB cluster, extracts data from the clone, and stores the data in an Amazon S3 bucket. The data is stored in an Apache Parquet format that is compressed and consistent. Individual Parquet files are usually 1–10 MB in size.

The faster performance that you can get with exporting snapshot data for Aurora MySQL version 2 and version 3 doesn't apply to exporting DB cluster data. For more information, see Exporting DB cluster snapshot data to Amazon S3.

You're charged for exporting the entire DB cluster, whether you export all or partial data. For more information, see the Amazon Aurora pricing page.

After the data is exported, you can analyze the exported data directly through tools like Amazon Athena or Amazon Redshift Spectrum. For more information on using Athena to read Parquet data, see Parquet SerDe in the *Amazon Athena User Guide*. For more information on using Redshift Spectrum to read Parquet data, see COPY from columnar data formats in the *Amazon Redshift Database Developer Guide*.

Feature availability and support varies across specific versions of each database engine and across AWS Regions. For more information on version and Region availability of exporting DB cluster data to S3, see Supported Regions and Aurora DB engines for exporting cluster data to Amazon S3.

Topics

- File naming convention
- Data conversion and storage format

Limitations

Exporting DB cluster data to Amazon S3 has the following limitations:

- You can't run multiple export tasks for the same DB cluster simultaneously. This applies to both full and partial exports.
- You can have up to five concurrent DB snapshot export tasks in progress per AWS account.
- Aurora Serverless v1 DB clusters don't support exports to S3.
- Aurora MySQL and Aurora PostgreSQL support exports to S3 only for the provisioned engine mode.
- Exports to S3 don't support S3 prefixes containing a colon (:).
- The following characters in the S3 file path are converted to underscores (_) during export:

```
\ ` " (space)
```

- If a database, schema, or table has characters in its name other than the following, partial export isn't supported. However, you can export the entire DB cluster.
  - Latin letters (A–Z)
  - Digits (0–9)
  - Dollar symbol ($)
  - Underscore (_)
- Spaces ( ) and certain characters aren't supported in database table column names. Tables with the following characters in column names are skipped during export:

```
, ; { } ( ) \n \t = (space)
```

- Tables with slashes (/) in their names are skipped during export.
- Aurora PostgreSQL temporary and unlogged tables are skipped during export.
- If the data contains a large object, such as a BLOB or CLOB, that is close to or greater than 500 MB, then the export fails.
- If a table contains a large row that is close to or greater than 2 GB, then the table is skipped during export.

- For partial exports, the `ExportOnly` list has a maximum size of 200 KB.
- We strongly recommend that you use a unique name for each export task. If you don't use a unique task name, you might receive the following error message:

  ExportTaskAlreadyExistsFault: An error occurred (ExportTaskAlreadyExists) when calling the StartExportTask operation: The export task with the ID *xxxxx* already exists.
- Because some tables might be skipped, we recommend that you verify row and table counts in the data after export.

Overview of exporting DB cluster data

You use the following process to export DB cluster data to an Amazon S3 bucket. For more details, see the following sections.

1. Identify the DB cluster whose data you want to export.
2. Set up access to the Amazon S3 bucket.

   A *bucket* is a container for Amazon S3 objects or files. To provide the information to access a bucket, take the following steps:

   a. Identify the S3 bucket where the DB cluster data is to be exported. The S3 bucket must be in the same AWS Region as the DB cluster. For more information, see Identifying the Amazon S3 bucket for export.

   b. Create an AWS Identity and Access Management (IAM) role that grants the DB cluster export task access to the S3 bucket. For more information, see Providing access to an Amazon S3 bucket using an IAM role.
3. Create a symmetric encryption AWS KMS key for the server-side encryption. The KMS key is used by the cluster export task to set up AWS KMS server-side encryption when writing the export data to S3.

   The KMS key policy must include both the `kms:CreateGrant` and `kms:DescribeKey` permissions. For more information on using KMS keys in Amazon Aurora, see AWS KMS key management.

   If you have a deny statement in your KMS key policy, make sure to explicitly exclude the AWS service principal `export.rds.amazonaws.com`.

You can use a KMS key within your AWS account, or you can use a cross-account KMS key. For more information, see Using a cross-account AWS KMS key.

4. Export the DB cluster to Amazon S3 using the console or the `start-export-task` CLI command. For more information, see Exporting DB cluster data to an Amazon S3 bucket.
5. To access your exported data in the Amazon S3 bucket, see Uploading, downloading, and managing objects in the *Amazon Simple Storage Service User Guide*.

Setting up access to an Amazon S3 bucket

You identify the Amazon S3 bucket, then you give the DB cluster export task permission to access it.

Topics

- Identifying the Amazon S3 bucket for export
- Providing access to an Amazon S3 bucket using an IAM role
- Using a cross-account Amazon S3 bucket

Identifying the Amazon S3 bucket for export

Identify the Amazon S3 bucket to export the DB cluster data to. Use an existing S3 bucket or create a new S3 bucket.

> ⓘ **Note**
>
> The S3 bucket must be in the same AWS Region as the DB cluster.

For more information about working with Amazon S3 buckets, see the following in the *Amazon Simple Storage Service User Guide*:

- How do I view the properties for an S3 bucket?
- How do I enable default encryption for an Amazon S3 bucket?
- How do I create an S3 bucket?

Providing access to an Amazon S3 bucket using an IAM role

Before you export DB cluster data to Amazon S3, give the export tasks write-access permission to the Amazon S3 bucket.

To grant this permission, create an IAM policy that provides access to the bucket, then create an IAM role and attach the policy to the role. Later, you can assign the IAM role to your DB cluster export task.

> ⚠ **Important**
>
> If you plan to use the AWS Management Console to export your DB cluster, you can choose to create the IAM policy and the role automatically when you export the DB cluster. For instructions, see Exporting DB cluster data to an Amazon S3 bucket.

To give tasks access to Amazon S3

1. Create an IAM policy. This policy provides the bucket and object permissions that allow your DB cluster export task to access Amazon S3.

   In the policy, include the following required actions to allow the transfer of files from Amazon Aurora to an S3 bucket:

   - `s3:PutObject*`
   - `s3:GetObject*`
   - `s3:ListBucket`
   - `s3:DeleteObject*`
   - `s3:GetBucketLocation`

   In the policy, include the following resources to identify the S3 bucket and objects in the bucket. The following list of resources shows the Amazon Resource Name (ARN) format for accessing Amazon S3.

   - `arn:aws:s3:::`*`DOC-EXAMPLE-BUCKET`*
   - `arn:aws:s3:::`*`DOC-EXAMPLE-BUCKET`*`/*`

For more information about creating an IAM policy for Amazon Aurora, see Creating and using an IAM policy for IAM database access. See also Tutorial: Create and attach your first customer managed policy in the *IAM User Guide*.

The following AWS CLI command creates an IAM policy named `ExportPolicy` with these options. It grants access to a bucket named *DOC-EXAMPLE-BUCKET*.

> **Note**
>
> After you create the policy, note the ARN of the policy. You need the ARN for a subsequent step when you attach the policy to an IAM role.

```
aws iam create-policy  --policy-name ExportPolicy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExportPolicy",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject*",
                "s3:ListBucket",
                "s3:GetObject*",
                "s3:DeleteObject*",
                "s3:GetBucketLocation"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
            ]
        }
    ]
}'
```

2. Create an IAM role, so that Aurora can assume this IAM role on your behalf to access your Amazon S3 buckets. For more information, see Creating a role to delegate permissions to an IAM user in the *IAM User Guide*.

The following example shows using the AWS CLI command to create a role named `rds-s3-export-role`.

```
aws iam create-role  --role-name rds-s3-export-role  --assume-role-policy-document
 '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "export.rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole"
       }
     ]
   }'
```

3. Attach the IAM policy that you created to the IAM role that you created.

   The following AWS CLI command attaches the policy created earlier to the role named `rds-s3-export-role`. Replace *`your-policy-arn`* with the policy ARN that you noted in an earlier step.

```
aws iam attach-role-policy  --policy-arn your-policy-arn  --role-name rds-s3-
export-role
```

Using a cross-account Amazon S3 bucket

You can use S3 buckets across AWS accounts. For more information, see Using a cross-account Amazon S3 bucket.

Exporting DB cluster data to an Amazon S3 bucket

You can have up to five concurrent DB cluster export tasks in progress per AWS account.

Note

Exporting DB cluster data can take a while depending on your database type and size. The export task first clones and scales the entire database before extracting the data to

Amazon S3. The task's progress during this phase displays as **Starting**. When the task switches to exporting data to S3, progress displays as **In progress**.

The time it takes for the export to complete depends on the data stored in the database. For example, tables with well-distributed numeric primary key or index columns export the fastest. Tables that don't contain a column suitable for partitioning and tables with only one index on a string-based column take longer because the export uses a slower single-threaded process.

You can export DB cluster data to Amazon S3 using the AWS Management Console, the AWS CLI, or the RDS API.

If you use a Lambda function to export the DB cluster data, add the `kms:DescribeKey` action to the Lambda function policy. For more information, see AWS Lambda permissions.

Console

The **Export to Amazon S3** console option appears only for DB clusters that can be exported to Amazon S3. A DB cluster might not be available for export because of the following reasons:

- The DB engine isn't supported for S3 export.
- The DB cluster version isn't supported for S3 export.
- S3 export isn't supported in the AWS Region where the DB cluster was created.

To export DB cluster data

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster whose data you want to export.
4. For **Actions**, choose **Export to Amazon S3**.

   The **Export to Amazon S3** window appears.
5. For **Export identifier**, enter a name to identify the export task. This value is also used for the name of the file created in the S3 bucket.
6. Choose the data to be exported:

- Choose **All** to export all data in the DB cluster.
- Choose **Partial** to export specific parts of the DB cluster. To identify which parts of the cluster to export, enter one or more databases, schemas, or tables for **Identifiers**, separated by spaces.

  Use the following format:

  ```
  database[.schema][.table] database2[.schema2][.table2] ... databasen[.scheman]
  [.tablen]
  ```

  For example:

  ```
  mydatabase mydatabase2.myschema1 mydatabase2.myschema2.mytable1
   mydatabase2.myschema2.mytable2
  ```

7. For **S3 bucket**, choose the bucket to export to.

   To assign the exported data to a folder path in the S3 bucket, enter the optional path for **S3 prefix**.
8. For **IAM role**, either choose a role that grants you write access to your chosen S3 bucket, or create a new role.

   - If you created a role by following the steps in Providing access to an Amazon S3 bucket using an IAM role, choose that role.
   - If you didn't create a role that grants you write access to your chosen S3 bucket, then choose **Create a new role** to create the role automatically. Next, enter a name for the role in **IAM role name**.
9. For **KMS key**, enter the ARN for the key to use for encrypting the exported data.
10. Choose **Export to Amazon S3**.

AWS CLI

To export DB cluster data to Amazon S3 using the AWS CLI, use the start-export-task command with the following required options:

- `--export-task-identifier`
- `--source-arn` – the Amazon Resource Name (ARN) of the DB cluster

- `--s3-bucket-name`
- `--iam-role-arn`
- `--kms-key-id`

In the following examples, the export task is named *my-cluster-export*, which exports the data to an S3 bucket named *DOC-EXAMPLE-DESTINATION-BUCKET*.

Example

For Linux, macOS, or Unix:

```
aws rds start-export-task \
    --export-task-identifier my-cluster-export \
    --source-arn arn:aws:rds:us-west-2:123456789012:cluster:my-cluster \
    --s3-bucket-name DOC-EXAMPLE-DESTINATION-BUCKET \
    --iam-role-arn iam-role \
    --kms-key-id my-key
```

For Windows:

```
aws rds start-export-task ^
    --export-task-identifier my-DB-cluster-export ^
    --source-arn arn:aws:rds:us-west-2:123456789012:cluster:my-cluster ^
    --s3-bucket-name DOC-EXAMPLE-DESTINATION-BUCKET ^
    --iam-role-arn iam-role ^
    --kms-key-id my-key
```

Sample output follows.

```
{
    "ExportTaskIdentifier": "my-cluster-export",
    "SourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:my-cluster",
    "S3Bucket": "DOC-EXAMPLE-DESTINATION-BUCKET",
    "IamRoleArn": "arn:aws:iam:123456789012:role/ExportTest",
    "KmsKeyId": "my-key",
    "Status": "STARTING",
    "PercentProgress": 0,
    "TotalExtractedDataInGB": 0,
}
```

To provide a folder path in the S3 bucket for the DB cluster export, include the `--s3-prefix` option in the start-export-task command.

RDS API

To export DB cluster data to Amazon S3 using the Amazon RDS API, use the StartExportTask operation with the following required parameters:

- `ExportTaskIdentifier`
- `SourceArn` – the ARN of the DB cluster
- `S3BucketName`
- `IamRoleArn`
- `KmsKeyId`

Monitoring DB cluster export tasks

You can monitor DB cluster exports using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To monitor DB cluster exports

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Exports in Amazon S3**.

   DB cluster exports are indicated in the **Source type** column. Export status is displayed in the **Status** column.
3. To view detailed information about a specific DB cluster export, choose the export task.

AWS CLI

To monitor DB cluster export tasks using the AWS CLI, use the describe-export-tasks command.

The following example shows how to display current information about all of your DB cluster exports.

Example

```
aws rds describe-export-tasks

{
    "ExportTasks": [
        {
            "Status": "CANCELED",
            "TaskEndTime": "2022-11-01T17:36:46.961Z",
            "S3Prefix": "something",
            "S3Bucket": "DOC-EXAMPLE-BUCKET",
            "PercentProgress": 0,
            "KmsKeyId": "arn:aws:kms:us-west-2:123456789012:key/K7MDENG/
bPxRfiCYEXAMPLEKEY",
            "ExportTaskIdentifier": "anewtest",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 0,
            "SourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:parameter-groups-
test"
        },
{
            "Status": "COMPLETE",
            "TaskStartTime": "2022-10-31T20:58:06.998Z",
            "TaskEndTime": "2022-10-31T21:37:28.312Z",
            "WarningMessage": "{\"skippedTables\":[],\"skippedObjectives\":[],\"general
\":[{\"reason\":\"FAILED_TO_EXTRACT_TABLES_LIST_FOR_DATABASE\"}]}",
            "S3Prefix": "",
            "S3Bucket": "DOC-EXAMPLE-BUCKET1",
            "PercentProgress": 100,
            "KmsKeyId": "arn:aws:kms:us-west-2:123456789012:key/2Zp9Utk/
h3yCo8nvbEXAMPLEKEY",
            "ExportTaskIdentifier": "thursday-events-test",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 263,
            "SourceArn": "arn:aws:rds:us-
west-2:123456789012:cluster:example-1-2019-10-31-06-44"
        },
        {
            "Status": "FAILED",
            "TaskEndTime": "2022-10-31T02:12:36.409Z",
            "FailureCause": "The S3 bucket DOC-EXAMPLE-BUCKET2 isn't located in the
 current AWS Region. Please, review your S3 bucket name and retry the export.",
            "S3Prefix": "",
            "S3Bucket": "DOC-EXAMPLE-BUCKET2",
            "PercentProgress": 0,
            "KmsKeyId": "arn:aws:kms:us-west-2:123456789012:key/2Zp9Utk/
h3yCo8nvbEXAMPLEKEY",
            "ExportTaskIdentifier": "wednesday-afternoon-test",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 0,
            "SourceArn": "arn:aws:rds:us-
west-2:123456789012:cluster:example-1-2019-10-30-06-45"
        }
    ]
}
```

To display information about a specific export task, include the `--export-task-identifier` option with the `describe-export-tasks` command. To filter the output, include the `--Filters` option. For more options, see the describe-export-tasks command.

RDS API

To display information about DB cluster exports using the Amazon RDS API, use the DescribeExportTasks operation.

To track completion of the export workflow or to initiate another workflow, you can subscribe to Amazon Simple Notification Service topics. For more information on Amazon SNS, see Working with Amazon RDS event notification.

Canceling a DB cluster export task

You can cancel a DB cluster export task using the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> Canceling an export task doesn't remove any data that was exported to Amazon S3. For information about how to delete the data using the console, see How do I delete objects from an S3 bucket? To delete the data using the CLI, use the delete-object command.

Console

To cancel a DB cluster export task

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Exports in Amazon S3**.

   DB cluster exports are indicated in the **Source type** column. Export status is displayed in the **Status** column.
3. Choose the export task that you want to cancel.
4. Choose **Cancel**.
5. Choose **Cancel export task** on the confirmation page.

AWS CLI

To cancel an export task using the AWS CLI, use the cancel-export-task command. The command requires the `--export-task-identifier` option.

Example

```
aws rds cancel-export-task --export-task-identifier my-export
{
    "Status": "CANCELING",
    "S3Prefix": "",
    "S3Bucket": "DOC-EXAMPLE-BUCKET",
    "PercentProgress": 0,
    "KmsKeyId": "arn:aws:kms:us-west-2:123456789012:key/K7MDENG/bPxRfiCYEXAMPLEKEY",
    "ExportTaskIdentifier": "my-export",
    "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
    "TotalExtractedDataInGB": 0,
    "SourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:export-example-1"
}
```

RDS API

To cancel an export task using the Amazon RDS API, use the CancelExportTask operation with the `ExportTaskIdentifier` parameter.

Failure messages for Amazon S3 export tasks

The following table describes the messages that are returned when Amazon S3 export tasks fail.

Failure message	Description
Failed to find or access the source DB cluster: [cluster name]	The source DB cluster can't be cloned.
An unknown internal error occurred.	The task has failed because of an unknown error, exception, or failure.
An unknown internal error occurred writing the export task's metadata to the S3 bucket [bucket name].	The task has failed because of an unknown error, exception, or failure.
The RDS export failed to write the export task's metadata because it can't assume the IAM role [role ARN].	The export task assumes your IAM role to validate whether it is allowed to write metadata to your S3 bucket. If the task can't assume your IAM role, it fails.
The RDS export failed to write the export task's metadata to the S3 bucket [bucket name] using the IAM role [role ARN] with the KMS key [key ID]. Error code: [error code]	One or more permissions are missing, so the export task can't access the S3 bucket. This failure message is raised when receiving one of the following error codes: • `AWSSecurityTokenServiceException` with the error code `AccessDenied` • `AmazonS3Exception` with the error code `NoSuchBucket` , `AccessDenied` , `KMS.KMSInvalidStateException` , `403 Forbidden` , or `KMS.DisabledException` These error codes indicate that settings are misconfigured for the IAM role, S3 bucket, or KMS key.
The IAM role [role ARN] isn't authorized to call [S3 action] on the S3 bucket	The IAM policy is misconfigured. Permission for the specific S3 action on the S3 bucket is missing, which causes the export task to fail.

Failure message	Description
[bucket name]. Review your permissions and retry the export.	
KMS key check failed. Check the credentials on your KMS key and try again.	The KMS key credential check failed.
S3 credential check failed. Check the permissions on your S3 bucket and IAM policy.	The S3 credential check failed.
The S3 bucket [bucket name] isn't valid. Either it isn't located in the current AWS Region or it doesn't exist. Review your S3 bucket name and retry the export.	The S3 bucket is invalid.
The S3 bucket [bucket name] isn't located in the current AWS Region. Review your S3 bucket name and retry the export.	The S3 bucket is in the wrong AWS Region.

Troubleshooting PostgreSQL permissions errors

When exporting PostgreSQL databases to Amazon S3, you might see a `PERMISSIONS_DO_NOT_EXIST` error stating that certain tables were skipped. This error usually occurs when the superuser, which you specified when creating the DB cluster, doesn't have permissions to access those tables.

To fix this error, run the following command:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO superuser_name
```

For more information on superuser privileges, see Master user account privileges.

File naming convention

Exported data for specific tables is stored in the format *base_prefix*/*files*, where the base prefix is the following:

```
export_identifier/database_name/schema_name.table_name/
```

For example:

```
export-1234567890123-459/rdststcluster/mycluster.DataInsert_7ADB5D19965123A2/
```

Output files use the following naming convention, where *partition_index* is alphanumeric:

```
partition_index/part-00000-random_uuid.format-based_extension
```

For example:

```
1/part-00000-c5a881bb-58ff-4ee6-1111-b41ecff340a3-c000.gz.parquet
a/part-00000-d7a881cc-88cc-5ab7-2222-c41ecab340a4-c000.gz.parquet
```

The file naming convention is subject to change. Therefore, when reading target tables, we recommend that you read everything inside the base prefix for the table.

Data conversion and storage format

When you export a DB cluster to an Amazon S3 bucket, Amazon Aurora converts data to, exports data in, and stores data in the Parquet format. For more information, see Data conversion when exporting to an Amazon S3 bucket.

Exporting DB cluster snapshot data to Amazon S3

You can export DB cluster snapshot data to an Amazon S3 bucket. The export process runs in the background and doesn't affect the performance of your active DB cluster.

When you export a DB cluster snapshot, Amazon Aurora extracts data from the snapshot and stores it in an Amazon S3 bucket. You can export manual snapshots and automated system snapshots. By default, all data in the snapshot is exported. However, you can choose to export specific sets of databases, schemas, or tables.

The data is stored in an Apache Parquet format that is compressed and consistent. Individual Parquet files are usually 1–10 MB in size.

After the data is exported, you can analyze the exported data directly through tools like Amazon Athena or Amazon Redshift Spectrum. For more information on using Athena to read Parquet data, see Parquet SerDe in the *Amazon Athena User Guide*. For more information on using Redshift Spectrum to read Parquet data, see COPY from columnar data formats in the *Amazon Redshift Database Developer Guide*.

Feature availability and support varies across specific versions of each database engine and across AWS Regions. For more information on version and Region availability of exporting DB cluster snapshot data to S3, see Supported Regions and Aurora DB engines for exporting snapshot data to Amazon S3.

Topics

- Data conversion when exporting to an Amazon S3 bucket

Limitations

Exporting DB snapshot data to Amazon S3 has the following limitations:

- You can't run multiple export tasks for the same DB cluster snapshot simultaneously. This applies to both full and partial exports.
- You can have up to five concurrent DB snapshot export tasks in progress per AWS account.
- You can't export snapshot data from Aurora Serverless v1 DB clusters to S3.
- Exports to S3 don't support S3 prefixes containing a colon (:).
- The following characters in the S3 file path are converted to underscores (_) during export:

```
\ ` " (space)
```

- If a database, schema, or table has characters in its name other than the following, partial export isn't supported. However, you can export the entire DB snapshot.
  - Latin letters (A–Z)
  - Digits (0–9)
  - Dollar symbol ($)
  - Underscore (_)
- Spaces ( ) and certain characters aren't supported in database table column names. Tables with the following characters in column names are skipped during export:

```
, ; { } ( ) \n \t = (space)
```

- Tables with slashes (/) in their names are skipped during export.
- Aurora PostgreSQL temporary and unlogged tables are skipped during export.
- If the data contains a large object, such as a BLOB or CLOB, that is close to or greater than 500 MB, then the export fails.
- If a table contains a large row that is close to or greater than 2 GB, then the table is skipped during export.
- For partial exports, the `ExportOnly` list has a maximum size of 200 KB.
- We strongly recommend that you use a unique name for each export task. If you don't use a unique task name, you might receive the following error message:

ExportTaskAlreadyExistsFault: An error occurred (ExportTaskAlreadyExists) when calling the StartExportTask operation: The export task with the ID *xxxxx* already exists.

- You can delete a snapshot while you're exporting its data to S3, but you're still charged for the storage costs for that snapshot until the export task has completed.
- You can't restore exported snapshot data from S3 to a new DB cluster.

Overview of exporting snapshot data

You use the following process to export DB snapshot data to an Amazon S3 bucket. For more details, see the following sections.

1. Identify the snapshot to export.

   Use an existing automated or manual snapshot, or create a manual snapshot of a DB instance.
2. Set up access to the Amazon S3 bucket.

   A *bucket* is a container for Amazon S3 objects or files. To provide the information to access a bucket, take the following steps:

   a. Identify the S3 bucket where the snapshot is to be exported to. The S3 bucket must be in the same AWS Region as the snapshot. For more information, see Identifying the Amazon S3 bucket for export.

   b. Create an AWS Identity and Access Management (IAM) role that grants the snapshot export task access to the S3 bucket. For more information, see Providing access to an Amazon S3 bucket using an IAM role.
3. Create a symmetric encryption AWS KMS key for the server-side encryption. The KMS key is used by the snapshot export task to set up AWS KMS server-side encryption when writing the export data to S3.

   The KMS key policy must include both the `kms:CreateGrant` and `kms:DescribeKey` permissions. For more information on using KMS keys in Amazon Aurora, see AWS KMS key management.

   If you have a deny statement in your KMS key policy, make sure to explicitly exclude the AWS service principal `export.rds.amazonaws.com`.

You can use a KMS key within your AWS account, or you can use a cross-account KMS key. For more information, see Using a cross-account AWS KMS key.

4. Export the snapshot to Amazon S3 using the console or the `start-export-task` CLI command. For more information, see Exporting a snapshot to an Amazon S3 bucket.
5. To access your exported data in the Amazon S3 bucket, see Uploading, downloading, and managing objects in the *Amazon Simple Storage Service User Guide*.

Setting up access to an Amazon S3 bucket

You identify the Amazon S3 bucket, then you give the snapshot permission to access it.

Topics

- Identifying the Amazon S3 bucket for export
- Providing access to an Amazon S3 bucket using an IAM role
- Using a cross-account Amazon S3 bucket
- Using a cross-account AWS KMS key

Identifying the Amazon S3 bucket for export

Identify the Amazon S3 bucket to export the DB snapshot to. Use an existing S3 bucket or create a new S3 bucket.

> **Note**
>
> The S3 bucket to export to must be in the same AWS Region as the snapshot.

For more information about working with Amazon S3 buckets, see the following in the *Amazon Simple Storage Service User Guide*:

- How do I view the properties for an S3 bucket?
- How do I enable default encryption for an Amazon S3 bucket?
- How do I create an S3 bucket?

Providing access to an Amazon S3 bucket using an IAM role

Before you export DB snapshot data to Amazon S3, give the snapshot export tasks write-access permission to the Amazon S3 bucket.

To grant this permission, create an IAM policy that provides access to the bucket, then create an IAM role and attach the policy to the role. Later, you can assign the IAM role to your snapshot export task.

> ⚠ **Important**
>
> If you plan to use the AWS Management Console to export your snapshot, you can choose to create the IAM policy and the role automatically when you export the snapshot. For instructions, see Exporting a snapshot to an Amazon S3 bucket.

To give DB snapshot tasks access to Amazon S3

1. Create an IAM policy. This policy provides the bucket and object permissions that allow your snapshot export task to access Amazon S3.

   In the policy, include the following required actions to allow the transfer of files from Amazon Aurora to an S3 bucket:

   - `s3:PutObject*`
   - `s3:GetObject*`
   - `s3:ListBucket`
   - `s3:DeleteObject*`
   - `s3:GetBucketLocation`

   In the policy, include the following resources to identify the S3 bucket and objects in the bucket. The following list of resources shows the Amazon Resource Name (ARN) format for accessing Amazon S3.

   - `arn:aws:s3:::DOC-EXAMPLE-BUCKET`
   - `arn:aws:s3:::DOC-EXAMPLE-BUCKET/*`

For more information on creating an IAM policy for Amazon Aurora, see Creating and using an IAM policy for IAM database access. See also Tutorial: Create and attach your first customer managed policy in the *IAM User Guide*.

The following AWS CLI command creates an IAM policy named `ExportPolicy` with these options. It grants access to a bucket named *`DOC-EXAMPLE-BUCKET`*.

> **Note**
>
> After you create the policy, note the ARN of the policy. You need the ARN for a subsequent step when you attach the policy to an IAM role.

```
aws iam create-policy  --policy-name ExportPolicy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ExportPolicy",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject*",
                "s3:ListBucket",
                "s3:GetObject*",
                "s3:DeleteObject*",
                "s3:GetBucketLocation"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
            ]
        }
    ]
}'
```

2. Create an IAM role, so that Aurora can assume this IAM role on your behalf to access your Amazon S3 buckets. For more information, see Creating a role to delegate permissions to an IAM user in the *IAM User Guide*.

The following example shows using the AWS CLI command to create a role named `rds-s3-export-role`.

```
aws iam create-role  --role-name rds-s3-export-role  --assume-role-policy-document
 '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "export.rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole"
       }
     ]
   }'
```

3. Attach the IAM policy that you created to the IAM role that you created.

   The following AWS CLI command attaches the policy created earlier to the role named `rds-s3-export-role`. Replace *`your-policy-arn`* with the policy ARN that you noted in an earlier step.

```
aws iam attach-role-policy  --policy-arn your-policy-arn  --role-name rds-s3-
export-role
```

Using a cross-account Amazon S3 bucket

You can use Amazon S3 buckets across AWS accounts. To use a cross-account bucket, add a bucket policy to allow access to the IAM role that you're using for the S3 exports. For more information, see Example 2: Bucket owner granting cross-account bucket permissions.

- Attach a bucket policy to your bucket, as shown in the following example.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/Admin"
            },
            "Action": [
                "s3:PutObject*",
                "s3:ListBucket",
                "s3:GetObject*",
                "s3:DeleteObject*",
                "s3:GetBucketLocation"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-DESTINATION-BUCKET",
                "arn:aws:s3:::DOC-EXAMPLE-DESTINATION-BUCKET/*"
            ]
        }
    ]
}
```

Using a cross-account AWS KMS key

You can use a cross-account AWS KMS key to encrypt Amazon S3 exports. First, you add a key policy to the local account, then you add IAM policies in the external account. For more information, see Allowing users in other accounts to use a KMS key.

To use a cross-account KMS key

1. Add a key policy to the local account.

   The following example gives `ExampleRole` and `ExampleUser` in the external account 444455556666 permissions in the local account 123456789012.

```
{
    "Sid": "Allow an external account to use this KMS key",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            "arn:aws:iam::444455556666:role/ExampleRole",
            "arn:aws:iam::444455556666:user/ExampleUser"
        ]
    },
    "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:CreateGrant",
        "kms:DescribeKey",
        "kms:RetireGrant"
    ],
    "Resource": "*"
}
```

2. Add IAM policies to the external account.

   The following example IAM policy allows the principal to use the KMS key in account 123456789012 for cryptographic operations. To give this permission to `ExampleRole` and `ExampleUser` in account 444455556666, [attach the policy](#) to them in that account.

```
{
    "Sid": "Allow use of KMS key in account 123456789012",
    "Effect": "Allow",
    "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:CreateGrant",
        "kms:DescribeKey",
        "kms:RetireGrant"
    ],
    "Resource": "arn:aws:kms:us-
west-2:123456789012:key/1234abcd-12ab-34cd-56ef-1234567890ab"
}
```

Exporting a snapshot to an Amazon S3 bucket

You can have up to five concurrent DB snapshot export tasks in progress per AWS account.

> **Note**
>
> Exporting RDS snapshots can take a while depending on your database type and size. The export task first restores and scales the entire database before extracting the data to Amazon S3. The task's progress during this phase displays as **Starting**. When the task switches to exporting data to S3, progress displays as **In progress**.

The time it takes for the export to complete depends on the data stored in the database. For example, tables with well-distributed numeric primary key or index columns export the fastest. Tables that don't contain a column suitable for partitioning and tables with only one index on a string-based column take longer. This longer export time occurs because the export uses a slower single-threaded process.

You can export a DB snapshot to Amazon S3 using the AWS Management Console, the AWS CLI, or the RDS API.

If you use a Lambda function to export a snapshot, add the `kms:DescribeKey` action to the Lambda function policy. For more information, see AWS Lambda permissions.

Console

The **Export to Amazon S3** console option appears only for snapshots that can be exported to Amazon S3. A snapshot might not be available for export because of the following reasons:

- The DB engine isn't supported for S3 export.
- The DB instance version isn't supported for S3 export.
- S3 export isn't supported in the AWS Region where the snapshot was created.

To export a DB snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. From the tabs, choose the type of snapshot that you want to export.
4. In the list of snapshots, choose the snapshot that you want to export.
5. For **Actions**, choose **Export to Amazon S3**.

   The **Export to Amazon S3** window appears.
6. For **Export identifier**, enter a name to identify the export task. This value is also used for the name of the file created in the S3 bucket.
7. Choose the data to be exported:

   - Choose **All** to export all data in the snapshot.

- Choose **Partial** to export specific parts of the snapshot. To identify which parts of the snapshot to export, enter one or more databases, schemas, or tables for **Identifiers**, separated by spaces.

  Use the following format:

  ```
  database[.schema][.table] database2[.schema2][.table2] ... databasen[.scheman]
  [.tablen]
  ```

  For example:

  ```
  mydatabase mydatabase2.myschema1 mydatabase2.myschema2.mytable1
   mydatabase2.myschema2.mytable2
  ```

8. For **S3 bucket**, choose the bucket to export to.

   To assign the exported data to a folder path in the S3 bucket, enter the optional path for **S3 prefix**.

9. For **IAM role**, either choose a role that grants you write access to your chosen S3 bucket, or create a new role.

   - If you created a role by following the steps in Providing access to an Amazon S3 bucket using an IAM role, choose that role.
   - If you didn't create a role that grants you write access to your chosen S3 bucket, then choose **Create a new role** to create the role automatically. Next, enter a name for the role in **IAM role name**.

10. For **AWS KMS key**, enter the ARN for the key to use for encrypting the exported data.

11. Choose **Export to Amazon S3**.

AWS CLI

To export a DB snapshot to Amazon S3 using the AWS CLI, use the start-export-task command with the following required options:

- `--export-task-identifier`
- `--source-arn`
- `--s3-bucket-name`

- `--iam-role-arn`
- `--kms-key-id`

In the following examples, the snapshot export task is named *my-snapshot-export*, which exports a snapshot to an S3 bucket named *DOC-EXAMPLE-DESTINATION-BUCKET*.

Example

For Linux, macOS, or Unix:

```
aws rds start-export-task \
    --export-task-identifier my-snapshot-export \
    --source-arn arn:aws:rds:AWS_Region:123456789012:snapshot:snapshot-name \
    --s3-bucket-name DOC-EXAMPLE-DESTINATION-BUCKET \
    --iam-role-arn iam-role \
    --kms-key-id my-key
```

For Windows:

```
aws rds start-export-task ^
    --export-task-identifier my-snapshot-export ^
    --source-arn arn:aws:rds:AWS_Region:123456789012:snapshot:snapshot-name ^
    --s3-bucket-name DOC-EXAMPLE-DESTINATION-BUCKET ^
    --iam-role-arn iam-role ^
    --kms-key-id my-key
```

Sample output follows.

```
{
    "Status": "STARTING",
    "IamRoleArn": "iam-role",
    "ExportTime": "2019-08-12T01:23:53.109Z",
    "S3Bucket": "DOC-EXAMPLE-DESTINATION-BUCKET",
    "PercentProgress": 0,
    "KmsKeyId": "my-key",
    "ExportTaskIdentifier": "my-snapshot-export",
    "TotalExtractedDataInGB": 0,
    "TaskStartTime": "2019-11-13T19:46:00.173Z",
    "SourceArn": "arn:aws:rds:AWS_Region:123456789012:snapshot:snapshot-name"
}
```

To provide a folder path in the S3 bucket for the snapshot export, include the `--s3-prefix` option in the start-export-task command.

RDS API

To export a DB snapshot to Amazon S3 using the Amazon RDS API, use the StartExportTask operation with the following required parameters:

- `ExportTaskIdentifier`
- `SourceArn`
- `S3BucketName`
- `IamRoleArn`
- `KmsKeyId`

Export performance in Aurora MySQL

Aurora MySQL version 2 and version 3 DB cluster snapshots use an advanced export mechanism to improve performance and reduce export time. The mechanism includes optimizations such as multiple export threads and Aurora MySQL parallel query to take advantage of the Aurora shared storage architecture. The optimizations are applied adaptively, depending on the data set size and structure.

You don't need to turn on parallel query to use the faster export process, but the process does have the same limitations as parallel query. In addition, some data values aren't supported, such as dates where the day of the month is `0` or the year is `0000`. For more information, see Working with parallel query for Amazon Aurora MySQL.

When performance optimizations are applied, you might also see much larger (~200 GB) Parquet files for Aurora MySQL version 2 and 3 exports.

If the faster export process can't be used, for example because of incompatible data types or values, Aurora automatically switches to a single-threaded export mode without parallel query. Depending on which process is used, and the amount of data to be exported, export performance can vary.

Monitoring snapshot exports

You can monitor DB snapshot exports using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To monitor DB snapshot exports

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Exports in Amazon S3**.

   DB snapshot exports are indicated in the **Source type** column. Export status is displayed in the **Status** column.
3. To view detailed information about a specific snapshot export, choose the export task.

AWS CLI

To monitor DB snapshot exports using the AWS CLI, use the describe-export-tasks command.

The following example shows how to display current information about all of your snapshot exports.

Example

```
aws rds describe-export-tasks

{
    "ExportTasks": [
        {
            "Status": "CANCELED",
            "TaskEndTime": "2019-11-01T17:36:46.961Z",
            "S3Prefix": "something",
            "ExportTime": "2019-10-24T20:23:48.364Z",
            "S3Bucket": "DOC-EXAMPLE-BUCKET",
            "PercentProgress": 0,
            "KmsKeyId": "arn:aws:kms:AWS_Region:123456789012:key/K7MDENG/
bPxRfiCYEXAMPLEKEY",
            "ExportTaskIdentifier": "anewtest",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 0,
            "TaskStartTime": "2019-10-25T19:10:58.885Z",
            "SourceArn": "arn:aws:rds:AWS_Region:123456789012:snapshot:parameter-
groups-test"
        },
{
            "Status": "COMPLETE",
            "TaskEndTime": "2019-10-31T21:37:28.312Z",
            "WarningMessage": "{\"skippedTables\":[],\"skippedObjectives\":[],\"general
\":[{\"reason\":\"FAILED_TO_EXTRACT_TABLES_LIST_FOR_DATABASE\"}]}",
            "S3Prefix": "",
            "ExportTime": "2019-10-31T06:44:53.452Z",
            "S3Bucket": "DOC-EXAMPLE-BUCKET1",
            "PercentProgress": 100,
            "KmsKeyId": "arn:aws:kms:AWS_Region:123456789012:key/2Zp9Utk/
h3yCo8nvbEXAMPLEKEY",
            "ExportTaskIdentifier": "thursday-events-test",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 263,
            "TaskStartTime": "2019-10-31T20:58:06.998Z",
            "SourceArn":
 "arn:aws:rds:AWS_Region:123456789012:snapshot:rds:example-1-2019-10-31-06-44"
        },
        {
            "Status": "FAILED",
            "TaskEndTime": "2019-10-31T02:12:36.409Z",
            "FailureCause": "The S3 bucket my-exports isn't located in the current AWS
 Region. Please, review your S3 bucket name and retry the export.",
            "S3Prefix": "",
            "ExportTime": "2019-10-30T06:45:04.526Z",
            "S3Bucket": "DOC-EXAMPLE-BUCKET2",
            "PercentProgress": 0,
            "KmsKeyId": "arn:aws:kms:AWS_Region:123456789012:key/2Zp9Utk/
h3yCo8nvbEXAMPLEKEY",
            "ExportTaskIdentifier": "wednesday-afternoon-test",
            "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
            "TotalExtractedDataInGB": 0,
            "TaskStartTime": "2019-10-30T22:43:40.034Z",
            "SourceArn":
 "arn:aws:rds:AWS_Region:123456789012:snapshot:rds:example-1-2019-10-30-06-45"
        }
    ]
}
```

To display information about a specific snapshot export, include the `--export-task-identifier` option with the `describe-export-tasks` command. To filter the output, include the `--Filters` option. For more options, see the describe-export-tasks command.

RDS API

To display information about DB snapshot exports using the Amazon RDS API, use the DescribeExportTasks operation.

To track completion of the export workflow or to initiate another workflow, you can subscribe to Amazon Simple Notification Service topics. For more information on Amazon SNS, see Working with Amazon RDS event notification.

Canceling a snapshot export task

You can cancel a DB snapshot export task using the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> Canceling a snapshot export task doesn't remove any data that was exported to Amazon S3. For information about how to delete the data using the console, see How do I delete objects from an S3 bucket? To delete the data using the CLI, use the delete-object command.

Console

To cancel a snapshot export task

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Exports in Amazon S3**.

   DB snapshot exports are indicated in the **Source type** column. Export status is displayed in the **Status** column.
3. Choose the snapshot export task that you want to cancel.
4. Choose **Cancel**.
5. Choose **Cancel export task** on the confirmation page.

AWS CLI

To cancel a snapshot export task using the AWS CLI, use the cancel-export-task command. The command requires the `--export-task-identifier` option.

Example

```
aws rds cancel-export-task --export-task-identifier my_export
{
    "Status": "CANCELING",
    "S3Prefix": "",
    "ExportTime": "2019-08-12T01:23:53.109Z",
    "S3Bucket": "DOC-EXAMPLE-BUCKET",
    "PercentProgress": 0,
    "KmsKeyId": "arn:aws:kms:AWS_Region:123456789012:key/K7MDENG/bPxRfiCYEXAMPLEKEY",
    "ExportTaskIdentifier": "my_export",
    "IamRoleArn": "arn:aws:iam::123456789012:role/export-to-s3",
    "TotalExtractedDataInGB": 0,
    "TaskStartTime": "2019-11-13T19:46:00.173Z",
    "SourceArn": "arn:aws:rds:AWS_Region:123456789012:snapshot:export-example-1"
}
```

RDS API

To cancel a snapshot export task using the Amazon RDS API, use the CancelExportTask operation with the `ExportTaskIdentifier` parameter.

Failure messages for Amazon S3 export tasks

The following table describes the messages that are returned when Amazon S3 export tasks fail.

Failure message	Description
An unknown internal error occurred.	The task has failed because of an unknown error, exception, or failure.
An unknown internal error occurred writing the export task's metadata to the S3 bucket [bucket name].	The task has failed because of an unknown error, exception, or failure.

Failure message	Description
The RDS export failed to write the export task's metadata because it can't assume the IAM role [role ARN].	The export task assumes your IAM role to validate whether it is allowed to write metadata to your S3 bucket. If the task can't assume your IAM role, it fails.
The RDS export failed to write the export task's metadata to the S3 bucket [bucket name] using the IAM role [role ARN] with the KMS key [key ID]. Error code: [error code]	One or more permissions are missing, so the export task can't access the S3 bucket. This failure message is raised when receiving one of the following error codes: • `AWSSecurityTokenServiceException` with the error code `AccessDenied` • `AmazonS3Exception` with the error code `NoSuchBucket`, `AccessDenied`, `KMS.KMSInvalidStateException`, `403 Forbidden`, or `KMS.DisabledException` These error codes indicate that settings are misconfigured for the IAM role, S3 bucket, or KMS key.
The IAM role [role ARN] isn't authorized to call [S3 action] on the S3 bucket [bucket name]. Review your permissions and retry the export.	The IAM policy is misconfigured. Permission for the specific S3 action on the S3 bucket is missing, which causes the export task to fail.
KMS key check failed. Check the credentials on your KMS key and try again.	The KMS key credential check failed.
S3 credential check failed. Check the permissions on your S3 bucket and IAM policy.	The S3 credential check failed.

Failure message	Description
The S3 bucket [bucket name] isn't valid. Either it isn't located in the current AWS Region or it doesn't exist. Review your S3 bucket name and retry the export.	The S3 bucket is invalid.
The S3 bucket [bucket name] isn't located in the current AWS Region. Review your S3 bucket name and retry the export.	The S3 bucket is in the wrong AWS Region.

Troubleshooting PostgreSQL permissions errors

When exporting PostgreSQL databases to Amazon S3, you might see a `PERMISSIONS_DO_NOT_EXIST` error stating that certain tables were skipped. This error usually occurs when the superuser, which you specified when creating the DB instance, doesn't have permissions to access those tables.

To fix this error, run the following command:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO superuser_name
```

For more information on superuser privileges, see Master user account privileges.

File naming convention

Exported data for specific tables is stored in the format *base_prefix/files*, where the base prefix is the following:

```
export_identifier/database_name/schema_name.table_name/
```

For example:

```
export-1234567890123-459/rdststdb/rdststdb.DataInsert_7ADB5D19965123A2/
```

There are two conventions for how files are named.

- Current convention:

  ```
  batch_index/part-partition_index-random_uuid.format-based_extension
  ```

  The batch index is a sequence number that represents a batch of data read from the table. If we can't partition your table into small chunks to be exported in parallel, there will be multiple batch indexes. The same thing happens if your table is partitioned into multiple tables. There will be multiple batch indexes, one for each of the table partitions of your main table.

  If we can partition your table into small chunks to be read in parallel, there will be only the batch index 1 folder.

  Inside the batch index folder, there are one or more Parquet files that contain your table's data. The prefix of the Parquet filename is `part-`*`partition_index`*. If your table is partitioned, there will be multiple files starting with the partition index `00000`.

  There can be gaps in the partition index sequence. This happens because each partition is obtained from a ranged query in your table. If there is no data in the range of that partition, then that sequence number is skipped.

  For example, suppose that the `id` column is the table's primary key, and its minimum and maximum values are `100` and `1000`. When we try to export this table with nine partitions, we read it with parallel queries such as the following:

  ```
  SELECT * FROM table WHERE id <= 100 AND id < 200
  SELECT * FROM table WHERE id <= 200 AND id < 300
  ```

  This should generate nine files, from `part-00000-`*`random_uuid`*`.gz.parquet` to `part-00008-`*`random_uuid`*`.gz.parquet`. However, if there are no rows with IDs between `200` and `350`, one of the completed partitions is empty, and no file is created for it. In the previous example, `part-00001-`*`random_uuid`*`.gz.parquet` isn't created.

- Older convention:

  ```
  part-partition_index-random_uuid.format-based_extension
  ```

  This is the same as the current convention, but without the *`batch_index`* prefix, for example:

  ```
  part-00000-c5a881bb-58ff-4ee6-1111-b41ecff340a3-c000.gz.parquet
  ```

```
part-00001-d7a881cc-88cc-5ab7-2222-c41ecab340a4-c000.gz.parquet
part-00002-f5a991ab-59aa-7fa6-3333-d41eccd340a7-c000.gz.parquet
```

The file naming convention is subject to change. Therefore, when reading target tables, we recommend that you read everything inside the base prefix for the table.

Data conversion when exporting to an Amazon S3 bucket

When you export a DB snapshot to an Amazon S3 bucket, Amazon Aurora converts data to, exports data in, and stores data in the Parquet format. For more information about Parquet, see the Apache Parquet website.

Parquet stores all data as one of the following primitive types:

- BOOLEAN
- INT32
- INT64
- INT96
- FLOAT
- DOUBLE
- BYTE_ARRAY – A variable-length byte array, also known as binary
- FIXED_LEN_BYTE_ARRAY – A fixed-length byte array used when the values have a constant size

The Parquet data types are few to reduce the complexity of reading and writing the format. Parquet provides logical types for extending primitive types. A *logical type* is implemented as an annotation with the data in a `LogicalType` metadata field. The logical type annotation explains how to interpret the primitive type.

When the `STRING` logical type annotates a `BYTE_ARRAY` type, it indicates that the byte array should be interpreted as a UTF-8 encoded character string. After an export task completes, Amazon Aurora notifies you if any string conversion occurred. The underlying data exported is always the same as the data from the source. However, due to the encoding difference in UTF-8, some characters might appear different from the source when read in tools such as Athena.

For more information, see Parquet logical type definitions in the Parquet documentation.

Topics

- MySQL data type mapping to Parquet
- PostgreSQL data type mapping to Parquet

MySQL data type mapping to Parquet

The following table shows the mapping from MySQL data types to Parquet data types when data is converted and exported to Amazon S3.

Source data type	Parquet primitive type	Logical type annotation	Conversion notes
Numeric data types			
BIGINT	INT64		
BIGINT UNSIGNED	FIXED_LEN _BYTE_ARRAY(9)	DECIMAL(20,0)	Parquet supports only signed types, so the mapping requires an additional byte (8 plus 1) to store the BIGINT_UNSIGNED type.
BIT	BYTE_ARRAY		
DECIMAL	INT32	DECIMAL(p,s)	If the source value is less than 2^{31}, it's stored as INT32.
	INT64	DECIMAL(p,s)	If the source value is 2^{31} or greater, but less than 2^{63}, it's stored as INT64.
	FIXED_LEN _BYTE_ARRAY(N)	DECIMAL(p,s)	If the source value is 2^{63} or greater, it's stored as FIXED_LEN _BYTE_ARRAY(N).

Source data type	Parquet primitive type	Logical type annotation	Conversion notes
	BYTE_ARRAY	STRING	Parquet doesn't support Decimal precision greater than 38. The Decimal value is converted to a string in a BYTE_ARRAY type and encoded as UTF8.
DOUBLE	DOUBLE		
FLOAT	DOUBLE		
INT	INT32		
INT UNSIGNED	INT64		
MEDIUMINT	INT32		
MEDIUMINT UNSIGNED	INT64		
NUMERIC	INT32	DECIMAL(p,s)	If the source value is less than 2^{31}, it's stored as INT32.
	INT64	DECIMAL(p,s)	If the source value is 2^{31} or greater, b ut less than 2^{63}, it's stored as INT64.

Source data type	Parquet primitive type	Logical type annotation	Conversion notes
	FIXED_LEN_ARRAY(N)	DECIMAL(p,s)	If the source value is 2^{63} or greater, it's stored as FIXED_LEN_BYTE_ARRAY(N).
	BYTE_ARRAY	STRING	Parquet doesn't support Numeric precision greater than 38. This Numeric value is converted to a string in a BYTE_ARRAY type and encoded as UTF8.
SMALLINT	INT32		
SMALLINT UNSIGNED	INT32		
TINYINT	INT32		
TINYINT UNSIGNED	INT32		
String data types			
BINARY	BYTE_ARRAY		
BLOB	BYTE_ARRAY		
CHAR	BYTE_ARRAY		
ENUM	BYTE_ARRAY	STRING	
LINESTRING	BYTE_ARRAY		
LONGBLOB	BYTE_ARRAY		

Source data type	Parquet primitive type	Logical type annotation	Conversion notes
LONGTEXT	BYTE_ARRAY	STRING	
MEDIUMBLOB	BYTE_ARRAY		
MEDIUMTEXT	BYTE_ARRAY	STRING	
MULTILINESTRING	BYTE_ARRAY		
SET	BYTE_ARRAY	STRING	
TEXT	BYTE_ARRAY	STRING	
TINYBLOB	BYTE_ARRAY		
TINYTEXT	BYTE_ARRAY	STRING	
VARBINARY	BYTE_ARRAY		
VARCHAR	BYTE_ARRAY	STRING	
Date and time data types			
DATE	BYTE_ARRAY	STRING	A date is converted to a string in a BYTE_ARRAY type and encoded as UTF8.
DATETIME	INT64	TIMESTAMP_MICROS	
TIME	BYTE_ARRAY	STRING	A TIME type is converted to a string in a BYTE_ARRAY and encoded as UTF8.
TIMESTAMP	INT64	TIMESTAMP_MICROS	
YEAR	INT32		

Source data type	Parquet primitive type	Logical type annotation	Conversion notes
Geometric data types			
GEOMETRY	BYTE_ARRAY		
GEOMETRYCOLLECTION	BYTE_ARRAY		
MULTIPOINT	BYTE_ARRAY		
MULTIPOLYGON	BYTE_ARRAY		
POINT	BYTE_ARRAY		
POLYGON	BYTE_ARRAY		
JSON data type			
JSON	BYTE_ARRAY	STRING	

PostgreSQL data type mapping to Parquet

The following table shows the mapping from PostgreSQL data types to Parquet data types when data is converted and exported to Amazon S3.

PostgreSQL data type	Parquet primitive type	Logical type annotation	Mapping notes
Numeric data types			
BIGINT	INT64		
BIGSERIAL	INT64		
DECIMAL	BYTE_ARRAY	STRING	A DECIMAL type is converted to a string in a BYTE_ARRAY

PostgreSQL data type	Parquet primitive type	Logical type annotation	Mapping notes
			type and encoded as UTF8. This conversion is to avoid complications due to data precision and data values that are not a number (NaN).
DOUBLE PRECISION	DOUBLE		
INTEGER	INT32		
MONEY	BYTE_ARRAY	STRING	
REAL	FLOAT		
SERIAL	INT32		
SMALLINT	INT32	INT_16	
SMALLSERIAL	INT32	INT_16	
String and related data types			

PostgreSQL data type	Parquet primitive type	Logical type annotation	Mapping notes
ARRAY	BYTE_ARRAY	STRING	An array is converted to a string and encoded as BINARY (UTF8). This conversion is to avoid complications due to data precision, data values that are not a number (NaN), and time data values.
BIT	BYTE_ARRAY	STRING	
BIT VARYING	BYTE_ARRAY	STRING	
BYTEA	BINARY		
CHAR	BYTE_ARRAY	STRING	
CHAR(N)	BYTE_ARRAY	STRING	
ENUM	BYTE_ARRAY	STRING	
NAME	BYTE_ARRAY	STRING	
TEXT	BYTE_ARRAY	STRING	
TEXT SEARCH	BYTE_ARRAY	STRING	
VARCHAR(N)	BYTE_ARRAY	STRING	
XML	BYTE_ARRAY	STRING	
Date and time data types			

PostgreSQL data type	Parquet primitive type	Logical type annotation	Mapping notes
DATE	BYTE_ARRAY	STRING	
INTERVAL	BYTE_ARRAY	STRING	
TIME	BYTE_ARRAY	STRING	
TIME WITH TIME ZONE	BYTE_ARRAY	STRING	
TIMESTAMP	BYTE_ARRAY	STRING	
TIMESTAMP WITH TIME ZONE	BYTE_ARRAY	STRING	
Geometric data types			
BOX	BYTE_ARRAY	STRING	
CIRCLE	BYTE_ARRAY	STRING	
LINE	BYTE_ARRAY	STRING	
LINESEGMENT	BYTE_ARRAY	STRING	
PATH	BYTE_ARRAY	STRING	
POINT	BYTE_ARRAY	STRING	
POLYGON	BYTE_ARRAY	STRING	
JSON data types			
JSON	BYTE_ARRAY	STRING	
JSONB	BYTE_ARRAY	STRING	
Other data types			
BOOLEAN	BOOLEAN		

PostgreSQL data type	Parquet primitive type	Logical type annotation	Mapping notes
CIDR	BYTE_ARRAY	STRING	Network data type
COMPOSITE	BYTE_ARRAY	STRING	
DOMAIN	BYTE_ARRAY	STRING	
INET	BYTE_ARRAY	STRING	Network data type
MACADDR	BYTE_ARRAY	STRING	
OBJECT IDENTIFIER	N/A		
PG_LSN	BYTE_ARRAY	STRING	
RANGE	BYTE_ARRAY	STRING	
UUID	BYTE_ARRAY	STRING	

Restoring a DB cluster to a specified time

You can restore a DB cluster to a specific point in time, creating a new DB cluster.

When you restore a DB cluster to a point in time, you can choose the default virtual private cloud (VPC) security group. Or you can apply a custom VPC security group to your DB cluster.

Restored DB clusters are automatically associated with the default DB cluster and DB parameter groups. However, you can apply custom parameter groups by specifying them during a restore.

Amazon Aurora uploads log records for DB clusters to Amazon S3 continuously. To see the latest restorable time for a DB cluster, use the AWS CLI describe-db-clusters command and look at the value returned in the `LatestRestorableTime` field for the DB cluster.

You can restore to any point in time within your backup retention period. To see the earliest restorable time for a DB cluster, use the AWS CLI describe-db-clusters command and look at the value returned in the `EarliestRestorableTime` field for the DB cluster.

The backup retention period of the restored DB cluster is the same as that of the source DB cluster.

> **Note**
>
> Information in this topic applies to Amazon Aurora. For information on restoring an Amazon RDS DB instance, see Restoring a DB instance to a specified time.
> For more information about backing up and restoring an Aurora DB cluster, see Overview of backing up and restoring an Aurora DB cluster.
> For Aurora MySQL, you can restore a provisioned DB cluster to an Aurora Serverless DB cluster. For more information, see Restoring an Aurora Serverless v1 DB cluster.
> You can also use AWS Backup to manage backups of Amazon Aurora DB clusters. If your DB cluster is associated with a backup plan in AWS Backup, that backup plan is used for point-in-time recovery. For information, see Restoring a DB cluster to a specified time using AWS Backup.

For information about restoring an Aurora DB cluster or a global cluster with an RDS Extended Support version, see Restoring an Aurora DB cluster or a global cluster with Amazon RDS Extended Support.

You can restore a DB cluster to a point in time using the AWS Management Console, the AWS CLI, or the RDS API.

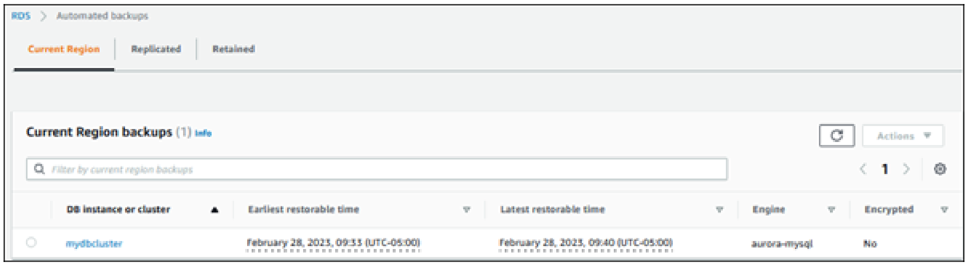
Console

To restore a DB cluster to a specified time

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Automated backups**.

   The automated backups are displayed on the **Current Region** tab.



Current Region backups (1) Info

Filter by current region backups

	DB instance or cluster	Earliest restorable time	Latest restorable time	Engine	Encrypted
○	mydbcluster	February 28, 2023, 09:33 (UTC-05:00)	February 28, 2023, 09:40 (UTC-05:00)	aurora-mysql	No

3. Choose the DB cluster that you want to restore.

4. For **Actions**, choose **Restore to point in time**.

   The **Restore to point in time** window appears.

5. Choose **Latest restorable time** to restore to the latest possible time, or choose **Custom** to choose a time.

   If you chose **Custom**, enter the date and time to which you want to restore the cluster.

> **Note**
>
> Times are shown in your local time zone, which is indicated by an offset from Coordinated Universal Time (UTC). For example, UTC-5 is Eastern Standard Time/Central Daylight Time.

6. For **DB cluster identifier**, enter the name of the target restored DB cluster. The name must be unique.

7. Choose other options as needed, such as the DB instance class and DB cluster storage configuration.

   For information about each setting, see [Settings for Aurora DB clusters](#).

8. Choose **Restore to point in time**.

AWS CLI

To restore a DB cluster to a specified time, use the AWS CLI command restore-db-cluster-to-point-in-time to create a new DB cluster.

You can specify other settings. For information about each setting, see Settings for Aurora DB clusters.

Resource tagging is supported for this operation. When you use the `--tags` option, the source DB cluster tags are ignored and the provided ones are used. Otherwise, the latest tags from the source cluster are used.

Example

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-to-point-in-time \
    --source-db-cluster-identifier mysourcedbcluster \
    --db-cluster-identifier mytargetdbcluster \
    --restore-to-time 2017-10-14T23:45:00.000Z
```

For Windows:

```
aws rds restore-db-cluster-to-point-in-time ^
    --source-db-cluster-identifier mysourcedbcluster ^
    --db-cluster-identifier mytargetdbcluster ^
    --restore-to-time 2017-10-14T23:45:00.000Z
```

⚠ **Important**

If you use the console to restore a DB cluster to a specified time, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the AWS CLI to restore a DB cluster to a specified time, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.

To create the primary instance for your DB cluster, call the create-db-instance AWS CLI command. Include the name of the DB cluster as the `--db-cluster-identifier` option value.

RDS API

To restore a DB cluster to a specified time, call the Amazon RDS API `RestoreDBClusterToPointInTime` operation with the following parameters:

- `SourceDBClusterIdentifier`
- `DBClusterIdentifier`
- `RestoreToTime`

> ⚠ **Important**
>
> If you use the console to restore a DB cluster to a specified time, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the RDS API to restore a DB cluster to a specified time, make sure to explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.
> To create the primary instance for your DB cluster, call the RDS API operation CreateDBInstance. Include the name of the DB cluster as the `DBClusterIdentifier` parameter value.

Restoring a DB cluster to a specified time from a retained automated backup

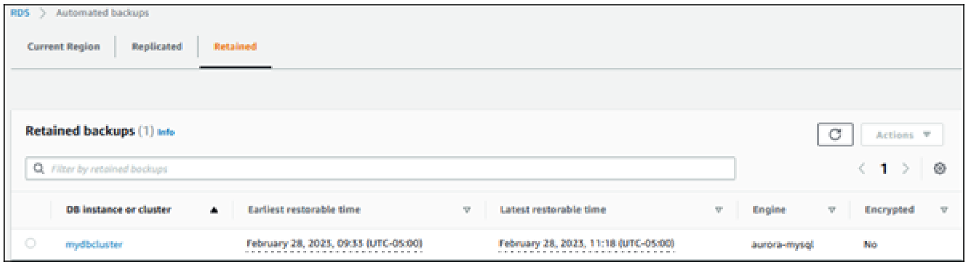
You can restore a DB cluster from a retained automated backup after you delete the source DB cluster, if the backup is within the retention period of the source cluster. The process is similar to restoring a DB cluster from an automated backup.

> ⓘ **Note**
>
> You can't restore an Aurora Serverless v1 DB cluster using this procedure, because automated backups for Aurora Serverless v1 clusters aren't retained.

Console

To restore a DB cluster to a specified time

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Automated backups**.
3. Choose the **Retained** tab.



Retained backups (1) Info

	DB instance or cluster	Earliest restorable time	Latest restorable time	Engine	Encrypted
○	mydbcluster	February 28, 2023, 09:33 (UTC-05:00)	February 28, 2023, 11:18 (UTC-05:00)	aurora-mysql	No

4. Choose the DB cluster that you want to restore.
5. For **Actions**, choose **Restore to point in time**.

   The **Restore to point in time** window appears.
6. Choose **Latest restorable time** to restore to the latest possible time, or choose **Custom** to choose a time.

   If you chose **Custom**, enter the date and time to which you want to restore the cluster.

   > **Note**
   >
   > Times are shown in your local time zone, which is indicated by an offset from Coordinated Universal Time (UTC). For example, UTC-5 is Eastern Standard Time/Central Daylight Time.
7. For **DB cluster identifier**, enter the name of the target restored DB cluster. The name must be unique.
8. Choose other options as needed, such as DB instance class.

   For information about each setting, see Settings for Aurora DB clusters.
9. Choose **Restore to point in time**.

AWS CLI

To restore a DB cluster to a specified time, use the AWS CLI command restore-db-cluster-to-point-in-time to create a new DB cluster.

You can specify other settings. For information about each setting, see Settings for Aurora DB clusters.

Resource tagging is supported for this operation. When you use the `--tags` option, the source DB cluster tags are ignored and the provided ones are used. Otherwise, the latest tags from the source cluster are used.

Example

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-to-point-in-time \
    --source-db-cluster-resource-id cluster-123ABCEXAMPLE \
    --db-cluster-identifier mytargetdbcluster \
    --restore-to-time 2017-10-14T23:45:00.000Z
```

For Windows:

```
aws rds restore-db-cluster-to-point-in-time ^
    --source-db-cluster-resource-id cluster-123ABCEXAMPLE ^
    --db-cluster-identifier mytargetdbcluster ^
    --restore-to-time 2017-10-14T23:45:00.000Z
```

> ⚠ **Important**
>
> If you use the console to restore a DB cluster to a specified time, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the AWS CLI to restore a DB cluster to a specified time, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.
>
> To create the primary instance for your DB cluster, call the create-db-instance AWS CLI command. Include the name of the DB cluster as the `--db-cluster-identifier` option value.

RDS API

To restore a DB cluster to a specified time, call the Amazon RDS API `RestoreDBClusterToPointInTime` operation with the following parameters:

- `SourceDbClusterResourceId`
- `DBClusterIdentifier`
- `RestoreToTime`

> ⚠ **Important**
>
> If you use the console to restore a DB cluster to a specified time, then Amazon RDS automatically creates the primary instance (writer) for your DB cluster. If you use the RDS API to restore a DB cluster to a specified time, make sure to explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.
>
> To create the primary instance for your DB cluster, call the RDS API operation CreateDBInstance. Include the name of the DB cluster as the `DBClusterIdentifier` parameter value.

Restoring a DB cluster to a specified time using AWS Backup

You can use AWS Backup to manage your automated backups, and then to restore them to a specified time. To do this, you create a backup plan in AWS Backup and assign your DB cluster as a resource. Then you enable continuous backups for PITR in the backup rule. For more information on backup plans and backup rules, see the *AWS Backup Developer Guide*.

Enabling continuous backups in AWS Backup

You enable continuous backups in backup rules.

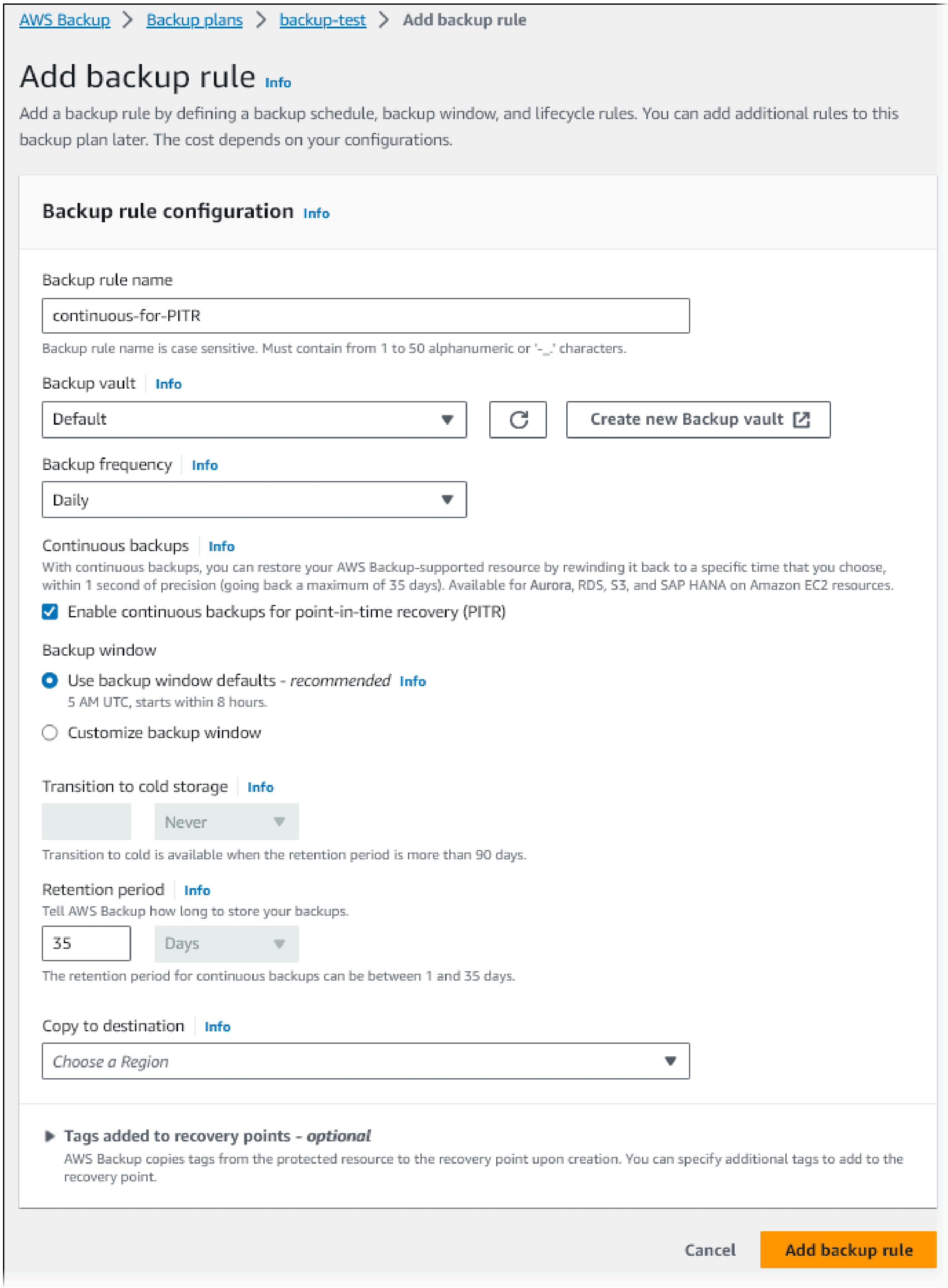
To enable continuous backups for PITR

1. Sign in to the AWS Management Console, and open the AWS Backup console at https://console.aws.amazon.com/backup.
2. In the navigation pane, choose **Backup plans**.
3. Under **Backup plan name**, select the backup plan that you use to back up your DB cluster.

4. Under the **Backup rules** section, choose **Add backup rule**.

   The **Add backup rule** page displays.

5. Select the **Enable continuous backups for point-in-time recovery (PITR)** check box.



AWS Backup > Backup plans > backup-test > Add backup rule

Add backup rule Info

Add a backup rule by defining a backup schedule, backup window, and lifecycle rules. You can add additional rules to this backup plan later. The cost depends on your configurations.

Backup rule configuration Info

Backup rule name

continuous-for-PITR

Backup rule name is case sensitive. Must contain from 1 to 50 alphanumeric or '-_.' characters.

Backup vault | Info

Default

Create new Backup vault

Backup frequency | Info

Daily

Continuous backups | Info

With continuous backups, you can restore your AWS Backup-supported resource by rewinding it back to a specific time that you choose, within 1 second of precision (going back a maximum of 35 days). Available for Aurora, RDS, S3, and SAP HANA on Amazon EC2 resources.

☑ Enable continuous backups for point-in-time recovery (PITR)

Backup window

● Use backup window defaults - *recommended* Info

5 AM UTC, starts within 8 hours.

○ Customize backup window

Transition to cold storage | Info

Never

Transition to cold is available when the retention period is more than 90 days.

Retention period | Info

Tell AWS Backup how long to store your backups.

35 Days

The retention period for continuous backups can be between 1 and 35 days.

Copy to destination | Info

Choose a Region

▶ Tags added to recovery points - *optional*

AWS Backup copies tags from the protected resource to the recovery point upon creation. You can specify additional tags to add to the recovery point.

Cancel | Add backup rule

6. Choose other settings as needed, then choose **Add backup rule**.

Restoring from a continuous backup in AWS Backup

You restore to a specified time from a backup vault.

Console

You can use the AWS Management Console to restore a DB cluster to a specified time.

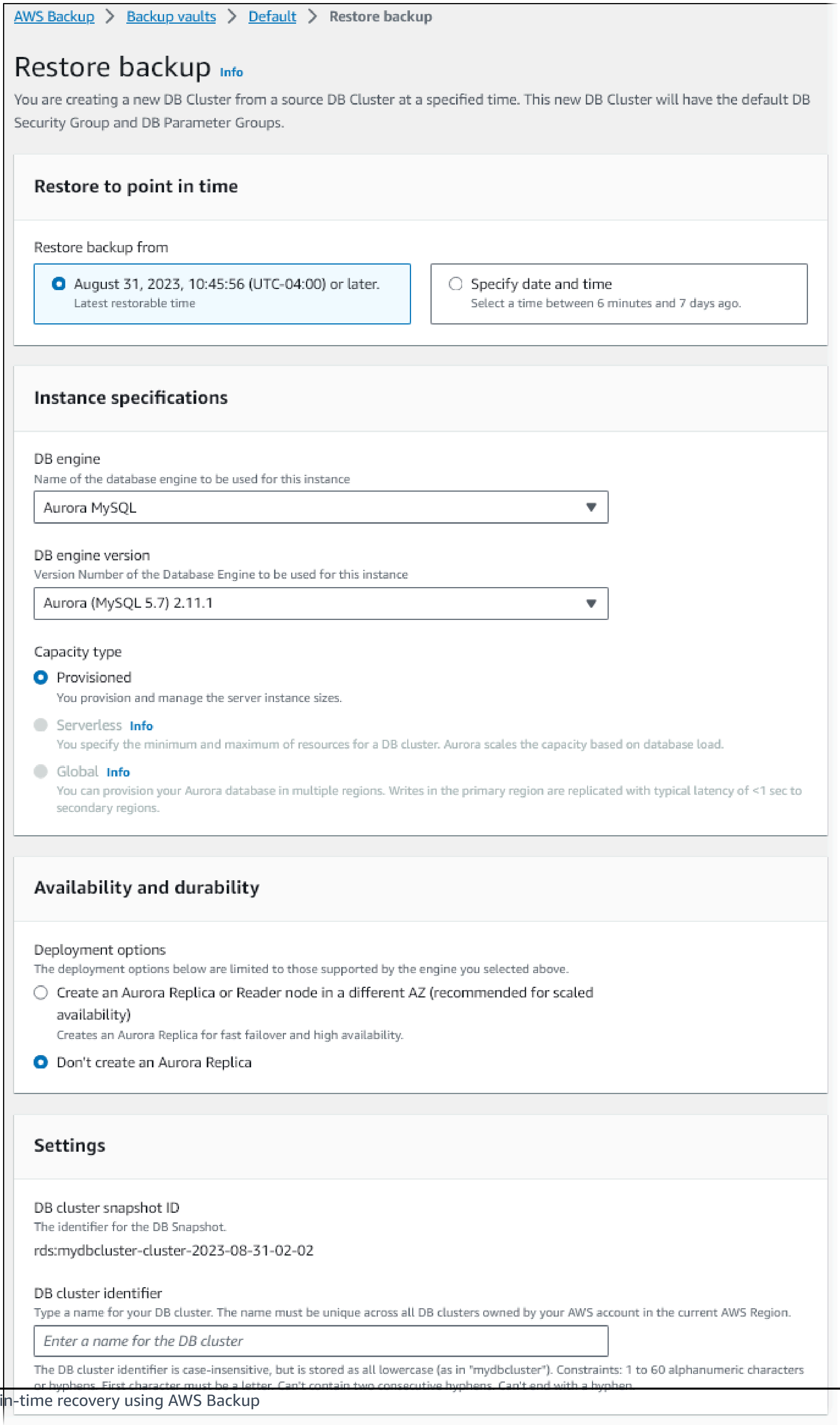
To restore from a continuous backup in AWS Backup

1. Sign in to the AWS Management Console, and open the AWS Backup console at https://console.aws.amazon.com/backup.
2. In the navigation pane, choose **Backup vaults**.
3. Choose the backup vault that contains your continuous backup, for example **Default**.

   The backup vault detail page displays.
4. Under **Recovery points**, select the recovery point for the automated backup.

   It has a backup type of **Continuous** and a name with `continuous:cluster-`*`AWS-Backup-job-number`*.
5. For **Actions**, choose **Restore**.

   The **Restore backup** page displays.



Restore backup

You are creating a new DB Cluster from a source DB Cluster at a specified time. This new DB Cluster will have the default DB Security Group and DB Parameter Groups.

Restore to point in time

Restore backup from

● August 31, 2023, 10:45:56 (UTC-04:00) or later.
Latest restorable time

○ Specify date and time
Select a time between 6 minutes and 7 days ago.

Instance specifications

DB engine
Name of the database engine to be used for this instance

Aurora MySQL ▼

DB engine version
Version Number of the Database Engine to be used for this instance

Aurora (MySQL 5.7) 2.11.1 ▼

Capacity type

● Provisioned
You provision and manage the server instance sizes.

○ Serverless
You specify the minimum and maximum of resources for a DB cluster. Aurora scales the capacity based on database load.

○ Global
You can provision your Aurora database in multiple regions. Writes in the primary region are replicated with typical latency of <1 sec to secondary regions.

Availability and durability

Deployment options
The deployment options below are limited to those supported by the engine you selected above.

○ Create an Aurora Replica or Reader node in a different AZ (recommended for scaled availability)
Creates an Aurora Replica for fast failover and high availability.

● Don't create an Aurora Replica

Settings

DB cluster snapshot ID
The identifier for the DB Snapshot.

rds:mydbcluster-cluster-2023-08-31-02-02

DB cluster identifier
Type a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

Enter a name for the DB cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

6. For **Restore to point in time**, select **Specify date and time** to restore to a specific point in time.
7. Choose other settings as needed for restoring the DB cluster, then choose **Restore backup**.

   The **Jobs** page displays, showing the **Restore jobs** pane. A message at the top of the page provides information about the restore job.

After the DB cluster is restored, you must add the primary (writer) DB instance to it. To create the primary instance for your DB cluster, call the create-db-instance AWS CLI command. Include the name of the DB cluster as the `--db-cluster-identifier` parameter value.

CLI

You use the start-restore-job AWS CLI command to restore the DB cluster to a specified time. The following parameters are required:

- `--recovery-point-arn` – The Amazon Resource Name (ARN) for the recovery point from which to restore.
- `--resource-type` – Use `Aurora`.
- `--iam-role-arn` – The ARN for the IAM role that you use for AWS Backup operations.
- `--metadata` – The metadata that you use to restore the DB cluster. The following parameters are required:
  - `DBClusterIdentifier`
  - `Engine`
  - `RestoreToTime` or `UseLatestRestorableTime`

The following example shows how to restore a DB cluster to a specified time.

```
aws backup start-restore-job \
--recovery-point-arn arn:aws:backup:eu-central-1:123456789012:recovery-
point:continuous:cluster-itsreallyjustanexample1234567890-487278c2 \
--resource-type Aurora \
--iam-role-arn arn:aws:iam::123456789012:role/service-role/AWSBackupDefaultServiceRole
 \
--metadata '{"DBClusterIdentifier":"backup-pitr-test","Engine":"aurora-
mysql","RestoreToTime":"2023-09-01T17:00:00.000Z"}'
```

The following example shows how to restore a DB cluster to the latest restorable time.

```
aws backup start-restore-job \
--recovery-point-arn arn:aws:backup:eu-central-1:123456789012:recovery-
point:continuous:cluster-itsreallyjustanexample1234567890-487278c2 \
--resource-type Aurora \
--iam-role-arn arn:aws:iam::123456789012:role/service-role/AWSBackupDefaultServiceRole
 \
--metadata '{"DBClusterIdentifier":"backup-pitr-latest","Engine":"aurora-
mysql","UseLatestRestorableTime":"true"}'
```

After the DB cluster is restored, you must add the primary (writer) DB instance to it. To create the primary instance for your DB cluster, call the create-db-instance AWS CLI command. Include the name of the DB cluster as the `--db-cluster-identifier` parameter value.

Deleting a DB cluster snapshot

You can delete DB cluster snapshots managed by Amazon RDS when you no longer need them.

> **Note**
>
> To delete backups managed by AWS Backup, use the AWS Backup console. For information about AWS Backup, see the *AWS Backup Developer Guide*.

Deleting a DB cluster snapshot

You can delete a DB cluster snapshot using the console, the AWS CLI, or the RDS API.

To delete a shared or public snapshot, you must sign in to the AWS account that owns the snapshot.

Console

To delete a DB cluster snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Choose the DB cluster snapshot that you want to delete.
4. For **Actions**, choose **Delete snapshot**.
5. Choose **Delete** on the confirmation page.

AWS CLI

You can delete a DB cluster snapshot by using the AWS CLI command delete-db-cluster-snapshot.

The following options are used to delete a DB cluster snapshot.

- `--db-cluster-snapshot-identifier` – The identifier for the DB cluster snapshot.

Example

The following code deletes the `mydbclustersnapshot` DB cluster snapshot.

For Linux, macOS, or Unix:

```
aws rds delete-db-cluster-snapshot \
    --db-cluster-snapshot-identifier mydbclustersnapshot
```

For Windows:

```
aws rds delete-db-cluster-snapshot ^
    --db-cluster-snapshot-identifier mydbclustersnapshot
```

RDS API

You can delete a DB cluster snapshot by using the Amazon RDS API operation DeleteDBClusterSnapshot.

The following parameters are used to delete a DB cluster snapshot.

- `DBClusterSnapshotIdentifier` – The identifier for the DB cluster snapshot.

Tutorial: Restore an Amazon Aurora DB cluster from a DB cluster snapshot

A common scenario when working with Amazon Aurora is to have a DB instance that you work with occasionally but that you don't need full time. For example, you might use a DB cluster to hold the data for a report that you run only quarterly. One way to save money on such a scenario is to take a DB cluster snapshot of the DB cluster after the report is completed. Then you delete the DB cluster, and restore it when you need to upload new data and run the report during the next quarter.

When you restore a DB cluster, you provide the name of the DB cluster snapshot to restore from. You then provide a name for the new DB cluster that's created from the restore operation. For more detailed information on restoring DB clusters from snapshots, see Restoring from a DB cluster snapshot.

In this tutorial, we also upgrade the restored DB cluster from Aurora MySQL version 2 (compatible with MySQL 5.7) to Aurora MySQL version 3 (compatible with MySQL 8.0).

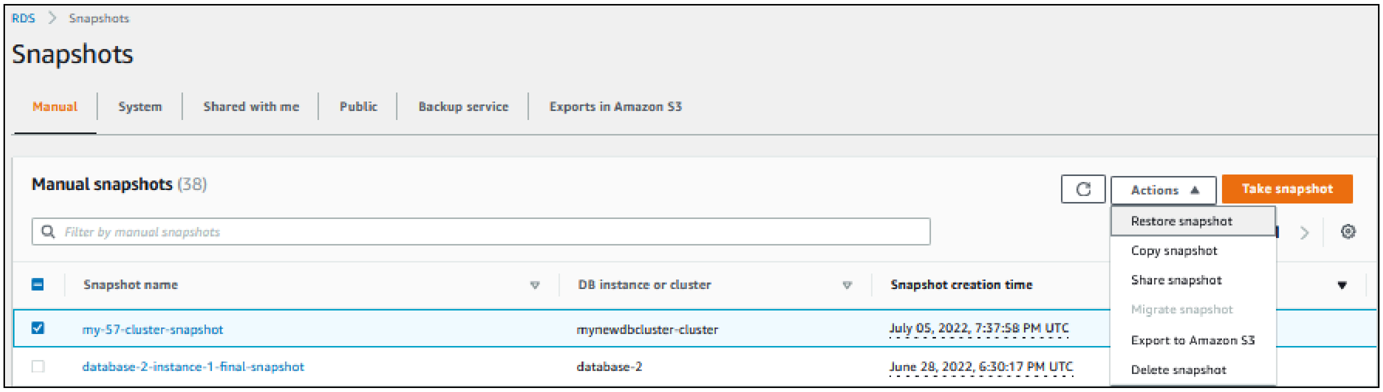
Restoring a DB cluster from a DB cluster snapshot using the Amazon RDS console

When you restore a DB cluster from a snapshot using the AWS Management Console, the primary (writer) DB instance is also created.

> **Note**
>
> While the primary DB instance is being created, it appears as a reader instance, but after creation it's a writer instance.

To restore a DB cluster from a DB cluster snapshot

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Snapshots**.
3. Choose the DB cluster snapshot that you want to restore from.
4. For **Actions**, choose **Restore snapshot**.



RDS > Snapshots

Snapshots

Manual | System | Shared with me | Public | Backup service | Exports in Amazon S3

Manual snapshots (38)

Filter by manual snapshots

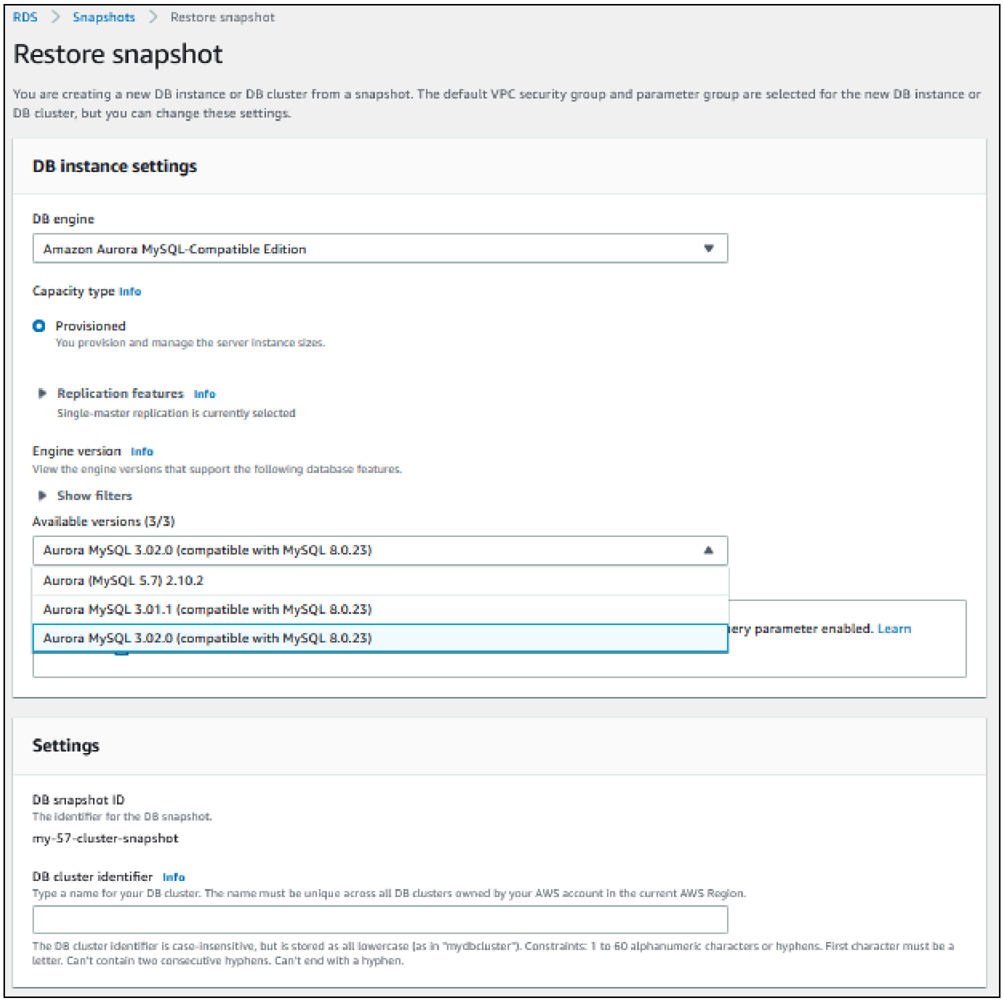
☐	Snapshot name	DB instance or cluster	Snapshot creation time
☑	my-57-cluster-snapshot	mynewdbcluster-cluster	July 05, 2022, 7:37:58 PM UTC
☐	database-2-instance-1-final-snapshot	database-2	June 28, 2022, 6:30:17 PM UTC

The **Restore snapshot** page appears.

5. Under **DB instance settings**, do the following:

   a. Use the default setting for **DB engine**.

   b. For **Available versions**, choose a MySQL–8.0 compatible version, such as **Aurora MySQL 3.02.0 (compatible with MySQL 8.0.23)**.



RDS > Snapshots > Restore snapshot

Restore snapshot

You are creating a new DB instance or DB cluster from a snapshot. The default VPC security group and parameter group are selected for the new DB instance or DB cluster, but you can change these settings.

DB instance settings

DB engine

Amazon Aurora MySQL-Compatible Edition ▼

Capacity type Info

● Provisioned
You provision and manage the server instance sizes.

▶ Replication features Info
Single-master replication is currently selected

Engine version Info
View the engine versions that support the following database features.

▶ Show filters

Available versions (3/3)

Aurora MySQL 3.02.0 (compatible with MySQL 8.0.23) ▲

Aurora (MySQL 5.7) 2.10.2

Aurora MySQL 3.01.1 (compatible with MySQL 8.0.23)

Aurora MySQL 3.02.0 (compatible with MySQL 8.0.23)

ery parameter enabled. Learn

Settings

DB snapshot ID
The identifier for the DB snapshot.

my-57-cluster-snapshot

DB cluster identifier Info
Type a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

6. Under **Settings**, for **DB cluster identifier** enter the unique name that you want to use for the restored DB cluster, for example **`my-80-cluster`**.

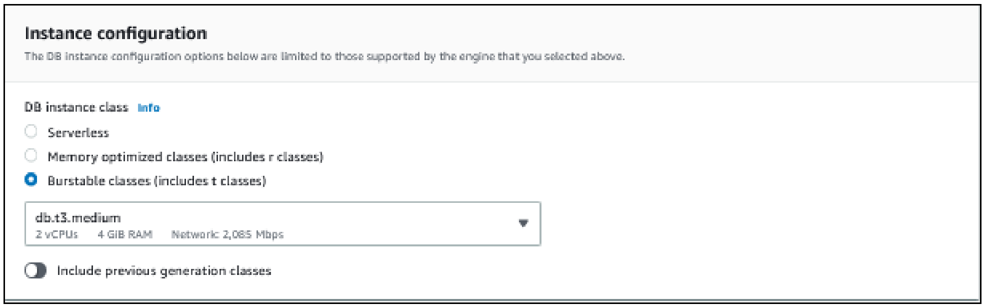
7. Under **Connectivity**, use the default settings for the following:

   - **Virtual private cloud (VPC)**
   - **DB subnet group**
   - **Public access**
   - **VPC security group (firewall)**

8. Choose the **DB instance class**.

   For this tutorial, choose **Burstable classes (includes t classes)**, and then choose **db.t3.medium**.

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class

○ Serverless

○ Memory optimized classes (includes r classes)

● Burstable classes (includes t classes)

db.t3.medium
2 vCPUs 4 GiB RAM Network: 2,085 Mbps

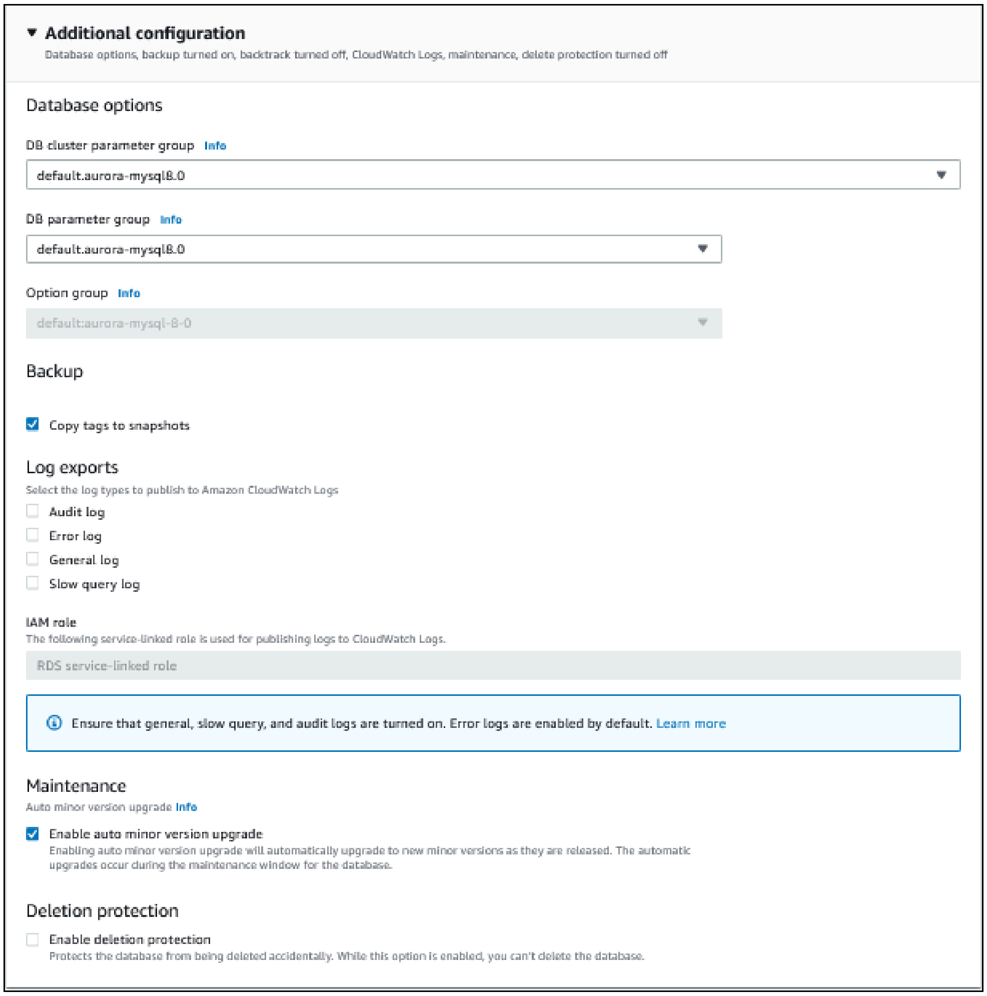
Include previous generation classes

9. For **Database authentication**, use the default setting.

10. For **Encryption**, use the default settings.

    If the source DB cluster for the snapshot was encrypted, the restored DB cluster is also encrypted. You can't make it unencrypted.

11. Expand **Additional configuration** at the bottom of the page.



▼ Additional configuration
Database options, backup turned on, backtrack turned off, CloudWatch Logs, maintenance, delete protection turned off

Database options

DB cluster parameter group Info
default.aurora-mysql8.0 ▼

DB parameter group Info
default.aurora-mysql8.0 ▼

Option group Info
default:aurora-mysql-8-0 ▼

Backup

☑ Copy tags to snapshots

Log exports
Select the log types to publish to Amazon CloudWatch Logs
□ Audit log
□ Error log
□ General log
□ Slow query log

IAM role
The following service-linked role is used for publishing logs to CloudWatch Logs.
RDS service-linked role

Ensure that general, slow query, and audit logs are turned on. Error logs are enabled by default. Learn more

Maintenance
Auto minor version upgrade Info
☑ Enable auto minor version upgrade
Enabling auto minor version upgrade will automatically upgrade to new minor versions as they are released. The automatic upgrades occur during the maintenance window for the database.

Deletion protection
□ Enable deletion protection
Protects the database from being deleted accidentally. While this option is enabled, you can't delete the database.

12. Make the following choices:

    a. For this tutorial, use the default value for **DB cluster parameter group**.

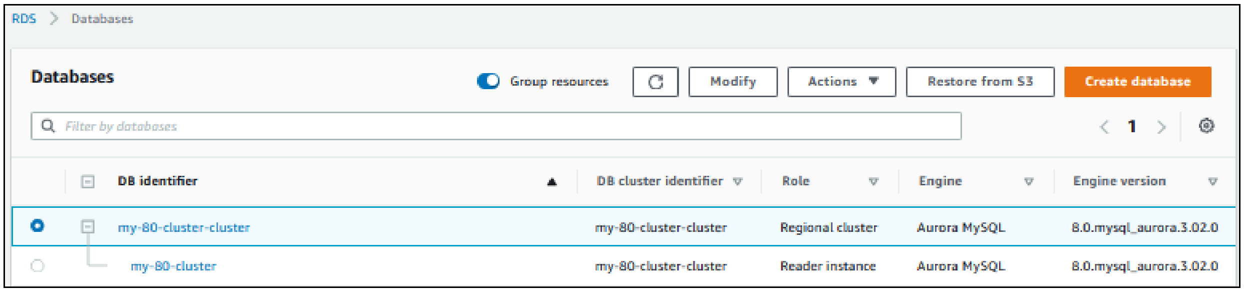
    b. For this tutorial, use the default value for **DB parameter group**.

    c. For **Log exports**, select all of the check boxes.

    d. For **Deletion protection**, select the **Enable deletion protection** check box.

13. Choose **Restore DB instance**.

The **Databases** page displays the restored DB cluster, with a status of `Creating`.



	DB identifier	DB cluster identifier	Role	Engine	Engine version
●	my-80-cluster-cluster	my-80-cluster-cluster	Regional cluster	Aurora MySQL	8.0.mysql_aurora.3.02.0
○	my-80-cluster	my-80-cluster-cluster	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.02.0

While the primary DB instance is being created, it appears as a reader instance, but after creation it's a writer instance.

Restoring a DB cluster from a DB cluster snapshot using the AWS CLI

Restoring a DB cluster from a snapshot using the AWS CLI has two steps:

1. Restoring the DB cluster using the restore-db-cluster-from-snapshot command
2. Creating the primary (writer) DB instance using the create-db-instance command

Restoring the DB cluster

You use the `restore-db-cluster-from-snapshot` command. The following options are required:

- `--db-cluster-identifier` – The name of the restored DB cluster.
- `--snapshot-identifier` – The name of the DB snapshot to restore from.
- `--engine` – The database engine of the restored DB cluster. It must be compatible with the database engine of the source DB cluster.

  The choices are the following:

  - `aurora-mysql` – Aurora MySQL 5.7 and 8.0 compatible.
  - `aurora-postgresql` – Aurora PostgreSQL compatible.

  In this example, we use `aurora-mysql`.
- `--engine-version` – The version of the restored DB cluster. In this example, we use a MySQL-8.0 compatible version.

The following example restores an Aurora MySQL 8.0–compatible DB cluster named `my-new-80-cluster` from a DB cluster snapshot named `my-57-cluster-snapshot`.

To restore the DB cluster

- Use one of the following commands.

  For Linux, macOS, or Unix:

  ```
  aws rds restore-db-cluster-from-snapshot \
      --db-cluster-identifier my-new-80-cluster \
      --snapshot-identifier my-57-cluster-snapshot \
      --engine aurora-mysql \
      --engine-version 8.0.mysql_aurora.3.02.0
  ```

  For Windows:

  ```
  aws rds restore-db-cluster-from-snapshot ^
      --db-cluster-identifier my-new-80-cluster ^
      --snapshot-identifier my-57-cluster-snapshot ^
      --engine aurora-mysql ^
      --engine-version 8.0.mysql_aurora.3.02.0
  ```

The output resembles the following.

```
{
    "DBCluster": {
        "AllocatedStorage": 1,
        "AvailabilityZones": [
            "eu-central-1b",
            "eu-central-1c",
            "eu-central-1a"
        ],
        "BackupRetentionPeriod": 14,
        "DatabaseName": "",
        "DBClusterIdentifier": "my-new-80-cluster",
        "DBClusterParameterGroup": "default.aurora-mysql8.0",
        "DBSubnetGroup": "default",
        "Status": "creating",
        "Endpoint": "my-new-80-cluster.cluster-############.eu-
central-1.rds.amazonaws.com",
        "ReaderEndpoint": "my-new-80-cluster.cluster-ro-############.eu-
central-1.rds.amazonaws.com",
        "MultiAZ": false,
```

```
        "Engine": "aurora-mysql",
        "EngineVersion": "8.0.mysql_aurora.3.02.0",
        "Port": 3306,
        "MasterUsername": "admin",
        "PreferredBackupWindow": "01:55-02:25",
        "PreferredMaintenanceWindow": "thu:21:14-thu:21:44",
        "ReadReplicaIdentifiers": [],
        "DBClusterMembers": [],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-########",
                "Status": "active"
            }
        ],
        "HostedZoneId": "Z1RLNU0EXAMPLE",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:eu-central-1:123456789012:key/#######-5ccc-49cc-8aaa-
############",
        "DbClusterResourceId": "cluster-ZZ12345678ITSJUSTANEXAMPLE",
        "DBClusterArn": "arn:aws:rds:eu-central-1:123456789012:cluster:my-new-80-
cluster",
        "AssociatedRoles": [],
        "IAMDatabaseAuthenticationEnabled": false,
        "ClusterCreateTime": "2022-07-05T20:45:42.171000+00:00",
        "EngineMode": "provisioned",
        "DeletionProtection": false,
        "HttpEndpointEnabled": false,
        "CopyTagsToSnapshot": false,
        "CrossAccountClone": false,
        "DomainMemberships": [],
        "TagList": []
    }
}
```

Creating the primary (writer) DB instance

To create the primary (writer) DB instance, you use the `create-db-instance` command. The following options are required:

- `--db-cluster-identifier` – The name of the restored DB cluster.
- `--db-instance-identifier` – The name of the primary DB instance.

- `--db-instance-class` – The instance class of the primary DB instance. In this example, we use `db.t3.medium`.

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.

- `--engine` – The database engine of the primary DB instance. It must be the same database engine as the restored DB cluster uses.

  The choices are the following:

  - `aurora-mysql` – Aurora MySQL 5.7 and 8.0 compatible.
  - `aurora-postgresql` – Aurora PostgreSQL compatible.

  In this example, we use `aurora-mysql`.

The following example creates a primary (writer) DB instance named `my-new-80-cluster-instance` in the restored Aurora MySQL 8.0–compatible DB cluster named `my-new-80-cluster`.

To create the primary DB instance

- Use one of the following commands.

  For Linux, macOS, or Unix:

  ```
  aws rds create-db-instance \
      --db-cluster-identifier my-new-80-cluster \
      --db-instance-identifier my-new-80-cluster-instance \
      --db-instance-class db.t3.medium \
      --engine aurora-mysql
  ```

  For Windows:

  ```
  aws rds create-db-instance ^
      --db-cluster-identifier my-new-80-cluster ^
      --db-instance-identifier my-new-80-cluster-instance ^
      --db-instance-class db.t3.medium ^
  ```

```
    --engine aurora-mysql
```

The output resembles the following.

```
{
    "DBInstance": {
        "DBInstanceIdentifier": "my-new-80-cluster-instance",
        "DBInstanceClass": "db.t3.medium",
        "Engine": "aurora-mysql",
        "DBInstanceStatus": "creating",
        "MasterUsername": "admin",
        "AllocatedStorage": 1,
        "PreferredBackupWindow": "01:55-02:25",
        "BackupRetentionPeriod": 14,
        "DBSecurityGroups": [],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-########",
                "Status": "active"
            }
        ],
        "DBParameterGroups": [
            {
                "DBParameterGroupName": "default.aurora-mysql8.0",
                "ParameterApplyStatus": "in-sync"
            }
        ],
        "DBSubnetGroup": {
            "DBSubnetGroupName": "default",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-2305ca49",
            "SubnetGroupStatus": "Complete",
            "Subnets": [
                {
                    "SubnetIdentifier": "subnet-########",
                    "SubnetAvailabilityZone": {
                        "Name": "eu-central-1a"
                    },
                    "SubnetOutpost": {},
                    "SubnetStatus": "Active"
                },
                {
                    "SubnetIdentifier": "subnet-########",
                    "SubnetAvailabilityZone": {
                        "Name": "eu-central-1b"
                    },
                    "SubnetOutpost": {},
                    "SubnetStatus": "Active"
                },
                {
                    "SubnetIdentifier": "subnet-########",
                    "SubnetAvailabilityZone": {
                        "Name": "eu-central-1c"
                    },
                    "SubnetOutpost": {},
                    "SubnetStatus": "Active"
                }
            ]
        },
        "PreferredMaintenanceWindow": "sat:02:41-sat:03:11",
        "PendingModifiedValues": {},
        "MultiAZ": false,
        "EngineVersion": "8.0.mysql_aurora.3.02.0",
        "AutoMinorVersionUpgrade": true,
        "ReadReplicaDBInstanceIdentifiers": [],
        "LicenseModel": "general-public-license",
        "OptionGroupMemberships": [
            {
                "OptionGroupName": "default:aurora-mysql-8-0",
                "Status": "in-sync"
            }
        ],
        "PubliclyAccessible": false,
        "StorageType": "aurora",
        "DbInstancePort": 0,
        "DBClusterIdentifier": "my-new-80-cluster",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:eu-central-1:534026745191:key/#######-5ccc-49cc-8aaa-
############",
        "DbiResourceId": "db-5C6UT5PU0YETANOTHEREXAMPLE",
        "CACertificateIdentifier": "rds-ca-2019",
        "DomainMemberships": [],
        "CopyTagsToSnapshot": false,
        "MonitoringInterval": 0,
        "PromotionTier": 1,
        "DBInstanceArn": "arn:aws:rds:eu-central-1:123456789012:db:my-new-80-cluster-instance",
        "IAMDatabaseAuthenticationEnabled": false,
        "PerformanceInsightsEnabled": false,
        "DeletionProtection": false,
        "AssociatedRoles": [],
        "TagList": []
    }
}
```

Monitoring metrics in an Amazon Aurora cluster

Amazon Aurora uses a cluster of replicated database servers. Typically, monitoring an Aurora cluster requires checking the health of multiple DB instances. The instances might have specialized roles, handling mostly write operations, only read operations, or a combination. You also monitor the overall health of the cluster by measuring the *replication lag*. This is the amount of time for changes made by one DB instance to be available to the other instances.

Topics

Overview of monitoring metrics in Amazon Aurora

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Aurora and your AWS solutions. To more easily debug multi-point failures, we recommend that you collect monitoring data from all parts of your AWS solution.

Topics

Monitoring plan

Before you start monitoring Amazon Aurora, create a monitoring plan. This plan should answer the following questions:

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?
- Whom should be notified when something goes wrong?

Performance baseline

To achieve your monitoring goals, you need to establish a baseline. To do this, measure performance under different load conditions at various times in your Amazon Aurora environment. You can monitor metrics such as the following:

- Network throughput
- Client connections
- I/O for read, write, or metadata operations
- Burst credit balances for your DB instances

We recommend that you store historical performance data for Amazon Aurora. Using the stored data, you can compare current performance against past trends. You can also distinguish normal performance patterns from anomalies, and devise techniques to address issues.

Performance guidelines

In general, acceptable values for performance metrics depend on what your application is doing relative to your baseline. Investigate consistent or trending variances from your baseline. The following metrics are often the source of performance issues:

- **High CPU or RAM consumption –** High values for CPU or RAM consumption might be appropriate, if they're in keeping with your goals for your application (like throughput or concurrency) and are expected.
- **Disk space consumption –** Investigate disk space consumption if space used is consistently at or above 85 percent of the total disk space. See if it is possible to delete data from the instance or archive data to a different system to free up space.
- **Network traffic –** For network traffic, talk with your system administrator to understand what expected throughput is for your domain network and internet connection. Investigate network traffic if throughput is consistently lower than expected.
- **Database connections –** If you see high numbers of user connections and also decreases in instance performance and response time, consider constraining database connections. The best number of user connections for your DB instance varies based on your instance class and the complexity of the operations being performed. To determine the number of database connections, associate your DB instance with a parameter group where the `User Connections` parameter is set to a value other than 0 (unlimited). You can either use an existing parameter group or create a new one. For more information, see [Working with parameter groups](#).
- **IOPS metrics –** The expected values for IOPS metrics depend on disk specification and server configuration, so use your baseline to know what is typical. Investigate if values are consistently different than your baseline. For best IOPS performance, make sure that your typical working set fits into memory to minimize read and write operations.

When performance falls outside your established baseline, you might need to make changes to optimize your database availability for your workload. For example, you might need to change the instance class of your DB instance. Or you might need to change the number of DB instances and read replicas that are available for clients.

Monitoring tools

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Aurora and your other AWS solutions. AWS provides various monitoring tools to watch Amazon Aurora, report when something is wrong, and take automatic actions when appropriate.

Topics

- Automated monitoring tools
- Manual monitoring tools

Automated monitoring tools

We recommend that you automate monitoring tasks as much as possible.

Topics

- Amazon Aurora cluster status and recommendations
- Amazon CloudWatch metrics for Amazon Aurora
- Amazon RDS Performance Insights and operating-system monitoring
- Integrated services

Amazon Aurora cluster status and recommendations

You can use the following automated tools to watch Amazon Aurora and report when something is wrong:

- **Amazon Aurora cluster status** — View details about the current status of your cluster by using the Amazon RDS console, the AWS CLI, or the RDS API.
- **Amazon Aurora recommendations** — Respond to automated recommendations for database resources, such as DB instances, DB clusters, and DB cluster parameter groups. For more information, see Viewing and responding to Amazon Aurora recommendations.

Amazon CloudWatch metrics for Amazon Aurora

Amazon Aurora integrates with Amazon CloudWatch for additional monitoring capabilities.

- **Amazon CloudWatch –** This service monitors your AWS resources and the applications you run on AWS in real time. You can use the following Amazon CloudWatch features with Amazon Aurora:
  - **Amazon CloudWatch metrics –** Amazon Aurora automatically sends metrics to CloudWatch every minute for each active database. You don't get additional charges for Amazon RDS metrics in CloudWatch. For more information, see Amazon CloudWatch metrics for Amazon Aurora
  - **Amazon CloudWatch alarms –** You can watch a single Amazon Aurora metric over a specific time period. You can then perform one or more actions based on the value of the metric relative to a threshold that you set.

Amazon RDS Performance Insights and operating-system monitoring

You can use the following automated tools to monitor Amazon Aurora performance:

- **Amazon RDS Performance Insights –** Assess the load on your database, and determine when and where to take action. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.
- **Amazon RDS Enhanced Monitoring** – Look at metrics in real time for the operating system. For more information, see Monitoring OS metrics with Enhanced Monitoring.

Integrated services

The following AWS services are integrated with Amazon Aurora:

- *Amazon EventBridge* is a serverless event bus service that makes it easy to connect your applications with data from a variety of sources. For more information, see Monitoring Amazon Aurora events.
- *Amazon CloudWatch Logs* lets you monitor, store, and access your log files from Amazon Aurora instances, CloudTrail, and other sources. For more information, see Monitoring Amazon Aurora log files.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. For more information, see Monitoring Amazon Aurora API calls in AWS CloudTrail.

- *Database Activity Streams* is an Amazon Aurora feature that provides a near-real-time stream of the activity in your DB cluster. For more information, see Monitoring Amazon Aurora with Database Activity Streams.
- *DevOps Guru for RDS* is a capability of Amazon DevOps Guru that applies machine learning to Performance Insights metrics for Amazon Aurora databases. For more information, see Analyzing Aurora performance anomalies with Amazon DevOps Guru for Amazon RDS.

Manual monitoring tools

You need to manually monitor those items that the CloudWatch alarms don't cover. The Amazon RDS, CloudWatch, AWS Trusted Advisor and other AWS console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on your DB instance.

- From the Amazon RDS console, you can monitor the following items for your resources:
  - The number of connections to a DB instance
  - The amount of read and write operations to a DB instance
  - The amount of storage that a DB instance is currently using
  - The amount of memory and CPU being used for a DB instance
  - The amount of network traffic to and from a DB instance
- From the Trusted Advisor dashboard, you can review the following cost optimization, security, fault tolerance, and performance improvement checks:
  - Amazon RDS Idle DB Instances
  - Amazon RDS Security Group Access Risk
  - Amazon RDS Backups
  - Amazon RDS Multi-AZ
  - Aurora DB Instance Accessibility

  For more information on these checks, see Trusted Advisor best practices (checks).
- CloudWatch home page shows:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

In addition, you can use CloudWatch to do the following:

- Create customized dashboards to monitor the services that you care about.
- Graph metric data to troubleshoot issues and discover trends.
- Search and browse all your AWS resource metrics.
- Create and edit alarms to be notified of problems.

Viewing cluster status

Using the Amazon RDS console, you can quickly access the status of your DB cluster.

Topics

Viewing an Amazon Aurora DB cluster

You have several options for viewing information about your Amazon Aurora DB clusters and the DB instances in your DB clusters.

- You can view DB clusters and DB instances in the Amazon RDS console by choosing **Databases** from the navigation pane.
- You can get DB cluster and DB instance information using the AWS Command Line Interface (AWS CLI).
- You can get DB cluster and DB instance information using the Amazon RDS API.

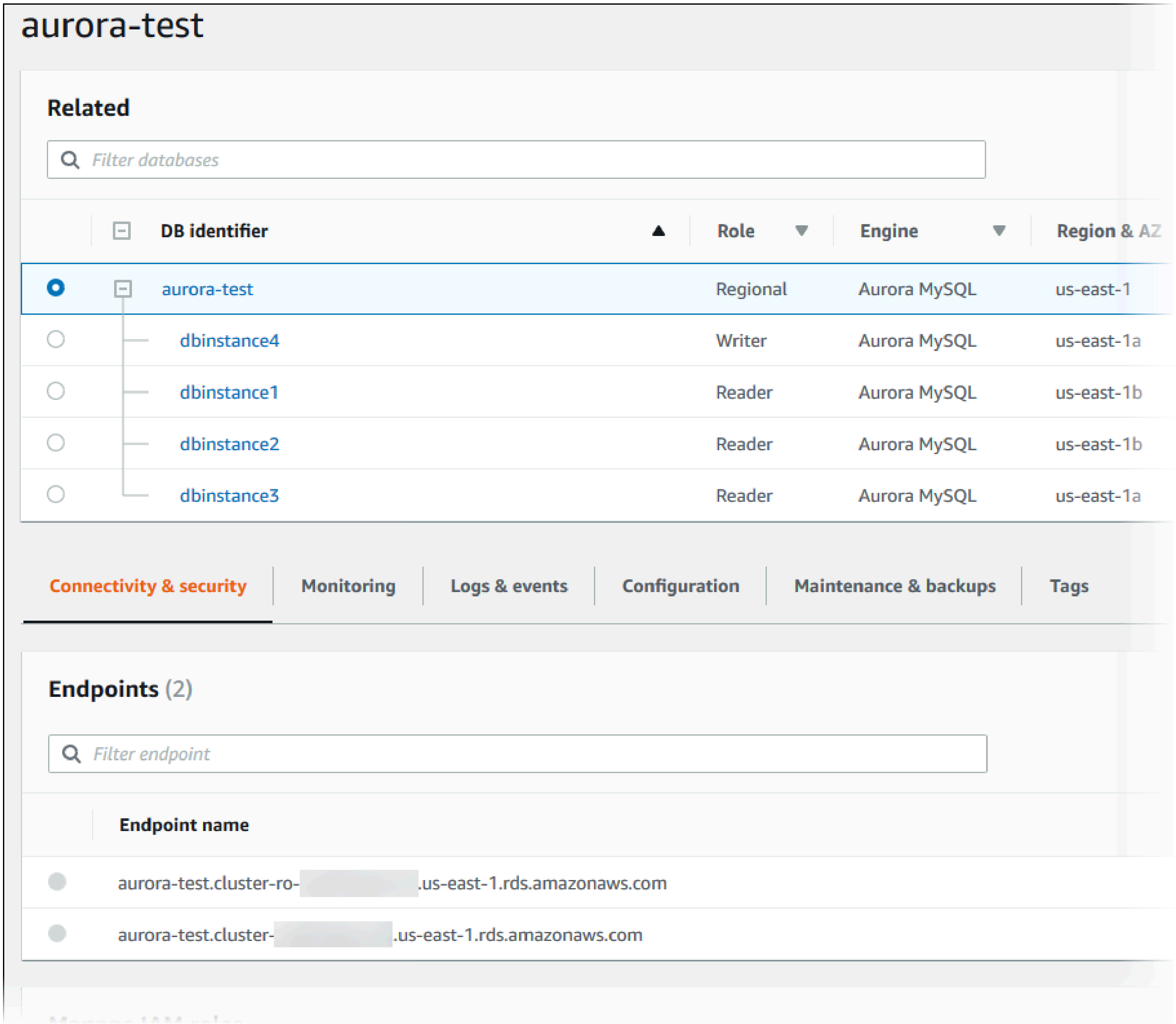
Console

In the Amazon RDS console, you can see details about a DB cluster by choosing **Databases** from the console's navigation pane. You can also see details about DB instances that are members of an Amazon Aurora DB cluster.

To view or modify DB clusters in the Amazon RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the Aurora DB cluster that you want to view from the list.

   For example, the following image shows the details page for the DB cluster named `aurora-test`. The DB cluster has four DB instances shown in the DB identifier list. The writer DB instance, `dbinstance4`, is the primary DB instance for the DB cluster.



aurora-test

Related

	DB identifier	Role	Engine	Region & AZ
●	aurora-test	Regional	Aurora MySQL	us-east-1
○	dbinstance4	Writer	Aurora MySQL	us-east-1a
○	dbinstance1	Reader	Aurora MySQL	us-east-1b
○	dbinstance2	Reader	Aurora MySQL	us-east-1b
○	dbinstance3	Reader	Aurora MySQL	us-east-1a

Endpoints (2)

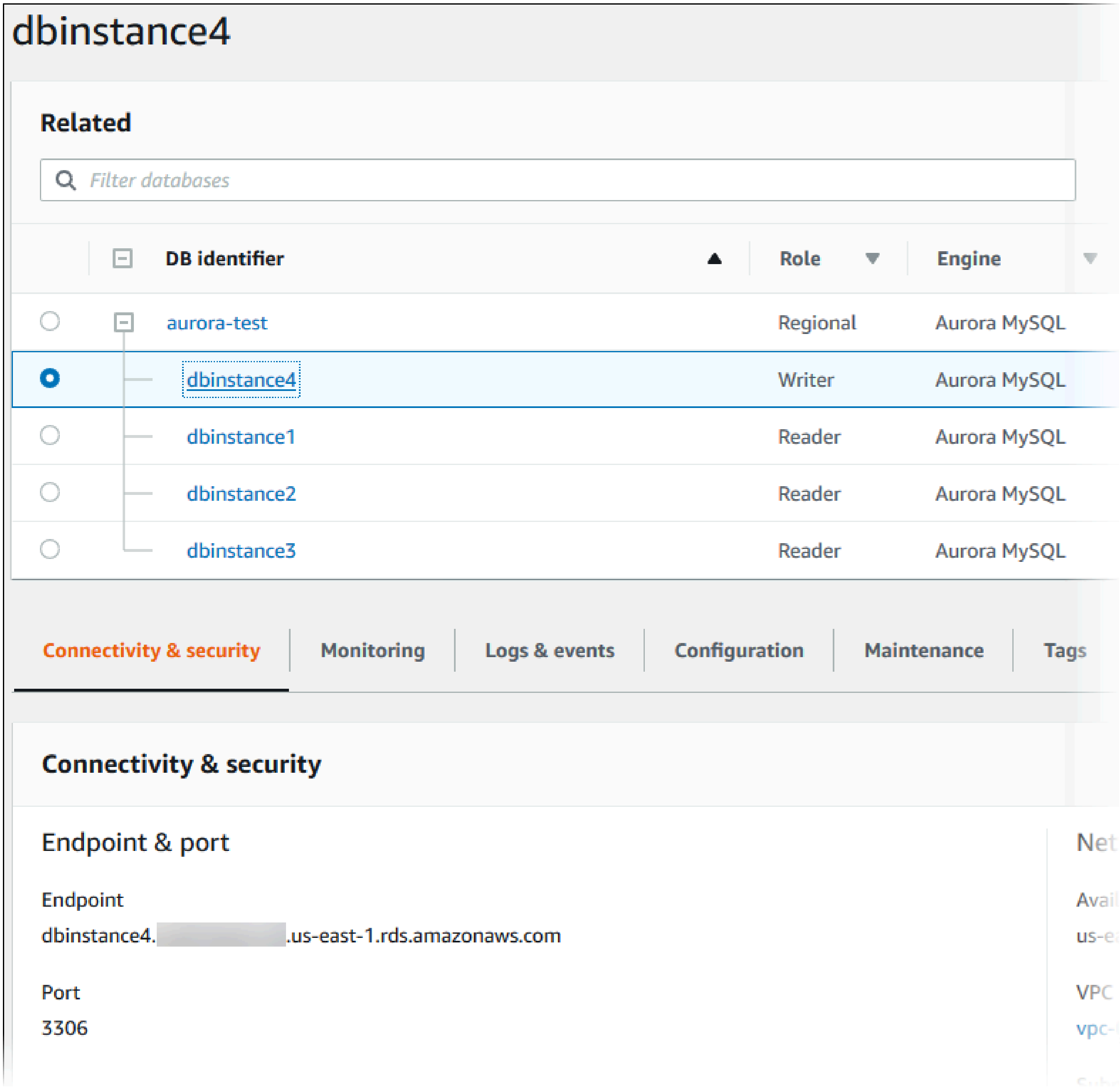
	Endpoint name
●	aurora-test.cluster-ro-[redacted].us-east-1.rds.amazonaws.com
●	aurora-test.cluster-[redacted].us-east-1.rds.amazonaws.com

4. To modify a DB cluster, select the DB cluster from the list and choose **Modify**.

To view or modify DB instances of a DB cluster in the Amazon RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Do one of the following:
   - To view a DB instance, choose one from the list that is a member of the Aurora DB cluster.

For example, if you choose the `dbinstance4` DB instance identifier, the console shows the details page for the `dbinstance4` DB instance, as shown in the following image.



dbinstance4

Related

	DB identifier	Role	Engine
○	aurora-test	Regional	Aurora MySQL
●	dbinstance4	Writer	Aurora MySQL
○	dbinstance1	Reader	Aurora MySQL
○	dbinstance2	Reader	Aurora MySQL
○	dbinstance3	Reader	Aurora MySQL

Connectivity & security

Endpoint & port

Endpoint
dbinstance4.[redacted].us-east-1.rds.amazonaws.com

Port
3306

- To modify a DB instance, choose the DB instance from the list and choose **Modify**. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

AWS CLI

To view DB cluster information by using the AWS CLI, use the describe-db-clusters command. For example, the following AWS CLI command lists the DB cluster information for all of the DB clusters in the modify `us-east-1` region for the configured AWS account.

```
aws rds describe-db-clusters --region us-east-1
```

The command returns the following output if your AWS CLI is configured for JSON output.

```
{
    "DBClusters": [
        {
            "Status": "available",
            "Engine": "aurora-mysql",
            "Endpoint": "sample-cluster1.cluster-123456789012.us-
east-1.rds.amazonaws.com"
            "AllocatedStorage": 1,
            "DBClusterIdentifier": "sample-cluster1",
            "MasterUsername": "mymasteruser",
            "EarliestRestorableTime": "2023-03-30T03:35:42.563Z",
            "DBClusterMembers": [
                {
                    "IsClusterWriter": false,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "DBInstanceIdentifier": "sample-replica"
                },
                {
                    "IsClusterWriter": true,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "DBInstanceIdentifier": "sample-primary"
                }
            ],
            "Port": 3306,
            "PreferredBackupWindow": "03:34-04:04",
            "VpcSecurityGroups": [
                {
                    "Status": "active",
                    "VpcSecurityGroupId": "sg-ddb65fec"
                }
            ],
            "DBSubnetGroup": "default",
            "StorageEncrypted": false,
            "DatabaseName": "sample",
            "EngineVersion": "5.7.mysql_aurora.2.11.0",
            "DBClusterParameterGroup": "default.aurora-mysql5.7",
            "BackupRetentionPeriod": 1,
            "AvailabilityZones": [
                "us-east-1b",
                "us-east-1c",
                "us-east-1d"
            ],
            "LatestRestorableTime": "2023-03-31T20:06:08.903Z",
            "PreferredMaintenanceWindow": "wed:08:15-wed:08:45"
        },
        {
            "Status": "available",
            "Engine": "aurora-mysql",
            "Endpoint": "aurora-sample.cluster-123456789012.us-
east-1.rds.amazonaws.com",
            "AllocatedStorage": 1,
            "DBClusterIdentifier": "aurora-sample-cluster",
            "MasterUsername": "mymasteruser",
            "EarliestRestorableTime": "2023-03-30T10:21:34.826Z",
            "DBClusterMembers": [
                {
                    "IsClusterWriter": false,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "DBInstanceIdentifier": "aurora-replica-sample"
                },
                {
                    "IsClusterWriter": true,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "DBInstanceIdentifier": "aurora-sample"
                }
            ],
            "Port": 3306,
            "PreferredBackupWindow": "10:20-10:50",
            "VpcSecurityGroups": [
                {
                    "Status": "active",
                    "VpcSecurityGroupId": "sg-55da224b"
                }
            ],
            "DBSubnetGroup": "default",
            "StorageEncrypted": false,
            "DatabaseName": "sample",
            "EngineVersion": "5.7.mysql_aurora.2.11.0",
            "DBClusterParameterGroup": "default.aurora-mysql5.7",
            "BackupRetentionPeriod": 1,
            "AvailabilityZones": [
                "us-east-1b",
                "us-east-1c",
                "us-east-1d"
            ],
            "LatestRestorableTime": "2023-03-31T20:00:11.491Z",
            "PreferredMaintenanceWindow": "sun:03:53-sun:04:23"
        }
    ]
}
```

RDS API

To view DB cluster information using the Amazon RDS API, use the DescribeDBClusters operation.

Viewing DB cluster status

The status of a DB cluster indicates its health. You can view the status of a DB cluster and the cluster instances by using the Amazon RDS console, the AWS CLI, or the API.

> **Note**
>
> Aurora also uses another status called *maintenance status*, which is shown in the **Maintenance** column of the Amazon RDS console. This value indicates the status of any maintenance patches that need to be applied to a DB cluster. Maintenance status is independent of DB cluster status. For more information about maintenance status, see Applying updates for a DB cluster.

Find the possible status values for DB clusters in the following table.

DB cluster status	Billed	Description
Available	Billed	The DB cluster is healthy and available. When an Aurora Serverless cluster is available and paused, you're billed for storage only.
Backing-up	Billed	The DB cluster is currently being backed up.
Backtracking	Billed	The DB cluster is currently being backtracked. This status only applies to Aurora MySQL.
Cloning-failed	Not billed	Cloning a DB cluster failed.
Creating	Not billed	The DB cluster is being created. The DB cluster is inaccessible while it is being created.
Deleting	Not billed	The DB cluster is being deleted.
Failing-over	Billed	A failover from the primary instance to an Aurora Replica is being performed.
Inaccessible-encryption-credentials	Not billed	The AWS KMS key used to encrypt or decrypt the DB cluster can't be accessed or recovered.

DB cluster status	Billed	Description
Inaccessible-encryption-credentials-recoverable	Billed for storage	The KMS key used to encrypt or decrypt the DB cluster can't be accessed. However, if the KMS key is active, restarting the DB cluster can recover it. For more information, see Encrypting an Amazon Aurora DB cluster.
Maintenance	Billed	Amazon RDS is applying a maintenance update to the DB cluster. This status is used for DB cluster-level maintenance that RDS schedules well in advance.
Migrating	Billed	A DB cluster snapshot is being restored to a DB cluster.
Migration-failed	Not billed	A migration failed.
Modifying	Billed	The DB cluster is being modified because of a customer request to modify the DB cluster.
Promoting	Billed	A read replica is being promoted to a standalone DB cluster.
Preparing-data-migration	Billed	Amazon RDS is preparing to migrate data to Aurora.
Renaming	Billed	The DB cluster is being renamed because of a customer request to rename it.
Resetting-master-credentials	Billed	The master credentials for the DB cluster are being reset because of a customer request to reset them.
Starting	Billed for storage	The DB cluster is starting.
Stopped	Billed for storage	The DB cluster is stopped.

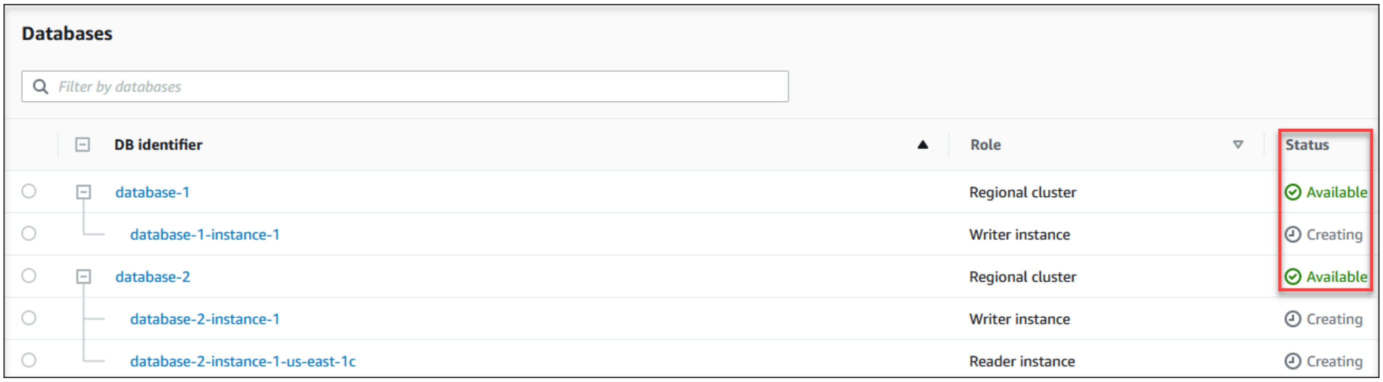
DB cluster status	Billed	Description
Stopping	Billed for storage	The DB cluster is being stopped.
Storage-optimization	Billed	Your DB instance is being modified to change the storage size or type. The DB instance is fully operational. However, while the status of your DB instance is **storage-optimization**, you can't request any changes to the storage of your DB instance. The storage optimization process is usually short, but can sometimes take up to and even beyond 24 hours.
Update-iam-db-auth	Billed	IAM authorization for the DB cluster is being updated.
Upgrading	Billed	The DB cluster engine version is being upgraded.

Console

To view the status of a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

   The **Databases page** appears with the list of DB clusters. For each DB cluster, the status value is displayed.



DB identifier	Role	Status
database-1	Regional cluster	✓ Available
database-1-instance-1	Writer instance	Creating
database-2	Regional cluster	✓ Available
database-2-instance-1	Writer instance	Creating
database-2-instance-1-us-east-1c	Reader instance	Creating

CLI

To view just the status of the DB clusters, use the following query in AWS CLI.

```
aws rds describe-db-clusters --query 'DBClusters[*].[DBClusterIdentifier,Status]' --
output table
```

Viewing DB instance status in an Aurora cluster

The status of a DB instance in an Aurora cluster indicates the health of the DB instance. You can use the following procedures to view the DB instance status of a cluster in the Amazon RDS console, the AWS CLI command, or the API operation.

> **Note**
>
> Amazon RDS also uses another status called *maintenance status*, which is shown in the **Maintenance** column of the Amazon RDS console. This value indicates the status of any maintenance patches that need to be applied to a DB instance. Maintenance status is independent of DB instance status. For more information about maintenance status, see Applying updates for a DB cluster.

Find the possible status values for DB instances in the following table. This table also shows whether you will be billed for the DB instance and storage, billed only for storage, or not billed. For all DB instance statuses, you are always billed for backup usage.

DB instance status	Billed	Description
Available	Billed	The DB instance is healthy and available.
Backing-up	Billed	The DB instance is currently being backed up.
Backtracking	Billed	The DB instance is currently being backtracked. This status only applies to Aurora MySQL.
Configuring-enhanced-monitoring	Billed	Enhanced Monitoring is being enabled or disabled for this DB instance.
Configuring-iam-database-auth	Billed	AWS Identity and Access Management (IAM) database authentication is being enabled or disabled for this DB instance.
Configuring-log-exports	Billed	Publishing log files to Amazon CloudWatch Logs is being enabled or disabled for this DB instance.

DB instance status	Billed	Description
Converting-to-vpc	Billed	The DB instance is being converted from a DB instance that is not in an Amazon Virtual Private Cloud (Amazon VPC) to a DB instance that is in an Amazon VPC.
Creating	Not billed	The DB instance is being created. The DB instance is inaccessible while it is being created.
Delete-precheck	Not billed	Amazon RDS is validating that read replicas are healthy and are safe to delete.
Deleting	Not billed	The DB instance is being deleted.
Failed	Not billed	The DB instance has failed and Amazon RDS can't recover it. Perform a point-in-time restore to the latest restorable time of the DB instance to recover the data.
Inaccessible-encryption-credentials	Not billed	The AWS KMS key used to encrypt or decrypt the DB instance can't be accessed or recovered.
Inaccessible-encryption-credentials-recoverable	Billed for storage	The KMS key used to encrypt or decrypt the DB instance can't be accessed. However, if the KMS key is active, restarting the DB instance can recover it. For more information, see Encrypting an Amazon Aurora DB cluster.
Incompatible-network	Not billed	Amazon RDS is attempting to perform a recovery action on a DB instance but can't do so because the VPC is in a state that prevents the action from being completed. This status can occur if, for example, all available IP addresses in a subnet are in use and Amazon RDS can't get an IP address for the DB instance.

DB instance status	Billed	Description
Incompatible-option-group	Billed	Amazon RDS attempted to apply an option group change but can't do so, and Amazon RDS can't roll back to the previous option group state. For more information, check the **Recent Events** list for the DB instance. This status can occur if, for example, the option group contains an option such as TDE and the DB instance doesn't contain encrypted information.
Incompatible-parameters	Billed	Amazon RDS can't start the DB instance because the parameters specified in the DB instance's DB parameter group aren't compatible with the DB instance. Revert the parameter changes or make them compatible with the DB instance to regain access to your DB instance. For more information about the incompatible parameters, check the **Recent Events** list for the DB instance.
Incompatible-restore	Not billed	Amazon RDS can't do a point-in-time restore. Common causes for this status include using temp tables or using MyISAM tables with MySQL.
Insufficient-capacity	Not billed	Amazon RDS can’t create your instance because sufficient capacity isn’t currently available. To create your DB instance in the same AZ with the same instance type, delete your DB instance, wait a few hours, and try to create again. Alternatively, create a new instance using a different instance class or AZ.
Maintenance	Billed	Amazon RDS is applying a maintenance update to the DB instance. This status is used for instance-level maintenance that RDS schedules well in advance.
Modifying	Billed	The DB instance is being modified because of a customer request to modify the DB instance.
Moving-to-vpc	Billed	The DB instance is being moved to a new Amazon Virtual Private Cloud (Amazon VPC).

DB instance status	Billed	Description
Rebooting	Billed	The DB instance is being rebooted because of a customer request or an Amazon RDS process that requires the rebooting of the DB instance.
Resetting-master-credentials	Billed	The master credentials for the DB instance are being reset because of a customer request to reset them.
Renaming	Billed	The DB instance is being renamed because of a customer request to rename it.
Restore-error	Billed	The DB instance encountered an error attempting to restore to a point-in-time or from a snapshot.
Starting	Billed for storage	The DB instance is starting.
Stopped	Billed for storage	The DB instance is stopped.
Stopping	Billed for storage	The DB instance is being stopped.
Storage-config-upgrade	Billed	The storage file system configuration of the DB instance is being upgraded. This status only applies to green databases within a blue/green deployment, or to DB instance read replicas.
Storage-full	Billed	The DB instance has reached its storage capacity allocation. This is a critical status, and we recommend that you fix this issue immediately. To do so, scale up your storage by modifying the DB instance. To avoid this situation, set Amazon CloudWatch alarms to warn you when storage space is getting low.

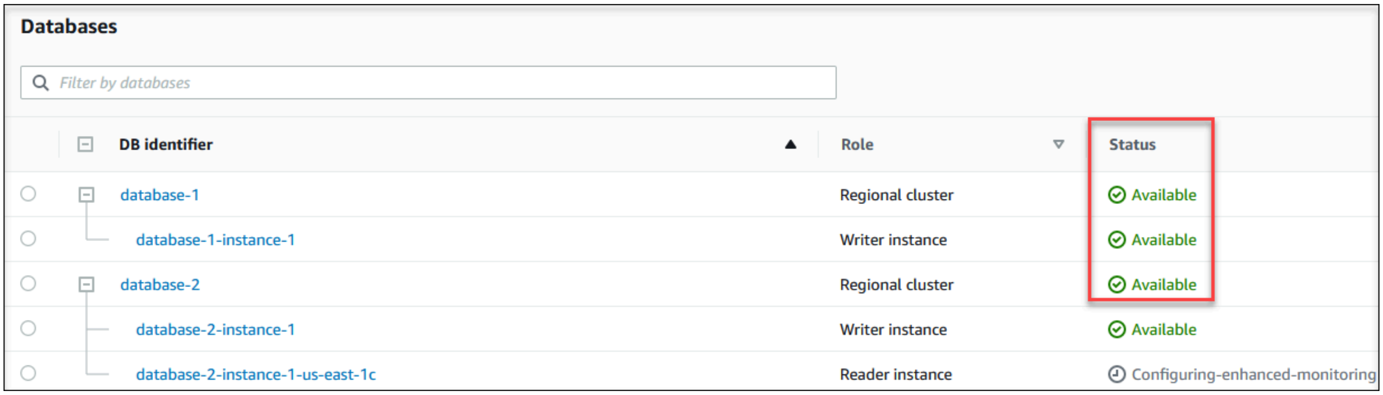
DB instance status	Billed	Description
Storage-optimization	Billed	Amazon RDS is optimizing the storage of your DB instance. The storage optimization process is usually short, but can sometimes take up to and even beyond 24 hours. During storage optimization, the DB instance remains available. Storage optimization is a background process that doesn't affect the instance's availability.
Upgrading	Billed	The database engine version is being upgraded.

Console

To view the status of a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

   The **Databases page** appears with the list of DB instances. For each DB instance in a cluster, the status value is displayed.



Databases

Filter by databases

	DB identifier ▲	Role ▽	Status
○	database-1	Regional cluster	✓ Available
○	database-1-instance-1	Writer instance	✓ Available
○	database-2	Regional cluster	✓ Available
○	database-2-instance-1	Writer instance	✓ Available
○	database-2-instance-1-us-east-1c	Reader instance	Configuring-enhanced-monitoring

CLI

To view DB instance and its status information by using the AWS CLI, use the [describe-db-instances](https://docs.aws.amazon.com/cli/latest/reference/rds/describe-db-instances.html) command. For example, the following AWS CLI command lists all the DB instances information .

```
aws rds describe-db-instances
```

To view a specific DB instance and its status, call the describe-db-instances command with the following option:

- `DBInstanceIdentifier` – The name of the DB instance.

```
aws rds describe-db-instances --db-instance-identifier mydbinstance
```

To view just the status of all the DB instances, use the following query in AWS CLI.

```
aws rds describe-db-instances --query 'DBInstances[*].
[DBInstanceIdentifier,DBInstanceStatus]' --output table
```

API

To view the status of the DB instance using the Amazon RDS API, call the DescribeDBInstances operation.

Viewing and responding to Amazon Aurora recommendations

Amazon Aurora provides automated recommendations for database resources, such as DB instances, DB clusters, and DB parameter groups. These recommendations provide best practice guidance by analyzing DB cluster configuration, DB instance configuration, usage, and performance data.

Amazon RDS Performance Insights monitors specific metrics and automatically creates thresholds by analyzing what levels are considered potentially problematic for a specified resource. When new metric values cross a predefined threshold over a given period of time, Performance Insights generates a proactive recommendation. This recommendation helps to prevent future database performance impact. For example, the "Idle In Transaction" recommendation is generated for Aurora PostgreSQL instances when the sessions connected to the database are not performing active work, but can keep database resources blocked. To receive proactive recommendations, you must turn on Performance Insights with a paid tier retention period. For information about turning on Performance Insights, see Turning Performance Insights on and off for Aurora. For information about pricing and data retention for Performance Insights see Pricing and data retention for Performance Insights.

DevOps Guru for RDS monitors certain metrics to detect when the metric's behavior becomes highly unusual or anomalous. These anomalies are reported as reactive insights with recommendations. For example, DevOps Guru for RDS might recommend you to consider increasing CPU capacity or investigate wait events that are contributing to DB load. DevOps Guru for RDS also provides threshold based proactive recommendations. For these recommendations, you must turn on DevOps Guru for RDS. For information about turning on DevOps Guru for RDS, see Turning on DevOps Guru and specifying resource coverage.

Recommendations will be in any of the following status: active, dismissed, pending, or resolved. Resolved recommendations are available for 365 days.

You can view or dismiss the recommendations. You can apply a configuration based active recommendation immediately, schedule it in the next maintenance window, or dismiss it. For threshold based proactive and machine learning based reactive recommendations, you need to review the suggested cause of the issue and then perform the recommended actions to fix the issue.

Topics

- Responding to Amazon Aurora recommendations

Viewing Amazon Aurora recommendations

Amazon Aurora generates recommendations for a resource when the resource is created or modified.

The configuration based recommendations are supported in the following regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (N. California)
- US West (Oregon)
- Asia Pacific (Mumbai)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- South America (São Paulo)

You can find examples of the configuration based recommendations in the following table.

Type	Description	Recommendation	Downt require	Additional informati on
Resource Automated backups is turned off	Automated backups aren't turned on for your DB instances. Automated backups are recommend ed because they	Turn on automated backups with a retention period of up to 14 days.	Yes	Overview of backing up and restoring an Aurora DB cluster Demystifying Amazon RDS backup

Type	Description	Recommendation	Downt require	Additional informati on
	enable point-in-time recovery of your DB instances.			storage costs on the AWS Database Blog
Engine minor version upgrade is required	Your database resources aren't running the latest minor DB engine version. The latest minor version contains the latest security fixes and other improvements.	Upgrade to latest engine version.	Yes	Maintaining an Amazon Aurora DB cluster
Enhanced Monitoring is turned off	Your database resources don't have Enhanced Monitoring turned on. Enhanced Monitoring provides real-time operating system metrics for monitoring and troubleshooting.	Turn on Enhanced Monitoring.	No	Monitoring OS metrics with Enhanced Monitoring

Type	Description	Recommendation	Downt require	Additional information
Storage encryption is turned off	Amazon RDS supports encryption at rest for all the database engines by using the keys that you manage in AWS Key Management Service (AWS KMS). On an active DB instance with Amazon RDS encryption, the data stored at rest in the storage is encrypted, similar to automated backups, read replicas, and snapshots. If encryption isn't turned on while creating an Aurora DB cluster, you must restore a decrypted snapshot to an encrypted DB cluster.	Turn on encryption of data at rest for your DB cluster.	Yes	Security in Amazon Aurora
DB clusters with all instances in the same Availability Zone	The DB clusters are currently in a single Availability Zone. Use multiple Availability Zones to improve the availability.	Add the DB instances to multiple Availability Zones in your DB cluster.	No	High availability for Amazon Aurora

Type	Description	Recommendation	Downt require	Additional informati on
DB instances in the clusters with heterogeneous instance sizes	We recommend that you use the same DB instance class and size for all the DB instances in your DB cluster.	Use the same instance class and size for all the DB instances in your DB cluster.	Yes	Replication with Amazon Aurora
DB instances in the clusters with heterogeneous instance classes	We recommend that you use the same DB instance class and size for all the DB instances in your DB cluster.	Use the same instance class and size for all the DB instances in your DB cluster.	Yes	Replication with Amazon Aurora
DB instances in the clusters with heterogeneous parameter groups	We recommend that all of the DB instances in the DB cluster use the same DB parameter group.	Associate the DB instance with the DB parameter group associated with the writer instance in your DB cluster.	No	Working with parameter groups
Amazon RDS DB clusters have one DB instance	Add at least one more DB instance to your DB cluster to improve availability and performance.	Add a reader DB instance to your DB cluster.	No	High availability for Amazon Aurora

Type	Description	Recommendation	Downt require	Additional information
Performance Insights is turned off	Performance Insights monitors your DB instance load to help you analyze and resolve database performance issues. We recommend that you turn on Performance Insights.	Turn on Performance Insights.	No	Monitoring DB load with Performance Insights on Amazon Aurora
RDS resources major versions update is required	Databases with the current major version for the DB engine won't be supported. We recommend that you upgrade to the latest major version which includes new functionality and enhancements.	Upgrade to the latest major version for the DB engine.	Yes	Amazon Aurora updates Creating a blue/green deployment

Type	Description	Recommendation	Downt require	Additional informati on
DB clusters support only up to 64 TiB volume	Your DB clusters support volumes up to 64 TiB. The latest engine versions support volumes up to 128 TiB for your DB cluster. We recommend that you upgrade the engine version of your DB cluster to the latest versions to support volumes up to 128 TiB.	Upgrade the engine version of your DB clusters to support volumes up to 128 TiB.	Yes	Amazon Aurora size limits

Type	Description	Recommendation	Downt require	Additional informati on
DB clusters with all reader instances in the same Availability Zone	Availability Zones (AZs) are locations that are distinct from each other to provide isolation in case of outages within each AWS Region. We recommend that you distribute the primary instance and reader instances in your DB cluster across multiple AZs to improve the availability of your DB cluster. You can create a Multi-AZ cluster using the AWS Management Console, AWS CLI, or Amazon RDS API when you create the cluster. You can modify the existing Aurora cluster to a Multi-AZ cluster by adding a new reader instance and specifying a different AZ.	Your DB cluster has all of its read instances in the same Availability Zone. We recommend that you distribute the reader instances across multiple Availability Zones. Distribution increases availability and improves response time by reducing network latency between clients and the database.	No	High availability for Amazon Aurora

Type	Description	Recommendation	Downt require	Additional informati on
DB memory parameters are diverging from default	The memory parameters of the DB instances are significantly different from the default values. These settings can impact performance and cause errors. We recommend that you reset the custom memory parameters for the DB instance to their default values in the DB parameter group.	Reset the memory parameters to their default values.	No	Working with parameter groups
Query cache parameter is turned on	When changes require that your query cache is purged, your DB instance will appear to stall. Most workloads don't benefit from a query cache. The query cache was removed from MySQL version 8.0. We recommend that you set the query_cache_type parameter to 0.	Set the `query_cache_type` parameter value to `0` in your DB parameter groups.	Yes	Working with parameter groups

Type	Description	Recommendation	Downt require	Additional information
`log_output` parameter is set to table	When `log_output` is set to `TABLE`, more storage is used than when `log_output` is set to `FILE`. We recommend that you set the parameter to `FILE`, to avoid reaching the storage size limit.	Set the `log_output` parameter value to `FILE` in your DB parameter groups.	No	Aurora MySQL database log files
`autovacuum` parameter is turned off	The autovacuum parameter is turned off for your DB clusters. Turning autovacuum off increases the table and index bloat and impacts the performance. We recommend that you turn on autovacuum in your DB parameter groups.	Turn on the autovacuum parameter in your DB cluster parameter groups.	No	Understanding autovacuum in Amazon RDS for PostgreSQL environments on the AWS Database Blog

Type	Description	Recommendation	Downt require	Additional information
`synchronous_commit` parameter is turned off	When `synchronous_commit` parameter is turned off, data can be lost in a database crash. The durability of the database is at risk. We recommend that you turn on the `synchronous_commit` parameter.	Turn on `synchronous_commit` parameter in your DB parameter groups.	Yes	[Amazon Aurora PostgreSQL parameters: Replication, security, and logging]() on the AWS Database Blog
`track_counts` parameter is turned off	When the `track_counts` parameter is turned off, the database doesn't collect the database activity statistics. Autovacuum requires these statistics to work correctly. We recommend that you set `track_counts` parameter to 1.	Set `track_counts` parameter to 1.	No	[Run-time Statistics for PostgreSQL]()

Type	Description	Recommendation	Downt require	Additional information
`enable_indexonlyscan` parameter is turned off	The query planner or optimizer can't use the index-only scan plan type when it is turned off. We recommend that you set the `enable_indexonlyscan` parameter value to 1.	Set the `enable_indexonlyscan` parameter value to 1.	No	Planner Method Configuration for PostgreSQL
`enable_indexscan` parameter is turned off	The query planner or optimizer can't use the index scan plan type when it is turned off. We recommend that you set the `enable_indexscan` value to 1.	Set the `enable_indexscan` parameter value to 1.	No	Planner Method Configuration for PostgreSQL

Type	Description	Recommendation	Downt require	Additional information
innodb_flush_log_at_trx parameter is turned off	The value of the innodb_flush_log_at_trx parameter of your DB instance isn't safe value. This parameter controls the persistence of commit operations to disk. We recommend that you set the innodb_flush_log_at_trx parameter to 1.	Set the innodb_flush_log_at_trx parameter value to 1.	No	Configuring how frequently the log buffer is flushed

Type	Description	Recommendation	Downt require	Additional informati on
`innodb_stats_persistent` parameter is turned off	Your DB instance isn't configured to persist the InnoDB statistics to the disk. When the statistics aren't stored, they are recalculated each time the instance restarts and the table accessed. This leads to variations in the query execution plan. You can modify the value of this global parameter at the table level. We recommend that you set the `innodb_stats_persistent` parameter value to `ON`.	Set the `innodb_stats_persistent` parameter value to `ON`.	No	[Working with parameter groups](#)

Type	Description	Recommendation	Downt require	Additional information
`innodb_open_files` parameter is low	The `innodb_open_files` parameter controls the number of files InnoDB can open at one time. InnoDB opens all of the log and system tablespace files when mysqld is running. Your DB instance has a low value for the maximum number of files InnoDB can open at one time. We recommend that you set the `innodb_open_files` parameter to a minimum value of 65.	Set the `innodb_open_files` parameter to a minimum value of 65.	Yes	InnoDB open files for MySQL

Type	Description	Recommendation	Downt require	Additional informati on
`max_user_connections` parameter is low	Your DB instance has a low value for the maximum number of simultaneous connections for each database account. We recommend setting the `max_user_connections` parameter to a number greater than 5.	Increase the value of the `max_user_connections` parameter to a number greater than 5.	Yes	[Setting Account Resource Limits for MySQL]()
Read Replicas are open in writable mode	Your DB instance has a read replica in writable mode, which allows updates from clients. We recommend that you set the the `read_only` parameter to `TrueIfReplica` so that the read replicas isn't in writable mode.	Set the `read_only` parameter value to `TrueIfReplica` .	No	[Working with parameter groups]()

Type	Description	Recommendation	Downtime required	Additional information
innodb_default_row_format parameter setting is unsafe	Your DB instance encounters a known issue: A table created in a MySQL version lower than 8.0.26 with the row_format set to COMPACT or REDUNDANT will be inaccessible and unrecoverable when the index exceeds 767 bytes. We recommend that you set the innodb_default_row_format parameter value to DYNAMIC.	Set the innodb_default_row_format parameter value to DYNAMIC.	No	Changes in MySQL 8.0.26

Type	Description	Recommendation	Downt require	Additional informati on
`general_logging` parameter is turned on	The general logging is turned on for your DB instance. This setting is useful while troubleshooting the database issues. However, turning on general logging increases the amount of I/O operations and allocated storage space, which might result in contention and performance degradation. Check your requirements for general logging usage. We recommend that you set the `general_logging` parameter value to `0`.	Check your requirements for general logging usage. If it isn't mandatory, we recommend that you to set the `general_logging` parameter value to `0`.	No	[Overview of Aurora MySQL database logs]()

Type	Description	Recommendation	Downt require	Additional informati on
DB cluster under-provisioned for read workload	We recommend that you add a reader DB instance to your DB cluster with the same instance class and size as the writer DB instance in the cluster. The current configuration has one DB instance with a continuously high database load caused mostly by read operations. Distribute these operations by adding another DB instance to the cluster and directing the read workload to the DB cluster read-only endpoint.	Add a reader DB instance to the cluster.	No	Adding Aurora Replicas to a DB cluster Managing performance and scaling for Aurora DB clusters Amazon RDS pricing

Type	Description	Recommendation	Downt require	Additional information
RDS instance under-provisioned for system memory capacity	We recommend that you tune your queries to use lesser memory or use a DB instance type with higher allocated memory. When the instance is running low on memory, then the database performance is impacted.	Use a DB instance with higher memory capacity	Yes	Scaling Your Amazon RDS Instance Vertically and Horizontally on the AWS Database Blog Amazon RDS instance types Amazon RDS pricing
RDS instance under-provisioned for system CPU capacity	We recommend that you tune your queries to use less CPU or modify your DB instance to use a DB instance class with higher allocated vCPUs. Database performance might decline when a DB instance is running low on CPU.	Use a DB instance with higher CPU capacity	Yes	Scaling Your Amazon RDS Instance Vertically and Horizontally on the AWS Database Blog Amazon RDS instance types Amazon RDS pricing

Type	Description	Recommendation	Downt require	Additional information
RDS resources are not utilizing connection pooling correctly	We recommend that you enable Amazon RDS Proxy to efficiently pool and share existing database connections. If you are already using a proxy for your database, configure it correctly to improve connection pooling and load balancing across multiple DB instances. RDS Proxy can help reduce the risk of connection exhaustion and downtime while improving availability and scalability.	Enable RDS Proxy or modify your existing proxy configuration	No	Scaling Your Amazon RDS Instance Vertically and Horizontally on the AWS Database Blog Using Amazon RDS Proxy for Aurora Amazon RDS Proxy Pricing

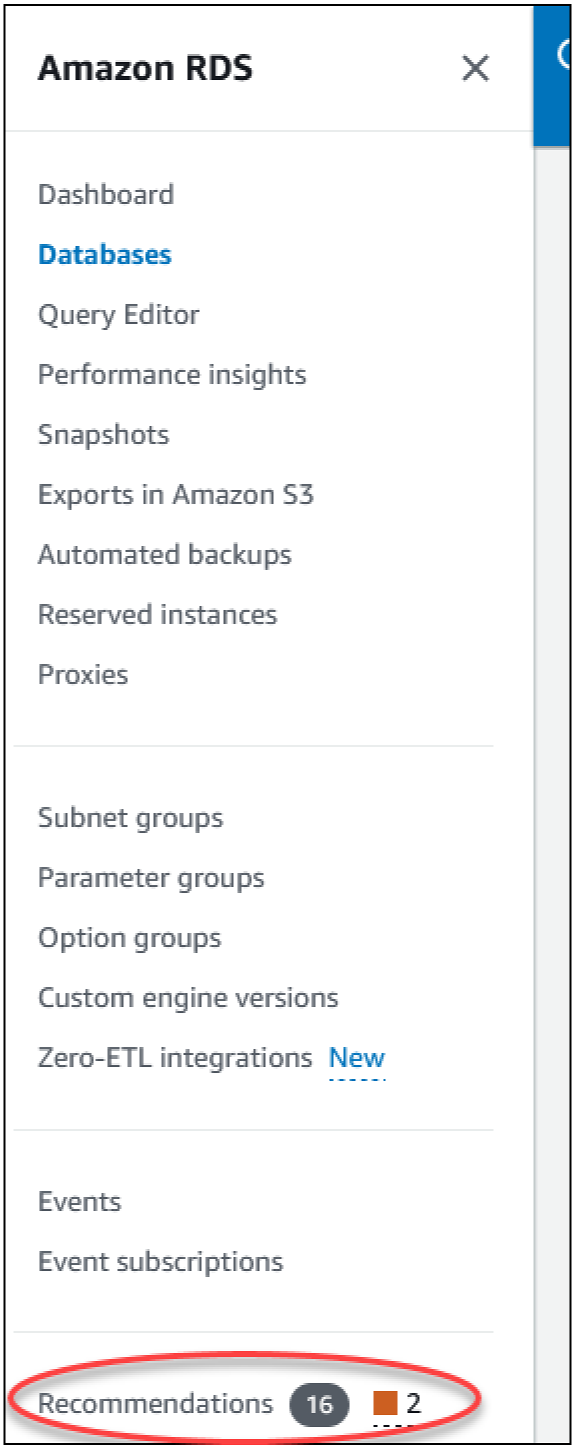
Using the Amazon RDS console, you can view Amazon Aurora recommendations for your database resources. For a DB cluster, the recommendations appear for the DB cluster and its instances.

Console

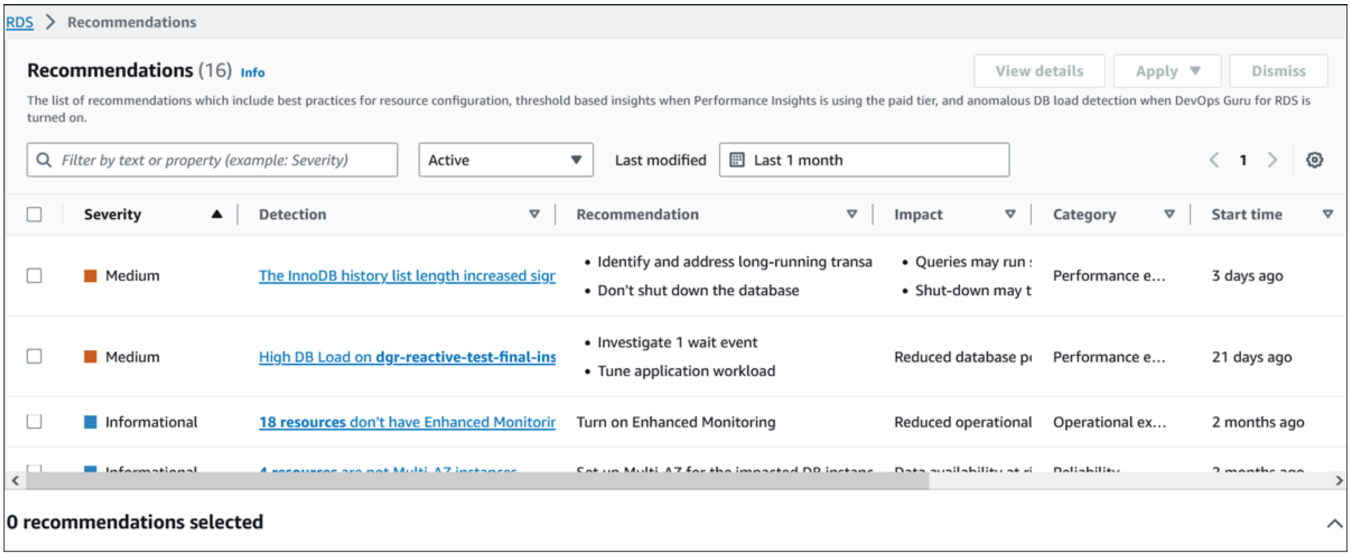
To view the Amazon Aurora recommendations

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, do any of the following:

   - Choose **Recommendations**. The number of active recommendations for your resources and the number of recommendations with the highest severity generated in the last month are available next to **Recommendations**. To find the number of active recommendations for each severity, choose the number that shows the highest severity.



By default, the **Recommendations** page displays a list of new recommendations in the last month. Amazon Aurora gives recommendations for all the resources in your account and sorts the recommendations by their severity.



RDS > Recommendations

Recommendations (16) Info

View details | Apply ▼ | Dismiss

The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using the paid tier, and anomalous DB load detection when DevOps Guru for RDS is turned on.

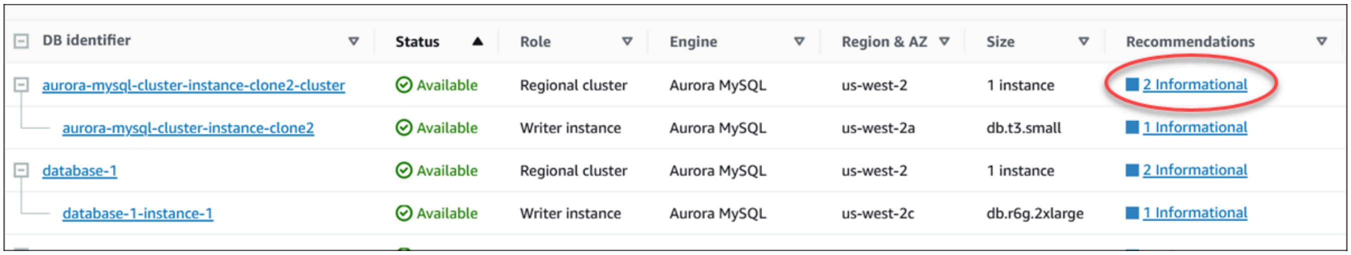
Filter by text or property (example: Severity) | Active ▼ | Last modified | Last 1 month | < 1 >

□	Severity ▲	Detection	Recommendation	Impact	Category	Start time
□	🟧 Medium	The InnoDB history list length increased sigr	• Identify and address long-running transa • Don't shut down the database	• Queries may run : • Shut-down may t	Performance e...	3 days ago
□	🟧 Medium	High DB Load on **dgr-reactive-test-final-ins**	• Investigate 1 wait event • Tune application workload	Reduced database pı	Performance e...	21 days ago
□	🟦 Informational	**18 resources** don't have Enhanced Monitorir	Turn on Enhanced Monitoring	Reduced operational	Operational ex...	2 months ago
□	🟦 Informational	4 resources are not Multi-AZ instances	Set up Multi-AZ for the impacted DB instanc	Data availability at ri	Reliability	2 months ago

0 recommendations selected

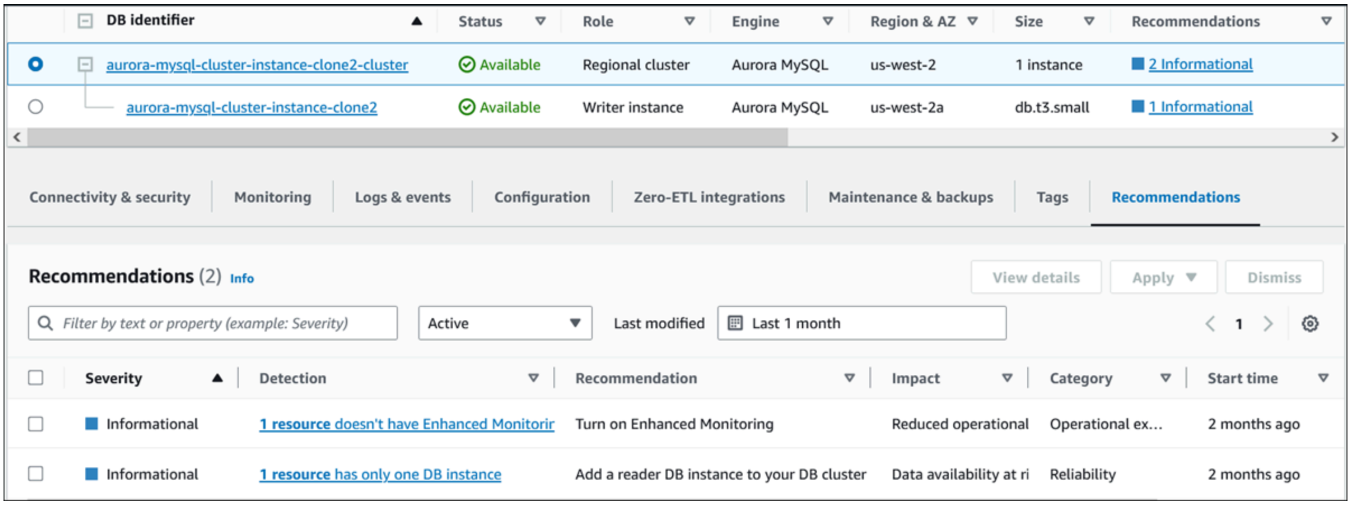
You can choose a recommendation to view a section at the bottom of the page which contains the affected resources and details of how the recommendation will be applied.

- In the **Databases** page, choose **Recommendations** for a resource.



⊟ DB identifier	Status ▲	Role	Engine	Region & AZ	Size	Recommendations
⊟ aurora-mysql-cluster-instance-clone2-cluster	✓ Available	Regional cluster	Aurora MySQL	us-west-2	1 instance	🟦 2 Informational
aurora-mysql-cluster-instance-clone2	✓ Available	Writer instance	Aurora MySQL	us-west-2a	db.t3.small	🟦 1 Informational
⊟ database-1	✓ Available	Regional cluster	Aurora MySQL	us-west-2	1 instance	🟦 2 Informational
database-1-instance-1	✓ Available	Writer instance	Aurora MySQL	us-west-2c	db.r6g.2xlarge	🟦 1 Informational

The **Recommendations** tab displays the recommendations and its details for the selected resource.



	⊟ DB identifier ▲	Status	Role	Engine	Region & AZ	Size	Recommendations
●	⊟ aurora-mysql-cluster-instance-clone2-cluster	✓ Available	Regional cluster	Aurora MySQL	us-west-2	1 instance	🟦 2 Informational
○	aurora-mysql-cluster-instance-clone2	✓ Available	Writer instance	Aurora MySQL	us-west-2a	db.t3.small	🟦 1 Informational

Connectivity & security | Monitoring | Logs & events | Configuration | Zero-ETL integrations | Maintenance & backups | Tags | Recommendations

Recommendations (2) Info

View details | Apply ▼ | Dismiss

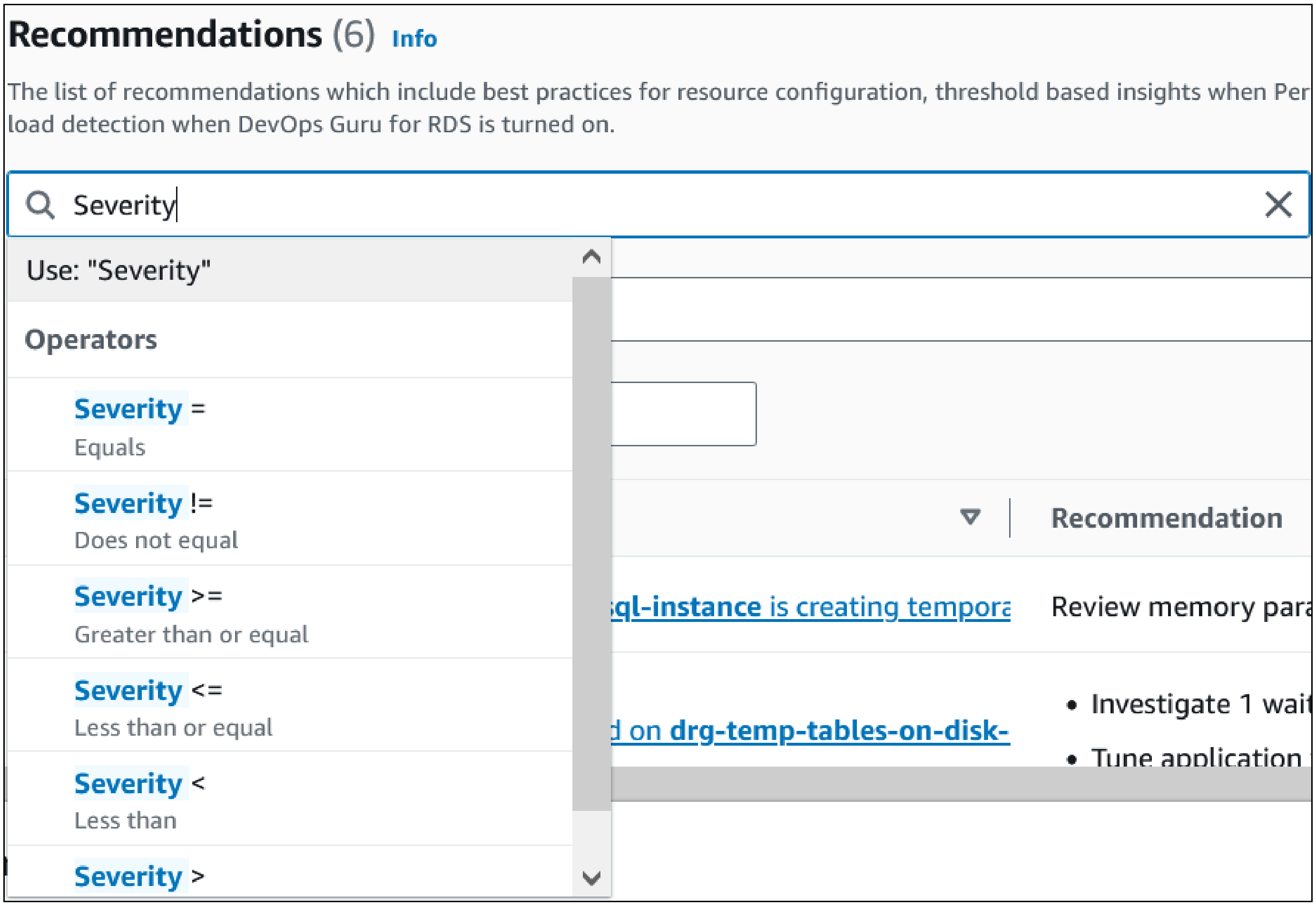
Filter by text or property (example: Severity) | Active ▼ | Last modified | Last 1 month | < 1 >

□	Severity ▲	Detection	Recommendation	Impact	Category	Start time
□	🟦 Informational	**1 resource** doesn't have Enhanced Monitorir	Turn on Enhanced Monitoring	Reduced operational	Operational ex...	2 months ago
□	🟦 Informational	**1 resource** has only one DB instance	Add a reader DB instance to your DB cluster	Data availability at ri	Reliability	2 months ago

The following details are available for the recommendations:

- **Severity –** The implication level of the issue. The severity levels are **High**, **Medium**, **Low**, and **Informational**.
- **Detection –** The number of affected resources and a short description of the issue. Choose this link to view the recommendation and the analysis details.
- **Recommendation –** A short description of the recommended action to apply.
- **Impact –** A short description of the possible impact when the recommendation isn't applied.
- **Category –** The type of recommendation. The categories are **Performance efficiency**, **Security**, **Reliability**, **Cost optimization**, **Operational excellence**, and **Sustainability**.
- **Status –** The current status of the recommendation. The possible statuses are **All**, **Active**, **Dismissed**, **Resolved**, and **Pending**.
- **Start time –** The time when the issue began. For example, **18 hours ago**.
- **Last modified –** The time when the recommendation was last updated by the system because of a change in the **Severity**, or the time you responded to the recommendation. For example, **10 hours ago**.
- **End time –** The time when the issue ended. The time won't display for any continuing issues.
- **Resource identifier –** The name of one or more resources.

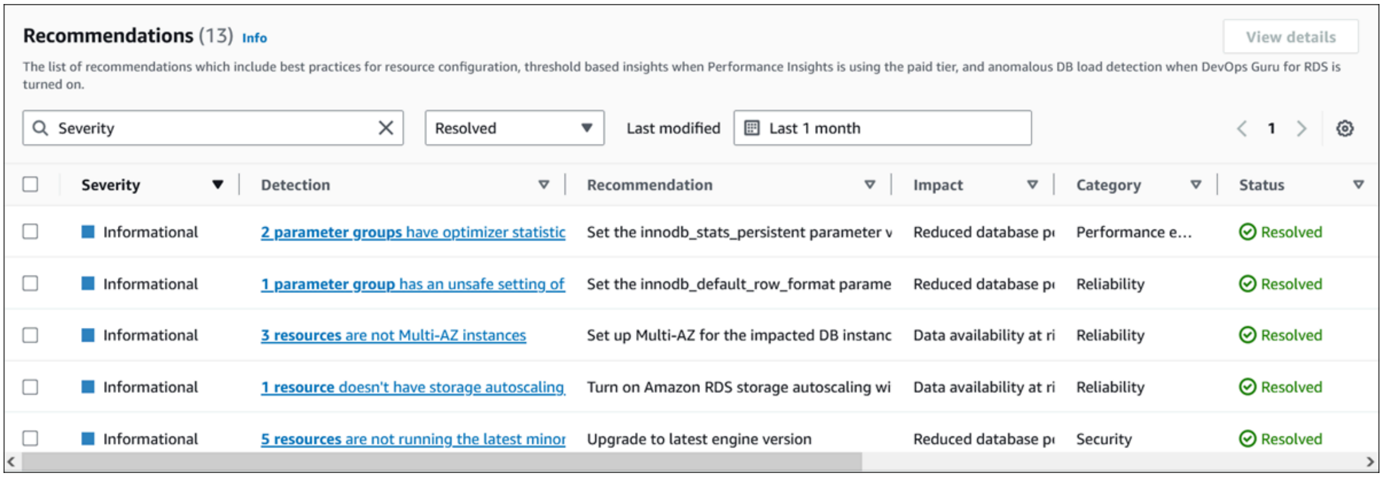
3. (Optional) Choose **Severity** or **Category** operators in the field to filter the list of recommendations.



The recommendations for the selected operation appear.

4. (Optional) Choose any of the following recommendation status:

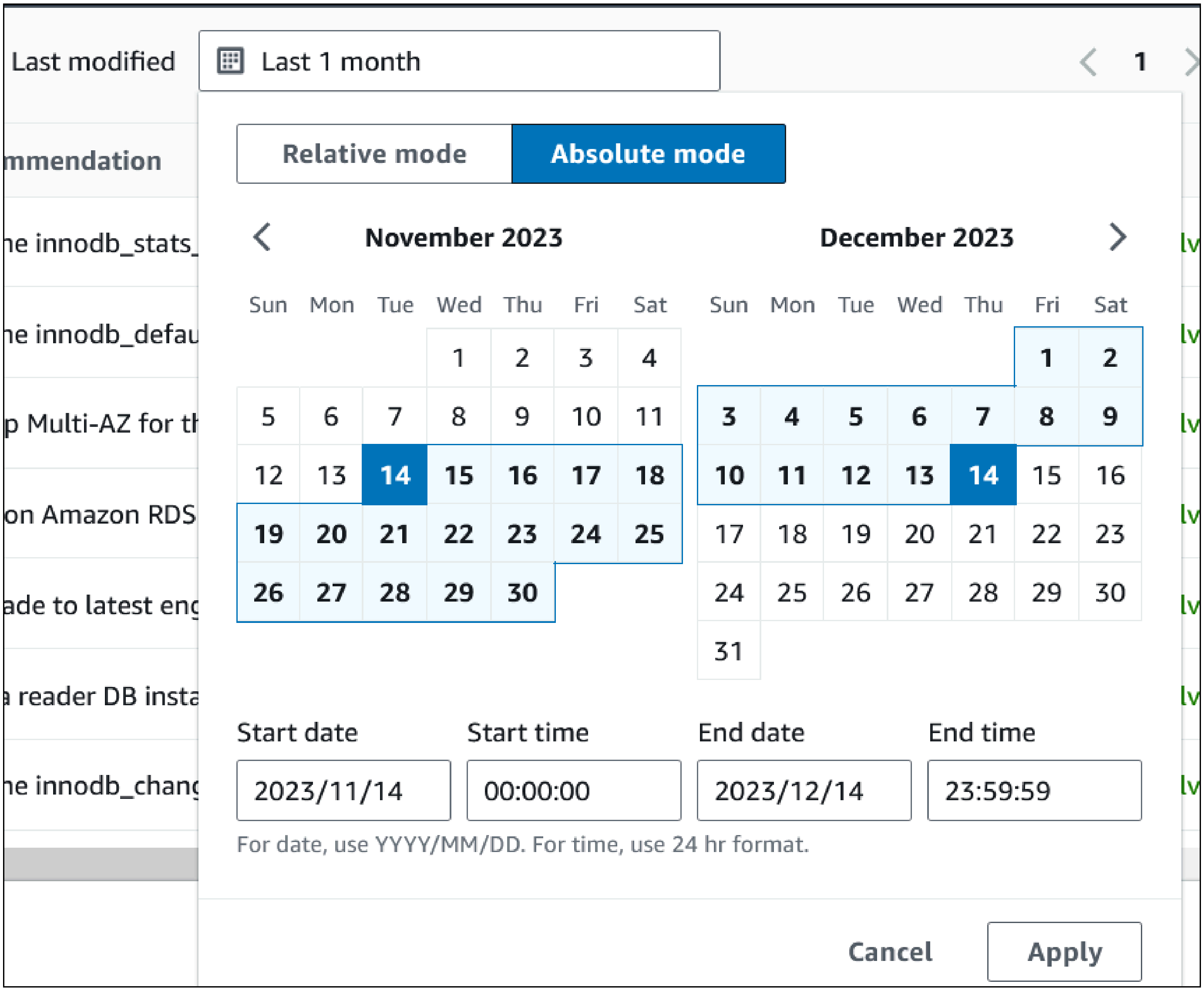
- **Active** (default) – Shows the current recommendations that you can apply, schedule it for the next maintenance window, or dismiss.
- **All** – Shows all the recommendations with the current status.
- **Dismissed** – Shows the dismissed recommendations.
- **Resolved** – Shows the recommendations that are resolved.
- **Pending** – Shows the recommendations whose recommended actions are in progress or scheduled for the next maintenance window.



The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using the paid tier, and anomalous DB load detection when DevOps Guru for RDS is turned on.

□	Severity	Detection	Recommendation	Impact	Category	Status
□	Informational	**2 parameter groups** have optimizer statistic	Set the innodb_stats_persistent parameter v	Reduced database pe	Performance e…	✓ Resolved
□	Informational	**1 parameter group** has an unsafe setting of	Set the innodb_default_row_format parame	Reduced database pe	Reliability	✓ Resolved
□	Informational	**3 resources** are not Multi-AZ instances	Set up Multi-AZ for the impacted DB instanc	Data availability at ri	Reliability	✓ Resolved
□	Informational	**1 resource** doesn't have storage autoscaling	Turn on Amazon RDS storage autoscaling wi	Data availability at ri	Reliability	✓ Resolved
□	Informational	**5 resources** are not running the latest minor	Upgrade to latest engine version	Reduced database pe	Security	✓ Resolved

5. (Optional) Choose **Relative mode** or **Absolute mode** in **Last modified** to modify the time period. The **Recommendations** page displays the recommendations generated in the time period. The default time period is the last month. In the **Absolute mode**, you can choose the time period, or enter the time in **Start date** and **End date** fields.



Last modified: Last 1 month

November 2023

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

December 2023

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

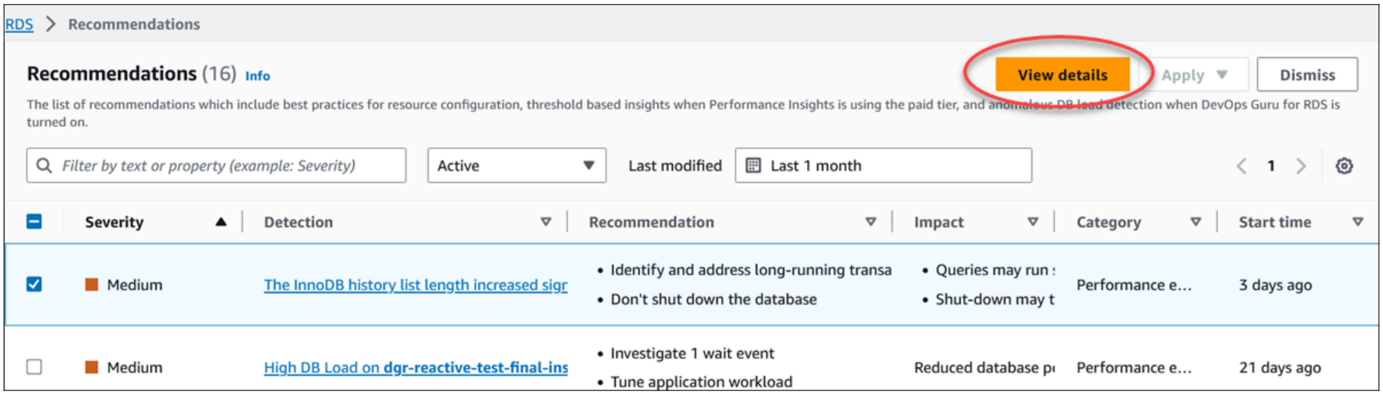
Start date	Start time	End date	End time
2023/11/14	00:00:00	2023/12/14	23:59:59

For date, use YYYY/MM/DD. For time, use 24 hr format.

The recommendations for the set time period display.

Note that you can see all recommendations for resources in your account by setting the range to **All**.

6. (Optional) Choose **Preferences** in the right to customize the details to display. You can choose a page size, wrap the lines of the text, and allow or hide the columns.
7. (Optional) Choose a recommendation and then choose **View details**.



RDS > Recommendations

Recommendations (16) Info

The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using the paid tier, and anomalous DB load detection when DevOps Guru for RDS is turned on.

Filter by text or property (example: Severity)

Active ▼ Last modified Last 1 month

☐	Severity ▲	Detection ▽	Recommendation ▽	Impact ▽	Category ▽	Start time ▽
☑	🟧 Medium	The InnoDB history list length increased sigr	• Identify and address long-running transa • Don't shut down the database	• Queries may run : • Shut-down may t	Performance e...	3 days ago
☐	🟧 Medium	High DB Load on dgr-reactive-test-final-ins	• Investigate 1 wait event • Tune application workload	Reduced database pı	Performance e...	21 days ago

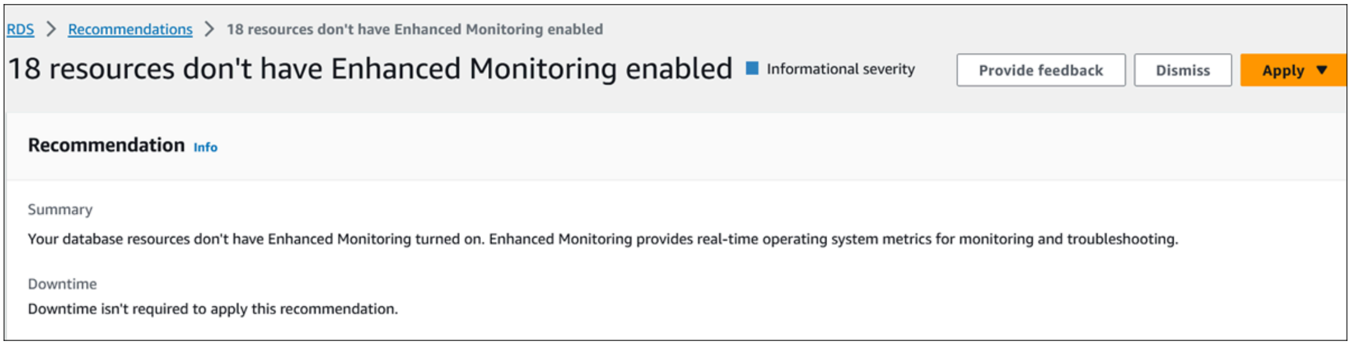
The recommendation details page appears. The title provides the total count of the resources with the issue detected and the severity.

For information about the components on the details page for an anomaly based reactive recommendation, see Viewing reactive anomalies in the *Amazon DevOps Guru User Guide*.

For information about the components on the details page for a threshold based proactive recommendation, see Viewing Performance Insights proactive recommendations.

The other automated recommendations display the following components on the recommendation details page:

- **Recommendation –** A summary of the recommendation and whether downtime is required to apply the recommendation.



RDS > Recommendations > 18 resources don't have Enhanced Monitoring enabled

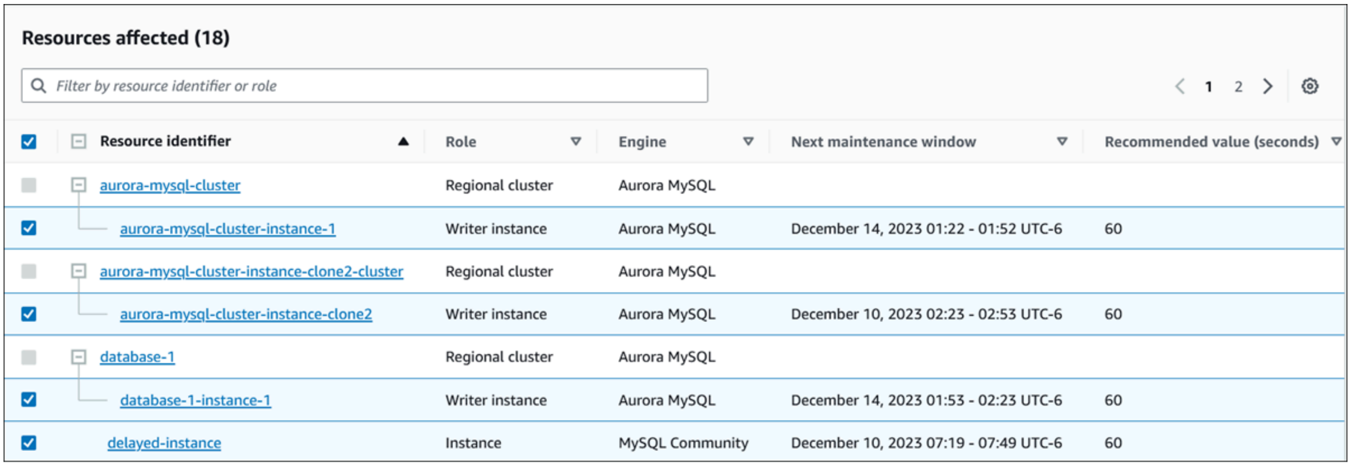
18 resources don't have Enhanced Monitoring enabled 🟦 Informational severity

Recommendation Info

Summary
Your database resources don't have Enhanced Monitoring turned on. Enhanced Monitoring provides real-time operating system metrics for monitoring and troubleshooting.

Downtime
Downtime isn't required to apply this recommendation.

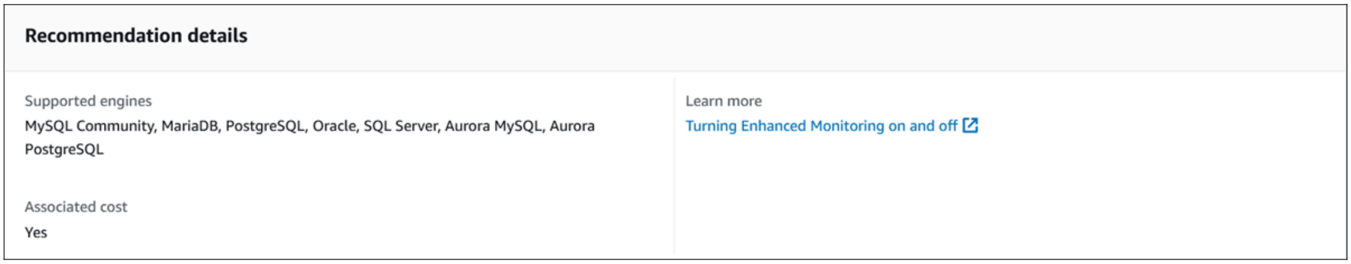
- **Resources affected –** Details of the affected resources.



Resources affected (18)

☑	Resource identifier ▲	Role	Engine	Next maintenance window	Recommended value (seconds)
□	aurora-mysql-cluster	Regional cluster	Aurora MySQL		
☑	aurora-mysql-cluster-instance-1	Writer instance	Aurora MySQL	December 14, 2023 01:22 - 01:52 UTC-6	60
□	aurora-mysql-cluster-instance-clone2-cluster	Regional cluster	Aurora MySQL		
☑	aurora-mysql-cluster-instance-clone2	Writer instance	Aurora MySQL	December 10, 2023 02:23 - 02:53 UTC-6	60
□	database-1	Regional cluster	Aurora MySQL		
☑	database-1-instance-1	Writer instance	Aurora MySQL	December 14, 2023 01:53 - 02:23 UTC-6	60
☑	delayed-instance	Instance	MySQL Community	December 10, 2023 07:19 - 07:49 UTC-6	60

- **Recommendation details –** Supported engine information, any required associated cost to apply the recommendation, and documentation link to learn more.



Recommendation details

Supported engines
MySQL Community, MariaDB, PostgreSQL, Oracle, SQL Server, Aurora MySQL, Aurora PostgreSQL

Associated cost
Yes

Learn more
Turning Enhanced Monitoring on and off

CLI

To view Amazon RDS recommendations of the DB instances or DB clusters, use the following command in AWS CLI.

```
aws rds describe-db-recommendations
```

RDS API

To view Amazon RDS recommendations using the Amazon RDS API, use the DescribeDBRecommendations operation.

Responding to Amazon Aurora recommendations

From the list of Aurora recommendations, you can:

- Apply a configuration based recommendation immediately or defer until the next maintenance window.
- Dismiss one or more recommendations.

- Move one or more dismissed recommendations to active recommendations.

Applying an Amazon Aurora recommendation

Using the Amazon RDS console, select a configuration based recommendation or an affected resource in the details page, and apply the recommendation immediately or schedule it for the next maintenance window. The resource might need to restart for the change to take effect. For a few DB parameter group recommendations, you might need to restart the resources.

The threshold based proactive or anomaly based reactive recommendations won't have the apply option and might need additional review.

Console

To apply a configuration based recommendation

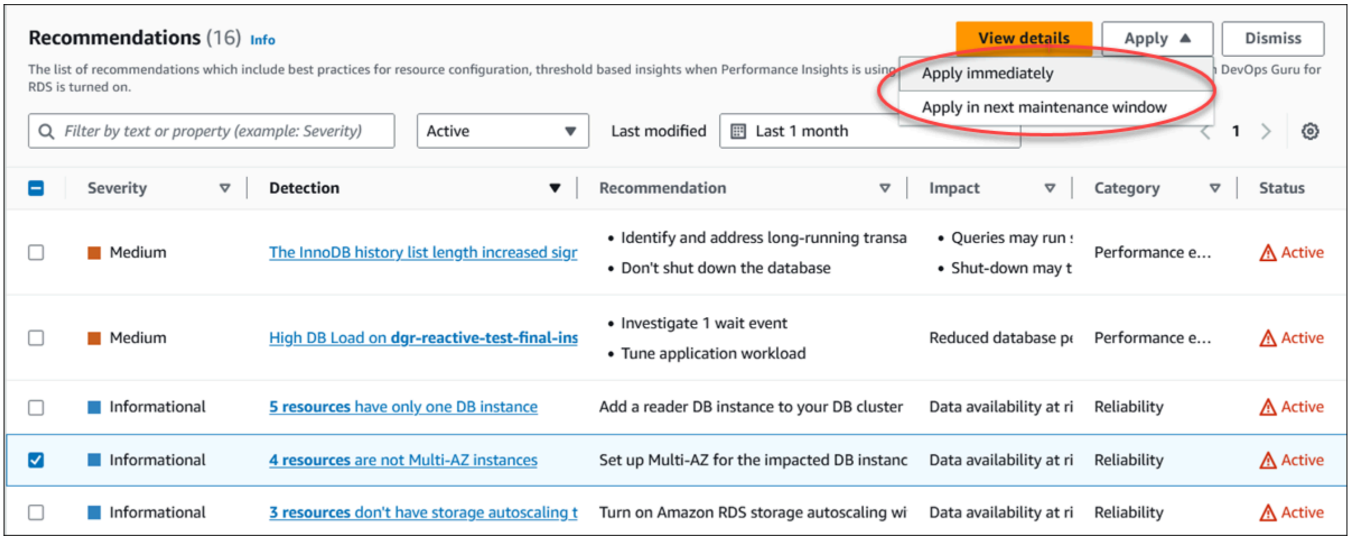
1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, perform any of the following:
   - Choose **Recommendations**.

     The **Recommendations** page appears with the list of all recommendations.
   - Choose **Databases** and then choose **Recommendations** for a resource in the databases page.

     The details appear in the **Recommendations** tab for the selected recommendation.
   - Choose **Detection** for an active recommendation in the **Recommendations** page or the **Recommendations** tab in the **Databases** page.

     The recommendation details page appears.
3. Choose a recommendation, or one or more affected resources in the recommendation details page, and do any of the following:
   - Choose **Apply** and then choose **Apply immediately** to apply the recommendation immediately.
   - Choose **Apply** and then choose **Apply in next maintenance window** to schedule in the next maintenance window.

The selected recommendation status is updated to pending until the next maintainance window.


View details
Apply ▲
Dismiss
Apply immediately
Apply in next maintenance window
Filter by text or property (example: Severity)
Active
Last modified
Last 1 month
1

Recommendations (16) Info

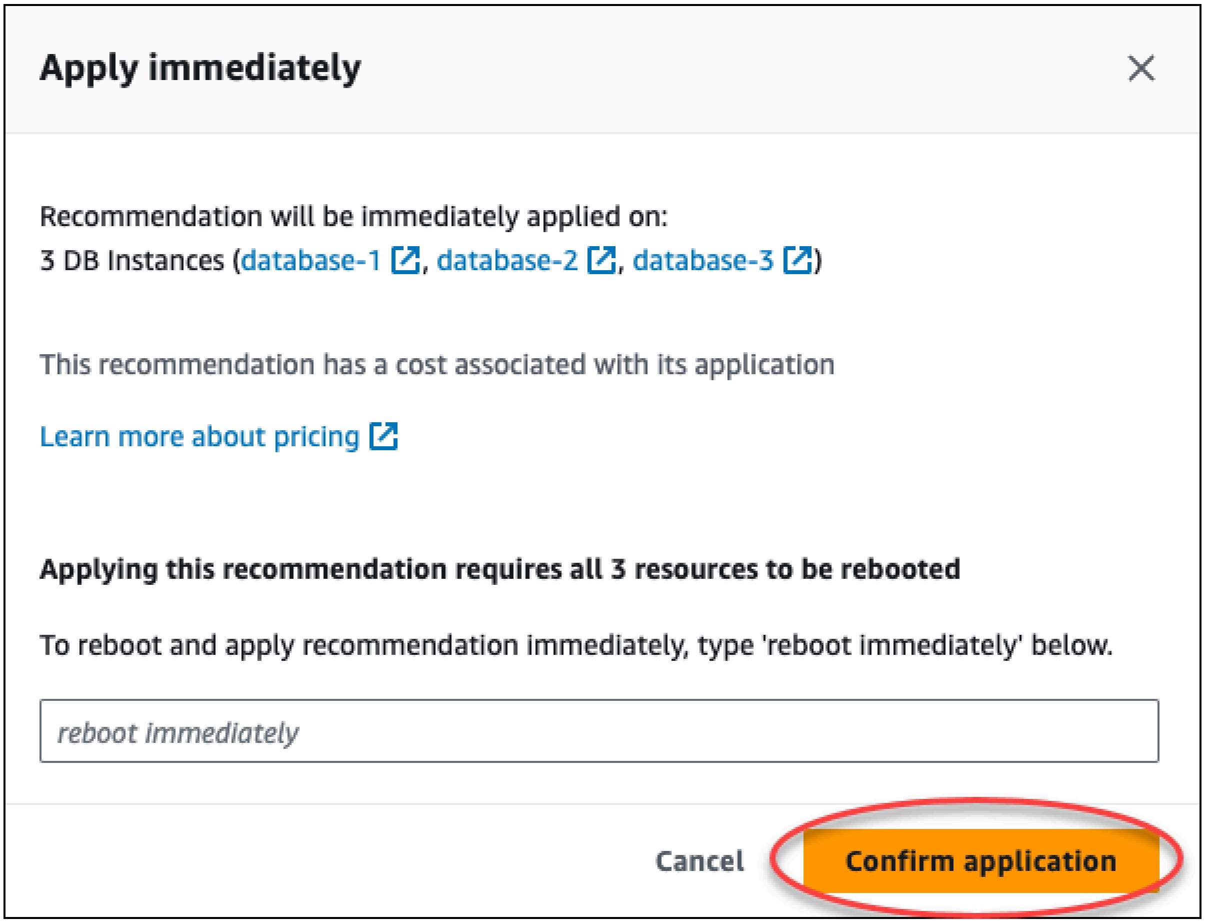
The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using DevOps Guru for RDS is turned on.

☐	Severity	Detection	Recommendation	Impact	Category	Status
☐	■ Medium	The InnoDB history list length increased sigr	• Identify and address long-running transa • Don't shut down the database	• Queries may run : • Shut-down may t	Performance e...	⚠ Active
☐	■ Medium	High DB Load on **dgr-reactive-test-final-ins**	• Investigate 1 wait event • Tune application workload	Reduced database pe	Performance e...	⚠ Active
☐	■ Informational	**5 resources** have only one DB instance	Add a reader DB instance to your DB cluster	Data availability at ri	Reliability	⚠ Active
☑	■ Informational	**4 resources** are not Multi-AZ instances	Set up Multi-AZ for the impacted DB instanc	Data availability at ri	Reliability	⚠ Active
☐	■ Informational	**3 resources** don't have storage autoscaling t	Turn on Amazon RDS storage autoscaling wi	Data availability at ri	Reliability	⚠ Active

A confirmation window appears.

4. Choose **Confirm application** to apply the recommendation. This window confirms whether the resources need an automatic or manual restart for the changes to take effect.

   The following example shows the confirmation window to apply the recommendation immediately.



Apply immediately

Recommendation will be immediately applied on:
3 DB Instances (database-1, database-2, database-3)

This recommendation has a cost associated with its application

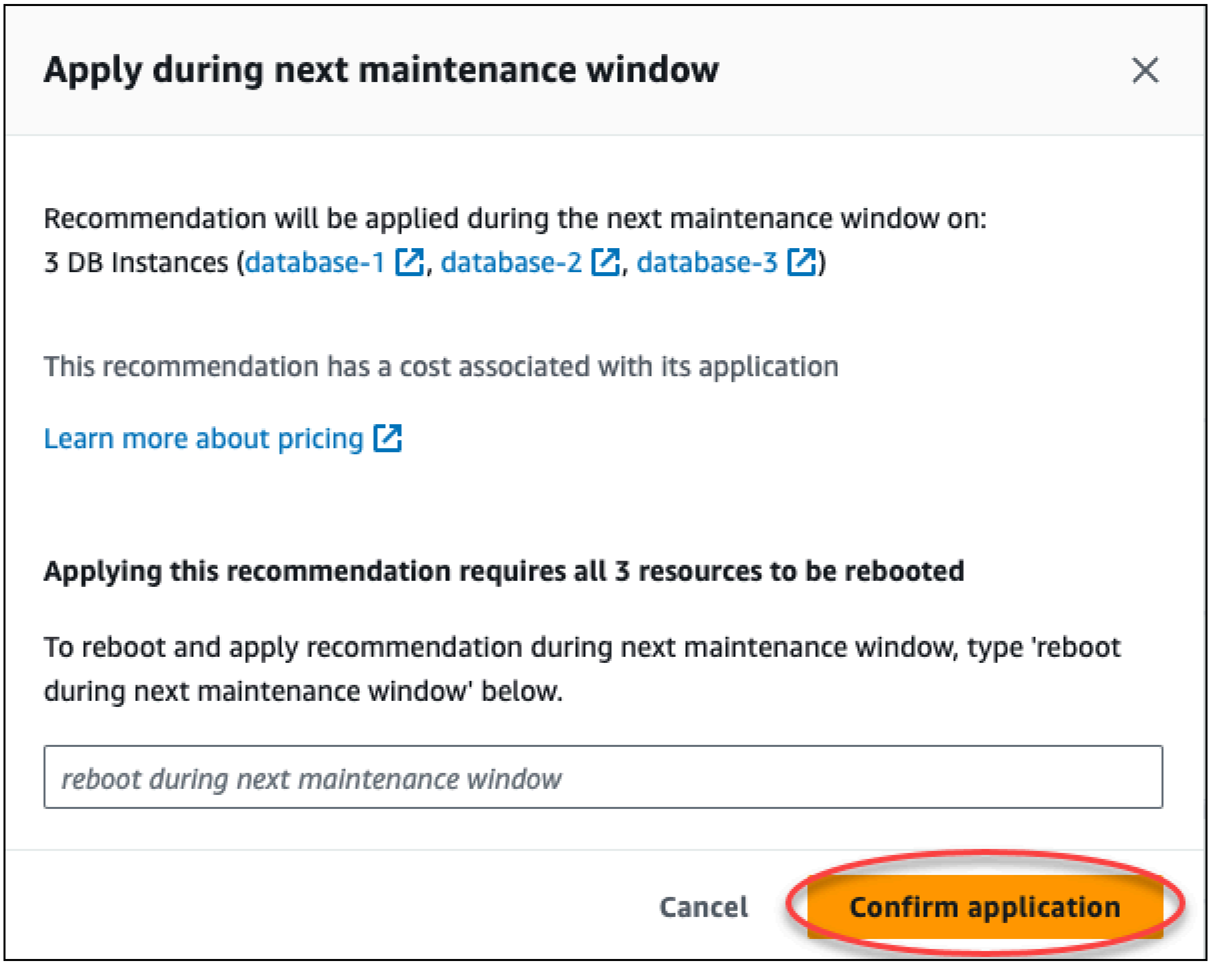
Learn more about pricing

Applying this recommendation requires all 3 resources to be rebooted

To reboot and apply recommendation immediately, type 'reboot immediately' below.

reboot immediately

The following example shows the confirmation window to schedule applying the recommendation in the next maintenance window.



Apply during next maintenance window

Recommendation will be applied during the next maintenance window on:
3 DB Instances (database-1, database-2, database-3)

This recommendation has a cost associated with its application

Learn more about pricing

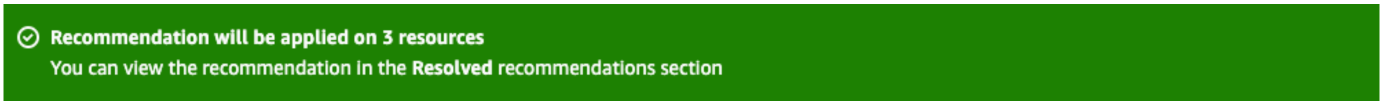
Applying this recommendation requires all 3 resources to be rebooted

To reboot and apply recommendation during next maintenance window, type 'reboot during next maintenance window' below.

reboot during next maintenance window

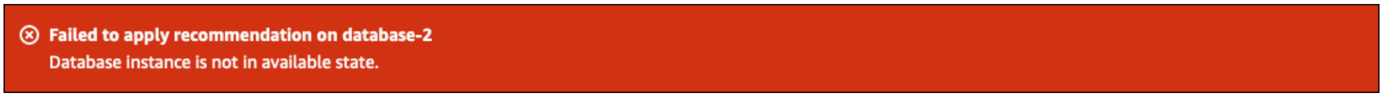
A banner displays a message when the recommendation applied is successful or has failed.

The following example shows the banner with the successful message.



Recommendation will be applied on 3 resources
You can view the recommendation in the **Resolved** recommendations section

The following example shows the banner with the failure message.



Failed to apply recommendation on database-2
Database instance is not in available state.

RDS API

To apply a configuration based Aurora recommendation using the Amazon RDS API

1. Use the DescribeDBRecommendations operation. The `RecommendedActions` in the output can have one or more recommended actions.

2. Use the RecommendedAction object for each recommended action from step 1. The output contains `Operation` and `Parameters`.

   The following example shows the output with one recommended action.

```
   "RecommendedActions": [
       {
           "ActionId": "0b19ed15-840f-463c-a200-b10af1b552e3",
           "Title": "Turn on auto backup", // localized
           "Description": "Turn on auto backup for my-mysql-instance-1", //
localized
           "Operation": "ModifyDbInstance",
           "Parameters": [
               {
                   "Key": "DbInstanceIdentifier",
                   "Value": "my-mysql-instance-1"
               },
               {
                   "Key": "BackupRetentionPeriod",
                   "Value": "7"
               }
           ],
           "ApplyModes": ["immediately", "next-maintenance-window"],
           "Status": "applied"
       },
       ... // several others
   ],
```

3. Use the `operation` for each recommended action from the output in step 2 and input the `Parameters` values.

4. After the operation in step 2 is successful, use the ModifyDBRecommendation operation to modify the recommendation status.

Dismissing the Amazon Aurora recommendations

You can dismiss one or more recommendations.

Console

To dismiss one or more recommendations

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, perform any of the following:

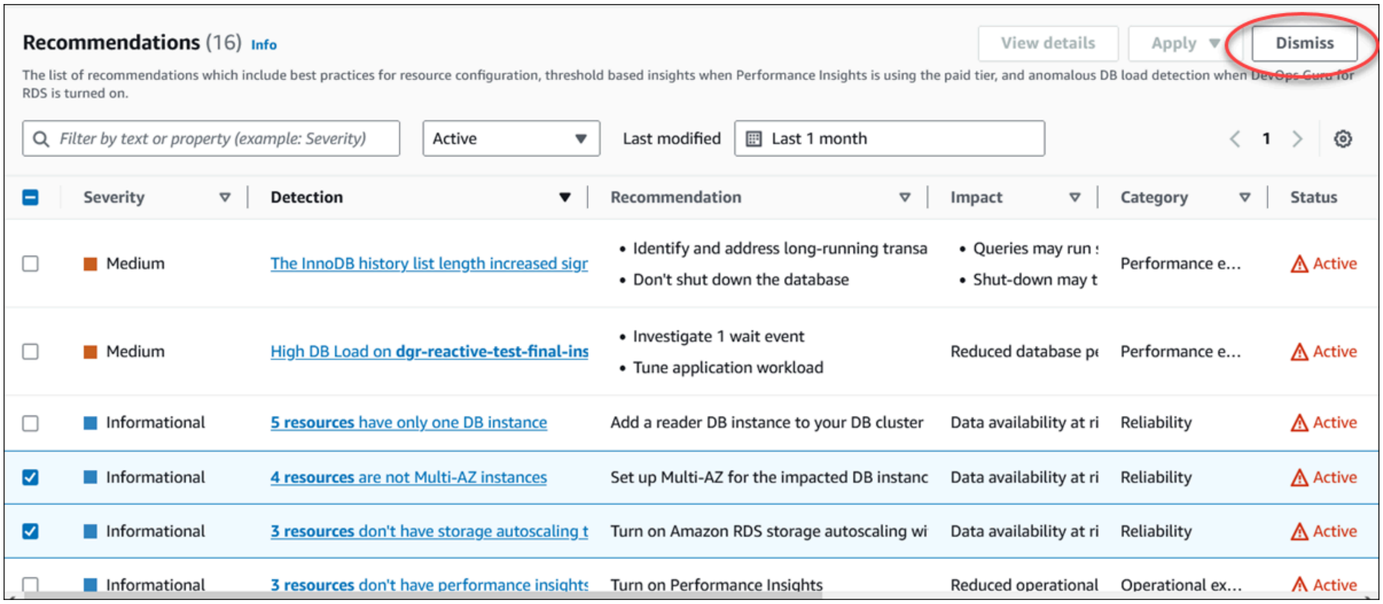
   - Choose **Recommendations**.

     The **Recommendations** page appears with the list of all recommendations.
   - Choose **Databases** and then choose **Recommendations** for a resource in the databases page.

     The details appear in the **Recommendations** tab for the selected recommendation.
   - Choose **Detection** for an active recommendation in the **Recommendations** page or the **Recommendations** tab in the **Databases** page.

     The recommendation details page displays the list of affected resources.
3. Choose one or more recommendation, or one or more affected resources in the recommendation details page, and then choose **Dismiss**.

   The following example shows the **Recommendations** page with multiple active recommendations selected to dismiss.



Recommendations (16) Info

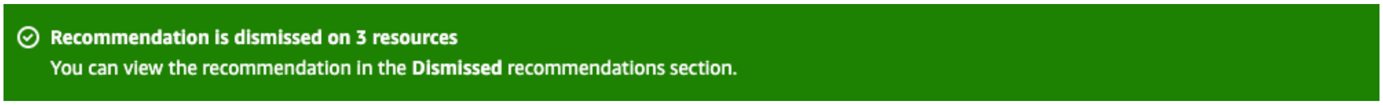
The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using the paid tier, and anomalous DB load detection when DevOps Guru for RDS is turned on.

Filter by text or property (example: Severity) | Active ▼ | Last modified | Last 1 month | < 1 >

▬	Severity	Detection	Recommendation	Impact	Category	Status
☐	🟧 Medium	The InnoDB history list length increased sigr	• Identify and address long-running transa • Don't shut down the database	• Queries may run : • Shut-down may t	Performance e...	⚠ Active
☐	🟧 Medium	High DB Load on **dgr-reactive-test-final-ins**	• Investigate 1 wait event • Tune application workload	Reduced database pe	Performance e...	⚠ Active
☐	🟦 Informational	**5 resources** have only one DB instance	Add a reader DB instance to your DB cluster	Data availability at ri	Reliability	⚠ Active
☑	🟦 Informational	**4 resources** are not Multi-AZ instances	Set up Multi-AZ for the impacted DB instanc	Data availability at ri	Reliability	⚠ Active
☑	🟦 Informational	**3 resources** don't have storage autoscaling t	Turn on Amazon RDS storage autoscaling wi	Data availability at ri	Reliability	⚠ Active
☐	🟦 Informational	**3 resources** don't have performance insights	Turn on Performance Insights	Reduced operational	Operational ex...	⚠ Active

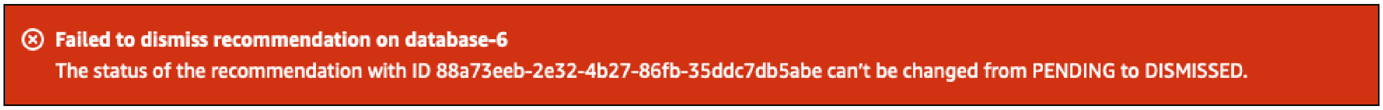
A banner displays a message when the selected one or more recommendations are dismissed.

The following example shows the banner with the successful message.



Recommendation is dismissed on 3 resources
You can view the recommendation in the **Dismissed** recommendations section.

The following example shows the banner with the failure message.



Failed to dismiss recommendation on database-6
The status of the recommendation with ID 88a73eeb-2e32-4b27-86fb-35ddc7db5abe can't be changed from PENDING to DISMISSED.

CLI

To dismiss an Aurora recommendation using the AWS CLI

1. Run the command `aws rds describe-db-recommendations --filters "Name=status,Values=active"`.

   The output provides a list of recommendations in `active` status.

2. Find the `recommendationId` for the recommendation that you want to dismiss from step 1.

3. Run the command `>aws rds modify-db-recommendation --status dismissed --recommendationId <ID>` with the `recommendationId` from step 2 to dismiss the recommendation.

RDS API

To dismiss an Aurora recommendation using the Amazon RDS API, use the ModifyDBRecommendation operation.

Modifying the dismissed Amazon Aurora recommendations to active recommendations

You can move one or more dismissed recommendations to active recommendations.

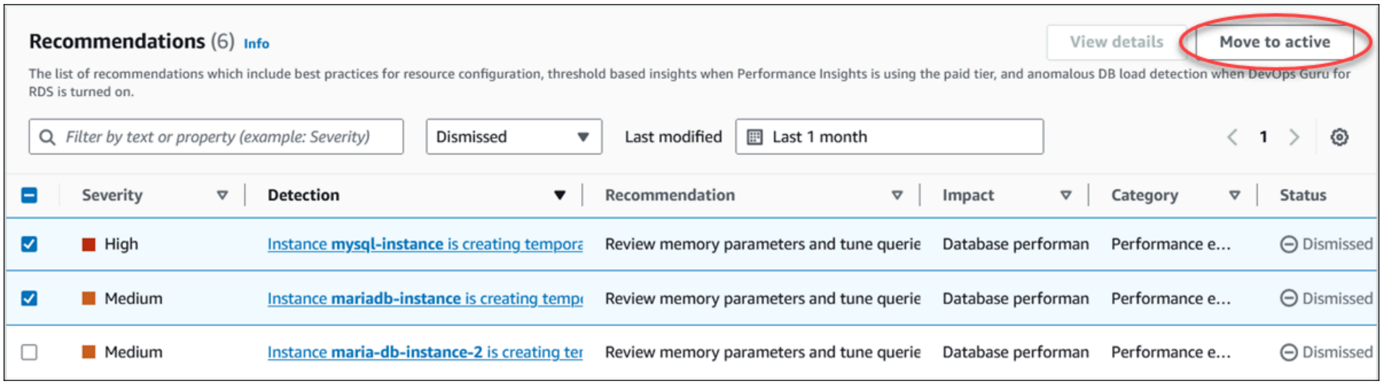
Console

To move one or more dismissed recommendations to active recommendations

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, perform any of the following:
   - Choose **Recommendations**.

     The **Recommendations** page displays a list of recommendations sorted by the severity for all the resources in your account.
   - Choose **Databases** and then choose **Recommendations** for a resource in the databases page.

     The **Recommendations** tab displays the recommendations and its details for the selected resource.
3. Choose one or more dismissed recommendations from the list and then choose **Move to active**.

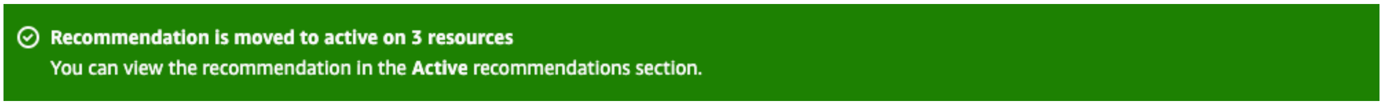


The list of recommendations which include best practices for resource configuration, threshold based insights when Performance Insights is using the paid tier, and anomalous DB load detection when DevOps Guru for RDS is turned on.

☐	Severity	Detection	Recommendation	Impact	Category	Status
☑	🟥 High	Instance **mysql-instance** is creating tempora	Review memory parameters and tune querie	Database performan	Performance e...	⊖ Dismissed
☑	🟧 Medium	Instance **mariadb-instance** is creating tempo	Review memory parameters and tune querie	Database performan	Performance e...	⊖ Dismissed
☐	🟧 Medium	Instance **maria-db-instance-2** is creating tei	Review memory parameters and tune querie	Database performan	Performance e...	⊖ Dismissed

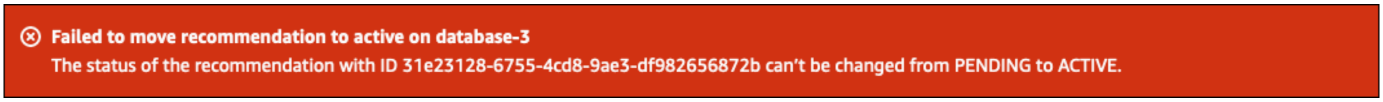
A banner displays a successful or failure message when the moving the selected recommendations from dismissed to active status.

The following example shows the banner with the successful message.



✓ **Recommendation is moved to active on 3 resources**
You can view the recommendation in the **Active** recommendations section.

The following example shows the banner with the failure message.



⊗ **Failed to move recommendation to active on database-3**
The status of the recommendation with ID 31e23128-6755-4cd8-9ae3-df982656872b can't be changed from PENDING to ACTIVE.

CLI

To change a dismissed Aurora recommendation to active recommendation using the AWS CLI

1. Run the command `aws rds describe-db-recommendations --filters "Name=status,Values=dismissed"`.

   The output provides a list of recommendations in `dismissed` status.
2. Find the `recommendationId` for the recommendation that you want to change the status from step 1.
3. Run the command `>aws rds modify-db-recommendation --status active --recommendationId <ID>` with the `recommendationId` from step 2 to change to active recommendation.

RDS API

To change a dismissed Aurora recommendation to active recommendation using the Amazon RDS API, use the ModifyDBRecommendation operation.

Viewing metrics in the Amazon RDS console

Amazon RDS integrates with Amazon CloudWatch to display a variety of Aurora DB cluster metrics in the RDS console. Some metrics are apply at the cluster level, whereas others apply at the instance level. For descriptions of the instance-level and cluster-level metrics, see Metrics reference for Amazon Aurora.

For your Aurora DB cluster, the following categories of metrics are monitored:

- **CloudWatch** – Shows the Amazon CloudWatch metrics for Aurora that you can access in the RDS console. You can also access these metrics in the CloudWatch console. Each metric includes a graph that shows the metric monitored over a specific time span. For a list of CloudWatch metrics, see Amazon CloudWatch metrics for Amazon Aurora.
- **Enhanced monitoring** – Shows a summary of operating-system metrics when your Aurora DB cluster has turned on Enhanced Monitoring. RDS delivers the metrics from Enhanced Monitoring to your Amazon CloudWatch Logs account. Each OS metric includes a graph showing the metric monitored over a specific time span. For an overview, see Monitoring OS metrics with Enhanced Monitoring. For a list of Enhanced Monitoring metrics, see OS metrics in Enhanced Monitoring.
- **OS Process list** – Shows details for each process running in your DB cluster.
- **Performance Insights** – Opens the Amazon RDS Performance Insights dashboard for a DB instance in your Aurora DB cluster. Performance Insights isn't supported at the cluster level. For an overview of Performance Insights, see Monitoring DB load with Performance Insights on Amazon Aurora. For a list of Performance Insights metrics, see Amazon CloudWatch metrics for Performance Insights.

Amazon RDS now provides a consolidated view of Performance Insights and CloudWatch metrics in the Performance Insights dashboard. Performance Insights must be turned on for your DB cluster to use this view. You can choose the new monitoring view in the **Monitoring** tab or **Performance Insights** in the navigation pane. To view the instructions for choosing this view, see Viewing combined metrics in the Amazon RDS console.

If you want to continue with the legacy monitoring view, continue with this procedure.

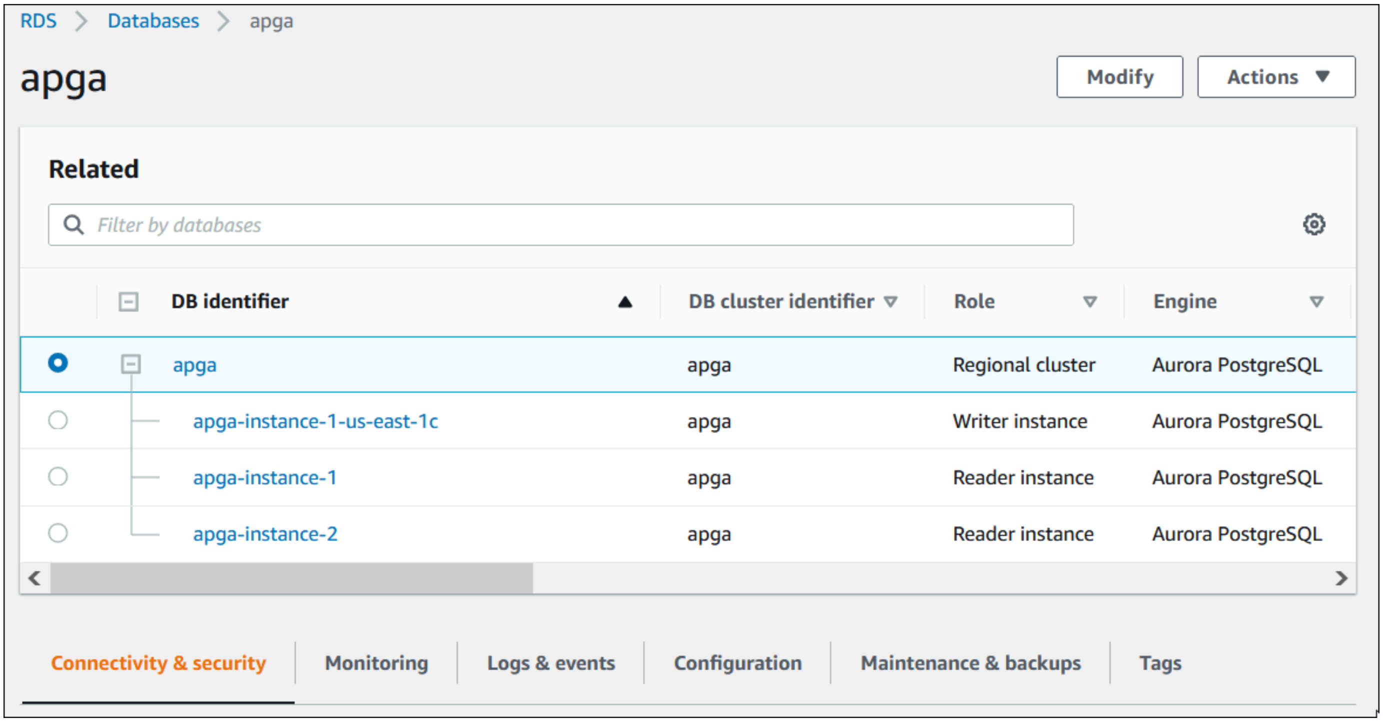
Note

The legacy monitoring view will be discontinued on December 15, 2023.

To view metrics for your DB cluster in the legacy monitoring view:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the Aurora DB cluster that you want to monitor.

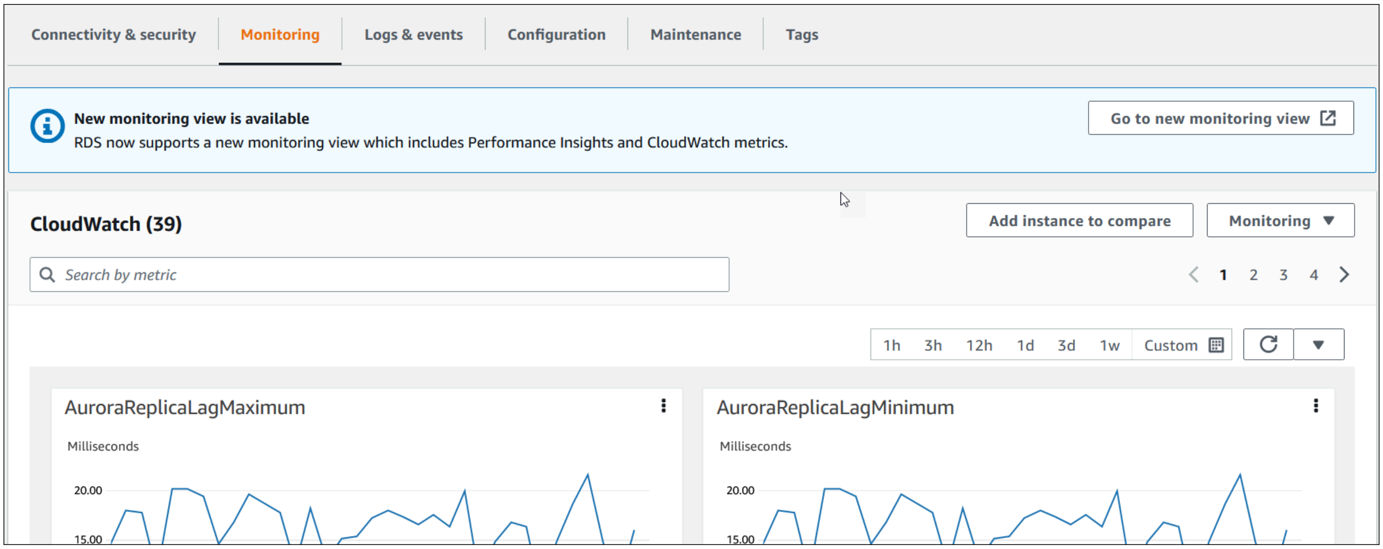
   The database page appears. The following example shows an Amazon Aurora PostgreSQL database named apga.



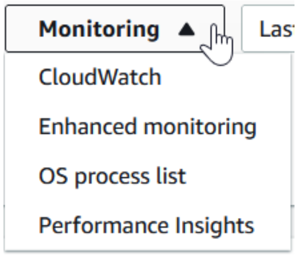
		DB identifier ▲	DB cluster identifier ▽	Role ▽	Engine ▽
●	⊟	apga	apga	Regional cluster	Aurora PostgreSQL
○		apga-instance-1-us-east-1c	apga	Writer instance	Aurora PostgreSQL
○		apga-instance-1	apga	Reader instance	Aurora PostgreSQL
○		apga-instance-2	apga	Reader instance	Aurora PostgreSQL

4. Scroll down and choose **Monitoring**.

   The monitoring section appears. By default, CloudWatch metrics are shown. For descriptions of these metrics, see [Amazon CloudWatch metrics for Amazon Aurora](#).

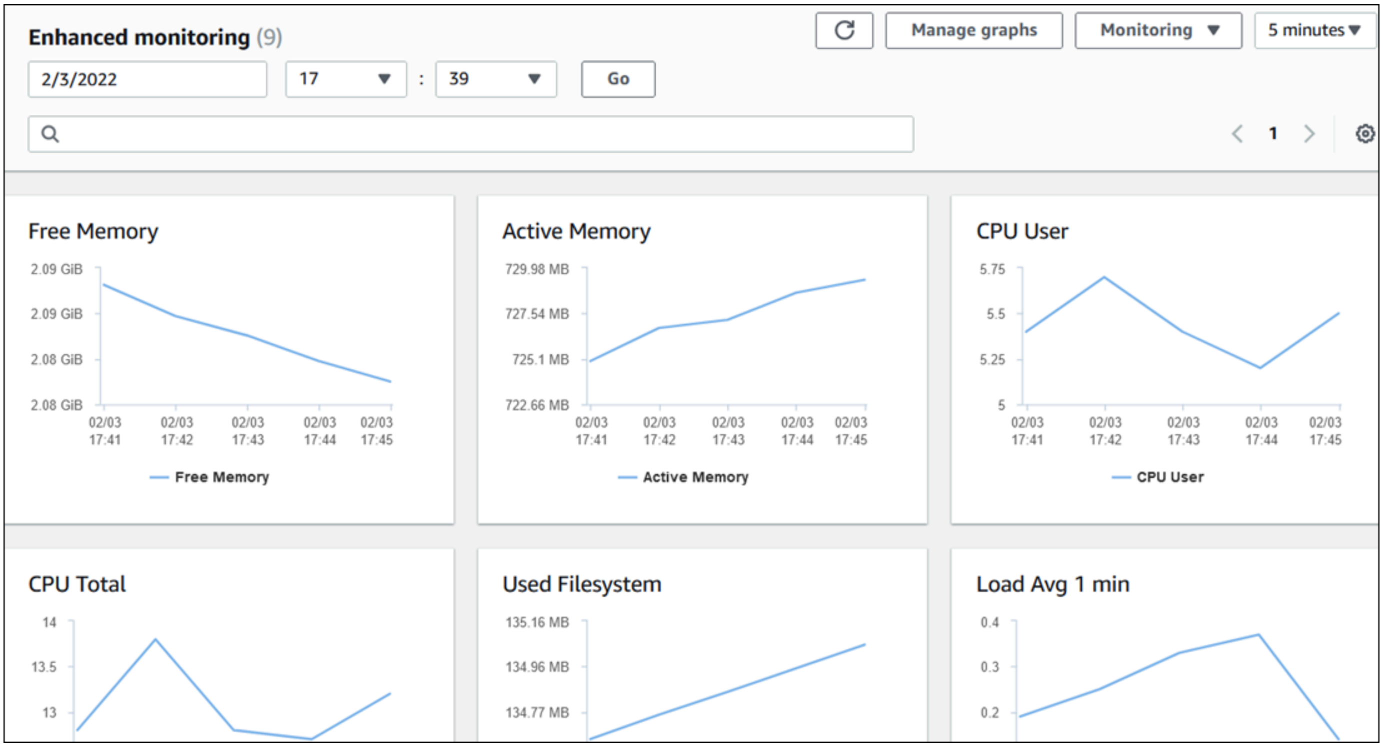


5. Choose **Monitoring** to see the metric categories.



6. Choose the category of metrics that you want to see.

   The following example shows Enhanced Monitoring metrics. For descriptions of these metrics, see OS metrics in Enhanced Monitoring.



Tip

To choose the time range of the metrics represented by the graphs, you can use the time range list.

To bring up a more detailed view, you can choose any graph. You can also apply metric-specific filters to the data.

Viewing combined metrics in the Amazon RDS console

Amazon RDS now provides a consolidated view of Performance Insights and CloudWatch metrics for your DB instance in the Performance Insights dashboard. You can use the preconfigured dashboard or create a custom dashboard. The preconfigured dashboard provides the most commonly used metrics to help diagnose performance issues for a database engine. Alternatively, you can create a custom dashboard with the metrics for a database engine that meet your analysis requirements. Then, use this dashboard for all the DB instances of that database engine type in your AWS account.

You can choose the new monitoring view in the **Monitoring** tab or **Performance Insights** in the navigation pane. When you navigate to the Performance Insights page, you see the options to choose between the new monitoring view and legacy view. The option you choose is saved as the default view.

Performance Insights must be turned on for your DB cluster to view the combined metrics in the Performance Insights dashboard. For more information about turning on Performance Insights, see Turning Performance Insights on and off for Aurora.

> **Note**
>
> We recommend that you choose the new monitoring view. You can continue to use the legacy monitoring view until it is discontinued on December 15, 2023.

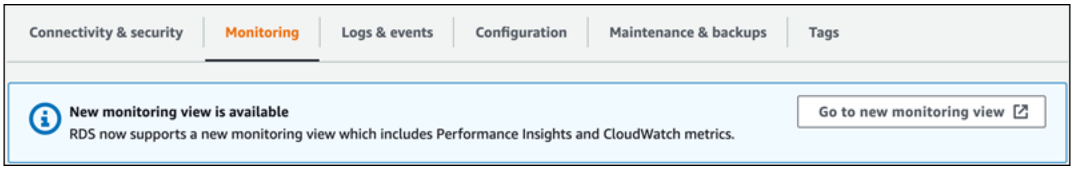
Choosing the new monitoring view in the Monitoring tab

To choose the new monitoring view in the Monitoring tab:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Databases**.
3. Choose the Aurora DB cluster that you want to monitor.

   The database page appears.
4. Scroll down and choose the **Monitoring** tab.

A banner appears with the option to choose the new monitoring view. The following example shows the banner to choose the new monitoring view.

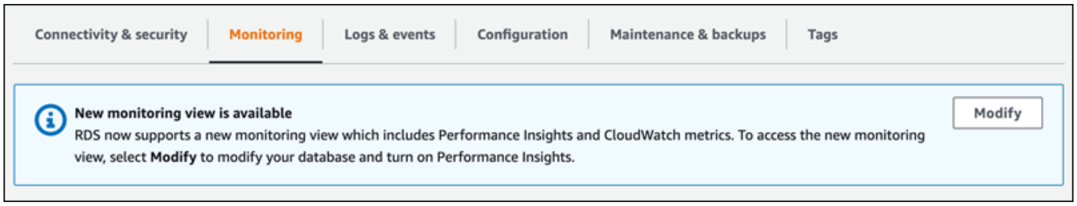


New monitoring view is available

RDS now supports a new monitoring view which includes Performance Insights and CloudWatch metrics.

5. Choose **Go to new monitoring view** to open the Performance Insights dashboard with Performance Insights and CloudWatch metrics for your DB cluster.
6. (Optional) If Performance Insights is turned off for your DB instance, a banner appears with the option to modify your DB instance and turn on Performance Insights.

   The following example shows the banner to modify the DB instance in the **Monitoring** tab .

   New monitoring view is available

   RDS now supports a new monitoring view which includes Performance Insights and CloudWatch metrics. To access the new monitoring view, select **Modify** to modify your database and turn on Performance Insights.

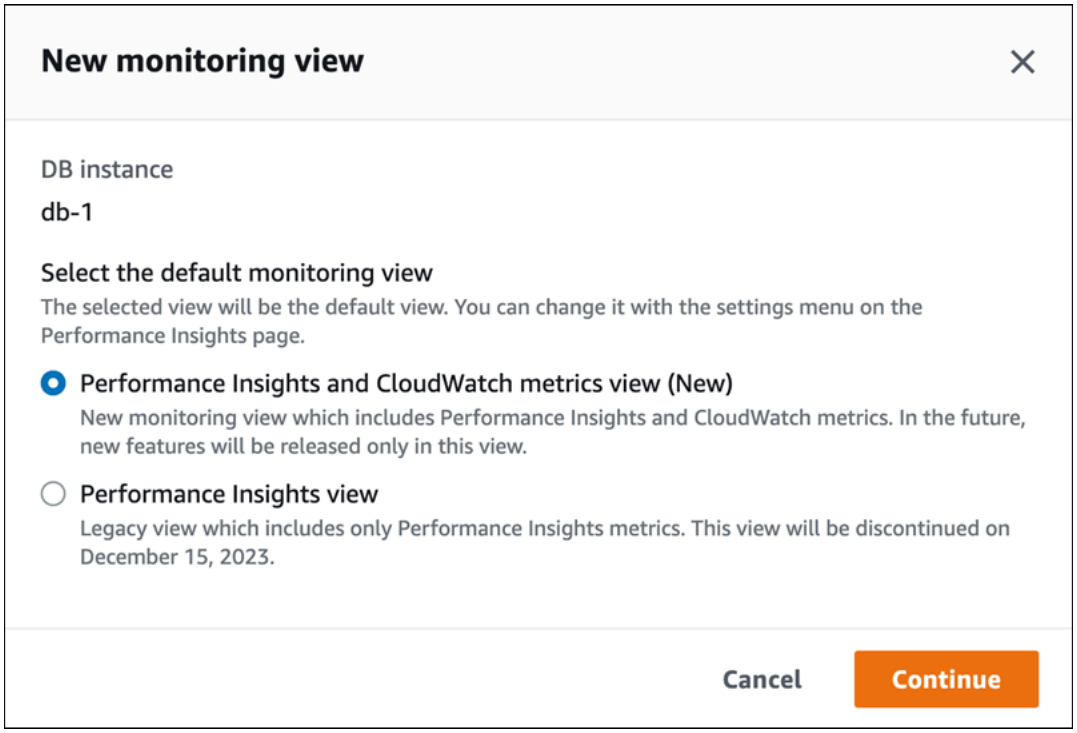
   Choose **Modify** to modify your DB instance and turn on Performance Insights. For more information about turning on Performance Insights, see Turning Performance Insights on and off for Aurora

Choosing the new monitoring view with Performance Insights in the navigation pane

To choose the new monitoring view with Performance Insights in the navigation pane:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance to open a window that has the monitoring view options.

   The following example shows the window with the monitoring view options.



New monitoring view

DB instance

db-1

Select the default monitoring view

The selected view will be the default view. You can change it with the settings menu on the Performance Insights page.

● Performance Insights and CloudWatch metrics view (New)

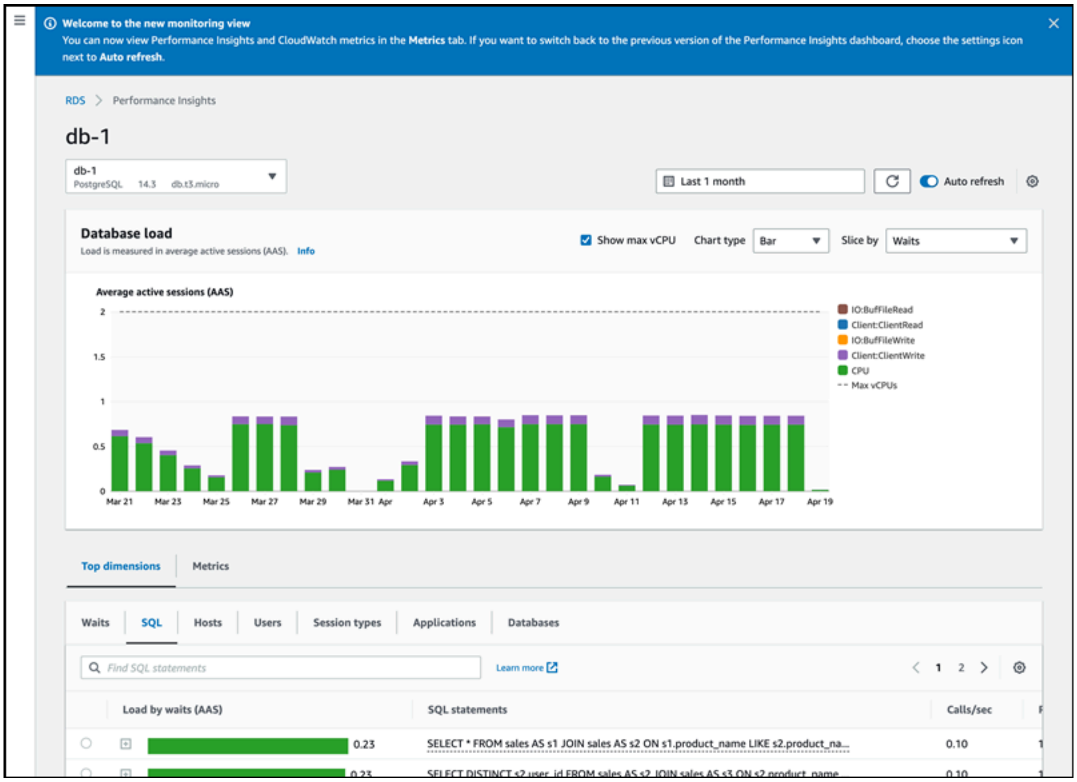
New monitoring view which includes Performance Insights and CloudWatch metrics. In the future, new features will be released only in this view.

○ Performance Insights view

Legacy view which includes only Performance Insights metrics. This view will be discontinued on December 15, 2023.

4. Choose the **Performance Insights and CloudWatch metrics view (New)** option, and then choose **Continue**.

   You can now view the Performance Insights dashboard that shows both Performance Insights and CloudWatch metrics for your DB instance. The following example shows the Performance Insights and CloudWatch metrics in the dashboard.



Welcome to the new monitoring view

You can now view Performance Insights and CloudWatch metrics in the **Metrics** tab. If you want to switch back to the previous version of the Performance Insights dashboard, choose the settings icon next to **Auto refresh**.

db-1

db-1 PostgreSQL 14.3 db.t3.micro

Last 1 month

Auto refresh

Database load

Load is measured in average active sessions (AAS). Info

☑ Show max vCPU

Chart type: Bar

Slice by: Waits

	Load by waits (AAS)	SQL statements	Calls/sec
○	0.23	SELECT * FROM sales AS s1 JOIN sales AS s2 ON s1.product_name LIKE s2.product_na...	0.10
○	0.23	SELECT DISTINCT s2.user_id FROM sales AS s2 JOIN sales AS s3 ON s2.product_name...	0.10

Choosing the legacy view with Performance Insights in the navigation pane

You can choose the legacy monitoring view to view only the Performance Insights metrics for your DB instance.

Note

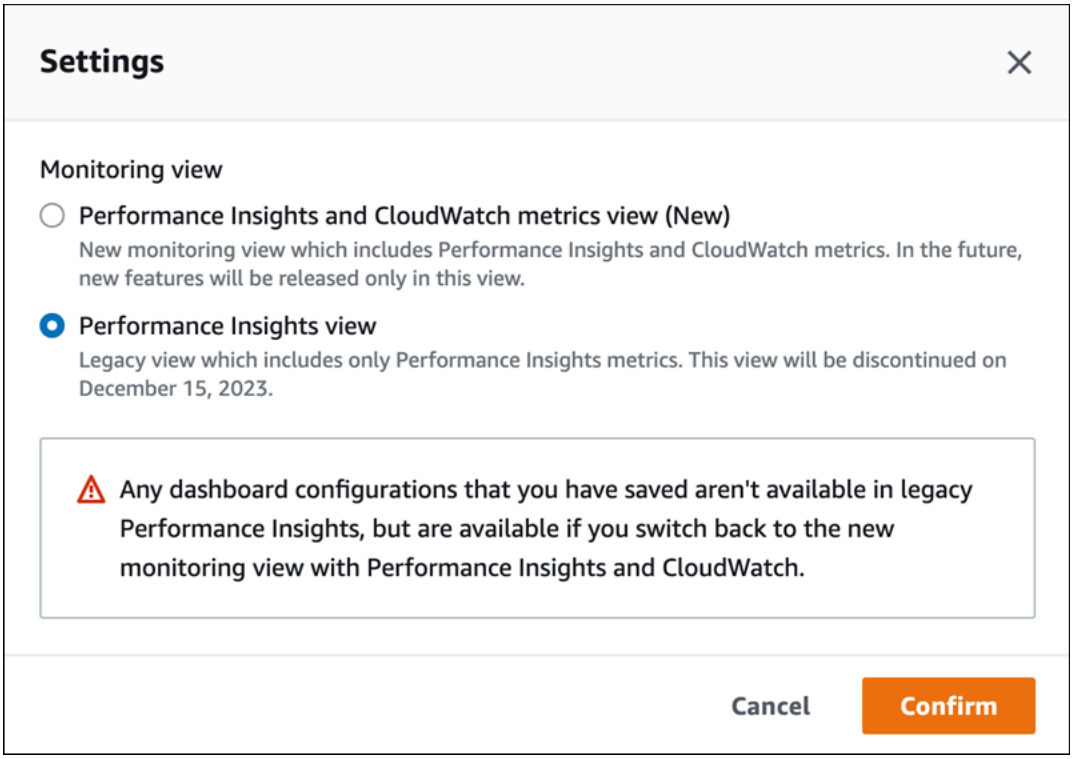
This view will be discontinued on December 15, 2023.

To choose the legacy monitoring view with Performance Insights in the navigation pane:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.
4. Choose the settings icon on the Performance Insights dashboard.

You can now see the **Settings** window that shows the option to choose the legacy Performance Insights view.

The following example shows the window with the option for the legacy monitoring view.



Monitoring view

○ Performance Insights and CloudWatch metrics view (New)
New monitoring view which includes Performance Insights and CloudWatch metrics. In the future, new features will be released only in this view.

● Performance Insights view
Legacy view which includes only Performance Insights metrics. This view will be discontinued on December 15, 2023.

⚠ Any dashboard configurations that you have saved aren't available in legacy Performance Insights, but are available if you switch back to the new monitoring view with Performance Insights and CloudWatch.

5. Select the **Performance Insights view** option and choose **Continue**.

   A warning message appears. Any dashboard configurations that you saved won't be available in this view.

6. Choose **Confirm** to continue to the legacy Performance Insights view.

   You can now view the Performance Insights dashboard that shows only Performance Insights metrics for the DB instance.

Creating a custom dashboard with Performance Insights in the navigation pane

In the new monitoring view, you can create a custom dashboard with the metrics you need to meet your analysis requirements.

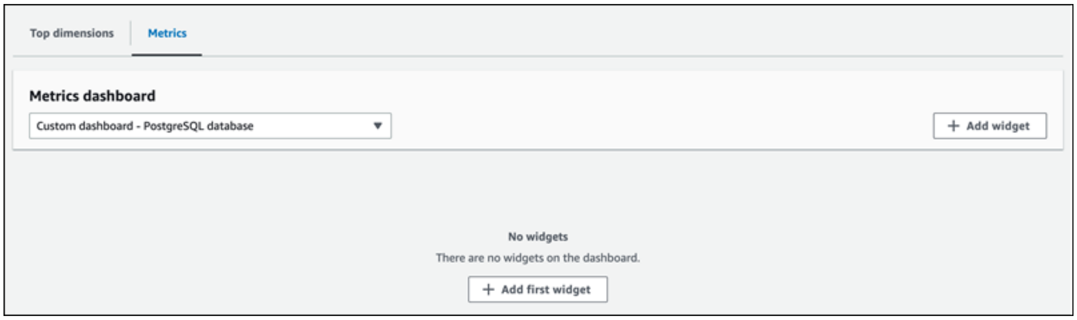
You can create a custom dashboard by selecting Performance Insights and CloudWatch metrics for your DB instance. You can use this custom dashboard for other DB instances of the same database engine type in your AWS account.

> **Note**
>
> The customized dashboard supports up to 50 metrics.

Use the widget settings menu to edit or delete the dashboard, and move or resize the widget window.

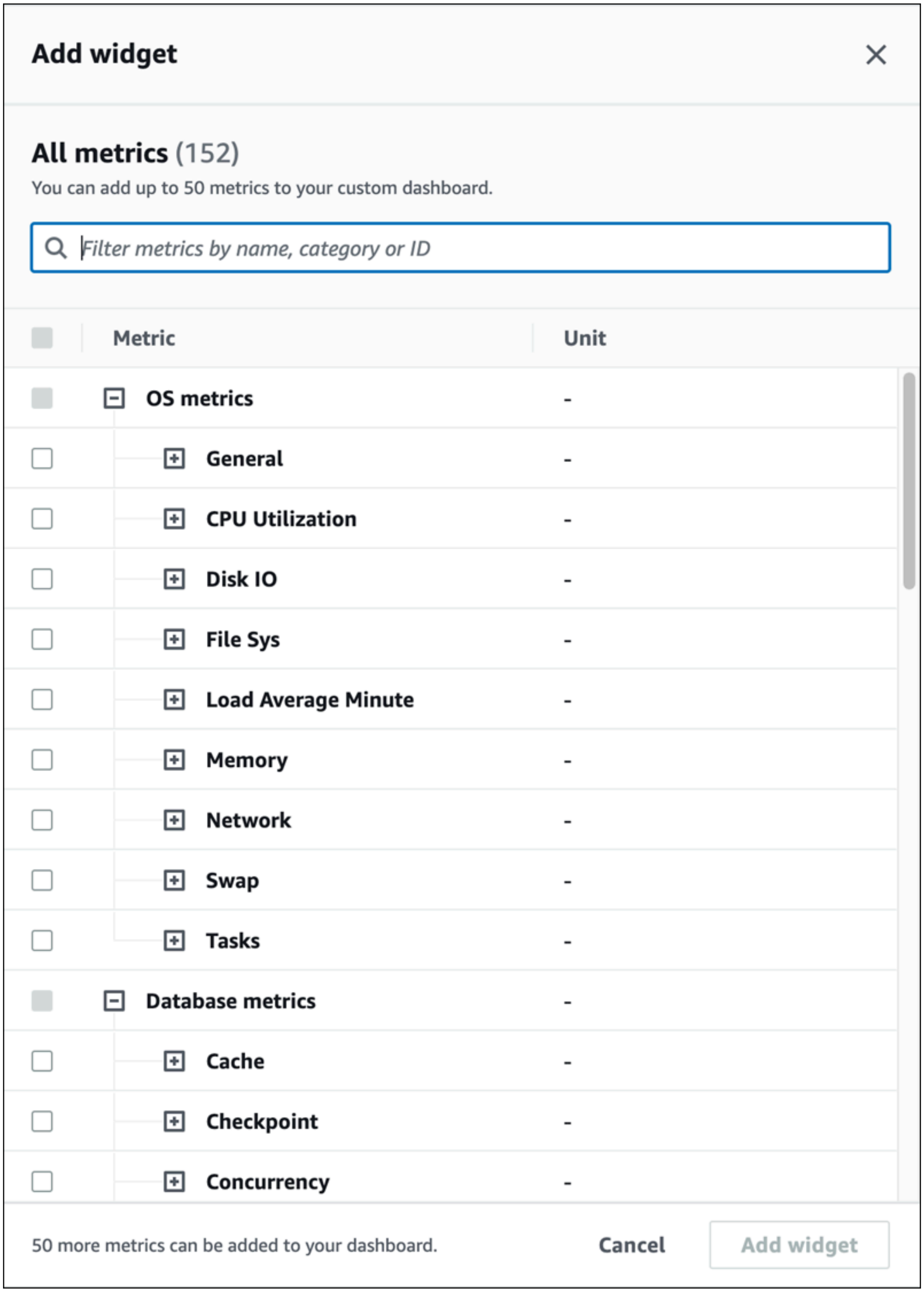
To create a custom dashboard with Performance Insights in the navigation pane:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.
4. Scroll down to the **Metrics tab** in the window.
5. Select the custom dashboard from the drop down list. The following example shows the custom dashboard creation.


Top dimensions
Metrics
Metrics dashboard
Custom dashboard - PostgreSQL database
Add widget
No widgets
There are no widgets on the dashboard.
Add first widget

6. Choose **Add widget** to open the **Add widget** window. You can open and view the available operating system (OS) metrics, database metrics, and CloudWatch metrics in the window.

   The following example shows the **Add widget** window with the metrics.



Add widget

All metrics (152)

You can add up to 50 metrics to your custom dashboard.

Filter metrics by name, category or ID

	Metric	Unit
□	⊟ OS metrics	-
□	⊞ General	-
□	⊞ CPU Utilization	-
□	⊞ Disk IO	-
□	⊞ File Sys	-
□	⊞ Load Average Minute	-
□	⊞ Memory	-
□	⊞ Network	-
□	⊞ Swap	-
□	⊞ Tasks	-
□	⊟ Database metrics	-
□	⊞ Cache	-
□	⊞ Checkpoint	-
□	⊞ Concurrency	-

50 more metrics can be added to your dashboard.

7. Select the metrics that you want to view in the dashboard and choose **Add widget**. You can use the search field to find a specific metric.

   The selected metrics appear on your dashboard.

8. (Optional) If you want to modify or delete your dashboard, choose the settings icon on the upper right of the widget, and then select one of the following actions in the menu.

   - **Edit –** Modify the metrics list in the window. Choose **Update widget** after you select the metrics for your dashboard.
   - **Delete –** Deletes the widget. Choose **Delete** in the confirmation window.

Choosing the preconfigured dashboard with Performance Insights in the navigation pane

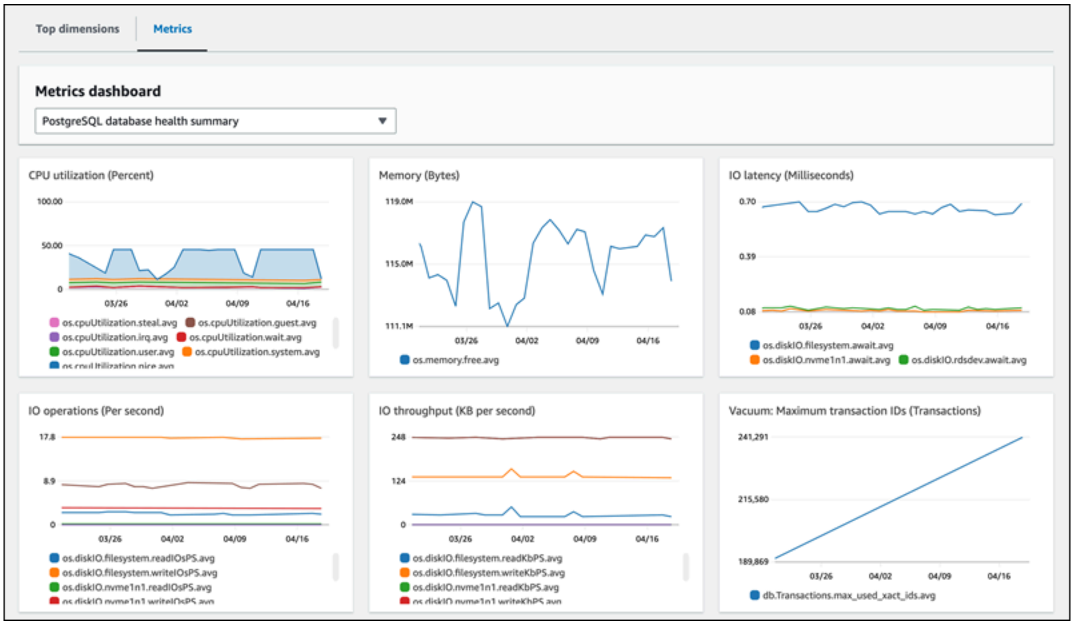
You can view the most commonly used metrics with the preconfigured dashboard. This dashboard helps diagnose performance issues with a database engine and reduce the average recovery time from hours to minutes.

> **Note**
>
> This dashboard can't be edited.

To choose the preconfigured dashboard with Performance Insights in the navigation pane:

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.
4. Scroll down to the **Metrics tab** in the window
5. Select a preconfigured dashboard from the drop down list.

   You can view the metrics for the DB instance in the dashboard. The following example shows a preconfigured metrics dashboard.


Top dimensions
Metrics
Metrics dashboard
PostgreSQL database health summary ▼
CPU utilization (Percent)
100.00
50.00
0
03/26
04/02
04/09
04/16
os.cpuUtilization.steal.avg
os.cpuUtilization.guest.avg
os.cpuUtilization.irq.avg
os.cpuUtilization.wait.avg
os.cpuUtilization.user.avg
os.cpuUtilization.system.avg
os.cpuUtilization.nice.avg
Memory (Bytes)
119.0M
115.0M
111.1M
03/26
04/02
04/09
04/16
os.memory.free.avg
IO latency (Milliseconds)
0.70
0.39
0.08
03/26
04/02
04/09
04/16
os.diskIO.filesystem.await.avg
os.diskIO.nvme1n1.await.avg
os.diskIO.rdsdev.await.avg
IO operations (Per second)
17.8
8.9
0
03/26
04/02
04/09
04/16
os.diskIO.filesystem.readIOsPS.avg
os.diskIO.filesystem.writeIOsPS.avg
os.diskIO.nvme1n1.readIOsPS.avg
os.diskIO.nvme1n1.writeIOsPS.avg
IO throughput (KB per second)
248
124
0
03/26
04/02
04/09
04/16
os.diskIO.filesystem.readKbPS.avg
os.diskIO.filesystem.writeKbPS.avg
os.diskIO.nvme1n1.readKbPS.avg
os.diskIO.nvme1n1.writeKbPS.avg
Vacuum: Maximum transaction IDs (Transactions)
241,291
215,580
189,869
03/26
04/02
04/09
04/16
db.Transactions.max_used_xact_ids.avg

Monitoring Amazon Aurora metrics with Amazon CloudWatch

Amazon CloudWatch is a metrics repository. The repository collects and processes raw data from Amazon Aurora into readable, near real-time metrics. For a complete list of Amazon Aurora metrics sent to CloudWatch, see Metrics reference for Amazon Aurora.

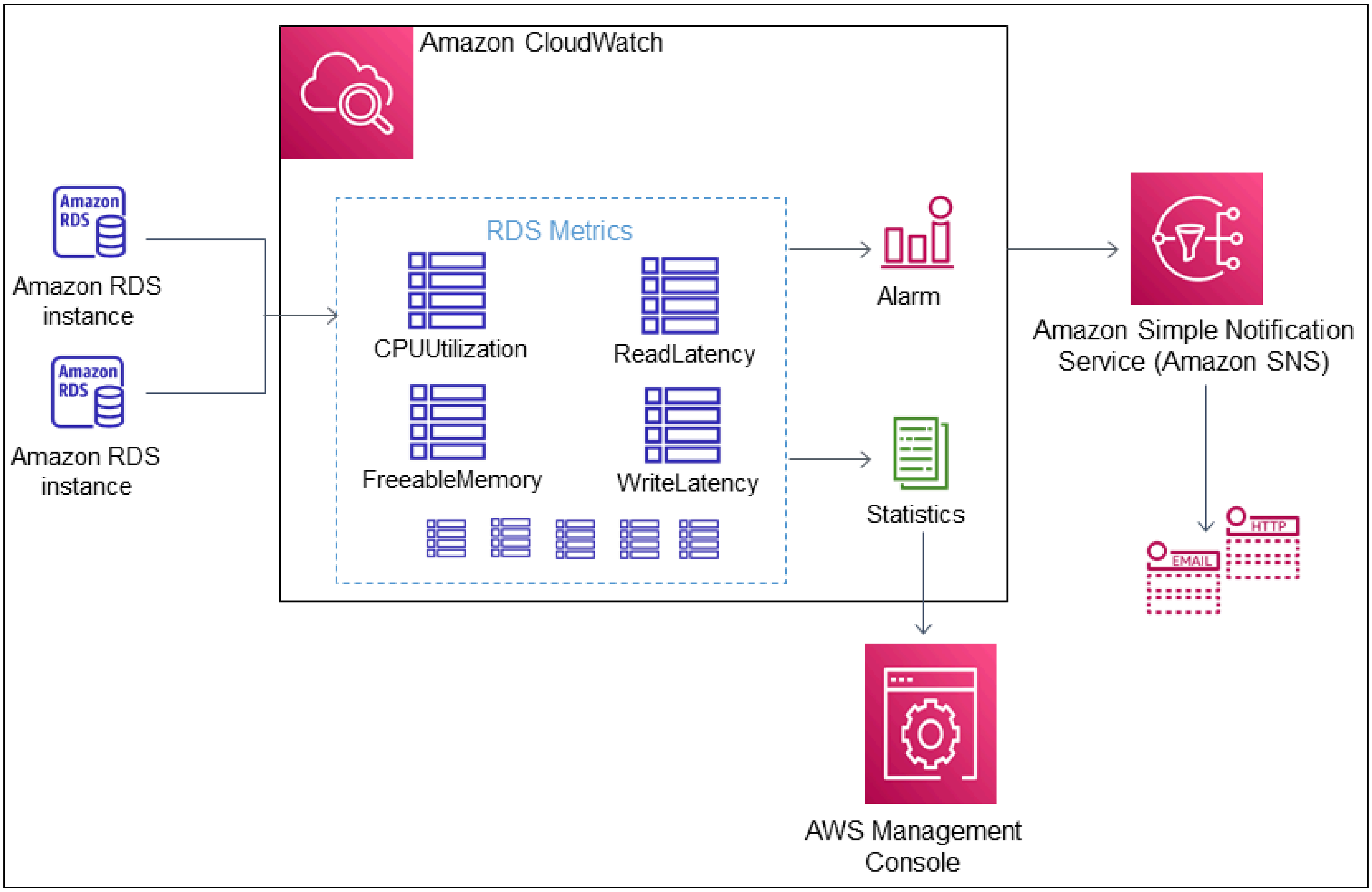
Topics

Overview of Amazon Aurora and Amazon CloudWatch

By default, Amazon Aurora automatically sends metric data to CloudWatch in 1-minute periods. For example, the `CPUUtilization` metric records the percentage of CPU utilization for a DB instance over time. Data points with a period of 60 seconds (1 minute) are available for 15 days. This means that you can access historical information and see how your web application or service is performing.

You can now export Performance Insights metrics dashboards from Amazon RDS to Amazon CloudWatch. You can export either the preconfigured or customized metrics dashboards as a new dashboard or add them to an existing CloudWatch dashboard. The exported dashboard is available to view in the CloudWatch console. For more information on how to export the Performance Insights metrics dashboards to CloudWatch, see Exporting Performance Insights metrics to CloudWatch.

As shown in the following diagram, you can set up alarms for your CloudWatch metrics. For example, you might create an alarm that signals when the CPU utilization for an instance is over 70%. You can configure Amazon Simple Notification Service to email you when the threshold is passed.



Amazon RDS publishes the following types of metrics to Amazon CloudWatch:

- Aurora metrics at both the cluster and instance level

  For a table of these metrics, see Amazon CloudWatch metrics for Amazon Aurora.
- Performance Insights metrics

  For a table of these metrics, see Amazon CloudWatch metrics for Performance Insights and Performance Insights counter metrics.
- Enhanced Monitoring metrics (published to Amazon CloudWatch Logs)

  For a table of these metrics, see OS metrics in Enhanced Monitoring.
- Usage metrics for the Amazon RDS service quotas in your AWS account

  For a table of these metrics, see Amazon CloudWatch usage metrics for Amazon Aurora. For more information about Amazon RDS quotas, see Quotas and constraints for Amazon Aurora.

For more information about CloudWatch, see What is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*. For more information about CloudWatch metrics retention, see Metrics retention.

Viewing DB cluster metrics in the CloudWatch console and AWS CLI

Following, you can find details about how to view metrics for your DB instance using CloudWatch. For information on monitoring metrics for your DB instance's operating system in real time using CloudWatch Logs, see Monitoring OS metrics with Enhanced Monitoring.

When you use Amazon Aurora resources, Amazon Aurora sends metrics and dimensions to Amazon CloudWatch every minute.

You can now export Performance Insights metrics dashboards from Amazon RDS to Amazon CloudWatch and view these metrics in the CloudWatch console. For more information on how to export the Performance Insights metrics dashboards to CloudWatch, see Exporting Performance Insights metrics to CloudWatch.

Use the following procedures to view the metrics for Amazon Aurora in the CloudWatch console and CLI.

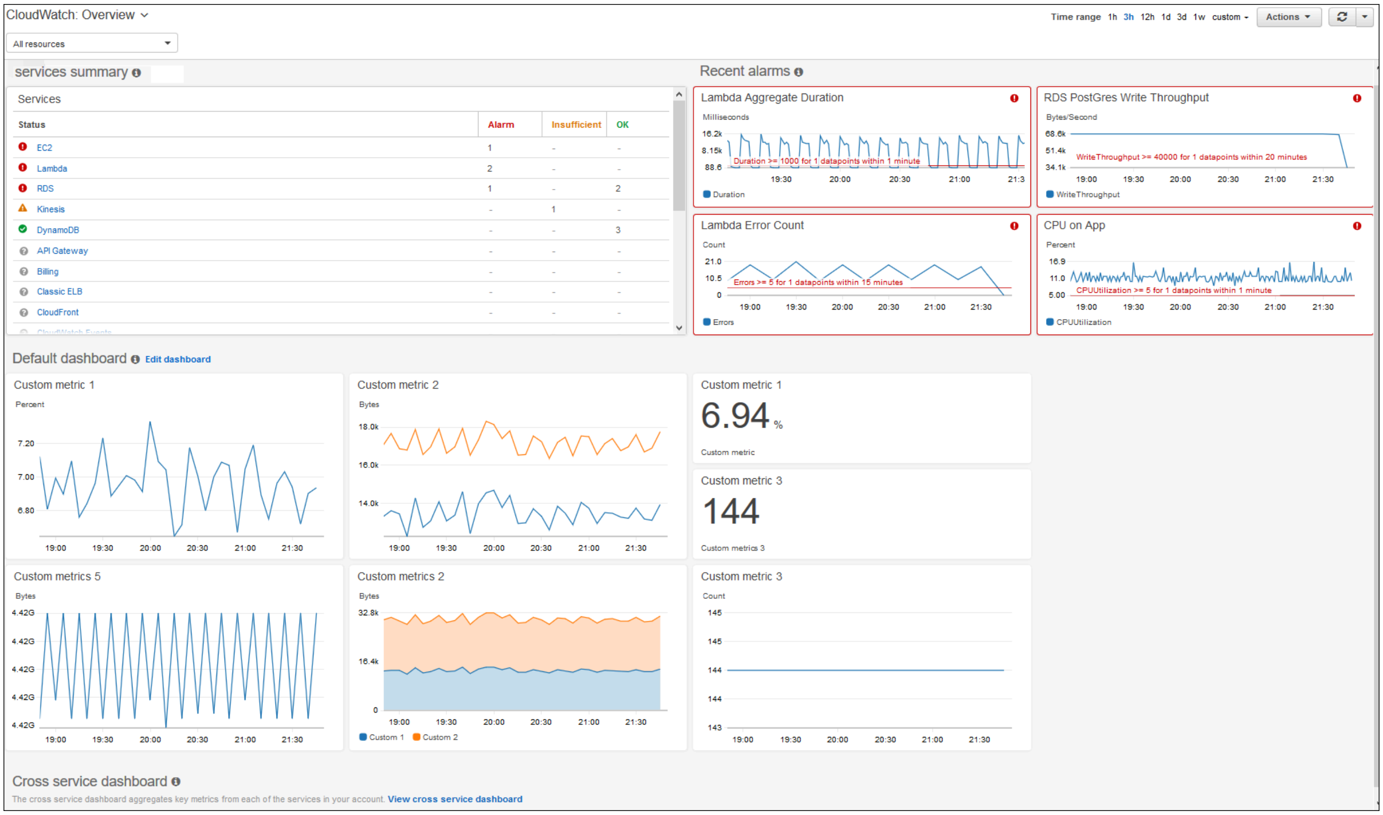
Console

To view metrics using the Amazon CloudWatch console

Metrics are grouped first by the service namespace, and then by the various dimension combinations within each namespace.

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

   The CloudWatch overview home page appears.



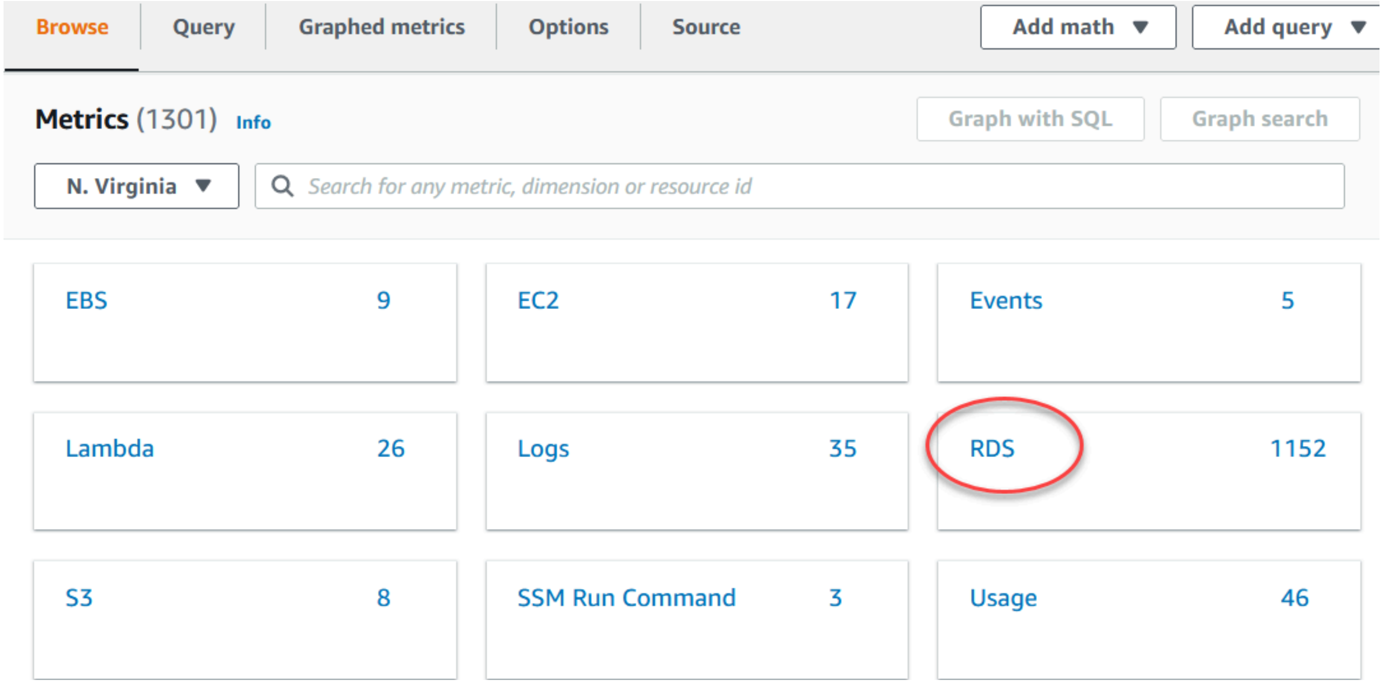
Services

Status	Alarm	Insufficient	OK
EC2	1	-	-
Lambda	2	-	-
RDS	1	-	2
Kinesis	-	1	-
DynamoDB	-	-	3
API Gateway	-	-	-
Billing	-	-	-
Classic ELB	-	-	-
CloudFront	-	-	-

The cross service dashboard aggregates key metrics from each of the services in your account. View cross service dashboard

2. If necessary, change the AWS Region. From the navigation bar, choose the AWS Region where your AWS resources are. For more information, see Regions and endpoints.

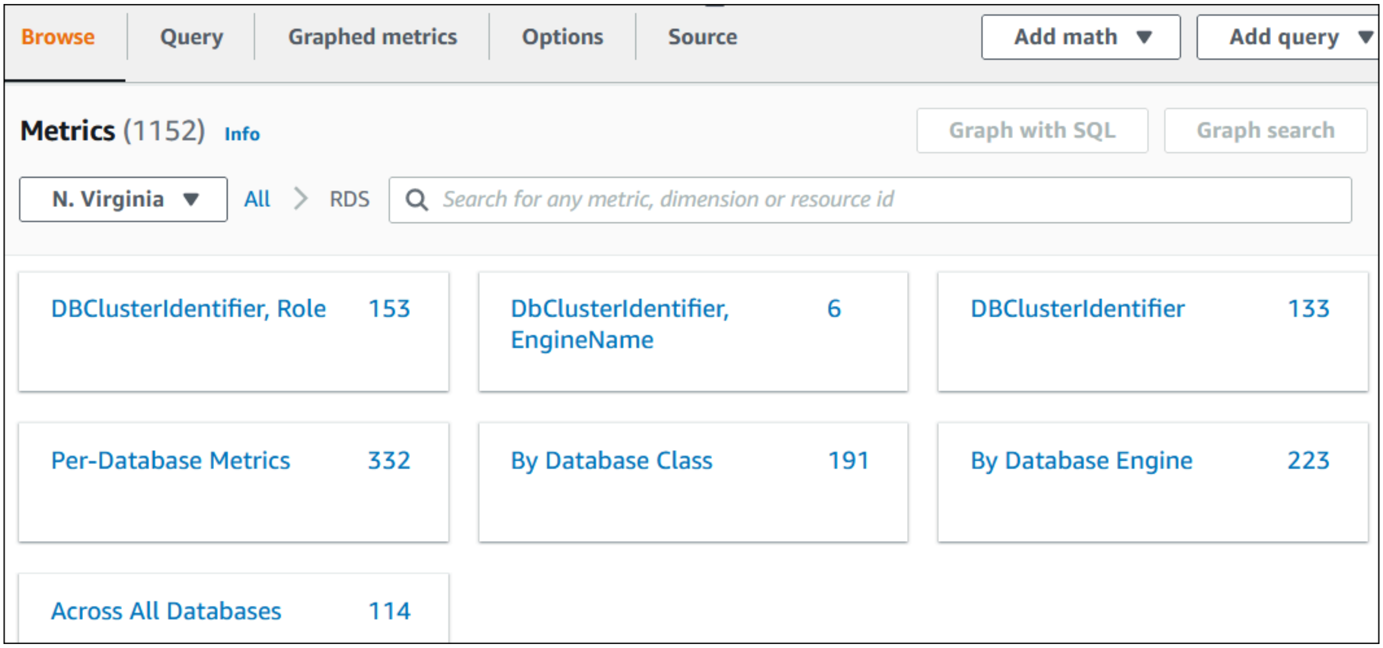
3. In the navigation pane, choose **Metrics** and then **All metrics**.



EBS	9
EC2	17
Events	5
Lambda	26
Logs	35
RDS	1152
S3	8
SSM Run Command	3
Usage	46

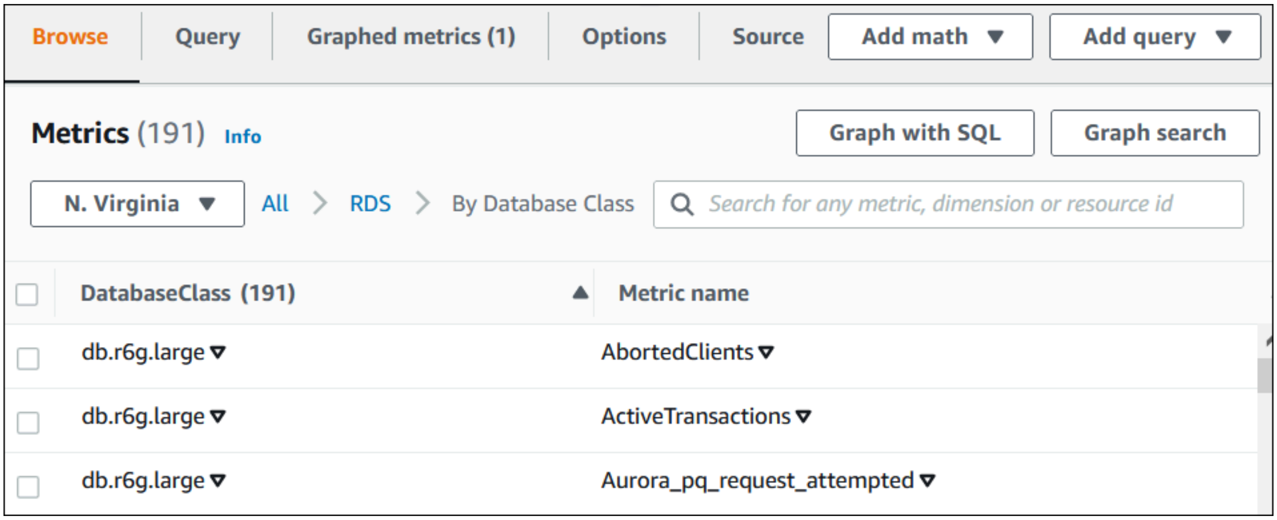
4. Scroll down and choose the **RDS** metric namespace.

The page displays the Amazon Aurora dimensions. For descriptions of these dimensions, see Amazon CloudWatch dimensions for Aurora.



DBClusterIdentifier, Role	153	DbClusterIdentifier, EngineName	6	DBClusterIdentifier	133
Per-Database Metrics	332	By Database Class	191	By Database Engine	223
Across All Databases	114				

5. Choose a metric dimension, for example **By Database Class**.

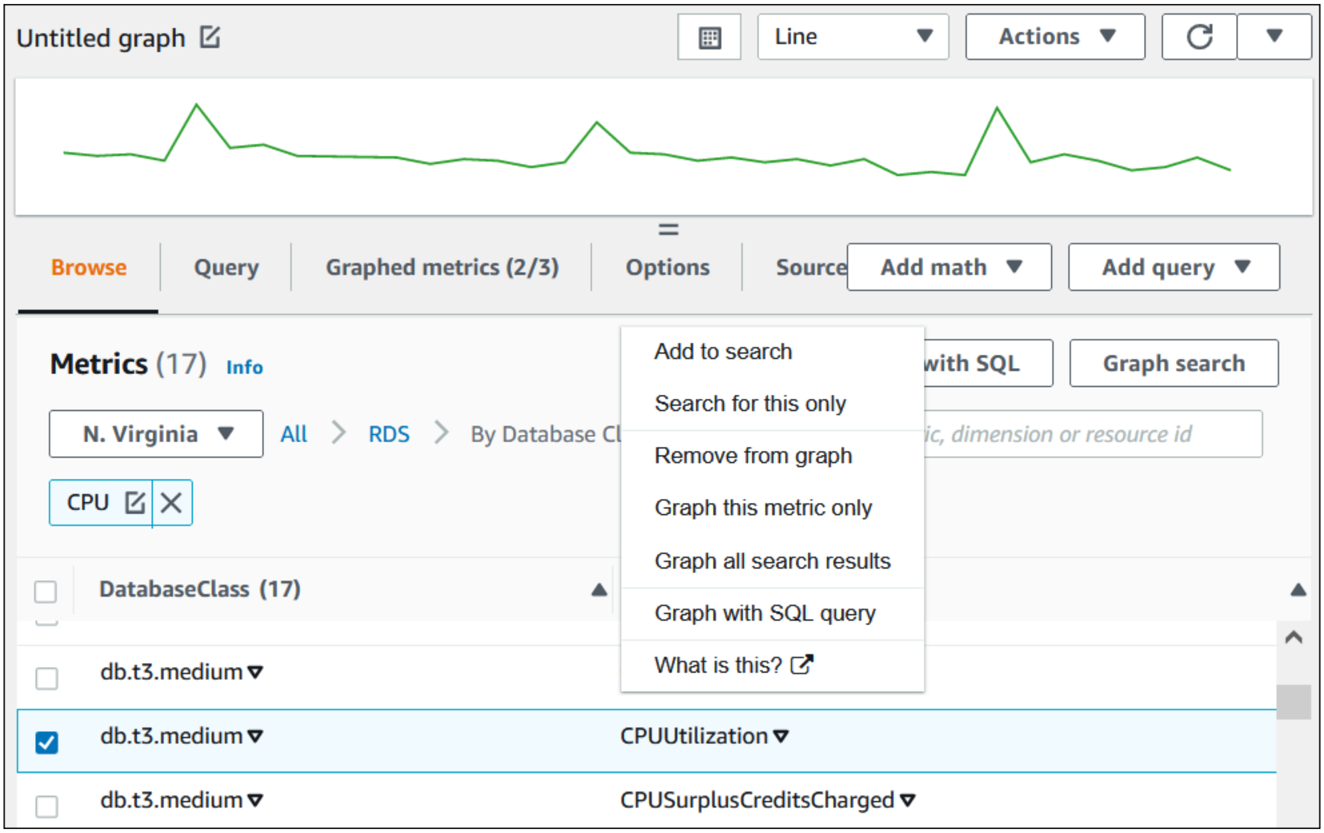


□	DatabaseClass (191) ▲	Metric name
□	db.r6g.large ▼	AbortedClients ▼
□	db.r6g.large ▼	ActiveTransactions ▼
□	db.r6g.large ▼	Aurora_pq_request_attempted ▼

6. Do any of the following actions:

- To sort the metrics, use the column heading.
- To graph a metric, select the check box next to the metric.
- To filter by resource, choose the resource ID, and then choose **Add to search**.
- To filter by metric, choose the metric name, and then choose **Add to search**.

The following example filters on the **db.t3.medium** class and graphs the **CPUUtilization** metric.



	DatabaseClass (17)	
□	db.t3.medium	
☑	db.t3.medium	CPUUtilization
□	db.t3.medium	CPUSurplusCreditsCharged

You can find details about how to analyze resource usage for Aurora PostgreSQL using CloudWatch metrics. For more information, see Using Amazon CloudWatch metrics to analyze resource usage for Aurora PostgreSQL

AWS CLI

To obtain metric information by using the AWS CLI, use the CloudWatch command `list-metrics`. In the following example, you list all metrics in the `AWS/RDS` namespace.

```
aws cloudwatch list-metrics --namespace AWS/RDS
```

To obtain metric data, use the command `get-metric-data`.

The following example gets `CPUUtilization` statistics for instance `my-instance` over the specific 24-hour period, with a 5-minute granularity.

Create a JSON file `CPU_metric.json` with the following contents.

```
{
   "StartTime" : "2023-12-25T00:00:00Z",
   "EndTime" : "2023-12-26T00:00:00Z",
   "MetricDataQueries" : [{
     "Id" : "cpu",
     "MetricStat" : {
    "Metric" : {
        "Namespace" : "AWS/RDS",
        "MetricName" : "CPUUtilization",
        "Dimensions" : [{ "Name" : "DBInstanceIdentifier" , "Value" : my-instance}]
    },
       "Period" : 360,
       "Stat" : "Minimum"
     }
   }]
}
```

Example

For Linux, macOS, or Unix:

```
aws cloudwatch get-metric-data \
    --cli-input-json file://CPU_metric.json
```

For Windows:

```
aws cloudwatch get-metric-data ^
     --cli-input-json file://CPU_metric.json
```

Sample output appears as follows:

```
{
    "MetricDataResults": [
        {
            "Id": "cpu",
            "Label": "CPUUtilization",
            "Timestamps": [
                "2023-12-15T23:48:00+00:00",
                "2023-12-15T23:42:00+00:00",
                "2023-12-15T23:30:00+00:00",
                "2023-12-15T23:24:00+00:00",
                ...
            ],
            "Values": [
                13.299778337027714,
                13.677507543049558,
                14.24976250395827,
                13.02521708695145,
                ...
            ],
            "StatusCode": "Complete"
        }
    ],
    "Messages": []
}
```

For more information, see Getting statistics for a metric in the *Amazon CloudWatch User Guide*.

Exporting Performance Insights metrics to CloudWatch

Performance Insights lets you export the preconfigured or custom metrics dashboard for your DB instance to Amazon CloudWatch. You can export the metrics dashboard as a new dashboard or add it to an existing CloudWatch dashboard. When you choose to add the dashboard to an existing CloudWatch dashboard, you can create a header label so that the metrics appear in a separate section in the CloudWatch dashboard.

You can view the exported metrics dashboard in the CloudWatch console. If you add new metrics to a Performance Insights metrics dashboard after you export it, you must export this dashboard again to view the new metrics in the CloudWatch console.

You can also select a metric widget in the Performance Insights dashboard and view the metrics data in the CloudWatch console.

For more information about viewing the metrics in the CloudWatch console, see Viewing DB cluster metrics in the CloudWatch console and AWS CLI.

Exporting Performance Insights metrics as a new dashboard to CloudWatch

Choose a preconfigured or custom metrics dashboard from the Performance Insights dashboard and export it as a new dashboard to CloudWatch. You can view the exported dashboard in the CloudWatch console.

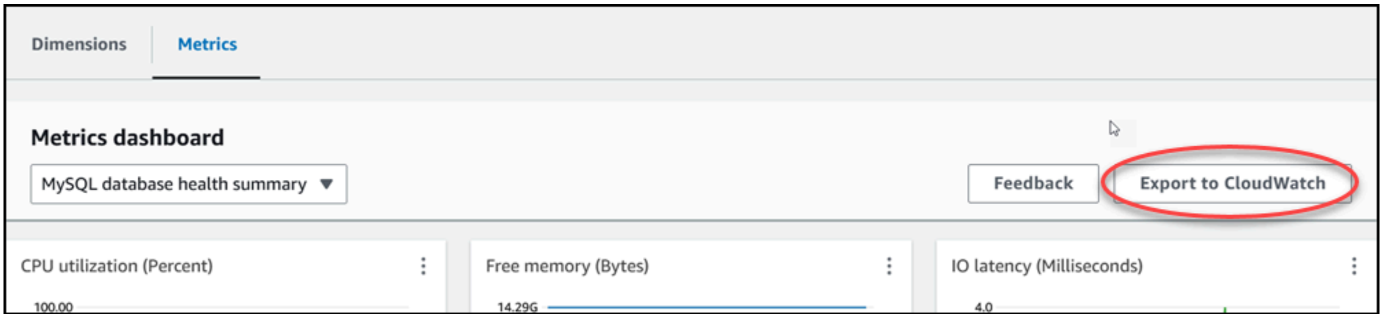
To export a Performance Insights metric dashboard as a new dashboard to CloudWatch

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

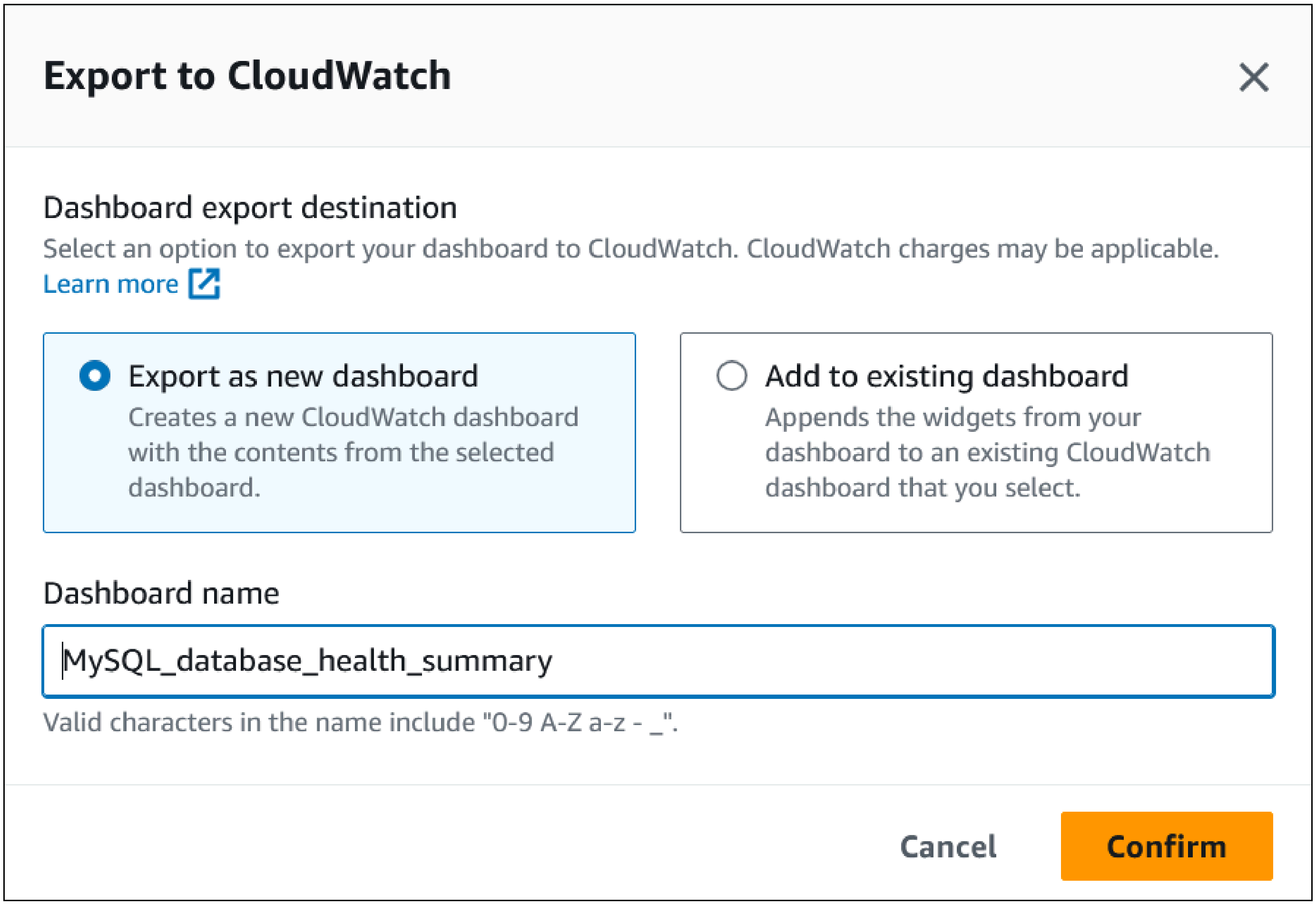
   The Performance Insights dashboard appears for the DB instance.
4. Scroll down and choose **Metrics**.

   By default, the preconfigured dashboard with Performance Insights metrics appears.
5. Choose a preconfigured or custom dashboard and then choose **Export to CloudWatch**.

   The **Export to CloudWatch** window appears.



6. Choose **Export as new dashboard**.



Export to CloudWatch

Dashboard export destination

Select an option to export your dashboard to CloudWatch. CloudWatch charges may be applicable. Learn more

● Export as new dashboard
Creates a new CloudWatch dashboard with the contents from the selected dashboard.

○ Add to existing dashboard
Appends the widgets from your dashboard to an existing CloudWatch dashboard that you select.

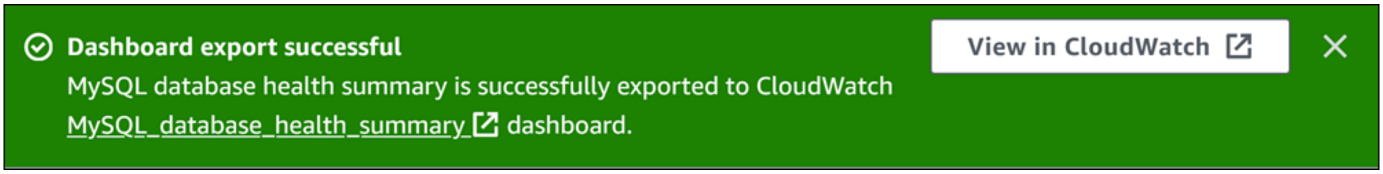
Dashboard name

MySQL_database_health_summary

Valid characters in the name include "0-9 A-Z a-z - _".

7. Enter a name for the new dashboard in the **Dashboard name** field and choose **Confirm**.

   A banner displays a message after the dashboard export is successful.



Dashboard export successful

MySQL database health summary is successfully exported to CloudWatch MySQL_database_health_summary dashboard.

8. Choose the link or **View in CloudWatch** in the banner to view the metrics dashboard in the CloudWatch console.

Adding Performance Insights metrics to an existing CloudWatch dashboard

Add a preconfigured or custom metrics dashboard to an existing CloudWatch dashboard. You can add a label to the metrics dashboard to appear in a separate section in the CloudWatch dashboard.

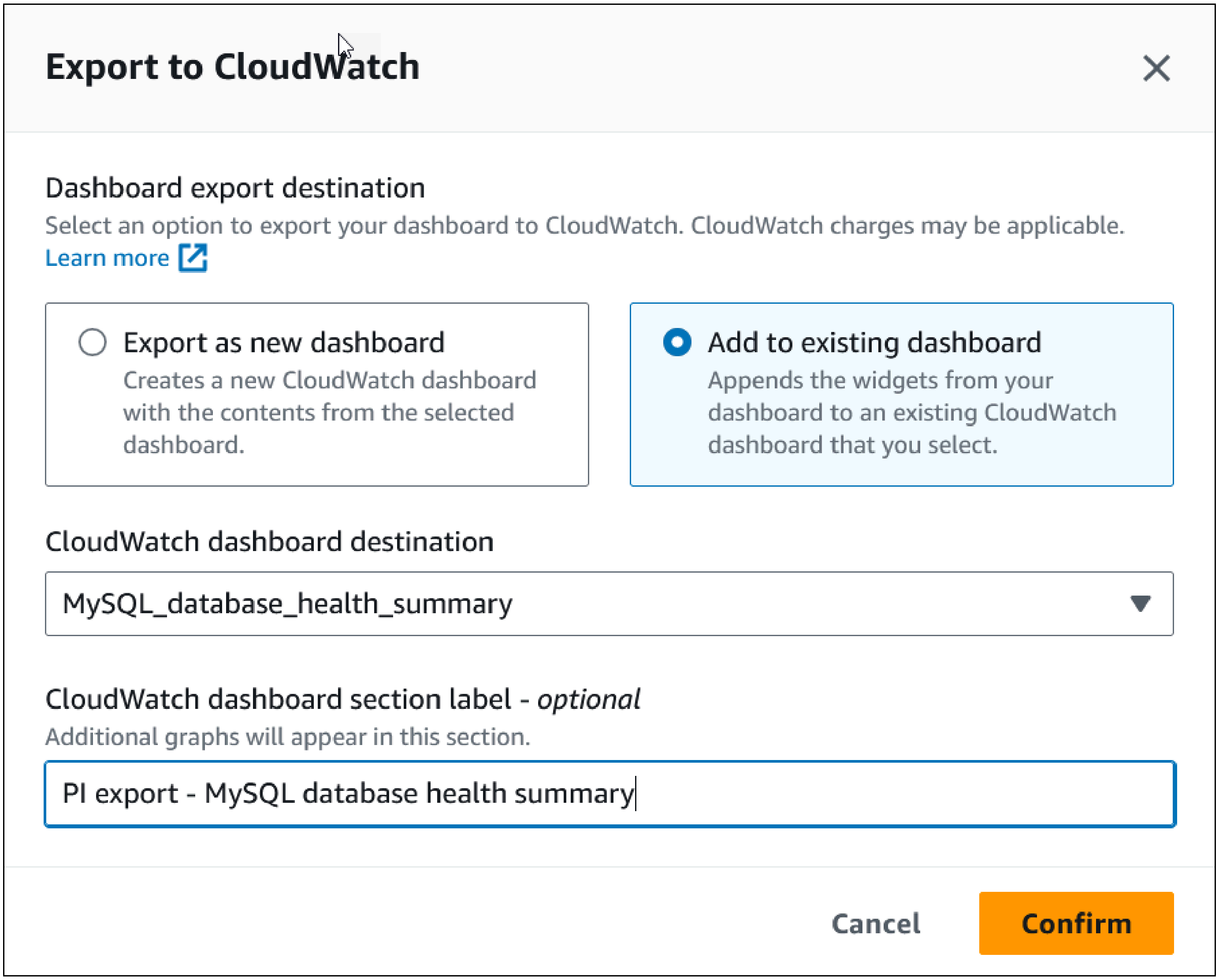
To export the metrics to an existing CloudWatch dashboard

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

   The Performance Insights dashboard appears for the DB instance.
4. Scroll down and choose **Metrics**.

   By default, the preconfigured dashboard with Performance Insights metrics appears.
5. Choose the preconfigured or custom dashboard and then choose **Export to CloudWatch**.

   The **Export to CloudWatch** window appears.
6. Choose **Add to existing dashboard**.



Export to CloudWatch

Dashboard export destination

Select an option to export your dashboard to CloudWatch. CloudWatch charges may be applicable. Learn more

○ Export as new dashboard
Creates a new CloudWatch dashboard with the contents from the selected dashboard.

● Add to existing dashboard
Appends the widgets from your dashboard to an existing CloudWatch dashboard that you select.

CloudWatch dashboard destination

MySQL_database_health_summary

CloudWatch dashboard section label - *optional*

Additional graphs will appear in this section.

PI export - MySQL database health summary

7. Specify the dashboard destination and label, and then choose **Confirm**.

   - **CloudWatch dashboard destination** - Choose an existing CloudWatch dashboard.
   - **CloudWatch dashboard section label - optional** - Enter a name for the Performance Insights metrics to appear in this section in the CloudWatch dashboard.

   A banner displays a message after the dashboard export is successful.

8. Choose the link or **View in CloudWatch** in the banner to view the metrics dashboard in the CloudWatch console.

Viewing a Performance Insights metric widget in CloudWatch

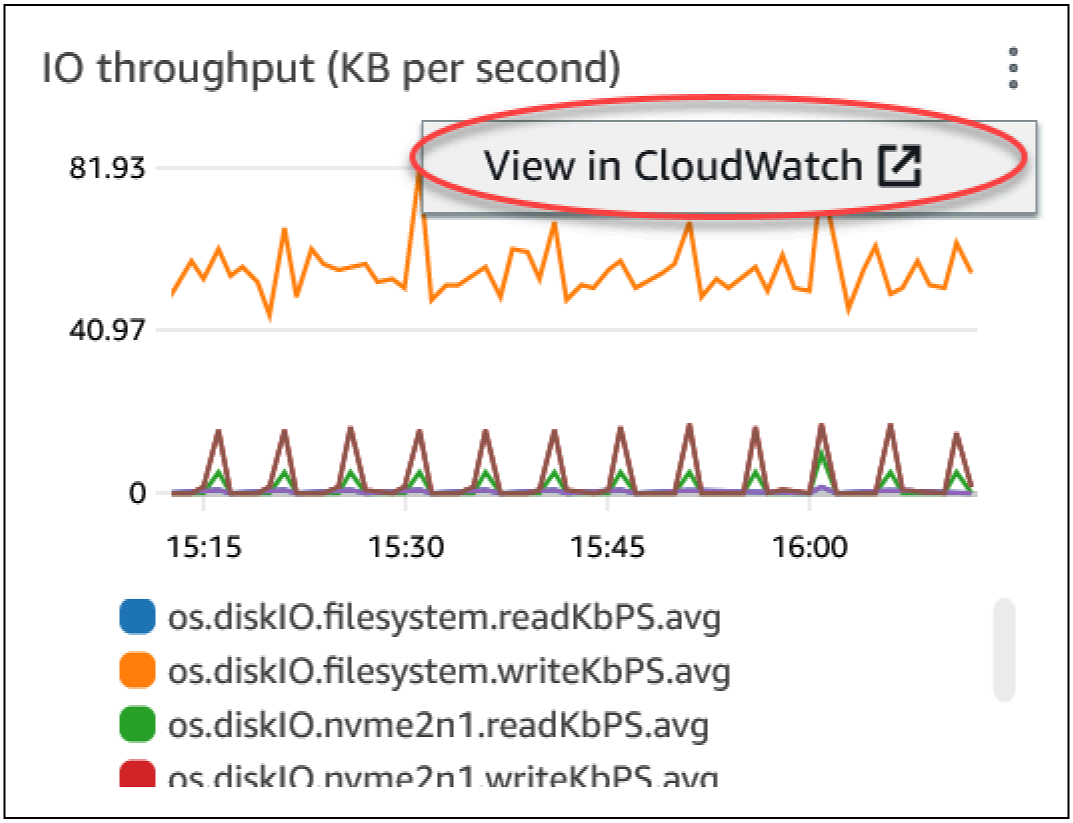
Select a Performance Insights metric widget in the Amazon RDS Performance Insights dashboard and view the metric data in the CloudWatch console.

To export a metric widget and view the metrics data in the CloudWatch console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

   The Performance Insights dashboard appears for the DB instance.
4. Scroll down to **Metrics**.

   By default, the preconfigured dashboard with Performance Insights metrics appears.
5. Choose a metric widget and then choose **View in CloudWatch** in the menu.


IO throughput (KB per second)
View in CloudWatch
81.93
40.97
0
15:15
15:30
15:45
16:00
os.diskIO.filesystem.readKbPS.avg
os.diskIO.filesystem.writeKbPS.avg
os.diskIO.nvme2n1.readKbPS.avg
os.diskIO.nvme2n1.writeKbPS.avg

   The metric data appears in the CloudWatch console.

Creating CloudWatch alarms to monitor Amazon Aurora

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period that you specify. The alarm can also perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods. The action is a notification sent to an Amazon SNS topic or Amazon EC2 Auto Scaling policy.

Alarms invoke actions for sustained state changes only. CloudWatch alarms don't invoke actions simply because they are in a particular state. The state must have changed and have been maintained for a specified number of time periods.

> **Note**
>
> For Aurora, use `WRITER` or `READER` role metrics to set up alarms instead of relying on metrics for specific DB instances. Aurora DB instance roles can change roles over time. You can find these role-based metrics in the CloudWatch console.
> Aurora Auto Scaling automatically sets alarms based on `READER` role metrics. For more information about Aurora Auto Scaling, see Using Amazon Aurora Auto Scaling with Aurora Replicas.

You can use the **DB_PERF_INSIGHTS** metric math function in the CloudWatch console to query Amazon RDS for Performance Insights counter metrics. The **DB_PERF_INSIGHTS** function also includes the DBLoad metric at sub-minute intervals. You can set CloudWatch alarms on these metrics.

For more details on how to create an alarm, see Create an alarm on Performance Insights counter metrics from an AWS database.

To set an alarm using the AWS CLI

- Call `put-metric-alarm`. For more information, see *AWS CLI Command Reference*.

To set an alarm using the CloudWatch API

- Call `PutMetricAlarm`. For more information, see *Amazon CloudWatch API Reference*

For more information about setting up Amazon SNS topics and creating alarms, see Using Amazon CloudWatch alarms.

Monitoring DB load with Performance Insights on Amazon Aurora

Performance Insights expands on existing Amazon Aurora monitoring features to illustrate and help you analyze your cluster performance. With the Performance Insights dashboard, you can visualize the database load on your Amazon Aurora cluster load and filter the load by waits, SQL statements, hosts, or users. For information about using Performance Insights with Amazon DocumentDB, see *Amazon DocumentDB Developer Guide*.

Topics

- Overview of Performance Insights on Amazon Aurora
- Turning Performance Insights on and off for Aurora
- Turning on the Performance Schema for Performance Insights on Aurora MySQL
- Configuring access policies for Performance Insights
- Analyzing metrics with the Performance Insights dashboard
- Viewing Performance Insights proactive recommendations
- Retrieving metrics with the Performance Insights API for Aurora
- Logging Performance Insights calls using AWS CloudTrail

Overview of Performance Insights on Amazon Aurora

By default, RDS enables Performance Insights in the console create wizard for all Amazon RDS engines. If you turn on Performance Insights at the DB cluster level, RDS enables Performance Insights for every DB instance in the cluster. If you have more than one database on a DB instance, Performance Insights aggregates performance data.

You can find an overview of Performance Insights for Amazon Aurora in the following video.

Using Performance Insights to Analyze Performance of Amazon Aurora PostgreSQL

Topics

- Database load
- Maximum CPU
- Amazon Aurora DB engine, Region, and instance class support for Performance Insights
- Pricing and data retention for Performance Insights

Database load

Database load (DB load) measures the level of session activity in your database. DBLoad is the key metric in Performance Insights, and Performance Insights collects DB load every second.

Topics

Active sessions

A *database session* represents an application's dialogue with a relational database. An *active session* is a connection that has submitted work to the DB engine and is waiting for a response.

A session is active when it's either running on CPU or waiting for a resource to become available so that it can proceed. For example, an active session might wait for a page (or block) to be read into memory, and then consume CPU while it reads data from the page.

Average active sessions

The *average active sessions (AAS)* is the unit for the DBLoad metric in Performance Insights. It measures how many sessions are concurrently active on the database.

Every second, Performance Insights samples the number of sessions concurrently running a query. For each active session, Performance Insights collects the following data:

- SQL statement
- Session state (running on CPU or waiting)
- Host
- User running the SQL

Performance Insights calculates the AAS by dividing the total number of sessions by the number of samples for a specific time period. For example, the following table shows 5 consecutive samples of a running query taken at 1-second intervals.

Sample	Number of sessions running query	AAS	Calculation
1	2	2	2 total sessions / 1 sample
2	0	1	2 total sessions / 2 samples
3	4	2	6 total sessions / 3 samples
4	0	1.5	6 total sessions / 4 samples
5	4	2	10 total sessions / 5 samples

In the preceding example, the DB load for the time interval was 2 AAS. This measurement means that, on average, 2 sessions were active at any given time during the interval when the 5 samples were taken.

Average active executions

The *average active executions (AAE)* per second is related to AAS. To calculate the AAE, Performance Insights divides the total execution time of a query by the time interval. The following table shows the AAE calculation for the same query in the preceding table.

Elapsed time (sec)	Total execution time (sec)	AAE	Calculation
60	120	2	120 execution seconds/60 elapsed seconds
120	120	1	120 execution seconds/120 elapsed seconds
180	380	2.11	380 execution seconds/180 elapsed seconds

Elapsed time (sec)	Total execution time (sec)	AAE	Calculation
240	380	1.58	380 execution seconds/240 elapsed seconds
300	600	2	600 execution seconds/300 elapsed seconds

In most cases, the AAS and AAE for a query are approximately the same. However, because the inputs to the calculations are different data sources, the calculations often vary slightly.

Dimensions

The `db.load` metric is different from the other time-series metrics because you can break it into subcomponents called *dimensions*. You can think of dimensions as "slice by" categories for the different characteristics of the `DBLoad` metric.

When you are diagnosing performance issues, the following dimensions are often the most useful:

Topics

- Wait events
- Top SQL

For a complete list of dimensions for the Aurora engines, see DB load sliced by dimensions.

Wait events

A *wait event* causes a SQL statement to wait for a specific event to happen before it can continue running. Wait events are an important dimension, or category, for DB load because they indicate where work is impeded.

Every active session is either running on the CPU or waiting. For example, sessions consume CPU when they search memory for a buffer, perform a calculation, or run procedural code. When sessions aren't consuming CPU, they might be waiting for a memory buffer to become free, a data file to be read, or a log to be written to. The more time that a session waits for resources, the less time it runs on the CPU.

When you tune a database, you often try to find out the resources that sessions are waiting for. For example, two or three wait events might account for 90 percent of DB load. This measure means that, on average, active sessions are spending most of their time waiting for a small number of resources. If you can find out the cause of these waits, you can attempt a solution.

Wait events vary by DB engine:

- For a list of the common wait events for Aurora MySQL, see Aurora MySQL wait events. To learn how to tune using these wait events, see Tuning Aurora MySQL.
- For information about all MySQL wait events, see Wait Event Summary Tables in the MySQL documentation.
- For a list of common wait events for Aurora PostgreSQL, see Amazon Aurora PostgreSQL wait events. To learn how to tune using these wait events, see Tuning with wait events for Aurora PostgreSQL.
- For information about all PostgreSQL wait events, see The Statistics Collector > Wait Event tables in the PostgreSQL documentation.

Top SQL

Where wait events show bottlenecks, top SQL shows which queries are contributing the most to DB load. For example, many queries might be currently running on the database, but a single query might consume 99 percent of the DB load. In this case, the high load might indicate a problem with the query.

By default, the Performance Insights console displays top SQL queries that are contributing to the database load. The console also shows relevant statistics for each statement. To diagnose performance problems for a specific statement, you can examine its execution plan.

Maximum CPU

In the dashboard, the **Database load** chart collects, aggregates, and displays session information. To see whether active sessions are exceeding the maximum CPU, look at their relationship to the **Max vCPU** line. Performance Insights determines the **Max vCPU** value by the number of vCPU (virtual CPU) cores for your DB instance. For Aurora Serverless v2, **Max vCPU** represents the estimated number of vCPUs.

One process can run on a vCPU at a time. If the number of processes exceed the number of vCPUs, the processes start queuing. When the queuing increase, the performance is impacted. If the DB load is often above the **Max vCPU** line, and the primary wait state is CPU, the CPU is overloaded. In this case, you might want to throttle connections to the instance, tune any SQL queries with a high CPU load, or consider a larger instance class. High and consistent instances of any wait state indicate that there might be bottlenecks or resource contention issues to resolve. This can be true even if the DB load doesn't cross the **Max vCPU** line.

Amazon Aurora DB engine, Region, and instance class support for Performance Insights

The following table provides Amazon Aurora DB engines that support Performance Insights.

Amazon Aurora DB engine	Supported engine versions and Regions	Instance class restrictions
Amazon Aurora MySQL-Compatible Edition	For more information on version and Region availability of Performance Insights with Aurora MySQL, see Performance Insights with Aurora MySQL.	Performance Insights has the following engine class restrictions: • db.t2 – Not supported • db.t3 – Not supported • db.t4g.micro and db.t4g.small – Not supported
Amazon Aurora PostgreSQL-Compatible Edition	For more information on version and Region availability of Performance Insights with Aurora PostgreSQL, see Performance Insights with Aurora PostgreSQL.	N/A

Amazon Aurora DB engine, Region, and instance class support for Performance Insights features

The following table provides Amazon Aurora DB engines that support Performance Insights features.

Feature	Pricing tier	Supported regions	Supported DB engines	Supported instance classes
SQL statistics for Performance Insights	All	All	All	All
Analyzing database performance for a period of time	Paid tier only	• US East (Ohio) • US East (N. Virginia) • US West (N. California) • US West (Oregon) • Asia Pacific (Mumbai) • Asia Pacific (Seoul) • Asia Pacific (Singapore) • Asia Pacific (Sydney) • Asia Pacific (Tokyo) • Canada (Central) • Europe (Frankfurt) • Europe (Ireland) • Europe (London) • Europe (Paris)	All	All except db.serverless (Aurora Serverless v2)

Feature	Pricing tier	Supported regions	Supported DB engines	Supported instance classes
		• Europe (Stockholm)		

Feature	Pricing tier	Supported regions	Supported DB engines	Supported instance classes
Viewing Performance Insights proactive recommendations	Paid tier only	US East (Ohio) US East (N. Virginia) US West (N. California) US West (Oregon) Asia Pacific (Mumbai) Asia Pacific (Seoul) Asia Pacific (Singapore) Asia Pacific (Sydney) Asia Pacific (Tokyo) Canada (Central) Europe (Frankfurt) Europe (Ireland) Europe (London) Europe (Paris) Europe (Stockholm)	All	All except db.serverless (Aurora Serverless v2)

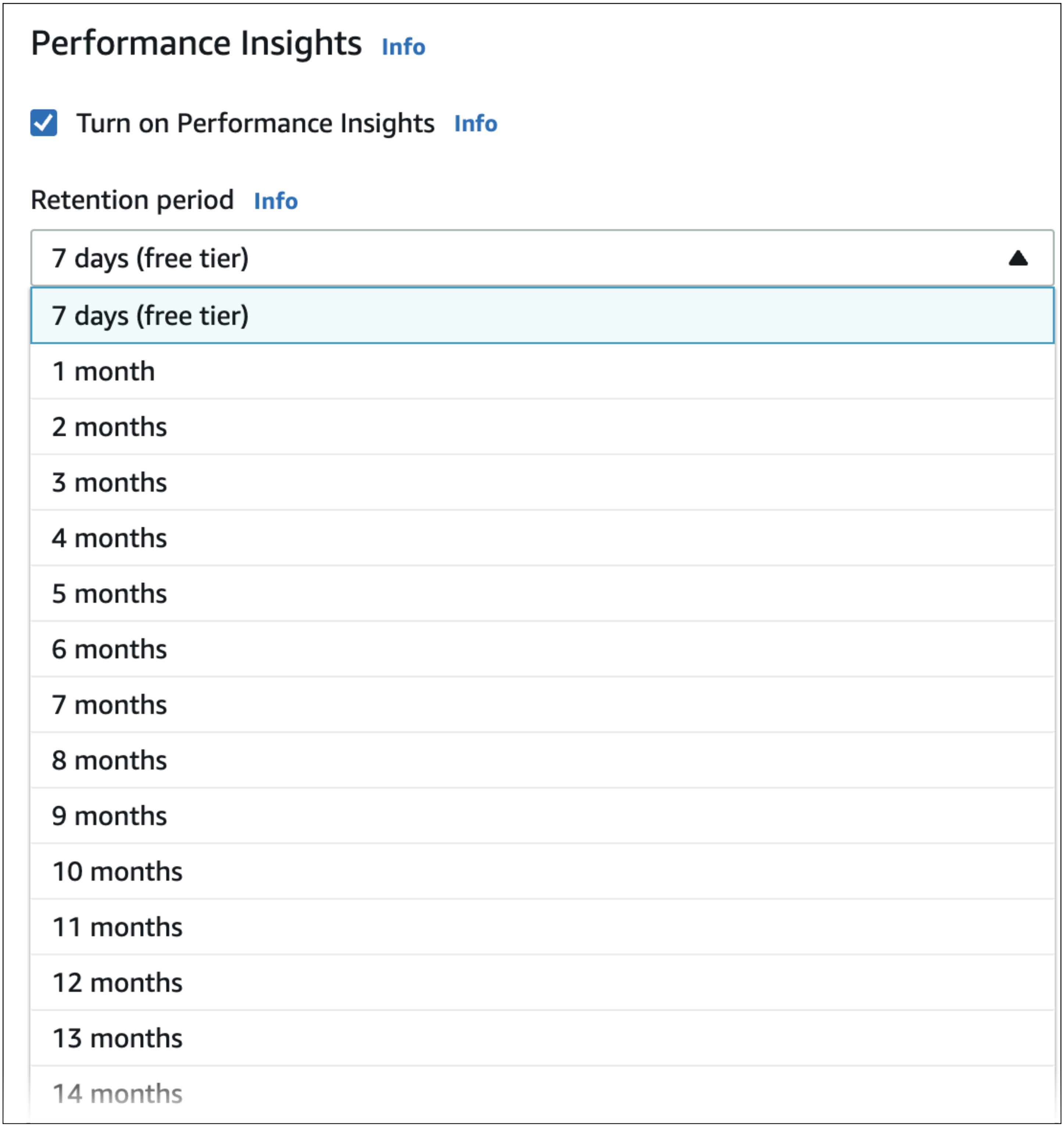
Feature	Pricing tier	Supported regions	Supported DB engines	Supported instance classes
		- South America (São Paulo)		

Pricing and data retention for Performance Insights

By default, Performance Insights offers a free tier that includes 7 days of performance data history and 1 million API requests per month. You can also purchase longer retention periods. For complete pricing information, see Performance Insights Pricing.

In the RDS console, you can choose any of the following retention periods for your Performance Insights data:

- **Default (7 days)**
- ***n*** **months**, where ***n*** is a number from 1–24



Performance Insights Info

☑ Turn on Performance Insights Info

Retention period Info

7 days (free tier)

To learn how to set a retention period using the AWS CLI, see AWS CLI.

Turning Performance Insights on and off for Aurora

You can turn on Performance Insights for your DB cluster when you create it. If needed, you can turn it off later at the instance level for any instance in your DB cluster. Turning Performance Insights on and off doesn't cause downtime, a reboot, or a failover.

> **Note**
>
> Performance Schema is an optional performance tool used by Aurora MySQL. If you turn Performance Schema on or off, you need to reboot. If you turn Performance Insights on or off, however, you don't need to reboot. For more information, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

If you use Performance Insights with Aurora global databases, turn on Performance Insights individually for the DB instances in each AWS Region. For details, see Monitoring an Amazon Aurora global database with Amazon RDS Performance Insights.

The Performance Insights agent consumes limited CPU and memory on the DB host. When the DB load is high, the agent limits the performance impact by collecting data less frequently.

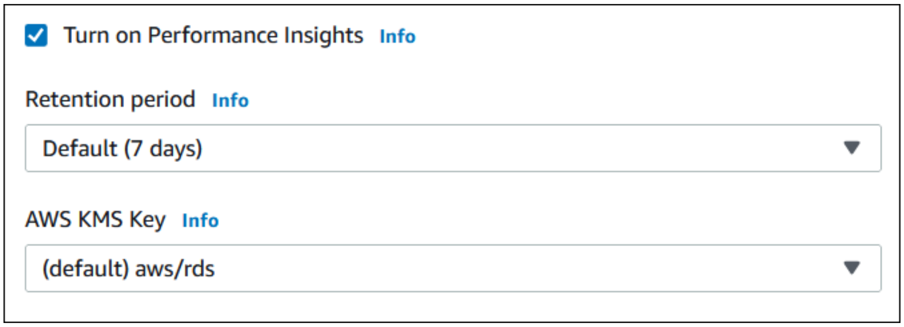
Console

In the console, you can turn Performance Insights on or off when you create a DB cluster. You can modify a DB instance in the cluster to turn Performance Insights on or off for the instance.

Turning Performance Insights on or off when creating a DB cluster

When you create a new DB cluster, turn on Performance Insights by choosing **Enable Performance Insights** in the **Performance Insights** section. Or choose **Disable Performance Insights**. To create a DB cluster, follow the instructions for your DB engine in Creating an Amazon Aurora DB cluster.

The following screenshot shows the **Performance Insights** section.



☑ Turn on Performance Insights Info

Retention period Info

Default (7 days) ▼

AWS KMS Key Info

(default) aws/rds ▼

If you choose **Enable Performance Insights**, you have the following options:

- **Retention** – The amount of time to retain Performance Insights data. The retention setting in the free tier is **Default (7 days)**. To retain your performance data for longer, specify 1–24 months. For more information about retention periods, see Pricing and data retention for Performance Insights.
- **AWS KMS key** – Specify your AWS KMS key. Performance Insights encrypts all potentially sensitive data using your KMS key. Data is encrypted in flight and at rest. For more information, see Configuring an AWS KMS policy for Performance Insights.

Turning Performance Insights on or off when modifying a DB instance in your DB cluster

In the console, you can modify a DB instance in your DB cluster to turn Performance Insights on or off. You can't turn Performance Insights on or off at the cluster level: you must do it for each instance in the cluster.

To turn Performance Insights on or off for a DB instance in your DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose a DB instance, and choose **Modify**.
4. In the **Performance Insights** section, choose either **Enable Performance Insights** or **Disable Performance Insights**.

   If you choose **Enable Performance Insights**, you have the following options:

   - **Retention** – The amount of time to retain Performance Insights data. The retention setting in the free tier is **Default (7 days)**. To retain your performance data for longer, specify 1–24 months. For more information about retention periods, see Pricing and data retention for Performance Insights.
   - **AWS KMS key** – Specify your KMS key. Performance Insights encrypts all potentially sensitive data using your KMS key. Data is encrypted in flight and at rest. For more information, see Encrypting Amazon Aurora resources.
5. Choose **Continue**.

6. For **Scheduling of Modifications**, choose Apply immediately. If you choose Apply during the next scheduled maintenance window, your instance ignores this setting and turns on Performance Insights immediately.
7. Choose **Modify instance**.

AWS CLI

When you use the create-db-instance AWS CLI command, turn on Performance Insights by specifying `--enable-performance-insights`. Or turn off Performance Insights by specifying `--no-enable-performance-insights`.

You can also specify these values using the following AWS CLI commands:

- create-db-instance-read-replica
- modify-db-instance
- restore-db-instance-from-s3

The following procedure describes how to turn Performance Insights on or off for an existing DB instance in your DB cluster using the AWS CLI.

To turn Performance Insights on or off for a DB instance in your DB cluster using the AWS CLI

- Call the modify-db-instance AWS CLI command and supply the following values:
  - `--db-instance-identifier` – The name of the DB instance in your DB cluster.
  - `--enable-performance-insights` to turn on or `--no-enable-performance-insights` to turn off

  The following example turns on Performance Insights for `sample-db-instance`.

  For Linux, macOS, or Unix:

  ```
  aws rds modify-db-instance \
      --db-instance-identifier sample-db-instance \
      --enable-performance-insights
  ```

  For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier sample-db-instance ^
    --enable-performance-insights
```

When you turn on Performance Insights in the CLI, you can optionally specify the number of days to retain Performance Insights data with the `--performance-insights-retention-period` option. You can specify 7, *month* * 31 (where *month* is a number from 1–23), or 731. For example, if you want to retain your performance data for 3 months, specify 93, which is 3 * 31. The default is 7 days. For more information about retention periods, see [Pricing and data retention for Performance Insights](#).

The following example turns on Performance Insights for `sample-db-instance` and specifies that Performance Insights data is retained for 93 days (3 months).

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier sample-db-instance \
    --enable-performance-insights \
    --performance-insights-retention-period 93
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier sample-db-instance ^
    --enable-performance-insights ^
    --performance-insights-retention-period 93
```

If you specify a retention period such as 94 days, which isn't a valid value, RDS issues an error.

```
An error occurred (InvalidParameterValue) when calling the CreateDBInstance operation:
Invalid Performance Insights retention period. Valid values are: [7, 31, 62, 93, 124,
 155, 186, 217,
248, 279, 310, 341, 372, 403, 434, 465, 496, 527, 558, 589, 620, 651, 682, 713, 731]
```

RDS API

When you create a new DB instance in your DB cluster using the [CreateDBInstance](#) operation Amazon RDS API operation, turn on Performance Insights by setting

EnablePerformanceInsights to True. To turn off Performance Insights, set EnablePerformanceInsights to False.

You can also specify the EnablePerformanceInsights value using the following API operations:

- ModifyDBInstance
- CreateDBInstanceReadReplica
- RestoreDBInstanceFromS3

When you turn on Performance Insights, you can optionally specify the amount of time, in days, to retain Performance Insights data with the PerformanceInsightsRetentionPeriod parameter. You can specify 7, *month* * 31 (where *month* is a number from 1–23), or 731. For example, if you want to retain your performance data for 3 months, specify 93, which is 3 * 31. The default is 7 days. For more information about retention periods, see Pricing and data retention for Performance Insights.

Turning on the Performance Schema for Performance Insights on Aurora MySQL

The Performance Schema is an optional feature for monitoring Aurora MySQL runtime performance at a low level of detail. The Performance Schema is designed to have minimal impact on database performance. Performance Insights is a separate feature that you can use with or without the Performance Schema.

Topics

Overview of the Performance Schema

The Performance Schema monitors events in Aurora MySQL databases. An *event* is a database server action that consumes time and has been instrumented so that timing information can be collected. Examples of events include the following:

- Function calls
- Waits for the operating system
- Stages of SQL execution
- Groups of SQL statements

The `PERFORMANCE_SCHEMA` storage engine is a mechanism for implementing the Performance Schema feature. This engine collects event data using instrumentation in the database source code. The engine stores events in memory-only tables in the `performance_schema` database. You can query `performance_schema` just as you can query any other tables. For more information, see MySQL Performance Schema in the *MySQL Reference Manual*.

Performance Insights and the Performance Schema

Performance Insights and the Performance Schema are separate features, but they are connected. The behavior of Performance Insights for Aurora MySQL depends on whether the Performance Schema is turned on, and if so, whether Performance Insights manages the Performance Schema automatically. The following table describes the behavior.

Performance Schema turned on	Performance Insights management mode	Performance Insights behavior
Yes	Automatic	• Collects detailed, low-level monitoring information • Collects active session metrics every second • Displays DB load categorized by detailed wait events, which you can use to identify bottlenecks
Yes	Manual	•

Performance Schema turned on	Performance Insights management mode	Performance Insights behavior
		Collects wait events and per-SQL metrics • Collects active session metrics every five seconds instead of every second • Reports user states such as inserting and sending, which don't help you identify bottlenecks
No	N/A	• Doesn't collect wait events, per-SQL metrics, or other detailed, low-level monitoring information • Collects active session metrics every five seconds instead of every second • Reports user states such as inserting and sending, which don't help you identify bottlenecks

Automatic management of the Performance Schema by Performance Insights

When you create an Aurora MySQL DB instance with Performance Insights turned on, the Performance Schema is also turned on. In this case, Performance Insights automatically manages your Performance Schema parameters. This is the recommended configuration.

When Performance Insights manages the Performance Schema automatically, the **Source** of `performance_schema` is `System`.

> **Note**
>
> Automatic management of the Performance Schema isn't supported for the t4g.medium instance class.

You can also manage the Performance Schema manually. If you choose this option, set the parameters according to the values in the following table.

Parameter name	Parameter value
performance_schema	1 (**Source** column has the value Modified)
performance-schema-consumer-events-waits-current	ON
performance-schema-instrument	wait/%=ON
performance_schema_consumer_global_instrumentation	1
performance_schema_consumer_thread_instrumentation	1

If you change the performance_schema parameter value manually, and then later want to change to automatic management, see Configuring the Performance Schema for automatic management.

⚠ **Important**

When Performance Insights turns on the Performance Schema, it doesn't change the parameter group values. However, the values are changed on the DB instances that are running. The only way to see the changed values is to run the SHOW GLOBAL VARIABLES command.

Effect of a reboot on the Performance Schema

Performance Insights and the Performance Schema differ in their requirements for DB instance reboots:

Performance Schema

To turn this feature on or off, you must reboot the DB instance.

Performance Insights

To turn this feature on or off, you don't need to reboot the DB instance.

If the Performance Schema isn't currently turned on, and you turn on Performance Insights without rebooting the DB instance, the Performance Schema won't be turned on.

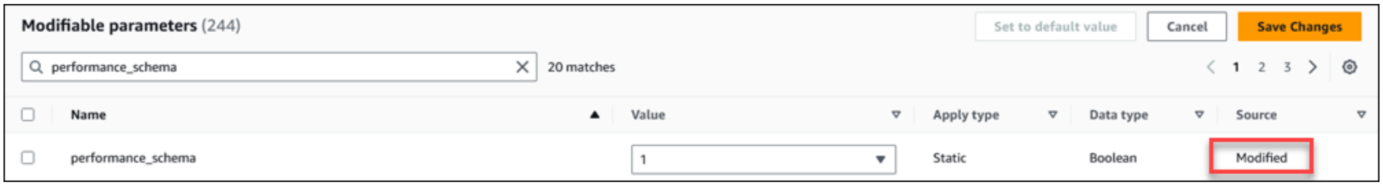
Determining whether Performance Insights is managing the Performance Schema

To find out whether Performance Insights is currently managing the Performance Schema for major engine versions 5.6, 5.7, and 8.0, review the following table.

Setting of performance_schema parameter	Setting of the Source column	Performance Insights is managing the Performance Schema?
`0`	`System`	Yes
`0` or `1`	`Modified`	No

To determine whether Performance Insights is managing the Performance Schema automatically

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Parameter groups**.
3. Select the name of the parameter group for your DB instance.
4. Enter **`performance_schema`** in the search bar.
5. Check whether **Source** is the system default and **Values** is **0**. If so, Performance Insights is managing the Performance Schema automatically. If not, Performance Insights isn't managing the Performance Schema automatically.



Modifiable parameters (244)

performance_schema — 20 matches

Name	Value	Apply type	Data type	Source
performance_schema	1	Static	Boolean	Modified

Configuring the Performance Schema for automatic management

Assume that Performance Insights is turned on for your DB instance but isn't currently managing the Performance Schema. If you want to allow Performance Insights to manage the Performance Schema automatically, complete the following steps.

To configure the Performance Schema for automatic management

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Parameter groups**.
3. Select the name of the parameter group for your DB instance.
4. Enter **`performance_schema`** in the search bar.
5. Select the `performance_schema` parameter.
6. Choose **Edit parameters**.
7. Select the `performance_schema` parameter.
8. In **Values**, choose **0**.
9. Choose **Save changes**.
10. Reboot the DB instance.

> ⚠ **Important**
>
> Whenever you turn the Performance Schema on or off, make sure to reboot the DB instance.

For more information about modifying instance parameters, see Modifying parameters in a DB parameter group. For more information about the dashboard, see Analyzing metrics with the Performance Insights dashboard. For more information about the MySQL performance schema, see MySQL 8.0 Reference Manual.

Configuring access policies for Performance Insights

To access Performance Insights, a principal must have the appropriate permissions from AWS Identity and Access Management (IAM). You can grant access in the following ways:

- Attach the `AmazonRDSPerformanceInsightsReadOnly` managed policy to a permission set or role to access all read-only operations of the Performance Insights API.
- Attach the `AmazonRDSPerformanceInsightsFullAccess` managed policy to a permission set or role to access all operations of the Performance Insights API.
- Create a custom IAM policy and attach it to a permission set or role.

If you specified a customer managed key when you turned on Performance Insights, make sure that users in your account have the `kms:Decrypt` and `kms:GenerateDataKey` permissions on the AWS KMS key

Attaching the AmazonRDSPerformanceInsightsReadOnly policy to an IAM principal

`AmazonRDSPerformanceInsightsReadOnly` is an AWS managed policy that grants access to all read-only operations of the Amazon RDS Performance Insights API.

If you attach `AmazonRDSPerformanceInsightsReadOnly` to a permission set or role, the recipient can use Performance Insights with other console features.

For more information, see AWS managed policy: AmazonRDSPerformanceInsightsReadOnly.

Attaching the AmazonRDSPerformanceInsightsFullAccess policy to an IAM principal

`AmazonRDSPerformanceInsightsFullAccess` is an AWS managed policy that grants access to all operations of the Amazon RDS Performance Insights API.

If you attach `AmazonRDSPerformanceInsightsFullAccess` to a permission set or role, the recipient can use Performance Insights with other console features.

For more information, see AWS managed policy: AmazonRDSPerformanceInsightsFullAccess.

Creating a custom IAM policy for Performance Insights

For users who don't have either the `AmazonRDSPerformanceInsightsReadOnly` or `AmazonRDSPerformanceInsightsFullAccess` policy, you can grant access to Performance Insights by creating or modifying a user-managed IAM policy. When you attach the policy to an IAM permission set or role, the recipient can use Performance Insights.

To create a custom policy

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Create Policy** page, choose the **JSON** option.
5. Copy and paste the text provided in the *JSON policy document* section in the *AWS Managed Policy Reference Guide* for AmazonRDSPerformanceInsightsReadOnly or AmazonRDSPerformanceInsightsFullAccess policy.
6. Choose **Review policy**.
7. Provide a name for the policy and optionally a description, and then choose **Create policy**.

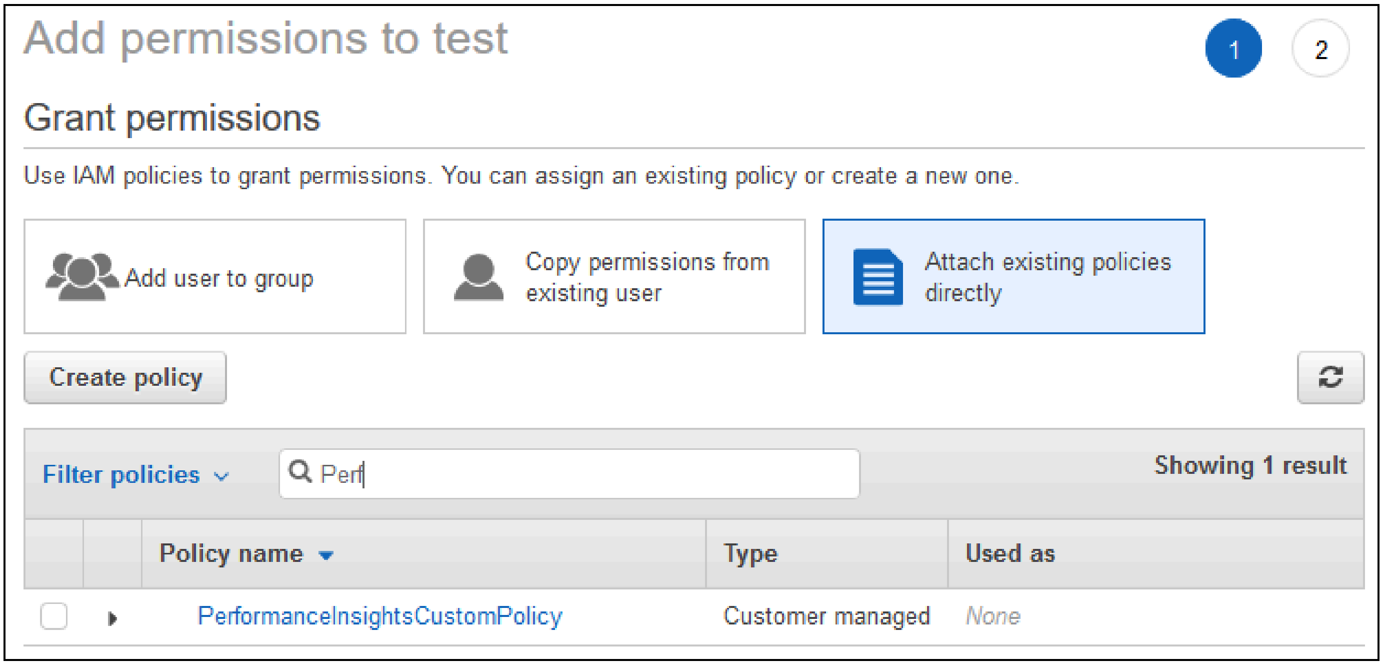
You can now attach the policy to a permission set or role. The following procedure assumes that you already have a user available for this purpose.

To attach the policy to a user

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**.
3. Choose an existing user from the list.

> ⚠ **Important**
>
> To use Performance Insights, make sure that you have access to Amazon RDS in addition to the custom policy. For example, the `AmazonRDSPerformanceInsightsReadOnly` predefined policy provides read-only access to Amazon RDS. For more information, see Managing access using policies.

4. On the **Summary** page, choose **Add permissions**.
5. Choose **Attach existing policies directly**. For **Search**, type the first few characters of your policy name, as shown in the following image.



Grant permissions

Use IAM policies to grant permissions. You can assign an existing policy or create a new one.

Search: Perf

		Policy name	Type	Used as
□	▸	PerformanceInsightsCustomPolicy	Customer managed	None

6. Choose your policy, and then choose **Next: Review**.

7. Choose **Add permissions**.

Configuring an AWS KMS policy for Performance Insights

Performance Insights uses an AWS KMS key to encrypt sensitive data. When you enable Performance Insights through the API or the console, you can do either of the following:

- Choose the default AWS managed key.

  Amazon RDS uses the AWS managed key for your new DB instance. Amazon RDS creates an AWS managed key for your AWS account. Your AWS account has a different AWS managed key for Amazon RDS for each AWS Region.

- Choose a customer managed key.

  If you specify a customer managed key, users in your account that call the Performance Insights API need the `kms:Decrypt` and `kms:GenerateDataKey` permissions on the KMS key. You can configure these permissions through IAM policies. However, we recommend that you manage these permissions through your KMS key policy. For more information, see Key policies in AWS KMS in the *AWS Key Management Service Developer Guide*.

Example

The following example shows how to add statements to your KMS key policy. These statements allow access to Performance Insights. Depending on how you use the KMS key, you might want to change some restrictions. Before adding statements to your policy, remove all comments.

```
{
"Version" : "2012-10-17",
 "Id" : "your-policy",
 "Statement" : [ {
    //This represents a statement that currently exists in your policy.
 }
 ....,
 //Starting here, add new statement to your policy for Performance Insights.
 //We recommend that you add one new statement for every RDS instance
{
    "Sid" : "Allow viewing RDS Performance Insights",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            //One or more principals allowed to access Performance Insights
            "arn:aws:iam::444455556666:role/Role1"
        ]
    },
    "Action": [
        "kms:Decrypt",
        "kms:GenerateDataKey"
    ],
    "Resource": "*",
    "Condition" : {
        "StringEquals" : {
            //Restrict access to only RDS APIs (including Performance Insights).
            //Replace region with your AWS Region.
            //For example, specify us-west-2.
            "kms:ViaService" : "rds.region.amazonaws.com"
        },
        "ForAnyValue:StringEquals": {
            //Restrict access to only data encrypted by Performance Insights.
            "kms:EncryptionContext:aws:pi:service": "rds",
            "kms:EncryptionContext:service": "pi",

            //Restrict access to a specific RDS instance.
            //The value is a DbiResourceId.
```

```
                "kms:EncryptionContext:aws:rds:db-id": "db-AAAAABBBBBCCCCDDDDDEEEEE"
            }
        }
    }
```

How Performance Insights uses AWS KMS customer managed key

Performance Insights uses customer managed keys to encrypt sensitive data. When you turn on Performance Insights, you can provide an AWS KMS key through the API. Performance Insights creates KMS permissions on this key. It uses the key and performs the necessary operations to process sensitive data. Sensitive data includes fields such as user, database, application, and SQL query text. Performance Insights ensures that the data remains encrypted both at rest and in-flight.

How Performance Insights IAM works with AWS KMS

IAM gives permissions to specific APIs. Performance Insights has the following public APIs, which you can restrict using IAM policies:

- `DescribeDimensionKeys`
- `GetDimensionKeyDetails`
- `GetResourceMetadata`
- `GetResourceMetrics`
- `ListAvailableResourceDimensions`
- `ListAvailableResourceMetrics`

You can use the following API requests to get sensitive data.

- `DescribeDimensionKeys`
- `GetDimensionKeyDetails`
- `GetResourceMetrics`

When you use the API to get sensitive data, Performance Insights leverages the caller's credentials. This check ensures that access to sensitive data is limited to those with access to the KMS key.

When calling these APIs, you need permissions to call the API through the IAM policy and permissions to invoke the `kms:decrypt` action through the AWS KMS key policy.

The GetResourceMetrics API can return both sensitive and non-sensitive data. The request parameters determine whether the response should include sensitive data. The API returns sensitive data when the request includes a sensitive dimension in either the filter or group-by parameters.

For more information about the dimensions that you can use with the GetResourceMetrics API, see DimensionGroup.

Example Examples

The following example requests the sensitive data for the db.user group:

```
POST / HTTP/1.1
Host: <Hostname>
Accept-Encoding: identity
X-Amz-Target: PerformanceInsightsv20180227.GetResourceMetrics
Content-Type: application/x-amz-json-1.1
User-Agent: <UserAgentString>
X-Amz-Date: <Date>
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
 Signature=<Signature>
Content-Length: <PayloadSizeBytes>
{
  "ServiceType": "RDS",
  "Identifier": "db-ABC1DEFGHIJKL2MNOPQRSTUV3W",
  "MetricQueries": [
    {
      "Metric": "db.load.avg",
      "GroupBy": {
        "Group": "db.user",
        "Limit": 2
      }
    }
  ],
  "StartTime": 1693872000,
  "EndTime": 1694044800,
  "PeriodInSeconds": 86400
}
```

Example

The following example requests the non-sensitive data for the `db.load.avg` metric:

```
POST / HTTP/1.1
Host: <Hostname>
Accept-Encoding: identity
X-Amz-Target: PerformanceInsightsv20180227.GetResourceMetrics
Content-Type: application/x-amz-json-1.1
User-Agent: <UserAgentString>
X-Amz-Date: <Date>
Authorization: AWS4-HMAC-SHA256 Credential=<Credential>, SignedHeaders=<Headers>,
 Signature=<Signature>
Content-Length: <PayloadSizeBytes>
{
    "ServiceType": "RDS",
    "Identifier": "db-ABC1DEFGHIJKL2MNOPQRSTUV3W",
    "MetricQueries": [
        {
            "Metric": "db.load.avg"
        }
    ],
    "StartTime": 1693872000,
    "EndTime": 1694044800,
    "PeriodInSeconds": 86400
}
```

Granting fine-grained access for Performance Insights

Fine-grained access control offers additional ways of controlling access to Performance Insights. This access control can allow or deny access to individual dimensions for `GetResourceMetrics`, `DescribeDimensionKeys`, and `GetDimensionKeyDetails` Performance Insights actions. To use fine-grained access, specify dimensions in the IAM policy by using condition keys. The evaluation of the access follows the IAM policy evaluation logic. For more information, see Policy evaluation logic in the *IAM User Guide*. If the IAM policy statement doesn't specify any dimension, then the statement controls access to all the dimensions for the specified action. For the list of available dimensions, see DimensionGroup.

To find out the dimensions that your credentials are authorized to access, use the `AuthorizedActions` parameter in `ListAvailableResourceDimensions` and specify the action. The allowed values for `AuthorizedActions` are as follows:

- `GetResourceMetrics`
- `DescribeDimensionKeys`
- `GetDimensionKeyDetails`

For example, if you specify `GetResourceMetrics` to the `AuthorizedActions` parameter, `ListAvailableResourceDimensions` returns the list of dimensions that the `GetResourceMetrics` action is authorized to access. If you specify multiple actions in the `AuthorizedActions` parameter, then `ListAvailableResourceDimensions` returns an intersection of dimensions that those actions are authorized to access.

Example

The following example provides access to the specified dimensions for `GetResourceMetrics` and `DescribeDimensionKeys` actions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowToDiscoverDimensions",
            "Effect": "Allow",
            "Action": [
                "pi:ListAvailableResourceDimensions"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ]
        },
        {
            "Sid": "SingleAllow",
            "Effect": "Allow",
            "Action": [
                "pi:GetResourceMetrics",
                "pi:DescribeDimensionKeys"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    // only these dimensions are allowed. Dimensions not included in
                    // a policy with "Allow" effect will be denied
                    "pi:Dimensions": [
                        "db.sql_tokenized.id",
                        "db.sql_tokenized.statement"
                    ]
                }
            }
        }


    ]
}
```

The following is the response for the requested dimension:

```
 // ListAvailableResourceDimensions API
// Request
{
    "ServiceType": "RDS",
    "Identifier": "db-ABC1DEFGHIJKL2MNOPQRSTUV3W",
    "Metrics": [ "db.load" ],
    "AuthorizedActions": ["DescribeDimensionKeys"]
}

// Response
{
    "MetricDimensions": [ {
        "Metric": "db.load",
        "Groups": [
            {
                "Group": "db.sql_tokenized",
                "Dimensions": [
                    { "Identifier": "db.sql_tokenized.id" },
                  //  { "Identifier": "db.sql_tokenized.db_id" }, // not included
 because not allows in the IAM Policy
```

```
                    { "Identifier": "db.sql_tokenized.statement" }
                ]
            }

        ] }
    ]
}
```

The following example specifies one allow and two deny access for the dimensions.

```
{
    "Version": "2012-10-17",
    "Statement": [
          {
            "Sid": "AllowToDiscoverDimensions",
            "Effect": "Allow",
            "Action": [
                "pi:ListAvailableResourceDimensions"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ]
          },

          {
            "Sid": "O01AllowAllWithoutSpecifyingDimensions",
            "Effect": "Allow",
            "Action": [
                "pi:GetResourceMetrics",
                "pi:DescribeDimensionKeys"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ]
        },

        {
            "Sid": "O01DenyAppDimensionForAll",
            "Effect": "Deny",
            "Action": [
                "pi:GetResourceMetrics",
                "pi:DescribeDimensionKeys"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "pi:Dimensions": [
                        "db.application.name"
                    ]
                }
            }
        },

        {
            "Sid": "O01DenySQLForGetResourceMetrics",
            "Effect": "Deny",
            "Action": [
                "pi:GetResourceMetrics"
            ],
            "Resource": [
                "arn:aws:pi:us-east-1:123456789012:metrics/rds/db-
ABC1DEFGHIJKL2MNOPQRSTUV3W"
            ],
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "pi:Dimensions": [
                        "db.sql_tokenized.statement"
                    ]
                }
            }
        }
    ]
}
```

The following are the responses for the requested dimensions:

```
   // ListAvailableResourceDimensions API
// Request
{
    "ServiceType": "RDS",
    "Identifier": "db-ABC1DEFGHIJKL2MNOPQRSTUV3W",
    "Metrics": [ "db.load" ],
    "AuthorizedActions": ["GetResourceMetrics"]
}

// Response
{
    "MetricDimensions": [ {
        "Metric": "db.load",
        "Groups": [
            {
                "Group": "db.application",
                "Dimensions": [

                  // removed from response because denied by the IAM Policy
                  //  { "Identifier": "db.application.name" }
                ]
            },
            {
                "Group": "db.sql_tokenized",
                "Dimensions": [
                    { "Identifier": "db.sql_tokenized.id" },
                    { "Identifier": "db.sql_tokenized.db_id" },

                  // removed from response because denied by the IAM Policy
                  //  { "Identifier": "db.sql_tokenized.statement" }
                ]
            },
            ...
        ] }
    ]
}
```

```

// ListAvailableResourceDimensions API
// Request
{
    "ServiceType": "RDS",
    "Identifier": "db-ABC1DEFGHIJKL2MNOPQRSTUV3W",
    "Metrics": [ "db.load" ],
    "AuthorizedActions": ["DescribeDimensionKeys"]
}
```

```
// Response
{
    "MetricDimensions": [ {
        "Metric": "db.load",
        "Groups": [
            {
                "Group": "db.application",
                "Dimensions": [
                  // removed from response because denied by the IAM Policy
                  //  { "Identifier": "db.application.name" }
                ]
            },
            {
                "Group": "db.sql_tokenized",
                "Dimensions": [
                    { "Identifier": "db.sql_tokenized.id" },
                    { "Identifier": "db.sql_tokenized.db_id" },

                  // allowed for DescribeDimensionKeys because our IAM Policy
                  // denies it only for GetResourceMetrics
                    { "Identifier": "db.sql_tokenized.statement" }
                ]
            },
            ...
        ] }
    ]
}
```

Analyzing metrics with the Performance Insights dashboard

The Performance Insights dashboard contains database performance information to help you analyze and troubleshoot performance issues. On the main dashboard page, you can view information about the database load. You can "slice" DB load by dimensions such as wait events or SQL.

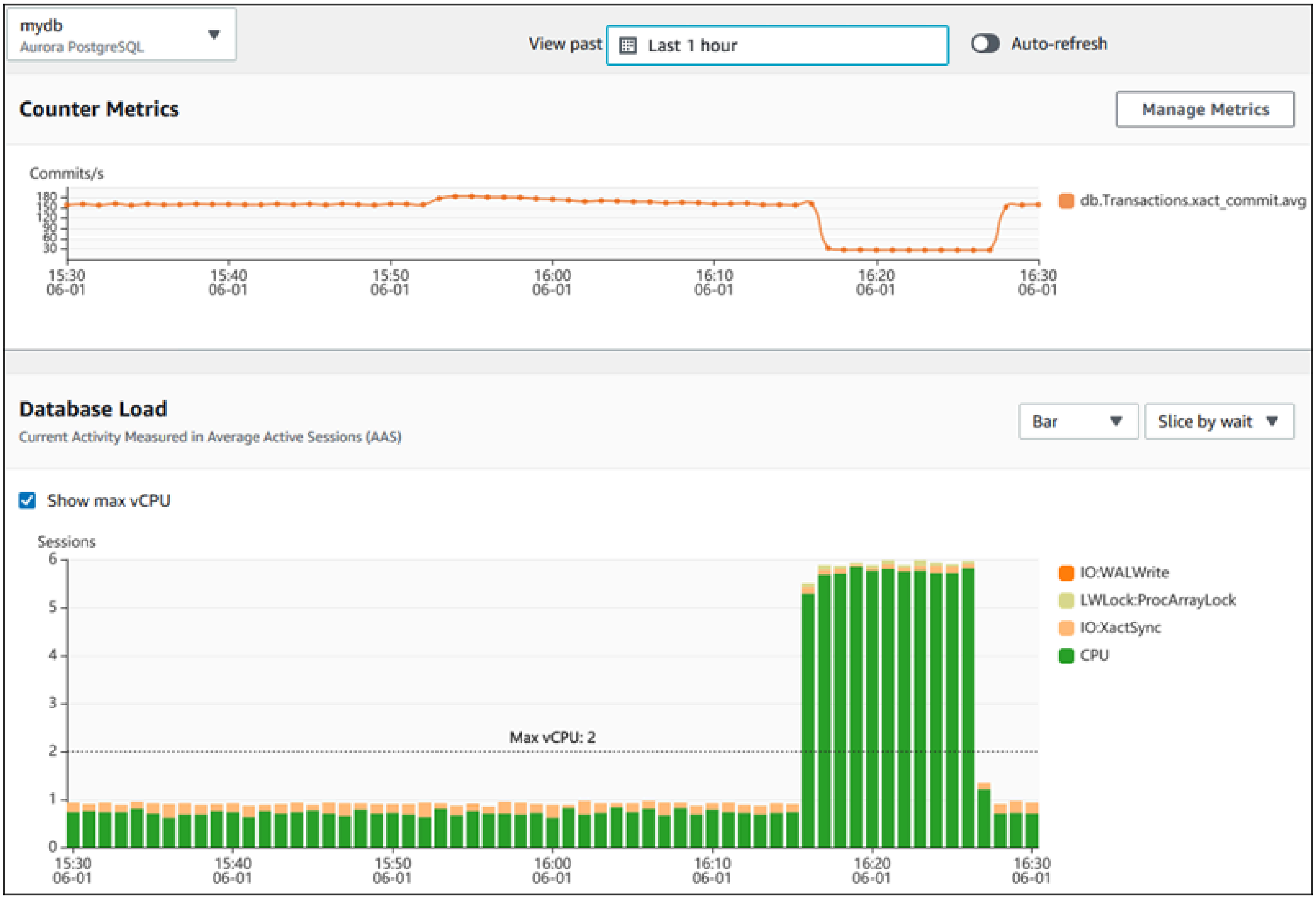
Performance Insights dashboard

- Overview of the Performance Insights dashboard
- Accessing the Performance Insights dashboard
- Analyzing DB load by wait events
- Analyzing database performance for a period of time

- Analyzing queries in the Performance Insights dashboard

Overview of the Performance Insights dashboard

The dashboard is the easiest way to interact with Performance Insights. The following example shows the dashboard for a MySQL DB instance.



Topics

- Time range filter
- Counter metrics chart
- Database load chart
- Top dimensions table

Time range filter

By default, the Performance Insights dashboard shows DB load for the last hour. You can adjust this range to be as short as 5 minutes or as long as 2 years. You can also select a custom relative range.

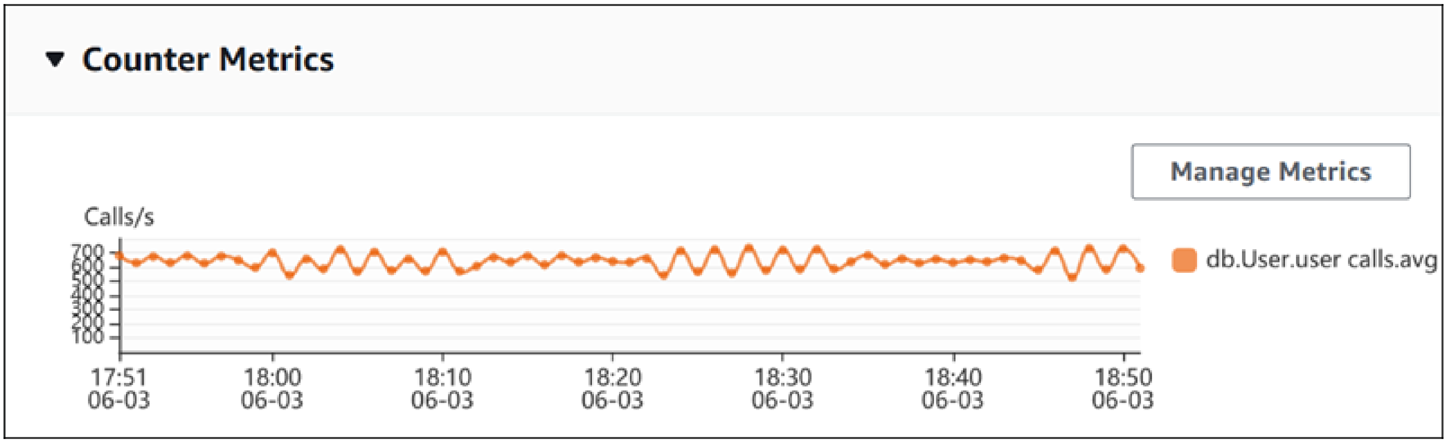
You can select an absolute range with a beginning and ending date and time. The following example shows the time range beginning at midnight on 4/11/22 and ending at 11:59 PM on 4/14/22.

Counter metrics chart

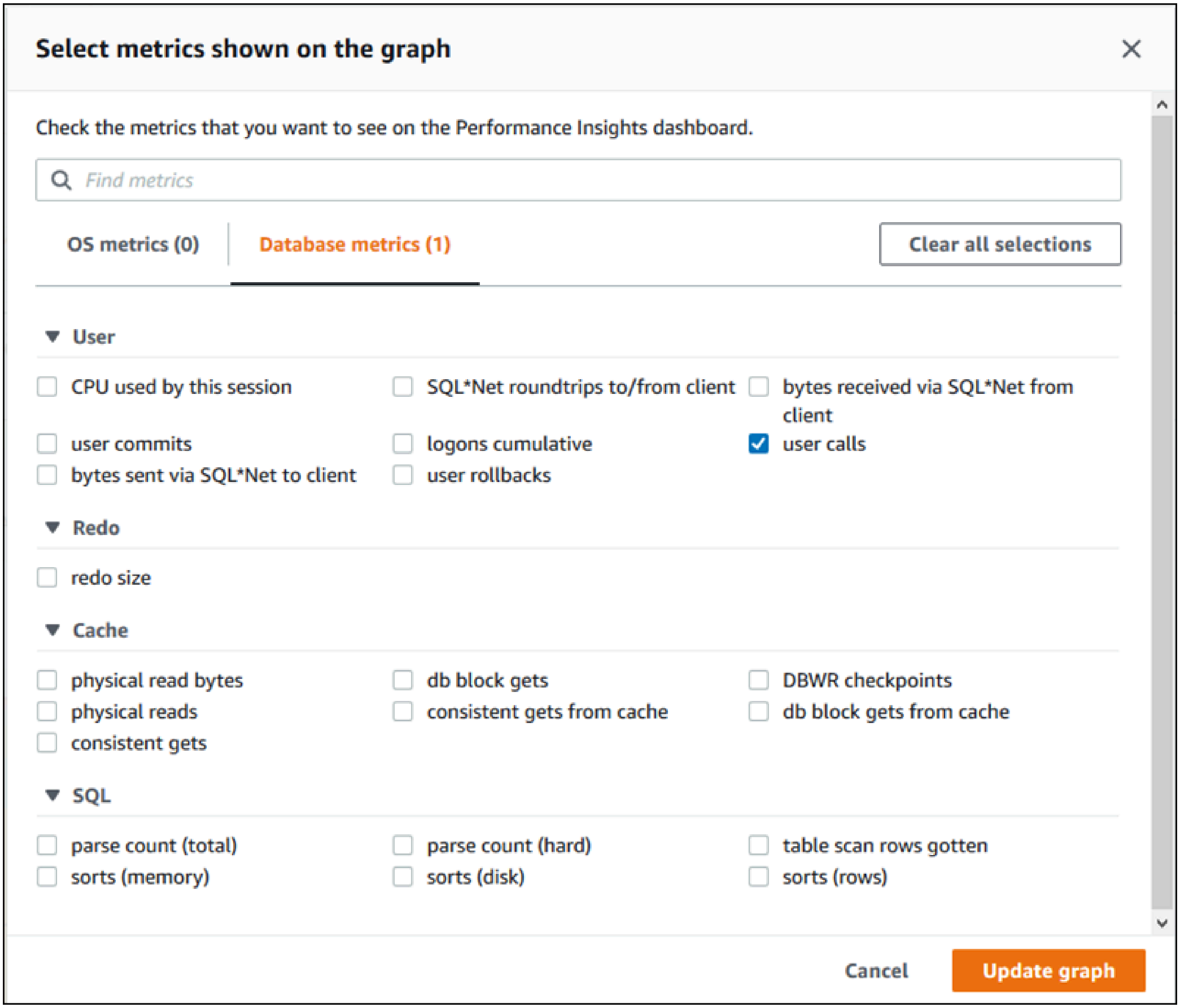
With counter metrics, you can customize the Performance Insights dashboard to include up to 10 additional graphs. These graphs show a selection of dozens of operating system and database performance metrics. You can correlate this information with DB load to help identify and analyze performance problems.

The **Counter metrics** chart displays data for performance counters. The default metrics depend on the DB engine:

- Aurora MySQL– `db.SQL.Innodb_rows_read.avg`
- Aurora PostgreSQL – `db.Transactions.xact_commit.avg`



To change the performance counters, choose **Manage Metrics**. You can select multiple **OS metrics** or **Database metrics**, as shown in the following screenshot. To see details for any metric, hover over the metric name.



Select metrics shown on the graph

Check the metrics that you want to see on the Performance Insights dashboard.

▼ User

☐ CPU used by this session
☐ user commits
☐ bytes sent via SQL*Net to client
☐ SQL*Net roundtrips to/from client
☐ logons cumulative
☐ user rollbacks
☐ bytes received via SQL*Net from client
☑ user calls

▼ Redo

☐ redo size

▼ Cache

☐ physical read bytes
☐ physical reads
☐ consistent gets
☐ db block gets
☐ consistent gets from cache
☐ DBWR checkpoints
☐ db block gets from cache

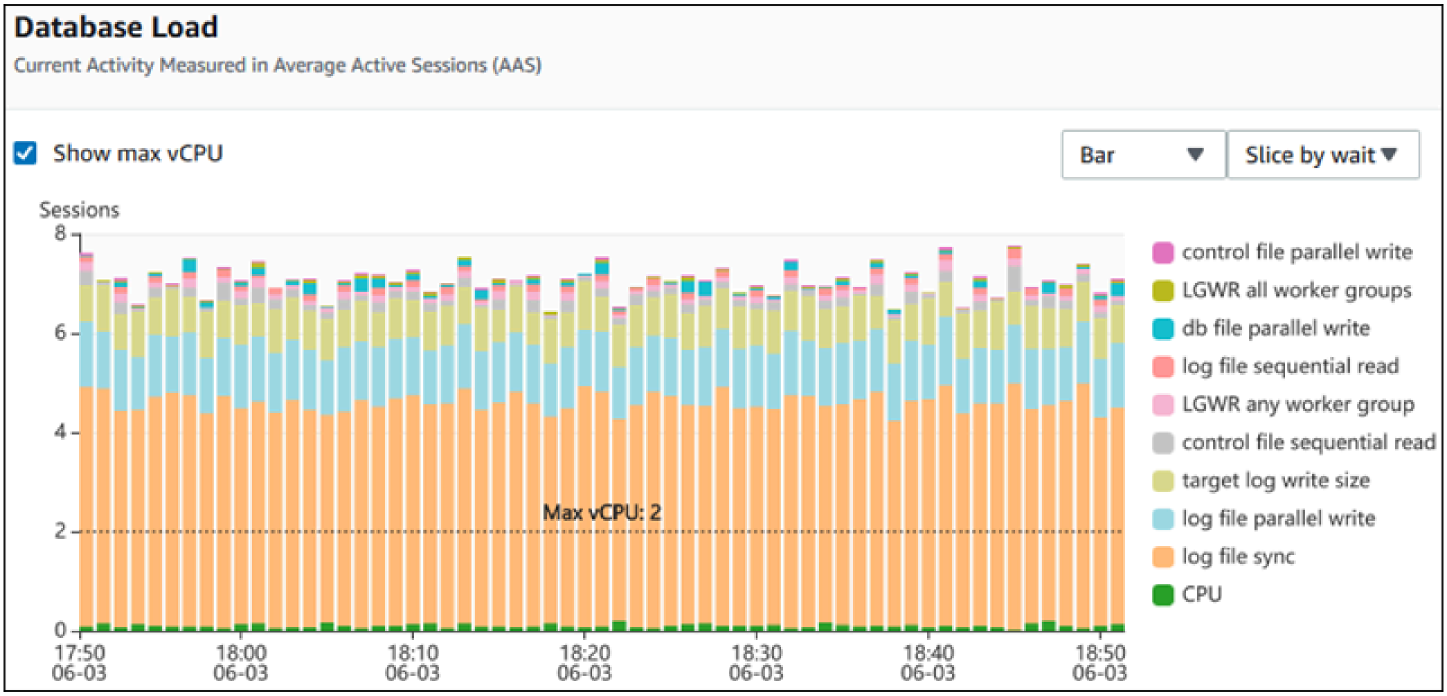
▼ SQL

☐ parse count (total)
☐ sorts (memory)
☐ parse count (hard)
☐ sorts (disk)
☐ table scan rows gotten
☐ sorts (rows)

For descriptions of the counter metrics that you can add for each DB engine, see Performance Insights counter metrics.

Database load chart

The **Database load** chart shows how the database activity compares to DB instance capacity as represented by the **Max vCPU** line. By default, the stacked line chart represents DB load as average active sessions per unit of time. The DB load is sliced (grouped) by wait states.

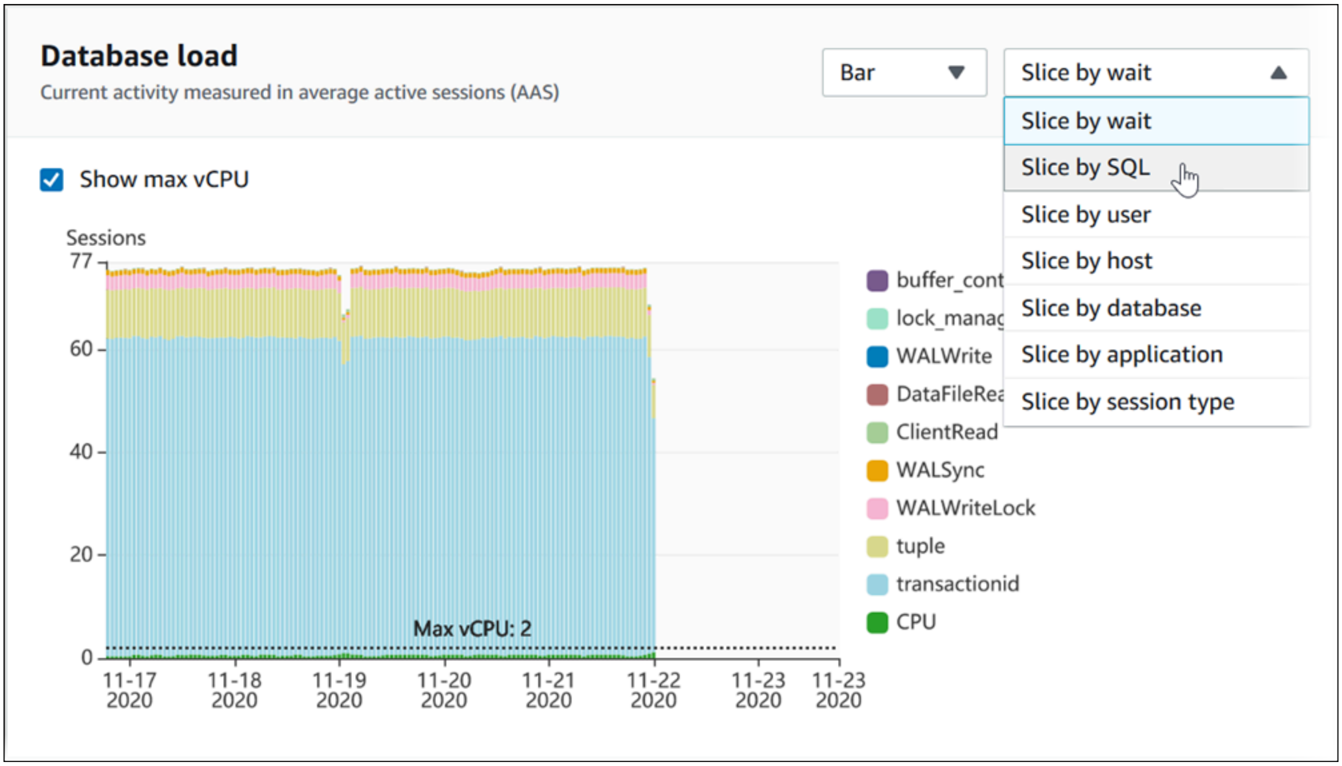


DB load sliced by dimensions

You can choose to display load as active sessions grouped by any supported dimensions. The following table shows which dimensions are supported for the different engines.

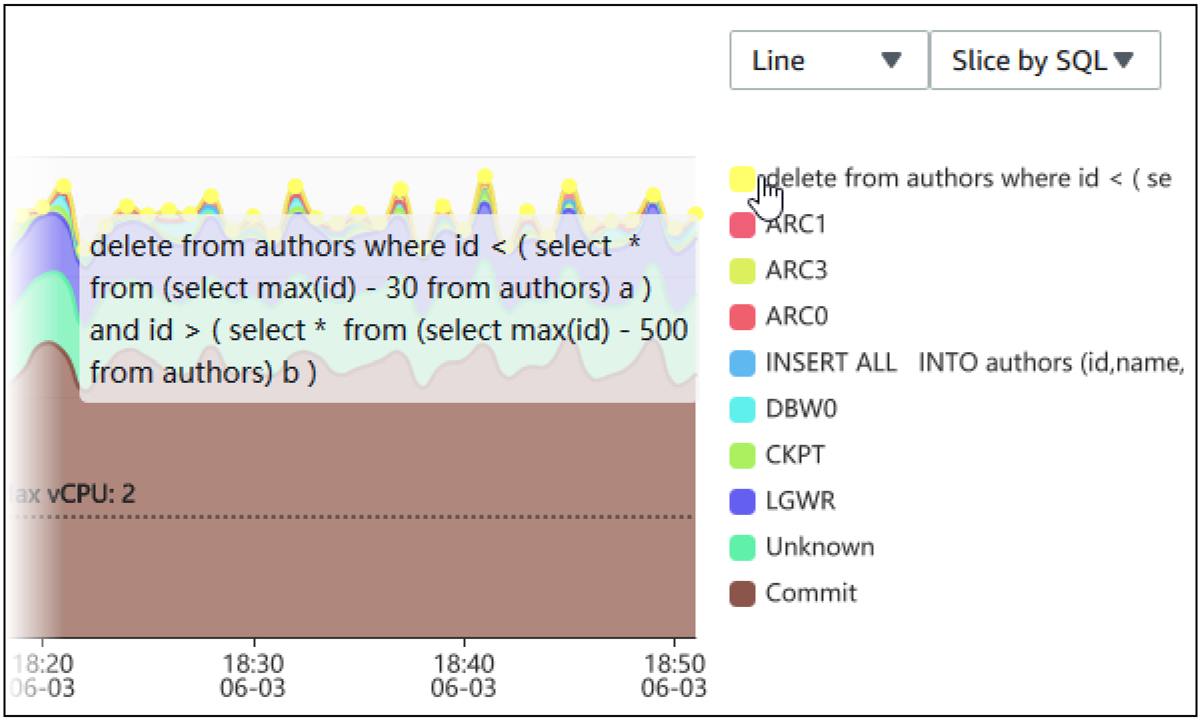
Dimension	Aurora PostgreSQL	Aurora MySQL
Host	Yes	Yes
SQL	Yes	Yes
User	Yes	Yes
Waits	Yes	Yes
Application	Yes	No
Database	Yes	Yes
Session type	Yes	No

The following image shows the dimensions for a PostgreSQL DB instance.



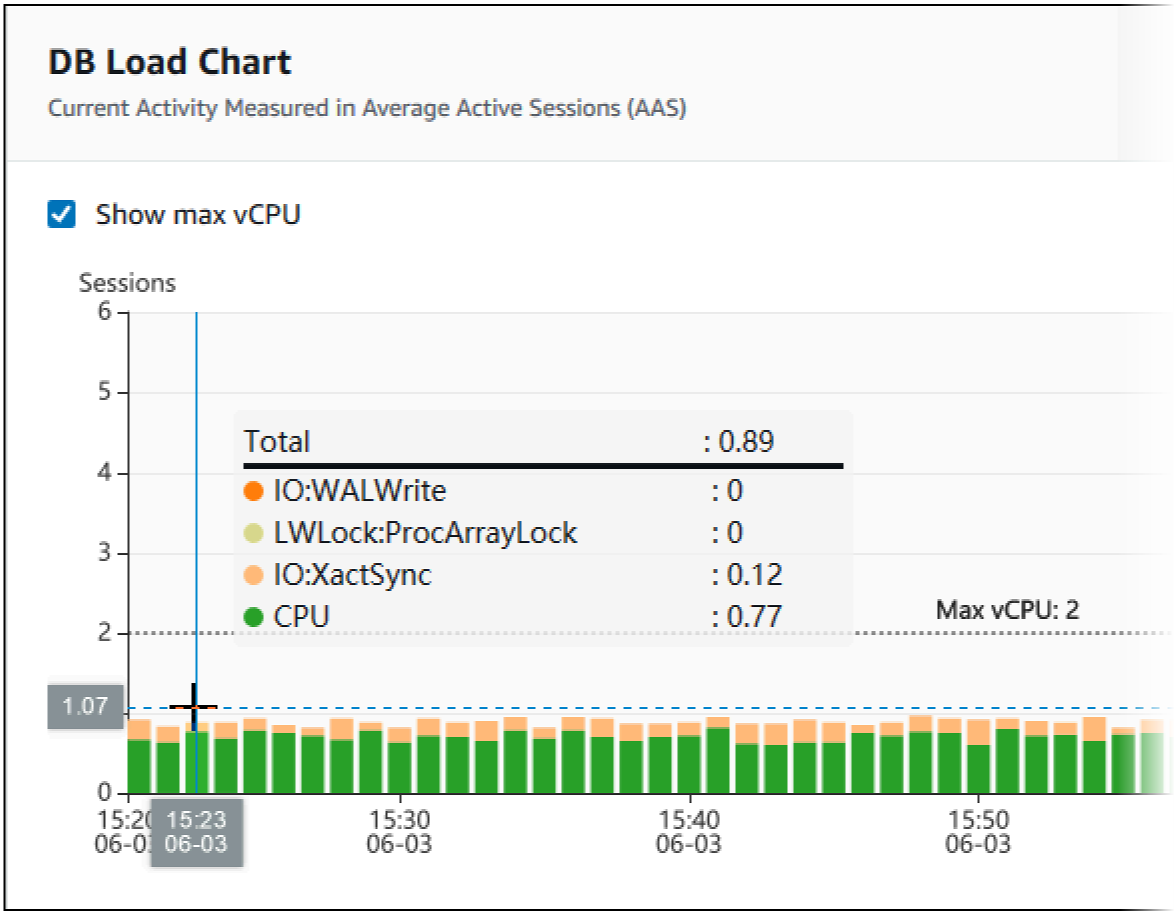
DB load details for a dimension item

To see details about a DB load item within a dimension, hover over the item name. The following image shows details for a SQL statement.



delete from authors where id < (select * from (select max(id) - 30 from authors) a) and id > (select * from (select max(id) - 500 from authors) b)

To see details for any item for the selected time period in the legend, hover over that item.



DB Load Chart

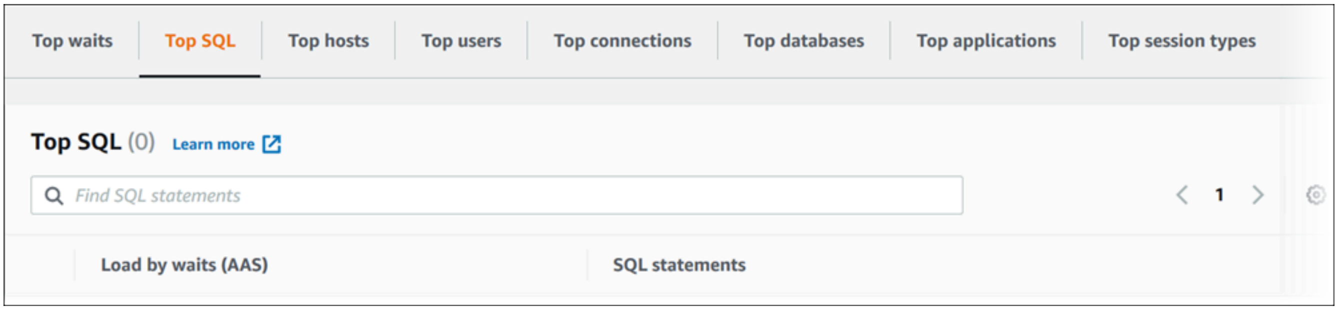
Current Activity Measured in Average Active Sessions (AAS)

☑ Show max vCPU

Total	: 0.89
● IO:WALWrite	: 0
● LWLock:ProcArrayLock	: 0
● IO:XactSync	: 0.12
● CPU	: 0.77

Top dimensions table

The Top dimensions table slices DB load by different dimensions. A dimension is a category or "slice by" for different characteristics of DB load. If the dimension is SQL, **Top SQL** shows the SQL statements that contribute the most to DB load.



Top SQL (0) Learn more

Load by waits (AAS)	SQL statements

Choose any of the following dimension tabs.

Tab	Description	Supported engines
Top SQL	The SQL statements that are currently running	All
Top waits	The event for which the database backend is waiting	All
Top hosts	The host name of the connected client	All
Top users	The user logged in to the database	All
Top applications	The name of the applicati on that is connected to the database	Aurora PostgreSQL only
Top session types	The type of the current session	Aurora PostgreSQL only

To learn how to analyze queries by using the **Top SQL** tab, see Overview of the Top SQL tab.

Accessing the Performance Insights dashboard

Amazon RDS provides a consolidated view of Performance Insights and CloudWatch metrics in the Performance Insights dashboard.

To access the Performance Insights dashboard, use the following procedure.

To view the Performance Insights dashboard in the AWS Management Console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.
4. Choose the default monitoring view in the displayed window.

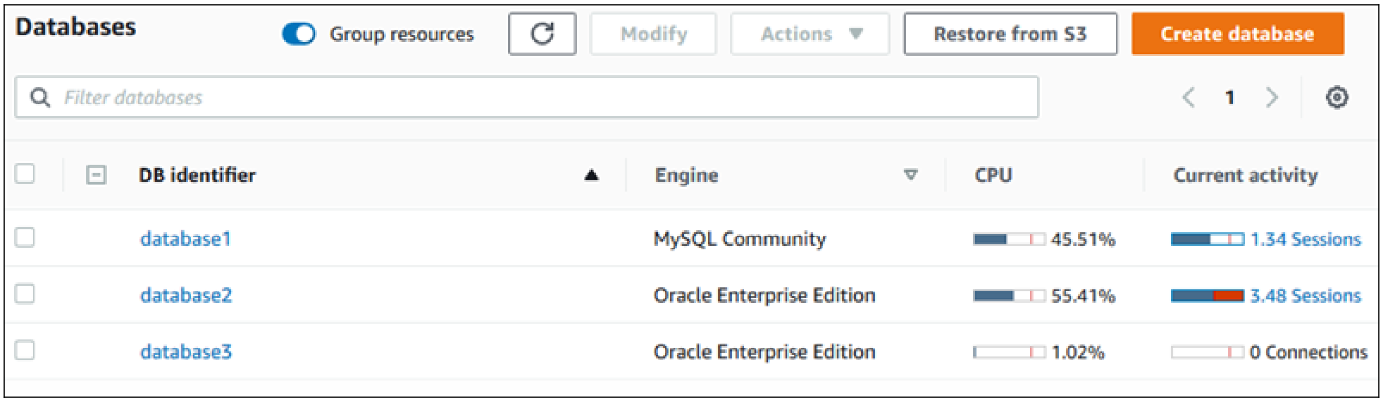
   - Select the **Performance Insights and CloudWatch metrics view (New)** option and choose **Continue** to view Performance Insights and CloudWatch metrics.

- Select the **Performance Insights view** option and choose **Continue** for the legacy monitoring view. Then, continue with this procedure.

> **Note**
>
> This view will be discontinued on December 15, 2023.

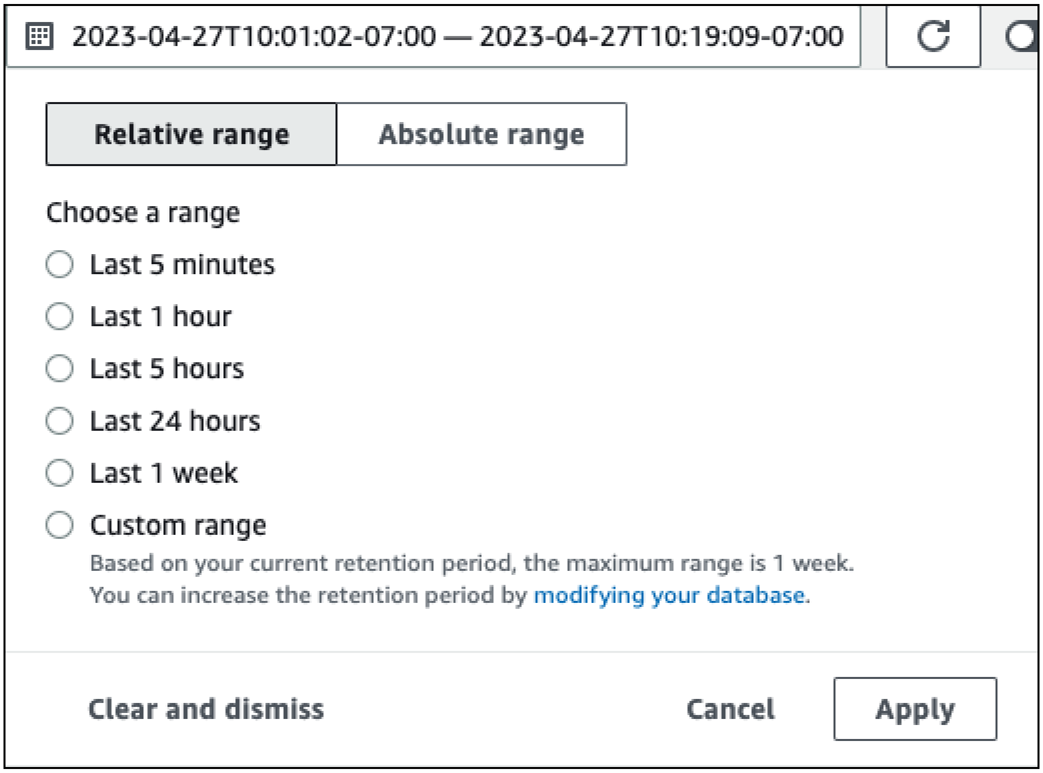
The Performance Insights dashboard appears for the DB instance.

For DB instances with Performance Insights turned on, you can also access the dashboard by choosing the **Sessions** item in the list of DB instances. Under **Current activity**, the **Sessions** item shows the database load in average active sessions over the last five minutes. The bar graphically shows the load. When the bar is empty, the DB instance is idle. As the load increases, the bar fills with blue. When the load passes the number of virtual CPUs (vCPUs) on the DB instance class, the bar turns red, indicating a potential bottleneck.



DB identifier	Engine	CPU	Current activity
database1	MySQL Community	45.51%	1.34 Sessions
database2	Oracle Enterprise Edition	55.41%	3.48 Sessions
database3	Oracle Enterprise Edition	1.02%	0 Connections

5. (Optional) Choose the date or time range in the upper right and specify a different relative or absolute time interval. You can now specify a time period, and generate a database performance analysis report. The report provides the identified insights and recommendations. For more information, see [Creating a performance analysis report](#).



Choose a range

○ Last 5 minutes

○ Last 1 hour

○ Last 5 hours

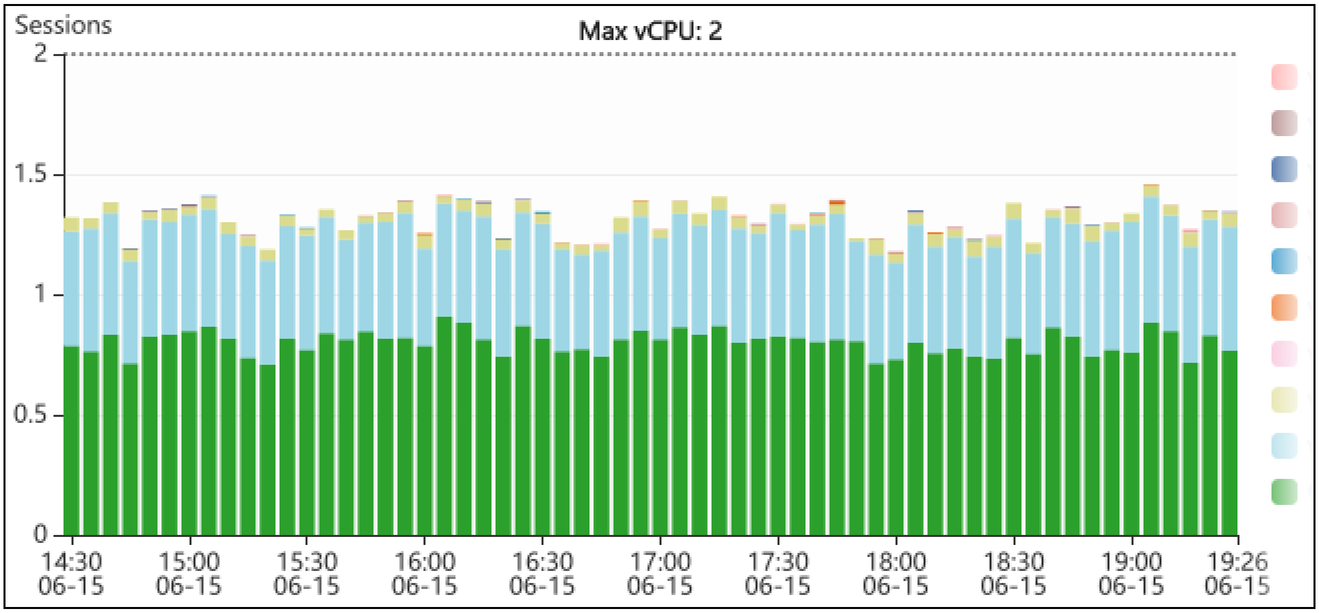
○ Last 24 hours

○ Last 1 week

○ Custom range

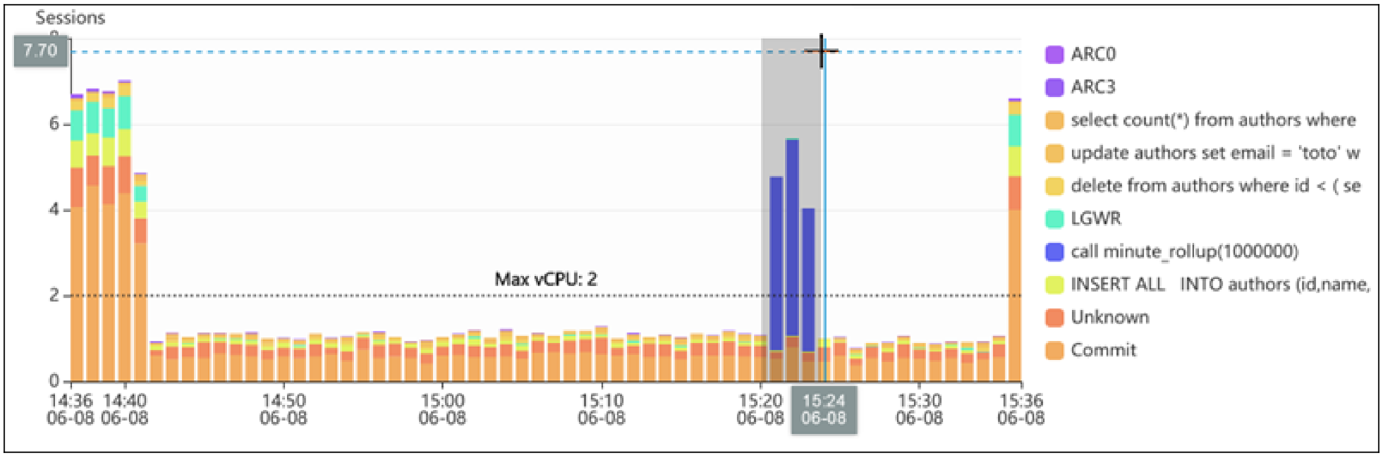
Based on your current retention period, the maximum range is 1 week. You can increase the retention period by modifying your database.

In the following screenshot, the DB load interval is 5 hours.

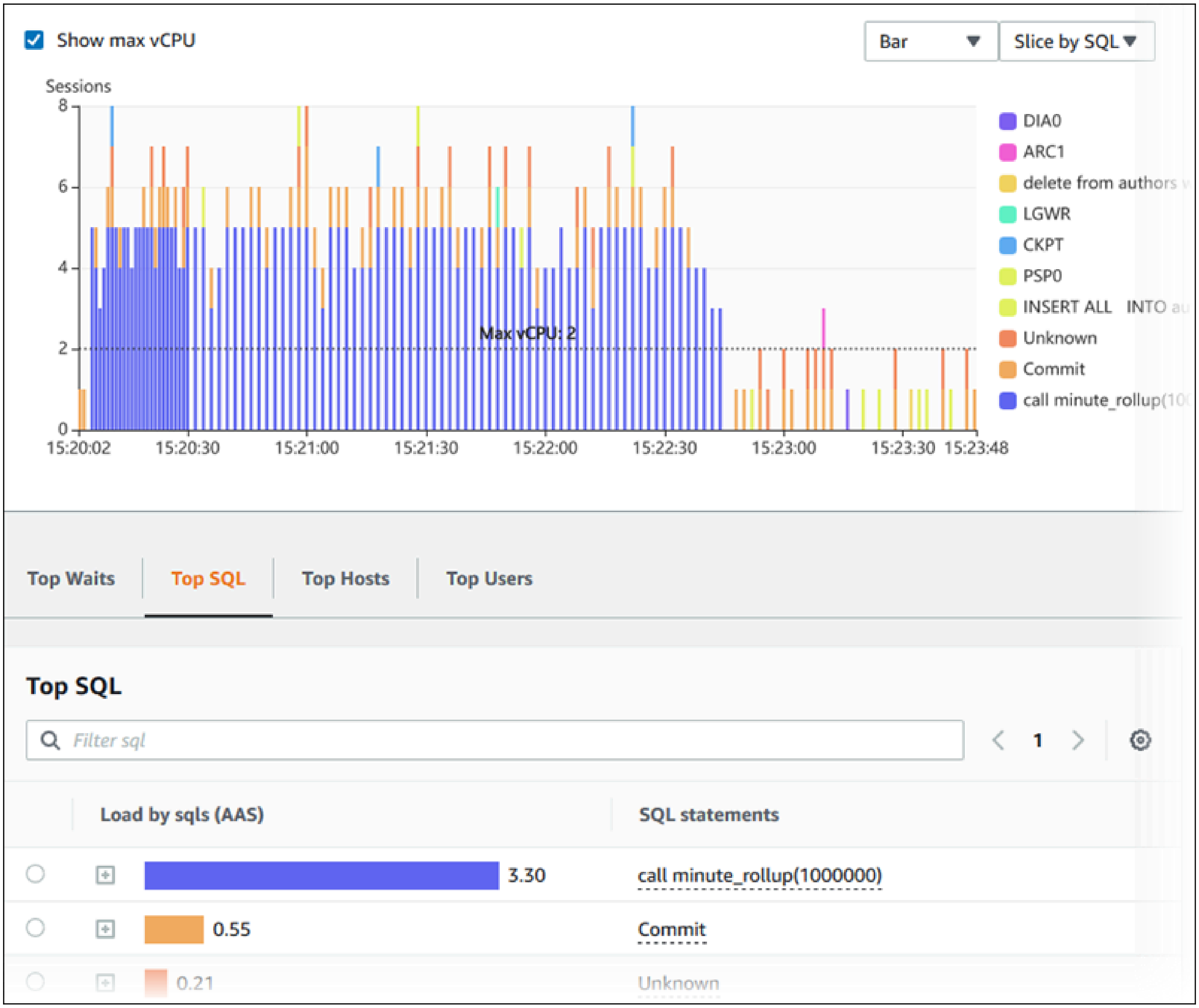


6. (Optional) To zoom in on a portion of the DB load chart, choose the start time and drag to the end of the time period you want.

   The selected area is highlighted in the DB load chart.



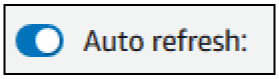
When you release the mouse, the DB load chart zooms in on the selected AWS Region, and the **Top *dimensions*** table is recalculated.



Top SQL

	Load by sqls (AAS)	SQL statements
○	3.30	call minute_rollup(1000000)
○	0.55	Commit
○	0.21	Unknown

7. (Optional) To refresh your data automatically, select **Auto refresh**.



The Performance Insights dashboard automatically refreshes with new data. The refresh rate depends on the amount of data displayed:

- 5 minutes refreshes every 10 seconds.
- 1 hour refreshes every 5 minutes.
- 5 hours refreshes every 5 minutes.
- 24 hours refreshes every 30 minutes.
- 1 week refreshes every day.

- 1 month refreshes every day.

Analyzing DB load by wait events

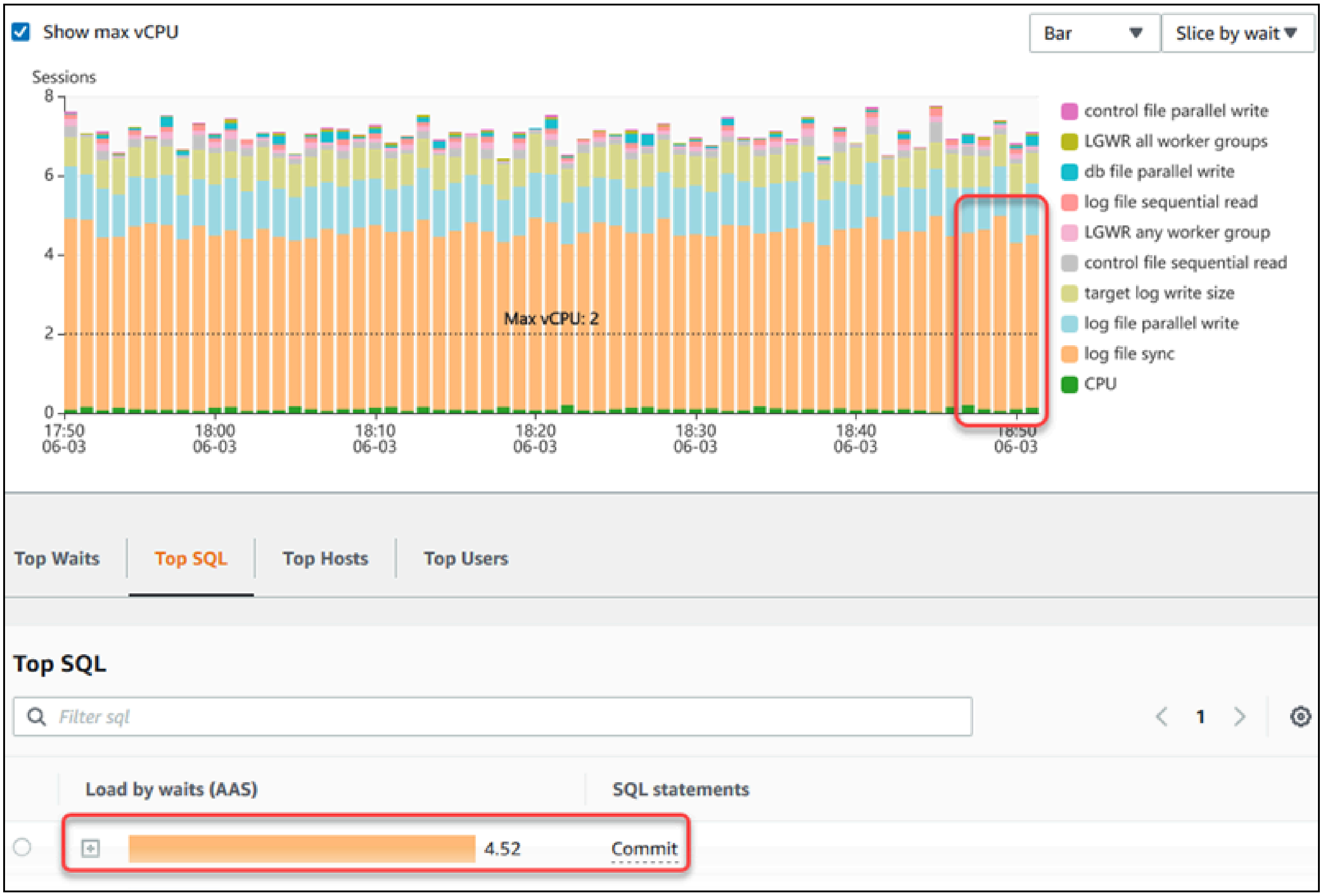
If the **Database load** chart shows a bottleneck, you can find out where the load is coming from. To do so, look at the top load items table below the **Database load** chart. Choose a particular item, like a SQL query or a user, to drill down into that item and see details about it.

DB load grouped by waits and top SQL queries is the default Performance Insights dashboard view. This combination typically provides the most insight into performance issues. DB load grouped by waits shows if there are any resource or concurrency bottlenecks in the database. In this case, the **SQL** tab of the top load items table shows which queries are driving that load.

Your typical workflow for diagnosing performance issues is as follows:

1. Review the **Database load** chart and see if there are any incidents of database load exceeding the **Max CPU** line.
2. If there is, look at the **Database load** chart and identify which wait state or states are primarily responsible.
3. Identify the digest queries causing the load by seeing which of the queries the **SQL** tab on the top load items table are contributing most to those wait states. You can identify these by the **DB Load by Wait** column.
4. Choose one of these digest queries in the **SQL** tab to expand it and see the child queries that it is composed of.

For example, in the dashboard following, **log file sync** waits account for most of the DB load. The **LGWR all worker groups** wait is also high. The **Top SQL** chart shows what is causing the **log file sync** waits: frequent `COMMIT` statements. In this case, committing less frequently will reduce DB load.



Top SQL

	Load by waits (AAS)	SQL statements
○	4.52	Commit

Analyzing database performance for a period of time

Analyze database performance with on-demand analysis by creating a performance analysis report for a period of time. View performance analysis reports to find performance issues, such as resource bottlenecks or changes in a query in your DB instance. The Performance Insights dashboard allows you to select a time period and create a performance analysis report. You can also add one or more tags to the report.

To use this feature, you must be using the paid tier retention period. For more information, see Pricing and data retention for Performance Insights

The report is available in the **Performance analysis reports - new** tab to select and view. The report contains the insights, related metrics, and recommendations to resolve the performance issue. The report is available to view for the duration of Performance Insights retention period.

The report is deleted if the start time of the report analysis period is outside of the retention period. You can also delete the report before the retention period ends.

To detect the performance issues and generate the analysis report for your DB instance, you must turn on Performance Insights. For more information about turning on Performance Insights, see Turning Performance Insights on and off for Aurora.

For the region, DB engine, and instance class support information for this feature, see Amazon Aurora DB engine, Region, and instance class support for Performance Insights features

Creating a performance analysis report

You can create a performance analysis report for a specific period in the Performance Insights dashboard. You can select a time period and add one or more tags to the analysis report.

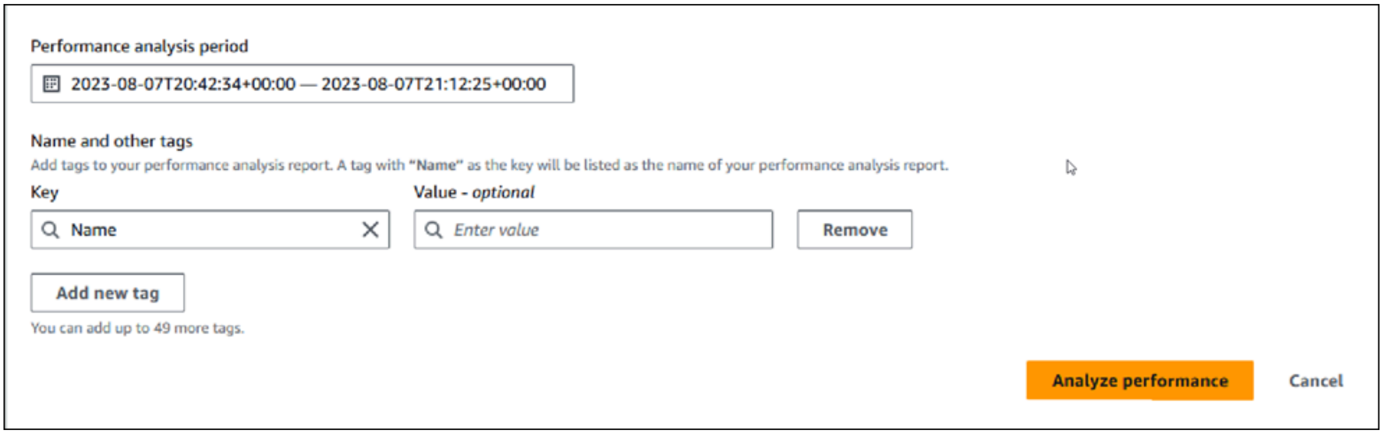
The analysis period can range from 5 minutes to 6 days. There must be at least 24 hours of performance data before the analysis start time.

To create a performance analysis report for a time period

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

   The Performance Insights dashboard appears for the DB instance.
4. Choose **Analyze performance** in **Database load** section on the dashboard.

   The fields to set the time period and add one or more tags to the performance analysis report are displayed.



Performance analysis period

2023-08-07T20:42:34+00:00 — 2023-08-07T21:12:25+00:00

Name and other tags

Add tags to your performance analysis report. A tag with "Name" as the key will be listed as the name of your performance analysis report.

Key: Name

Value - *optional*: *Enter value*

You can add up to 49 more tags.

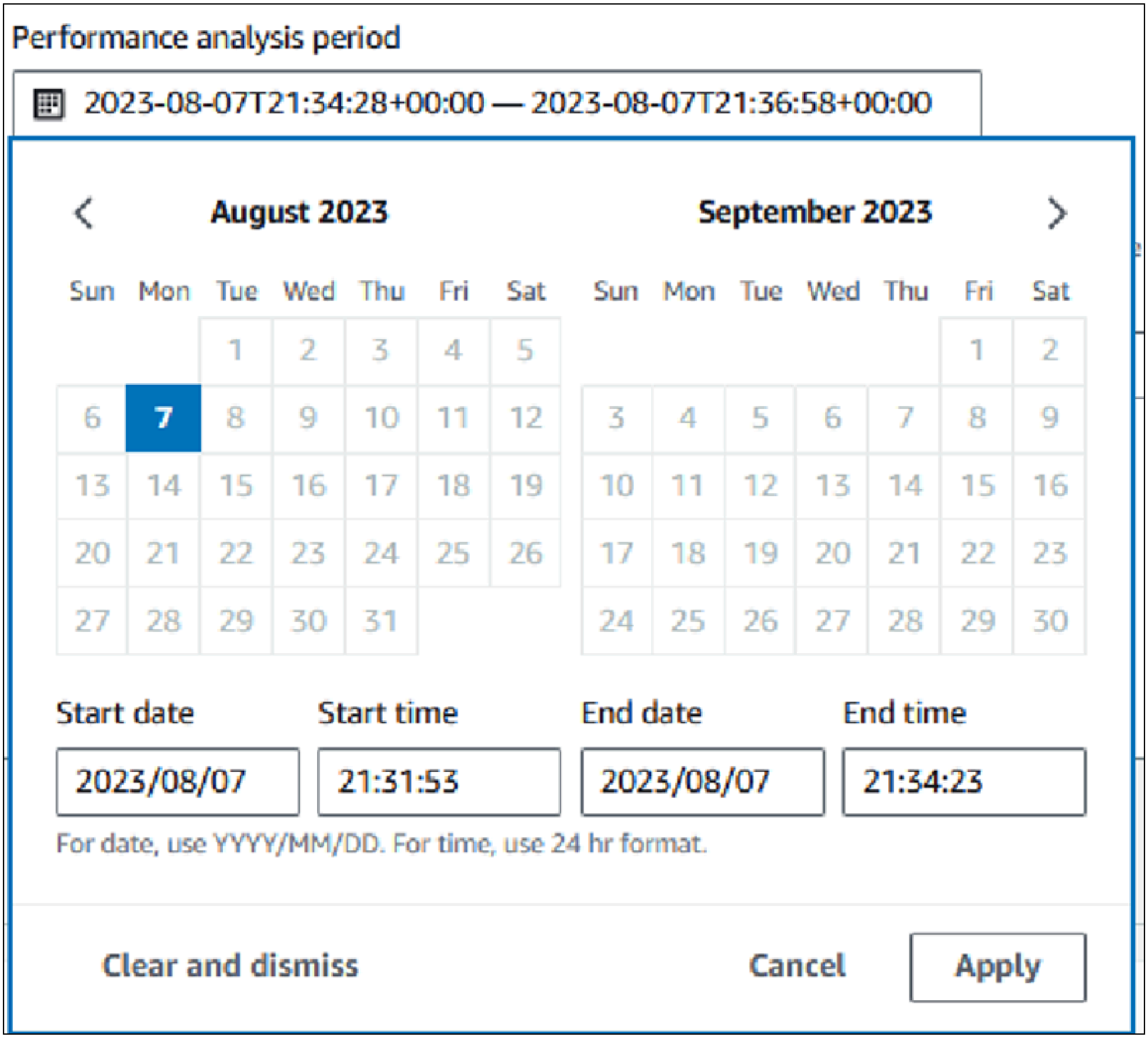
5. Choose the time period. If you set a time period in the **Relative range** or **Absolute range** in the upper right, you can only enter or select the analysis report date and time within this time period. If you select the analysis period outside of this time period, an error message displays.

To set the time period, you can do any of the following:

- Press and drag any of the sliders on the DB load chart.

  The **Performance analysis period** box displays the selected time period and DB load chart highlights the selected time period.

- Choose the **Start date**, **Start time**, **End date**, and **End time** in the **Performance analysis period** box.



Performance analysis period

2023-08-07T21:34:28+00:00 — 2023-08-07T21:36:58+00:00

August 2023

Sun	Mon	Tue	Wed	Thu	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

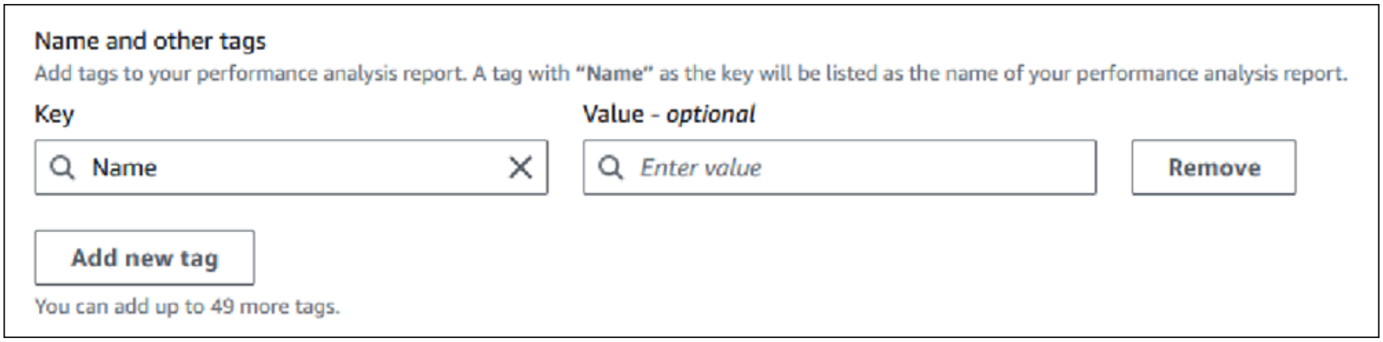
September 2023

Sun	Mon	Tue	Wed	Thu	Fri	Sat
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Start date	Start time	End date	End time
2023/08/07	21:31:53	2023/08/07	21:34:23

For date, use YYYY/MM/DD. For time, use 24 hr format.

6. (Optional) Enter **Key** and **Value-*optional*** to add a tag for the report.



Name and other tags

Add tags to your performance analysis report. A tag with "Name" as the key will be listed as the name of your performance analysis report.

Key: Name

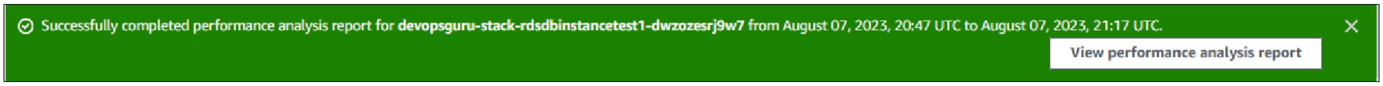
Value - *optional*: Enter value

You can add up to 49 more tags.

7. Choose **Analyze performance**.

   A banner displays a message whether the report generation is successful or failed. The message also provides the link to view the report.

   The following example shows the banner with the report creation successful message.



Successfully completed performance analysis report for **devopsguru-stack-rdsdbinstancetest1-dwzozesrj9w7** from August 07, 2023, 20:47 UTC to August 07, 2023, 21:17 UTC.

   The report is available to view in **Performance analysis reports - new** tab.

You can create a performance analysis report using the AWS CLI. For an example on how to create a report using AWS CLI, see Creating a performance analysis report for a time period.

Viewing a performance analysis report

The **Performance analysis reports - new** tab lists all the reports that are created for the DB instance. The following are displayed for each report:

- **ID**: Unique identifier of the report.
- **Name**: Tag key added to the report.
- **Report creation time**: Time you created the report.
- **Analysis start time**: Start time of the analysis in the report.
- **Analysis end time**: End time of the analysis in the report.

To view a performance analysis report

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance for which you want to view the analysis report.

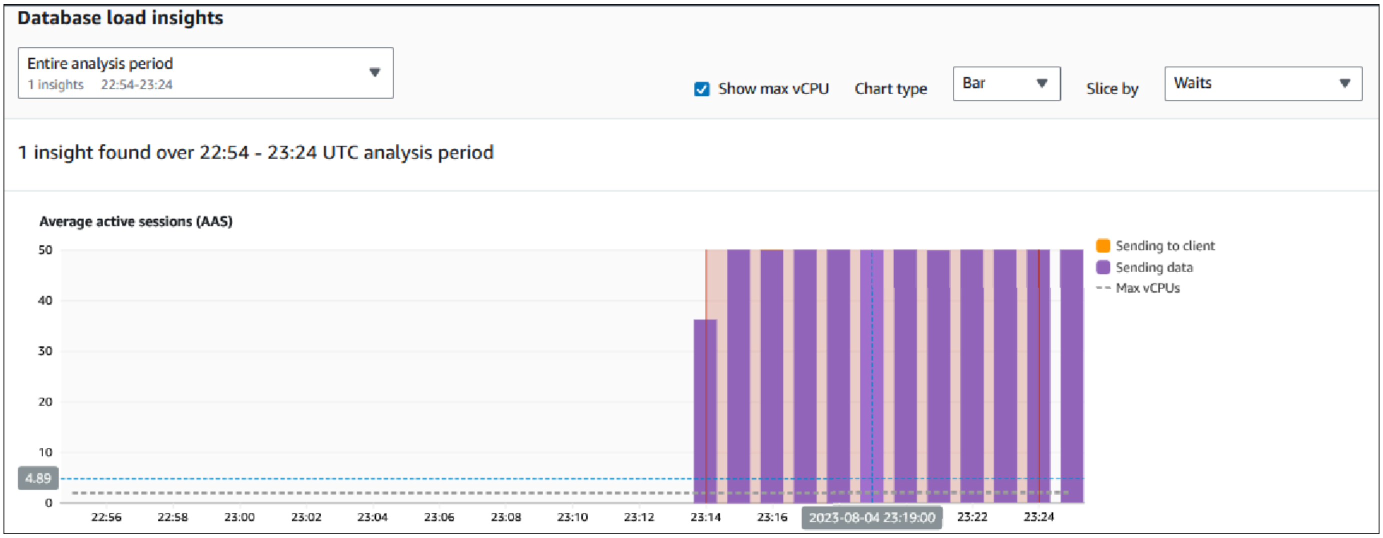
   The Performance Insights dashboard appears for the DB instance.
4. Scroll down and choose **Performance analysis reports - new** tab.

   All the analysis reports for the different time periods are displayed.
5. Choose **ID** of the report you want to view.

   The DB load chart displays the entire analysis period by default if more than one insight is identified. If the report has identified one insight then the DB load chart displays the insight by default.

   The dashboard also lists the tags for the report in the **Tags** section.

   The following example shows the entire analysis period for the report.



Database load insights

Entire analysis period
1 insights 22:54-23:24

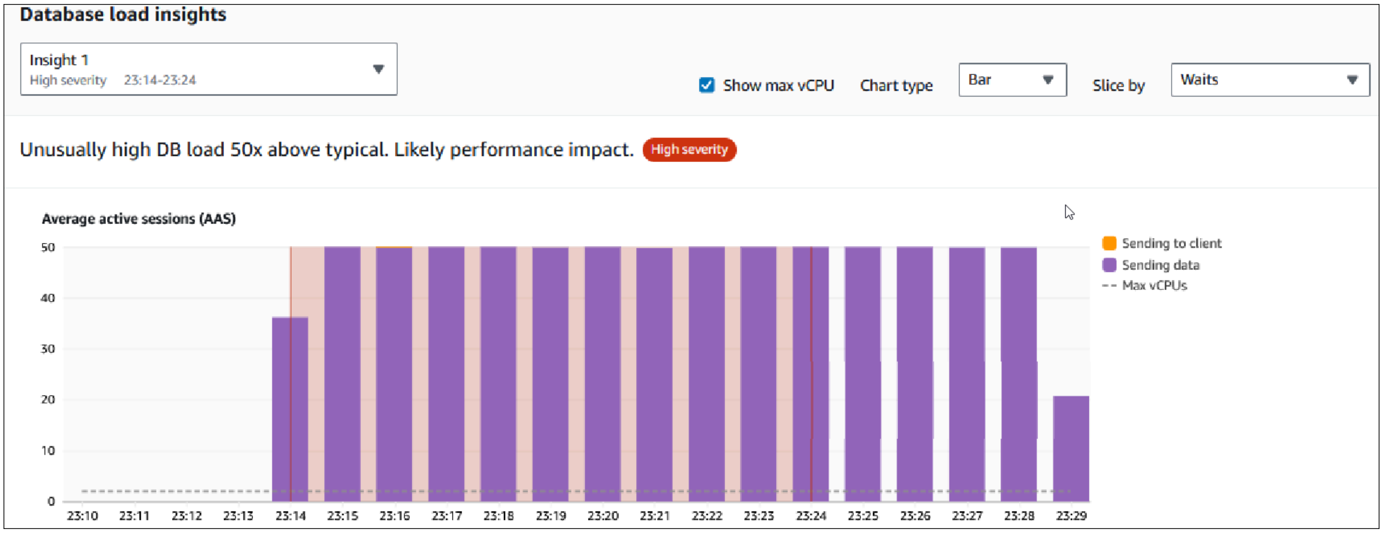
☑ Show max vCPU Chart type Bar Slice by Waits

1 insight found over 22:54 - 23:24 UTC analysis period

6. Choose the insight in the **Database load insights** list you want to view if more than one insight is identified in the report.

   The dashboard displays the insight message, DB load chart highlighting the time period of the insight, analysis and recommendations, and the list of report tags.

   The following example shows the DB load insight in the report.



Database load insights

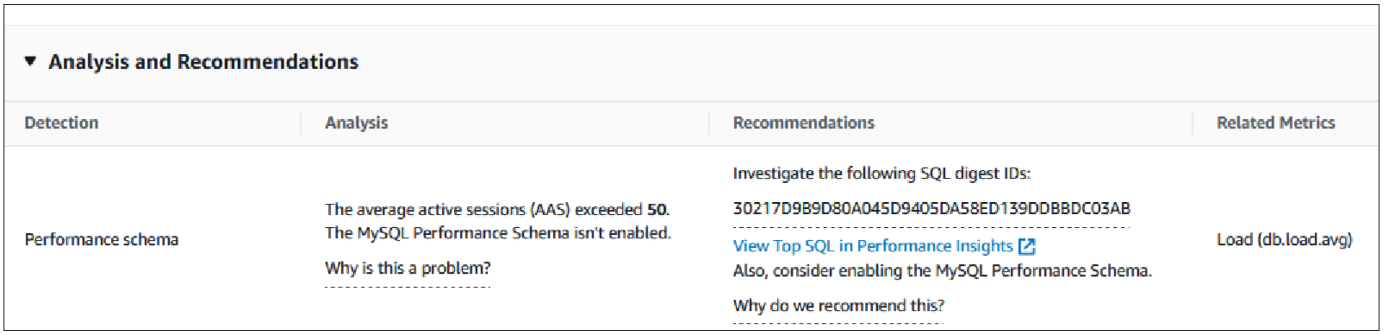
Insight 1
High severity 23:14-23:24

☑ Show max vCPU

Chart type: Bar

Slice by: Waits

Unusually high DB load 50x above typical. Likely performance impact. High severity



▼ Analysis and Recommendations

Detection	Analysis	Recommendations	Related Metrics
Performance schema	The average active sessions (AAS) exceeded **50**. The MySQL Performance Schema isn't enabled. Why is this a problem?	Investigate the following SQL digest IDs: 30217D9B9D80A045D9405DA58ED139DDBBDC03AB View Top SQL in Performance Insights Also, consider enabling the MySQL Performance Schema. Why do we recommend this?	Load (db.load.avg)

Adding tags to a performance analysis report

You can add a tag when you create or view a report. You can add up to 50 tags for a report.

You need permissions to add the tags. For more information about the access policies for Performance Insights, see Configuring access policies for Performance Insights

To add one or more tags while creating a report, see step 6 in the procedure Creating a performance analysis report.

To add one or more tags when viewing a report

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

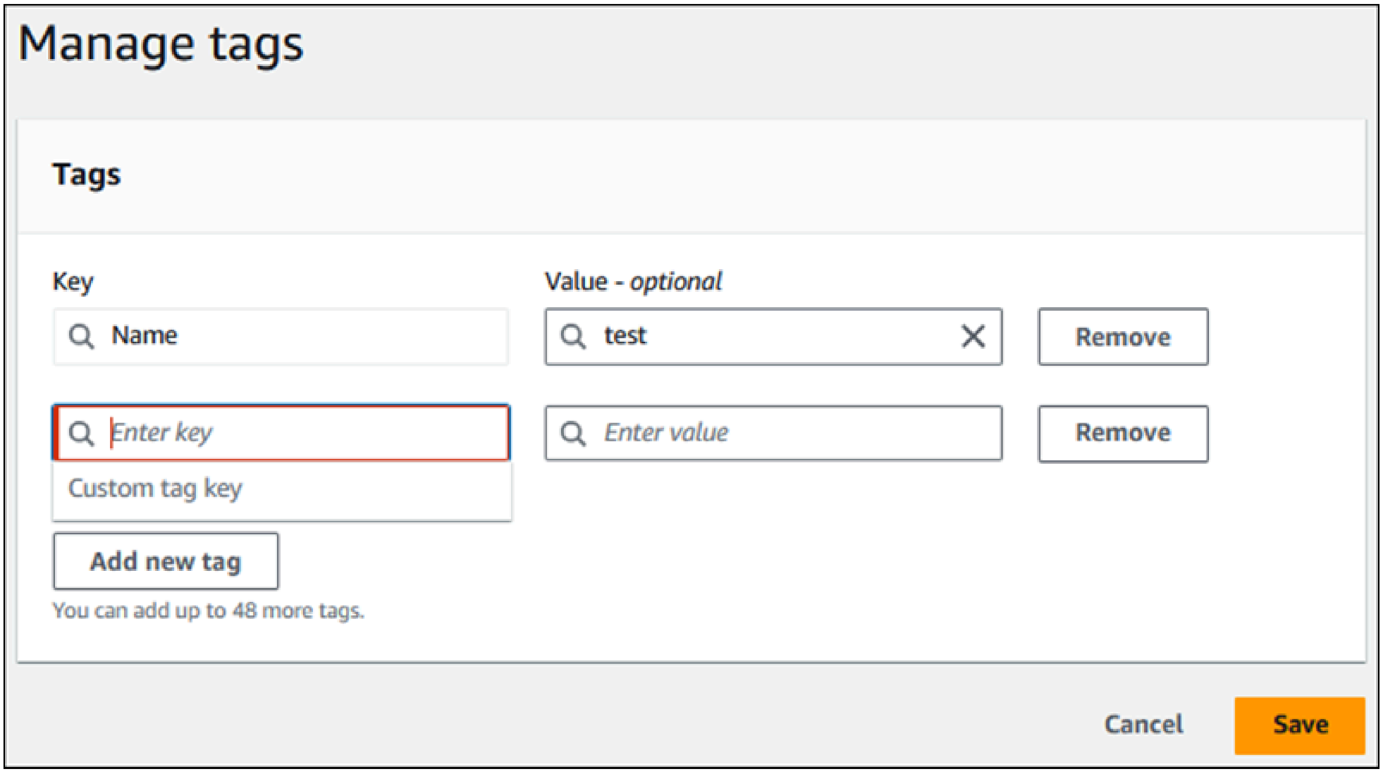
   The Performance Insights dashboard appears for the DB instance.
4. Scroll down and choose **Performance analysis reports - new** tab.

5. Choose the report for which you want to add the tags.

   The dashboard displays the report.

6. Scroll down to **Tags** and choose **Manage tags**.
7. Choose **Add new tag**.
8. Enter the **Key** and **Value - *optional***, and choose **Add new tag**.

   The following example provides the option to add a new tag for the selected report.



   A new tag is created for the report.

   The list of tags for the report is displayed in the **Tags** section on the dashboard. If you want to remove a tag from the report, choose **Remove** next to the tag.

Deleting a performance analysis report

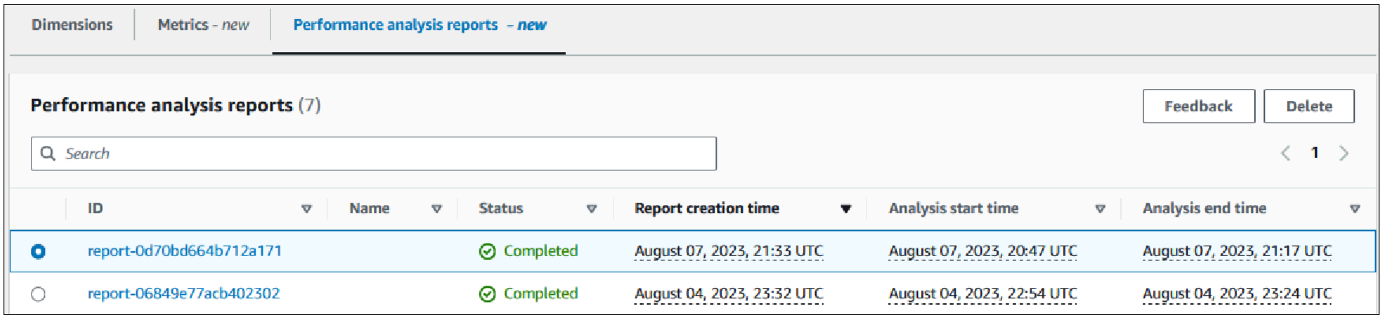
You can delete a report from the list of reports displayed in the **Performance analysis reports** tab or while viewing a report.

To delete a report

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the left navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

   The Performance Insights dashboard appears for the DB instance.
4. Scroll down and choose **Performance analysis reports - new** tab.
5. Select the report you want to delete and choose **Delete** in the upper right.



Performance analysis reports (7)

	ID	Name	Status	Report creation time	Analysis start time	Analysis end time
●	report-0d70bd664b712a171		Completed	August 07, 2023, 21:33 UTC	August 07, 2023, 20:47 UTC	August 07, 2023, 21:17 UTC
○	report-06849e77acb402302		Completed	August 04, 2023, 23:32 UTC	August 04, 2023, 22:54 UTC	August 04, 2023, 23:24 UTC

   A confirmation window is displayed. The report is deleted after you choose confirm.
6. (Optional) Choose **ID** of the report you want to delete.

   In the report page, choose **Delete** in the upper right.

   A confirmation window is displayed. The report is deleted after you choose confirm.

Analyzing queries in the Performance Insights dashboard

In the Amazon RDS Performance Insights dashboard, you can find information about running and recent queries in the **Top SQL** tab in the **Top dimensions** table. You can use this information to tune your queries.

Topics

- Overview of the Top SQL tab
- Accessing more SQL text in the Performance Insights dashboard
- Viewing SQL statistics in the Performance Insights dashboard

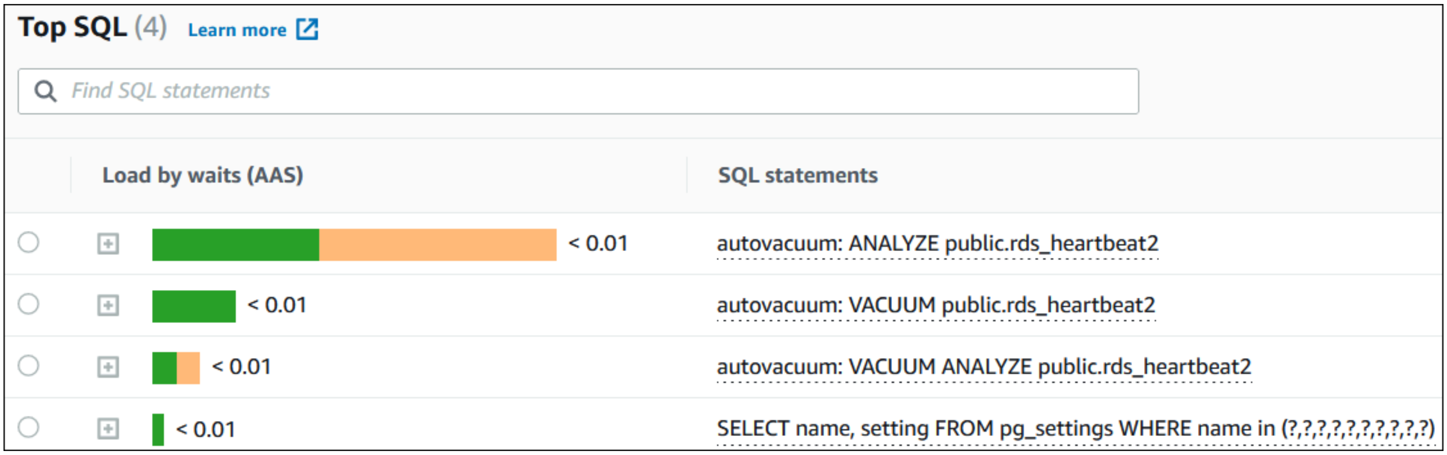
Overview of the Top SQL tab

By default, the **Top SQL** tab shows the 25 queries that are contributing the most to DB load. To help tune your queries, you can analyze information such as the query text and SQL statistics. You can also choose the statistics that you want to appear in the **Top SQL** tab.

Topics

SQL text

By default, each row in the **Top SQL** table shows 500 bytes of text for each statement.



Top SQL (4) Learn more

Find SQL statements

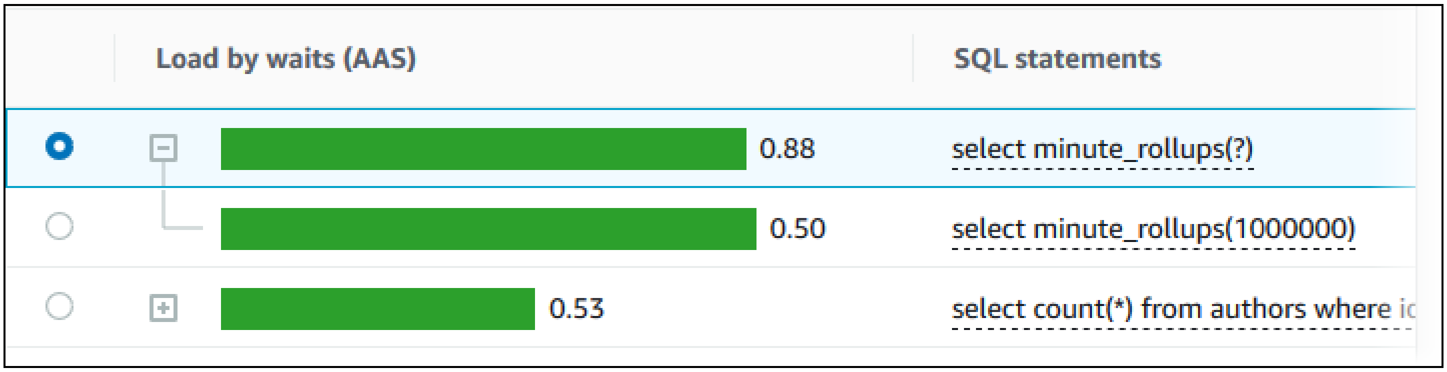
		Load by waits (AAS)	SQL statements
○	⊞	< 0.01	autovacuum: ANALYZE public.rds_heartbeat2
○	⊞	< 0.01	autovacuum: VACUUM public.rds_heartbeat2
○	⊞	< 0.01	autovacuum: VACUUM ANALYZE public.rds_heartbeat2
○	⊞	< 0.01	SELECT name, setting FROM pg_settings WHERE name in (?,?,?,?,?,?,?,?,?,?)

To learn how to see more than the default 500 bytes of SQL text, see Accessing more SQL text in the Performance Insights dashboard.

A *SQL digest* is a composite of multiple actual queries that are structurally similar but might have different literal values. The digest replaces hardcoded values with a question mark. For example, a digest might be `SELECT * FROM emp WHERE lname= ?`. This digest might include the following child queries:

```
SELECT * FROM emp WHERE lname = 'Sanchez'
SELECT * FROM emp WHERE lname = 'Olagappan'
SELECT * FROM emp WHERE lname = 'Wu'
```

To see the literal SQL statements in a digest, select the query, and then choose the plus symbol (+). In the following example, the selected query is a digest.



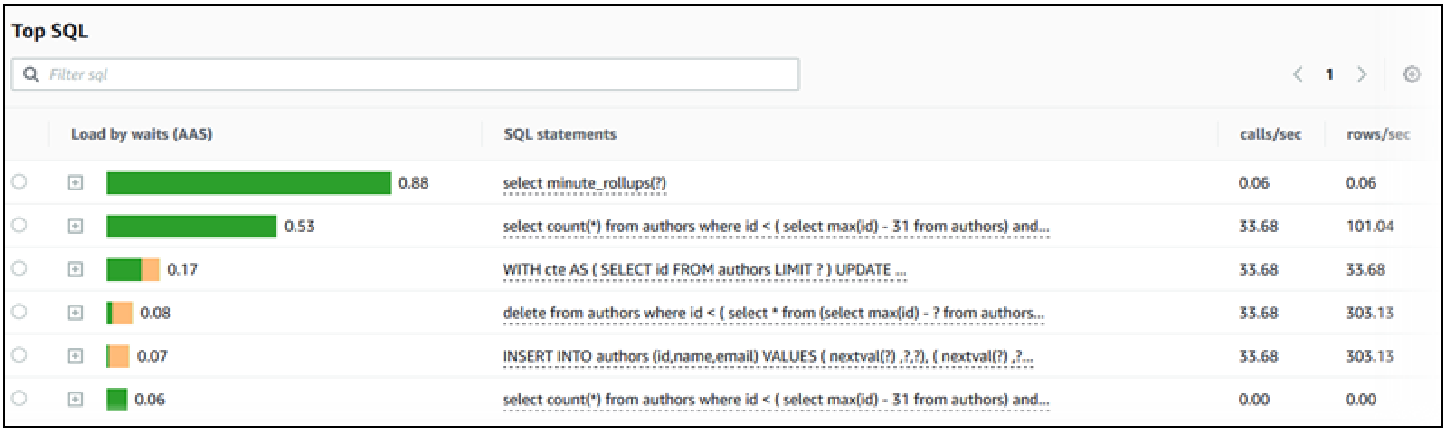
	Load by waits (AAS)	SQL statements
●	⊟ 0.88	select minute_rollups(?)
○	0.50	select minute_rollups(1000000)
○	⊞ 0.53	select count(*) from authors where ic

> **Note**
>
> A SQL digest groups similar SQL statements, but doesn't redact sensitive information.

SQL statistics

SQL statistics are performance-related metrics about SQL queries. For example, Performance Insights might show executions per second or rows processed per second. Performance Insights collects statistics for only the most common queries. Typically, these match the top queries by load shown in the Performance Insights dashboard.

Every line in the **Top SQL** table shows relevant statistics for the SQL statement or digest, as shown in the following example.



Top SQL

		Load by waits (AAS)	SQL statements	calls/sec	rows/sec
○	⊞	0.88	select minute_rollups(?)	0.06	0.06
○	⊞	0.53	select count(*) from authors where id < (select max(id) - 31 from authors) and...	33.68	101.04
○	⊞	0.17	WITH cte AS (SELECT id FROM authors LIMIT ?) UPDATE ...	33.68	33.68
○	⊞	0.08	delete from authors where id < (select * from (select max(id) - ? from authors...	33.68	303.13
○	⊞	0.07	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?), (nextval(?) ,?...	33.68	303.13
○	⊞	0.06	select count(*) from authors where id < (select max(id) - 31 from authors) and...	0.00	0.00

Performance Insights can report `0.00` and `-` (unknown) for SQL statistics. This situation occurs under the following conditions:

- Only one sample exists. For example, Performance Insights calculates rates of change for Aurora PostgreSQL queries based on multiple samples from the `pg_stat_statements` view. When a workload runs for a short time, Performance Insights might collect only one sample, which means that it can't calculate a rate of change. The unknown value is represented with a dash (-).

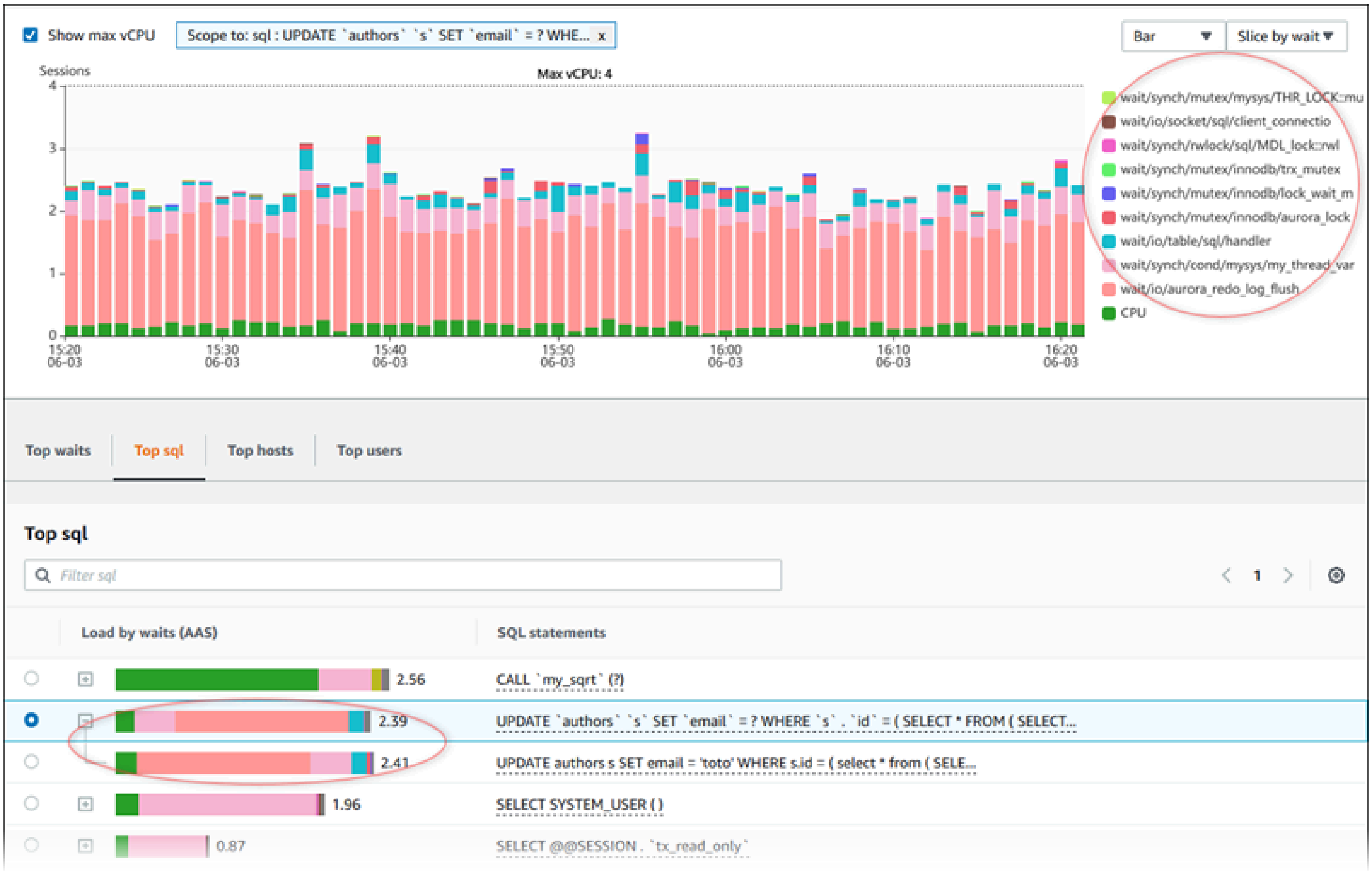
- Two samples have the same values. Performance Insights can't calculate a rate of change because no change has occurred, so it reports the rate as `0.00`.
- An Aurora PostgreSQL statement lacks a valid identifier. PostgreSQL creates a identifier for a statement only after parsing and analysis. Thus, a statement can exist in the PostgreSQL internal in-memory structures with no identifier. Because Performance Insights samples internal in-memory structures once per second, low-latency queries might appear for only a single sample. If the query identifier isn't available for this sample, Performance Insights can't associate this statement with its statistics. The unknown value is represented with a dash (-).

For a description of the SQL statistics for the Aurora engines, see SQL statistics for Performance Insights.

Load by waits (AAS)

In **Top SQL**, the **Load by waits (AAS)** column illustrates the percentage of the database load associated with each top load item. This column reflects the load for that item by whatever grouping is currently selected in the **DB Load Chart**. For more information about Average active sessions (AAS), see Average active sessions.

For example, you might group the **DB load** chart by wait states. You examine SQL queries in the top load items table. In this case, the **DB Load by Waits** bar is sized, segmented, and color-coded to show how much of a given wait state that query is contributing to. It also shows which wait states are affecting the selected query.

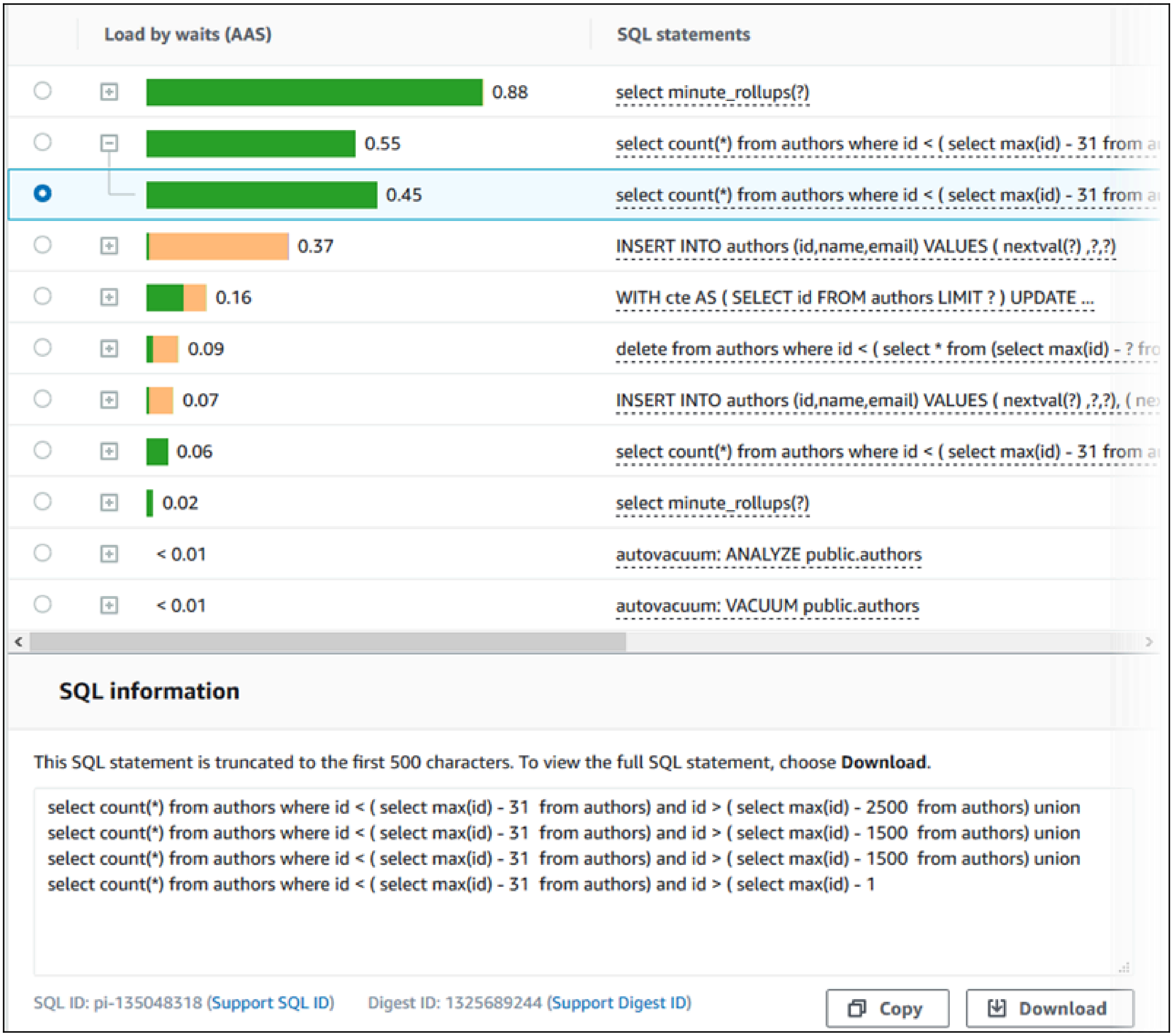

☑ Show max vCPU
Scope to: sql : UPDATE `authors` `s` SET `email` = ? WHE... x
Bar ▼
Slice by wait ▼
Sessions
Max vCPU: 4
4
3
2
1
0
15:20 06-03
15:30 06-03
15:40 06-03
15:50 06-03
16:00 06-03
16:10 06-03
16:20 06-03
wait/synch/mutex/mysys/THR_LOCK::mu
wait/io/socket/sql/client_connectio
wait/synch/rwlock/sql/MDL_lock::rwl
wait/synch/mutex/innodb/trx_mutex
wait/synch/mutex/innodb/lock_wait_m
wait/synch/mutex/innodb/aurora_lock
wait/io/table/sql/handler
wait/synch/cond/mysys/my_thread_var
wait/io/aurora_redo_log_flush
CPU
Top waits
Top sql
Top hosts
Top users
Filter sql
1

Top sql

	Load by waits (AAS)	SQL statements
○	2.56	CALL `my_sqrt` (?)
●	2.39	UPDATE `authors` `s` SET `email` = ? WHERE `s` . `id` = (SELECT * FROM (SELECT...
○	2.41	UPDATE authors s SET email = 'toto' WHERE s.id = (select * from (SELE...
○	1.96	SELECT SYSTEM_USER ( )
○	0.87	SELECT @@SESSION . `tx_read_only`

SQL information

In the **Top SQL** table, you can open a statement to view its information. The information appears in the bottom pane.



		Load by waits (AAS)	SQL statements
○	⊞	0.88	select minute_rollups(?)
○	⊟	0.55	select count(*) from authors where id < (select max(id) - 31 from a
●		0.45	select count(*) from authors where id < (select max(id) - 31 from a
○	⊞	0.37	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?)
○	⊞	0.16	WITH cte AS (SELECT id FROM authors LIMIT ?) UPDATE ...
○	⊞	0.09	delete from authors where id < (select * from (select max(id) - ? fro
○	⊞	0.07	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?), (ne
○	⊞	0.06	select count(*) from authors where id < (select max(id) - 31 from a
○	⊞	0.02	select minute_rollups(?)
○	⊞	< 0.01	autovacuum: ANALYZE public.authors
○	⊞	< 0.01	autovacuum: VACUUM public.authors

SQL information

This SQL statement is truncated to the first 500 characters. To view the full SQL statement, choose **Download**.

```
select count(*) from authors where id < ( select max(id) - 31  from authors) and id > ( select max(id) - 2500  from authors) union
select count(*) from authors where id < ( select max(id) - 31  from authors) and id > ( select max(id) - 1500  from authors) union
select count(*) from authors where id < ( select max(id) - 31  from authors) and id > ( select max(id) - 1500  from authors) union
select count(*) from authors where id < ( select max(id) - 31  from authors) and id > ( select max(id) - 1
```

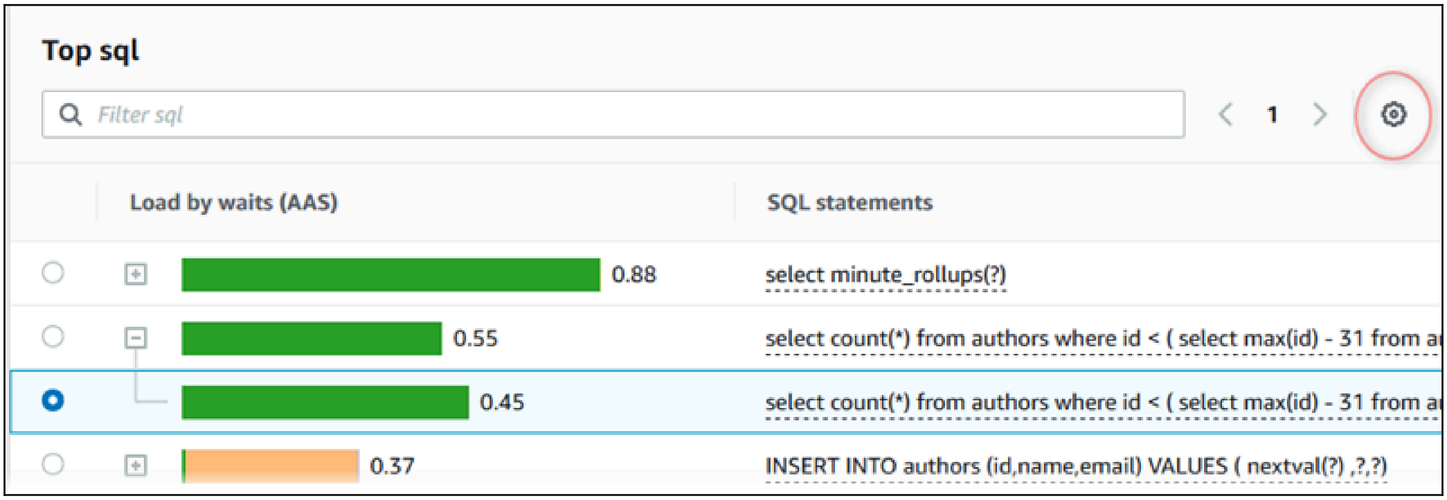
SQL ID: pi-135048318 (Support SQL ID)   Digest ID: 1325689244 (Support Digest ID)

The following types of identifiers (IDs) that are associated with SQL statements:

- **Support SQL ID** – A hash value of the SQL ID. This value is only for referencing a SQL ID when you are working with AWS Support. AWS Support doesn't have access to your actual SQL IDs and SQL text.
- **Support Digest ID** – A hash value of the digest ID. This value is only for referencing a digest ID when you are working with AWS Support. AWS Support doesn't have access to your actual digest IDs and SQL text.

Preferences

You can control the statistics displayed in the **Top SQL** tab by choosing the **Preferences** icon.

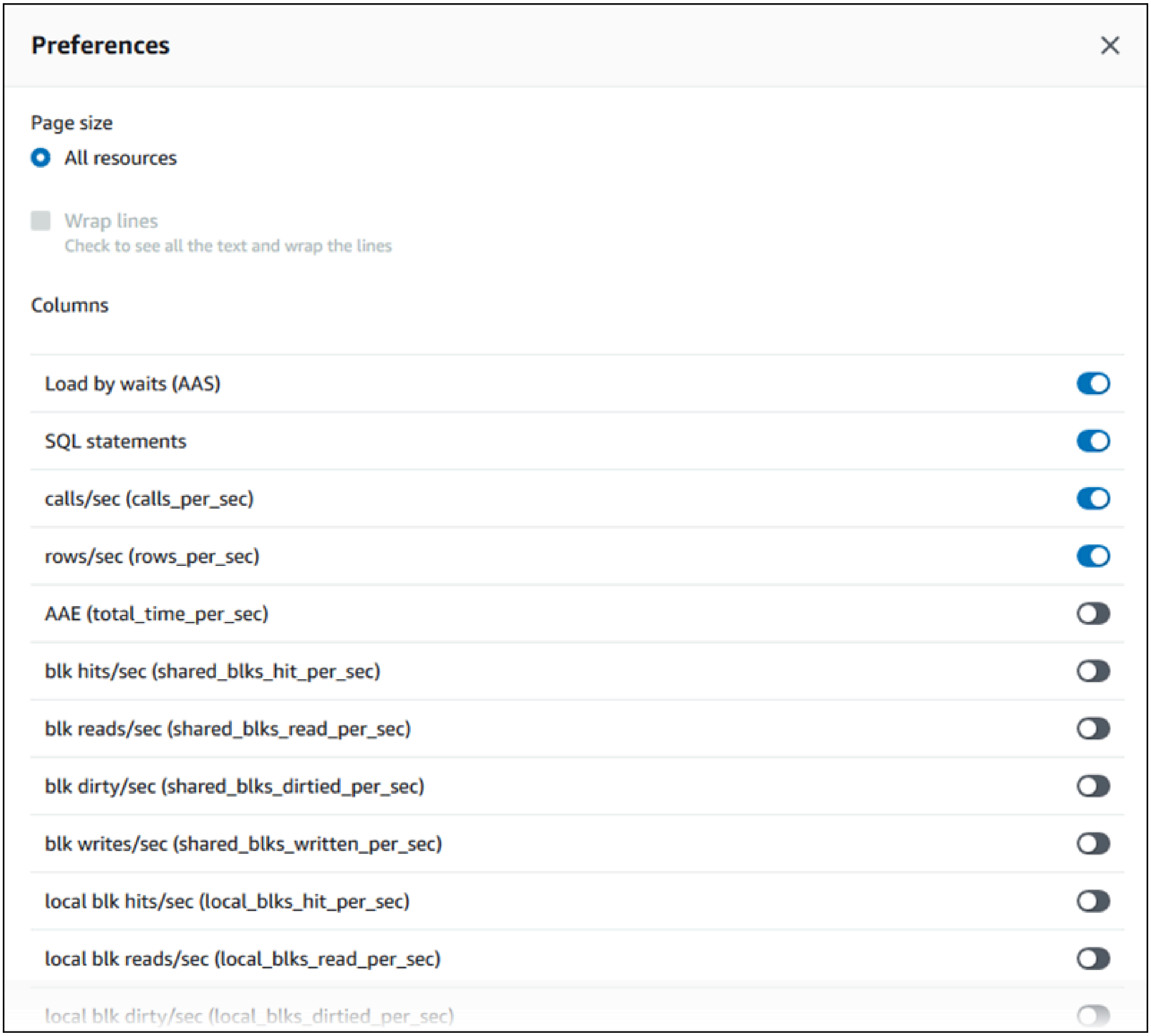


Top sql

Filter sql

	Load by waits (AAS)	SQL statements
○	⊞ 0.88	select minute_rollups(?)
○	⊟ 0.55	select count(*) from authors where id < (select max(id) - 31 from a
●	0.45	select count(*) from authors where id < (select max(id) - 31 from a
○	⊞ 0.37	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?)

When you choose the **Preferences** icon, the **Preferences** window opens. The following screenshot is an example of the **Preferences** window.



Preferences

Page size

● All resources

□ Wrap lines
Check to see all the text and wrap the lines

Columns

Load by waits (AAS)	● on
SQL statements	● on
calls/sec (calls_per_sec)	● on
rows/sec (rows_per_sec)	● on
AAE (total_time_per_sec)	○ off
blk hits/sec (shared_blks_hit_per_sec)	○ off
blk reads/sec (shared_blks_read_per_sec)	○ off
blk dirty/sec (shared_blks_dirtied_per_sec)	○ off
blk writes/sec (shared_blks_written_per_sec)	○ off
local blk hits/sec (local_blks_hit_per_sec)	○ off
local blk reads/sec (local_blks_read_per_sec)	○ off
local blk dirty/sec (local_blks_dirtied_per_sec)	○ off

To enable the statistics that you want to appear in the **Top SQL** tab, use your mouse to scroll to the bottom of the window, and then choose **Continue**.

For more information about per-second or per-call statistics for the Aurora engines, see the engine specific SQL statistics section in SQL statistics for Performance Insights

Accessing more SQL text in the Performance Insights dashboard

By default, each row in the **Top SQL** table shows 500 bytes of SQL text for each SQL statement.

0.01 select name, to_char(next_time,'YYYY/MM/DD HH24:MI:SS') As restorable_time, reci...

When a SQL statement exceeds 500 bytes, you can view more text in the **SQL text** section below the **Top SQL** table. In this case, the maximum length for the text displayed in **SQL text** is 4 KB. This limit is introduced by the console and is subject to the limits set by the database engine. To save the text shown in **SQL text**, choose **Download**.

Topics

- Text size limits for Aurora MySQL
- Setting the SQL text limit for Aurora PostgreSQL DB instances
- Viewing and downloading SQL text in the Performance Insights dashboard

Text size limits for Aurora MySQL

When you download SQL text, the database engine determines its maximum length. You can download SQL text up to the following per-engine limits.

DB engine	Maximum length of downloaded text
Aurora MySQL	4,096 bytes

The **SQL text** section of the Performance Insights console displays up to the maximum that the engine returns. For example, if Aurora MySQL returns at most 1 KB to Performance Insights, it can only collect and show 1 KB, even if the original query is larger. Thus, when you view the query in **SQL text** or download it, Performance Insights returns the same number of bytes.

If you use the AWS CLI or API, Performance Insights doesn't have the 4 KB limit enforced by the console. `DescribeDimensionKeys` and `GetResourceMetrics` return at most 500 bytes.

> **Note**
>
> `GetDimensionKeyDetails` returns the full query, but the size is subject to the engine limit.

Setting the SQL text limit for Aurora PostgreSQL DB instances

Aurora PostgreSQL handles text differently. You can set the text size limit with the DB instance parameter `track_activity_query_size`. This parameter has the following characteristics:

Default text size

On Aurora PostgreSQL version 9.6, the default setting for the `track_activity_query_size` parameter is 1,024 bytes. On Aurora PostgreSQL version 10 or higher, the default is 4,096 bytes.

Maximum text size

The limit for `track_activity_query_size` is 102,400 bytes for Aurora PostgreSQL version 12 and lower. The maximum is 1 MB for version 13 and higher.

If the engine returns 1 MB to Performance Insights, the console displays only the first 4 KB. If you download the query, you get the full 1 MB. In this case, viewing and downloading return different numbers of bytes. For more information about the `track_activity_query_size` DB instance parameter, see Run-time Statistics in the PostgreSQL documentation.

To increase the SQL text size, increase the `track_activity_query_size` limit. To modify the parameter, change the parameter setting in the parameter group that is associated with the Aurora PostgreSQL DB instance.

To change the setting when the instance uses the default parameter group

1. Create a new DB instance parameter group for the appropriate DB engine and DB engine version.
2. Set the parameter in the new parameter group.
3. Associate the new parameter group with the DB instance.

For information about setting a DB instance parameter, see Modifying parameters in a DB parameter group.

Viewing and downloading SQL text in the Performance Insights dashboard

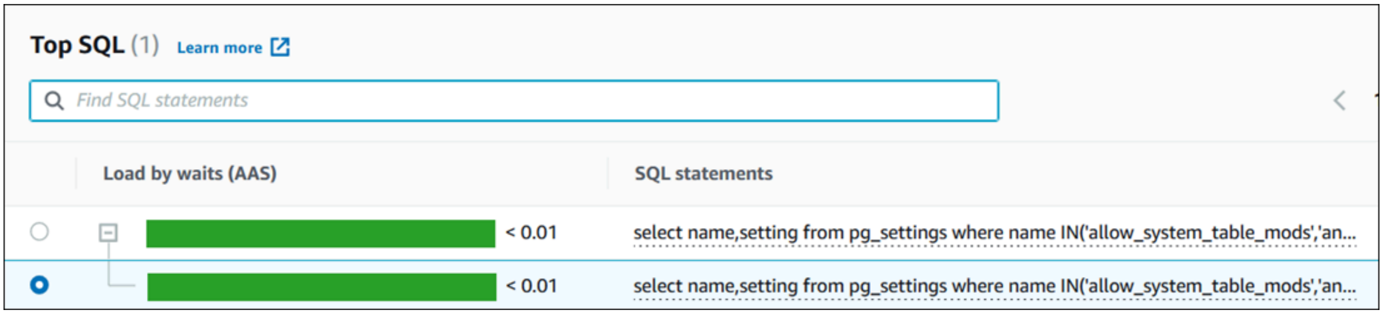
In the Performance Insights dashboard, you can view or download SQL text.

To view more SQL text in the Performance Insights dashboard

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance.

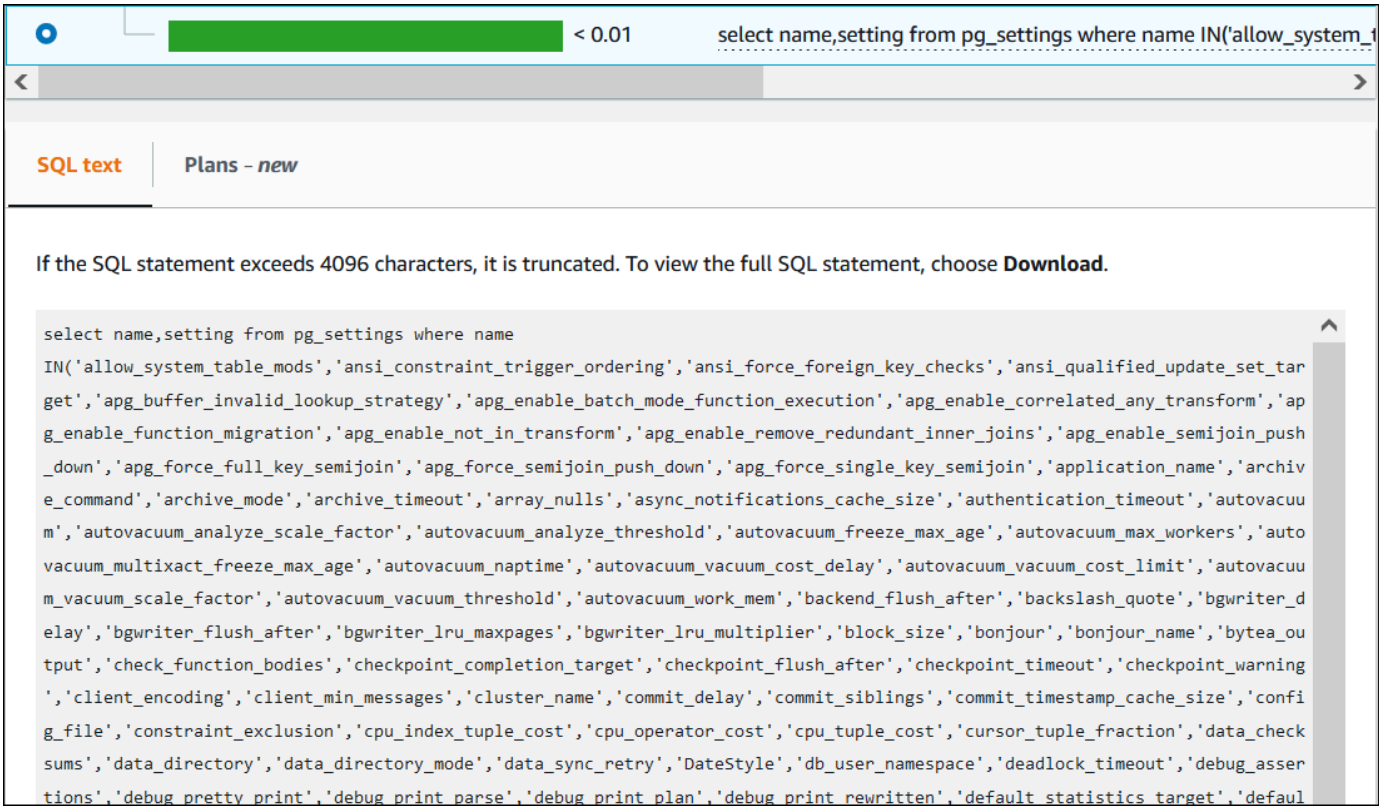
   The Performance Insights dashboard is displayed for your DB instance.
4. Scroll down to the **Top SQL** tab.
5. Choose the plus sign to expand a SQL digest and choose one of the digest's child queries.

   SQL statements with text larger than 500 bytes look similar to the following image.



	Load by waits (AAS)	SQL statements
○	< 0.01	select name,setting from pg_settings where name IN('allow_system_table_mods','an...
●	< 0.01	select name,setting from pg_settings where name IN('allow_system_table_mods','an...

6. Scroll down to the **SQL text** tab.



< 0.01
select name,setting from pg_settings where name IN('allow_system_t
SQL text
Plans - *new*

If the SQL statement exceeds 4096 characters, it is truncated. To view the full SQL statement, choose **Download**.

```
select name,setting from pg_settings where name
IN('allow_system_table_mods','ansi_constraint_trigger_ordering','ansi_force_foreign_key_checks','ansi_qualified_update_set_tar
get','apg_buffer_invalid_lookup_strategy','apg_enable_batch_mode_function_execution','apg_enable_correlated_any_transform','ap
g_enable_function_migration','apg_enable_not_in_transform','apg_enable_remove_redundant_inner_joins','apg_enable_semijoin_push
_down','apg_force_full_key_semijoin','apg_force_semijoin_push_down','apg_force_single_key_semijoin','application_name','archiv
e_command','archive_mode','archive_timeout','array_nulls','async_notifications_cache_size','authentication_timeout','autovacuu
m','autovacuum_analyze_scale_factor','autovacuum_analyze_threshold','autovacuum_freeze_max_age','autovacuum_max_workers','auto
vacuum_multixact_freeze_max_age','autovacuum_naptime','autovacuum_vacuum_cost_delay','autovacuum_vacuum_cost_limit','autovacuu
m_vacuum_scale_factor','autovacuum_vacuum_threshold','autovacuum_work_mem','backend_flush_after','backslash_quote','bgwriter_d
elay','bgwriter_flush_after','bgwriter_lru_maxpages','bgwriter_lru_multiplier','block_size','bonjour','bonjour_name','bytea_ou
tput','check_function_bodies','checkpoint_completion_target','checkpoint_flush_after','checkpoint_timeout','checkpoint_warning
','client_encoding','client_min_messages','cluster_name','commit_delay','commit_siblings','commit_timestamp_cache_size','confi
g_file','constraint_exclusion','cpu_index_tuple_cost','cpu_operator_cost','cpu_tuple_cost','cursor_tuple_fraction','data_check
sums','data_directory','data_directory_mode','data_sync_retry','DateStyle','db_user_namespace','deadlock_timeout','debug_asser
tions','debug_pretty_print','debug_print_parse','debug_print_plan','debug_print_rewritten','default_statistics_target','defaul
```

The Performance Insights dashboard can display up to 4,096 bytes for each SQL statement.

7. (Optional) Choose **Copy** to copy the displayed SQL statement, or choose **Download** to download the SQL statement to view the SQL text up to the DB engine limit.

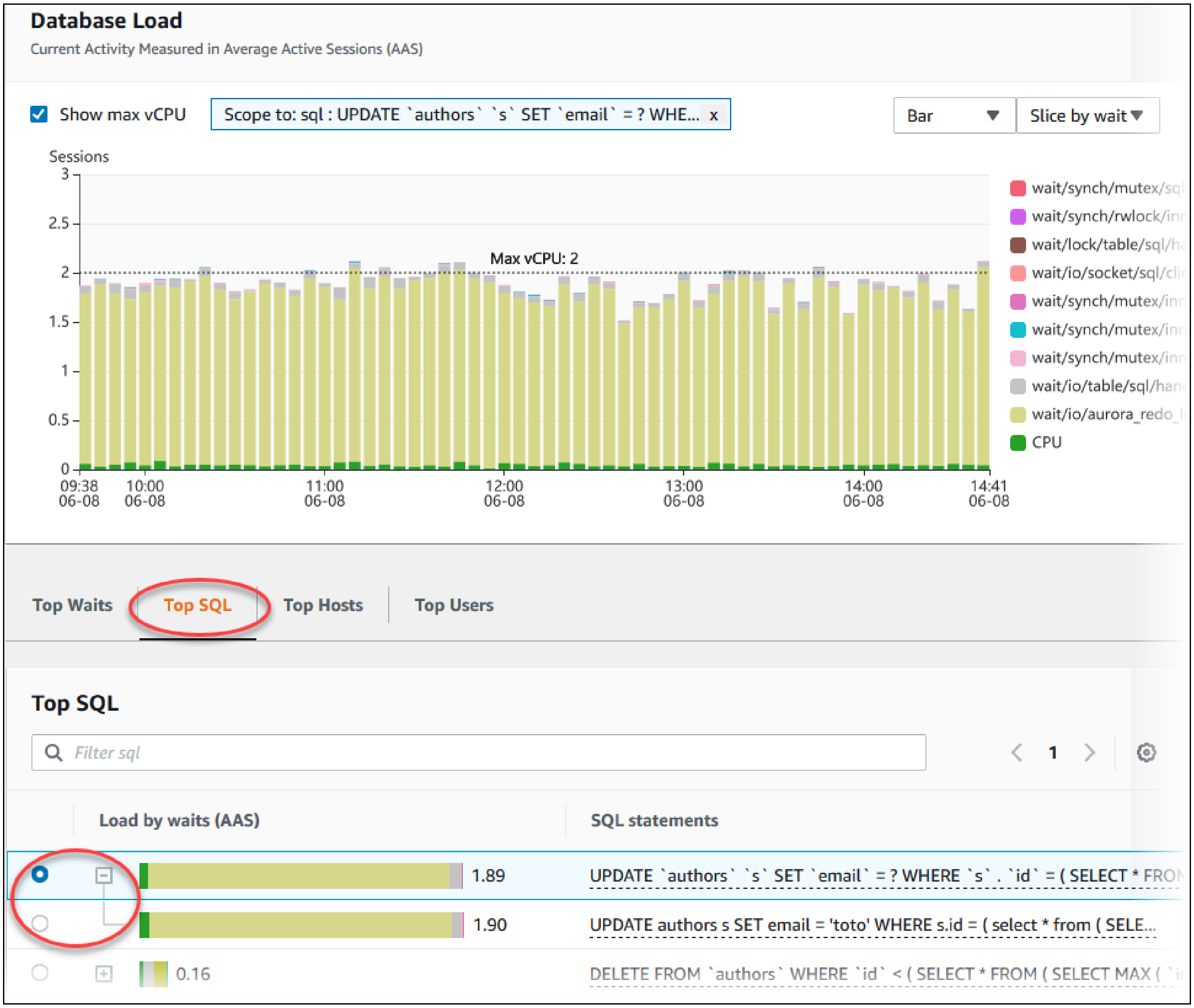
   > **Note**
   >
   > To copy or download the SQL statement, disable pop-up blockers.

Viewing SQL statistics in the Performance Insights dashboard

In the Performance Insights dashboard, SQL statistics are available in the **Top SQL** tab of the **Database load** chart.

To view SQL statistics

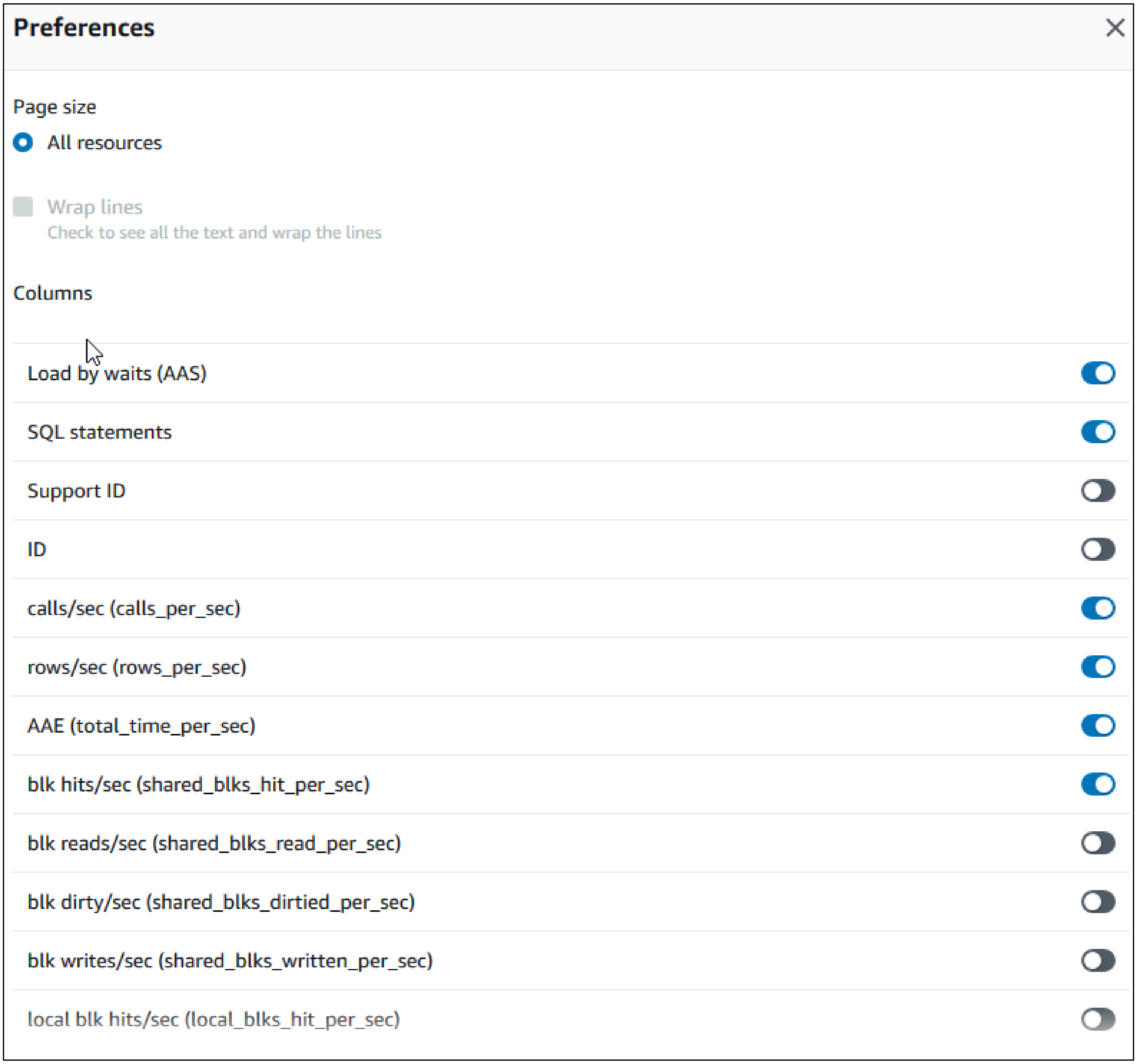
1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the left navigation pane, choose **Performance Insights**.
3. At the top of the page, choose the database whose SQL statistics you want to see.
4. Scroll to the bottom of the page and choose the **Top SQL** tab.
5. Choose an individual statement (Aurora MySQL only) or digest query.



	Load by waits (AAS)	SQL statements
● ⊟	1.89	UPDATE `authors` `s` SET `email` = ? WHERE `s` . `id` = (SELECT * FRO
○	1.90	UPDATE authors s SET email = 'toto' WHERE s.id = (select * from (SELE...
○ ⊞	0.16	DELETE FROM `authors` WHERE `id` < (SELECT * FROM (SELECT MAX (`i

6. Choose which statistics to display by choosing the gear icon in the upper-right corner of the chart. For descriptions of the SQL statistics for the Amazon RDSAurora engines, see SQL statistics for Performance Insights.

   The following example shows the preferences for Aurora PostgreSQL.



Preferences

Page size

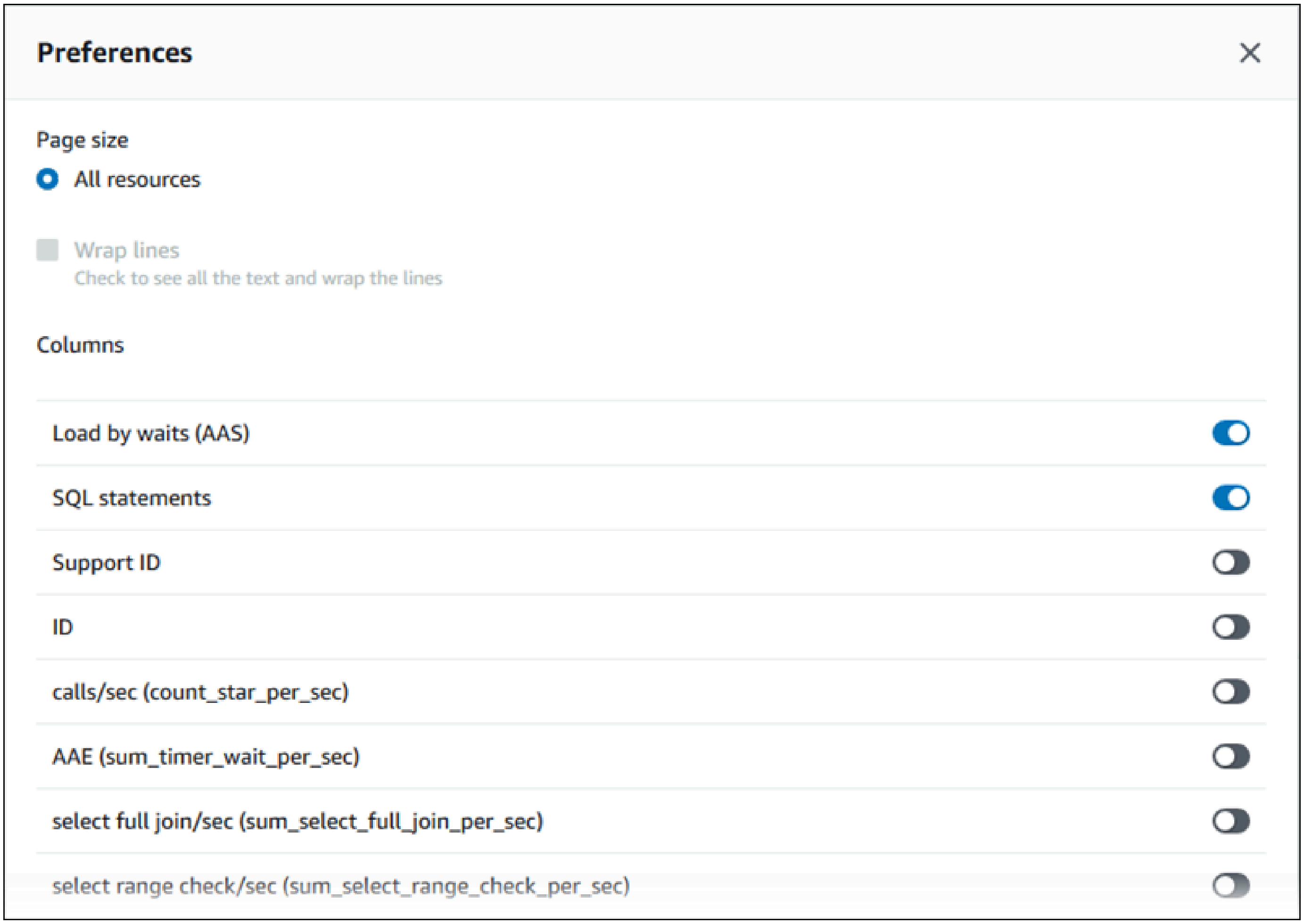
● All resources

□ Wrap lines
Check to see all the text and wrap the lines

Columns

Load by waits (AAS)	On
SQL statements	On
Support ID	Off
ID	Off
calls/sec (calls_per_sec)	On
rows/sec (rows_per_sec)	On
AAE (total_time_per_sec)	On
blk hits/sec (shared_blks_hit_per_sec)	On
blk reads/sec (shared_blks_read_per_sec)	Off
blk dirty/sec (shared_blks_dirtied_per_sec)	Off
blk writes/sec (shared_blks_written_per_sec)	Off
local blk hits/sec (local_blks_hit_per_sec)	Off

The following example shows the preferences for Aurora MySQL DB instances.



Preferences

Page size

● All resources

□ Wrap lines
Check to see all the text and wrap the lines

Columns

Load by waits (AAS)	●
SQL statements	●
Support ID	○
ID	○
calls/sec (count_star_per_sec)	○
AAE (sum_timer_wait_per_sec)	○
select full join/sec (sum_select_full_join_per_sec)	○
select range check/sec (sum_select_range_check_per_sec)	○

7. Choose Save to save your preferences.

   The **Top SQL** table refreshes.

Viewing Performance Insights proactive recommendations

Amazon RDS Performance Insights monitors specific metrics and automatically creates thresholds by analyzing what levels might be potentially problematic for a specified resource. When the new metric values cross a predefined threshold over a given period of time, Performance Insights generates a proactive recommendation. This recommendation helps to prevent future database performance impact. To receive these proactive recommendations, you must turn on Performance Insights with a paid tier retention period.

For more information about turning on Performance Insights, see Turning Performance Insights on and off for Aurora. For information about pricing and data retention for Performance Insights, see Pricing and data retention for Performance Insights.

To find out the regions, DB engines, and instance classes supported for the proactive recommendations, see Amazon Aurora DB engine, Region, and instance class support for Performance Insights features.

You can view the detailed analysis and recommended investigations of proactive recommendations in the recommendation details page.

For more information about recommendations, see Viewing and responding to Amazon Aurora recommendations.

To view the detailed analysis of a proactive recommendation

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, do any of the following:

   - Choose **Recommendations**.

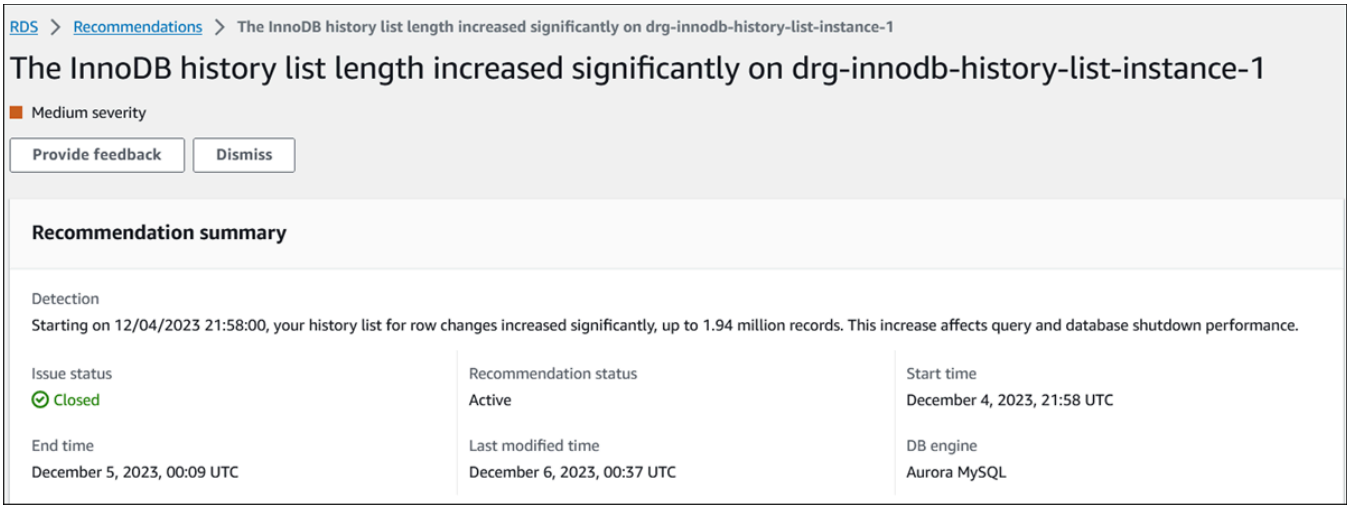
     The **Recommendations** page displays a list of recommendations sorted by the severity for all the resources in your account.
   - Choose **Databases** and then choose **Recommendations** for a resource in the databases page.

     The **Recommendations** tab displays the recommendations and its details for the selected resource.
3. Find a proactive recommendation and choose **View details**.

   The recommendation details page appears. The title provides the name of the affected resource with the issue detected and the severity.

   The following are the components on the recommendation details page:

   - **Recommendation summary** – The detected issue, recommendation and issue status, issue start and end time, recommendation modified time, and the engine type.



RDS > Recommendations > The InnoDB history list length increased significantly on drg-innodb-history-list-instance-1

The InnoDB history list length increased significantly on drg-innodb-history-list-instance-1

🟧 Medium severity

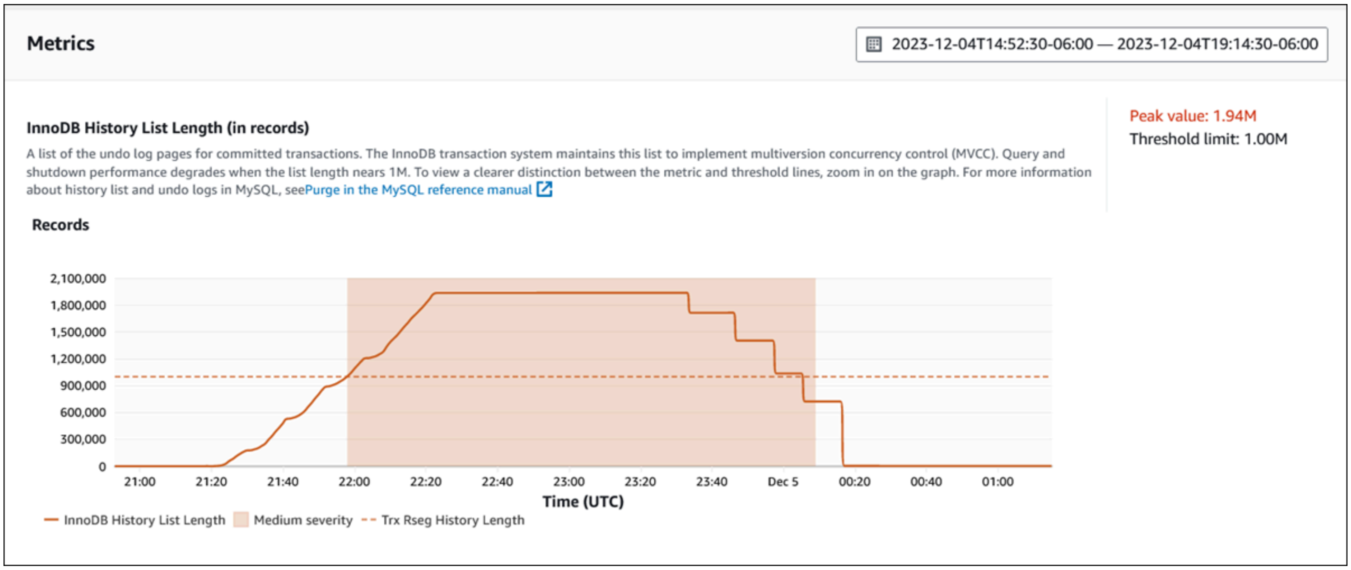
Recommendation summary

Detection

Starting on 12/04/2023 21:58:00, your history list for row changes increased significantly, up to 1.94 million records. This increase affects query and database shutdown performance.

Issue status	Recommendation status	Start time
✓ Closed	Active	December 4, 2023, 21:58 UTC
End time	**Last modified time**	**DB engine**
December 5, 2023, 00:09 UTC	December 6, 2023, 00:37 UTC	Aurora MySQL

- **Metrics** – The graphs of the detected issue. Each graph displays a threshold determined by the resource's baseline behavior and data of the metric reported from the issue start time.



Metrics

2023-12-04T14:52:30-06:00 — 2023-12-04T19:14:30-06:00

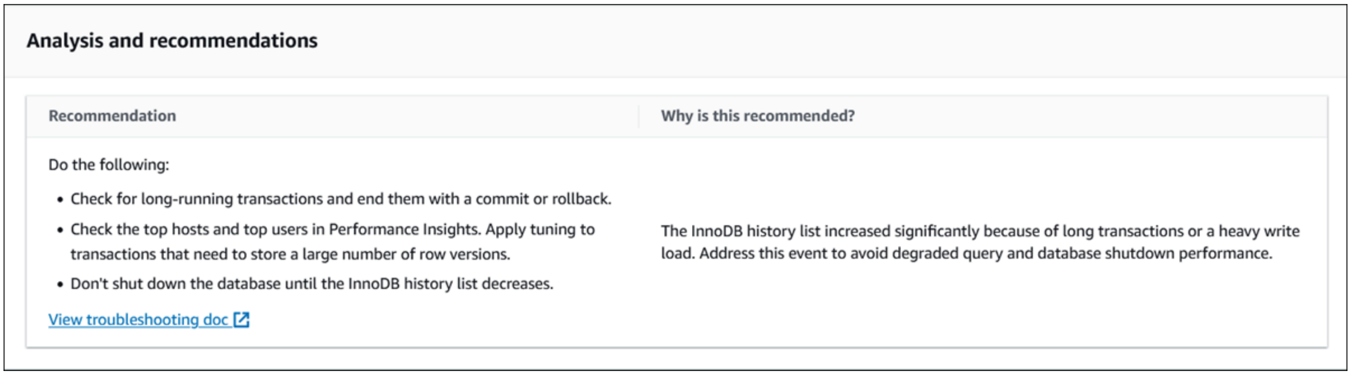
InnoDB History List Length (in records)

A list of the undo log pages for committed transactions. The InnoDB transaction system maintains this list to implement multiversion concurrency control (MVCC). Query and shutdown performance degrades when the list length nears 1M. To view a clearer distinction between the metric and threshold lines, zoom in on the graph. For more information about history list and undo logs in MySQL, seePurge in the MySQL reference manual

Peak value: 1.94M

Threshold limit: 1.00M

- **Analysis and recommendations** – The recommendation and the reason for the suggested recommendation.



Analysis and recommendations

Recommendation	Why is this recommended?
Do the following: • Check for long-running transactions and end them with a commit or rollback. • Check the top hosts and top users in Performance Insights. Apply tuning to transactions that need to store a large number of row versions. • Don't shut down the database until the InnoDB history list decreases. View troubleshooting doc	The InnoDB history list increased significantly because of long transactions or a heavy write load. Address this event to avoid degraded query and database shutdown performance.

You can review the cause of the issue and then perform the suggested recommended actions to fix the issue, or choose **Dismiss** in the upper right to dismiss the recommendation.

Retrieving metrics with the Performance Insights API for Aurora

When Performance Insights is turned on, the API provides visibility into instance performance. Amazon CloudWatch Logs provides the authoritative source for vended monitoring metrics for AWS services.

Performance Insights offers a domain-specific view of database load measured as average active sessions (AAS). This metric appears to API consumers as a two-dimensional time-series dataset. The time dimension of the data provides DB load data for each time point in the queried time range. Each time point decomposes overall load in relation to the requested dimensions, such as `SQL`, `Wait-event`, `User`, or `Host`, measured at that time point.

Amazon RDS Performance Insights monitors your Amazon Aurora cluster so that you can analyze and troubleshoot database performance. One way to view Performance Insights data is in the AWS Management Console. Performance Insights also provides a public API so that you can query your own data. You can use the API to do the following:

- Offload data into a database
- Add Performance Insights data to existing monitoring dashboards
- Build monitoring tools

To use the Performance Insights API, enable Performance Insights on one of your Amazon RDS DB instances. For information about enabling Performance Insights, see Turning Performance Insights on and off for Aurora. For more information about the Performance Insights API, see the Amazon RDS Performance Insights API Reference.

The Performance Insights API provides the following operations.

Performance Insights action	AWS CLI command	Description
`CreatePerformanceAnalysisReport`	`aws pi create-performance-analysis-report`	Creates a performance analysis report for a specific time period for the DB

Performance Insights action	AWS CLI command	Description
		instance. The result is `AnalysisReportId` which is the unique identifier of the report.
`DeletePerformanceAnalysisReport`	`aws pi delete-performance-analysis-report`	Deletes a performance analysis report.
`DescribeDimensionKeys`	`aws pi describe-dimension-keys`	Retrieves the top N dimension keys for a metric for a specific time period.
`GetDimensionKeyDetails`	`aws pi get-dimension-key-details`	Retrieves the attributes of the specified dimension group for a DB instance or data source. For example, if you specify a SQL ID, and if the dimension details are available, `GetDimensionKeyDetails` retrieves the full text of the dimension `db.sql.statement` associated with this ID. This operation is useful because `GetResourceMetrics` and `DescribeDimensionKeys` don't support retrieval of large SQL statement text.
`GetPerformanceAnalysisReport`	`aws pi get-performance-analysis-report`	Retrieves the report including the insights for the report. The result includes the report status, report ID, report time details, insights, and recommendations.

Performance Insights action	AWS CLI command	Description
GetResourceMetadata	aws pi get-resource-metadata	Retrieve the metadata for different features. For example, the metadata might indicate that a feature is turned on or off on a specific DB instance.
GetResourceMetrics	aws pi get-resource-metrics	Retrieves Performance Insights metrics for a set of data sources over a time period. You can provide specific dimension groups and dimensions, and provide aggregation and filtering criteria for each group.
ListAvailableResourceDimensions	aws pi list-available-resource-dimensions	Retrieve the dimensions that can be queried for each specified metric type on a specified instance.
ListAvailableResourceMetrics	aws pi list-available-resource-metrics	Retrieve all available metrics of the specified metric types that can be queried for a specified DB instance.
ListPerformanceAnalysisReports	aws pi list-performance-analysis-reports	Retrieves all the analysis reports available for the DB instance. The reports are listed based on the start time of each report.
ListTagsForResource	aws pi list-tags-for-resource	Lists all the metadata tags added to the resource. The list includes the name and value of the tag.

Performance Insights action	AWS CLI command	Description
TagResource	aws pi tag-resource	Adds metadata tags to the Amazon RDS resource. The tag includes a name and a value.
UntagResource	aws pi untag-resource	Removes the metadata tag from the resource.

Topics

- AWS CLI for Performance Insights
- Retrieving time-series metrics
- AWS CLI examples for Performance Insights

AWS CLI for Performance Insights

You can view Performance Insights data using the AWS CLI. You can view help for the AWS CLI commands for Performance Insights by entering the following on the command line.

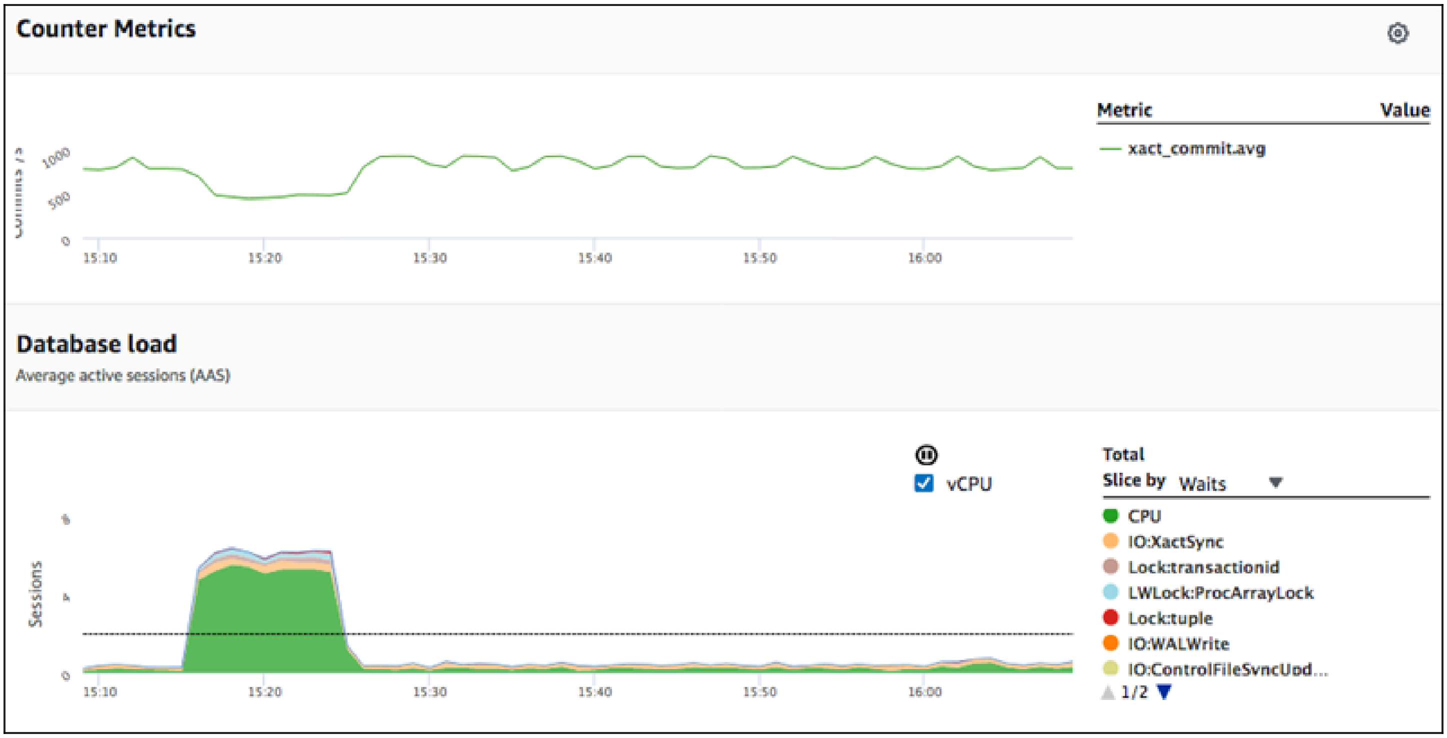
```
aws pi help
```

If you don't have the AWS CLI installed, see Installing the AWS CLI in the *AWS CLI User Guide* for information about installing it.

Retrieving time-series metrics

The `GetResourceMetrics` operation retrieves one or more time-series metrics from the Performance Insights data. `GetResourceMetrics` requires a metric and time period, and returns a response with a list of data points.

For example, the AWS Management Console uses `GetResourceMetrics` to populate the **Counter Metrics** chart and the **Database Load** chart, as seen in the following image.



All metrics returned by `GetResourceMetrics` are standard time-series metrics, with the exception of `db.load`. This metric is displayed in the **Database Load** chart. The `db.load` metric is different from the other time-series metrics because you can break it into subcomponents called *dimensions*. In the previous image, `db.load` is broken down and grouped by the waits states that make up the `db.load`.

Note

`GetResourceMetrics` can also return the `db.sampleload` metric, but the `db.load` metric is appropriate in most cases.

For information about the counter metrics returned by `GetResourceMetrics`, see Performance Insights counter metrics.

The following calculations are supported for the metrics:

- Average – The average value for the metric over a period of time. Append `.avg` to the metric name.
- Minimum – The minimum value for the metric over a period of time. Append `.min` to the metric name.

- Maximum – The maximum value for the metric over a period of time. Append `.max` to the metric name.
- Sum – The sum of the metric values over a period of time. Append `.sum` to the metric name.
- Sample count – The number of times the metric was collected over a period of time. Append `.sample_count` to the metric name.

For example, assume that a metric is collected for 300 seconds (5 minutes), and that the metric is collected one time each minute. The values for each minute are 1, 2, 3, 4, and 5. In this case, the following calculations are returned:

- Average – 3
- Minimum – 1
- Maximum – 5
- Sum – 15
- Sample count – 5

For information about using the `get-resource-metrics` AWS CLI command, see [`get-resource-metrics`]().

For the `--metric-queries` option, specify one or more queries that you want to get results for. Each query consists of a mandatory `Metric` and optional `GroupBy` and `Filter` parameters. The following is an example of a `--metric-queries` option specification.

```
{
   "Metric": "string",
   "GroupBy": {
     "Group": "string",
     "Dimensions": ["string", ...],
     "Limit": integer
   },
   "Filter": {"string": "string"
     ...}
```

AWS CLI examples for Performance Insights

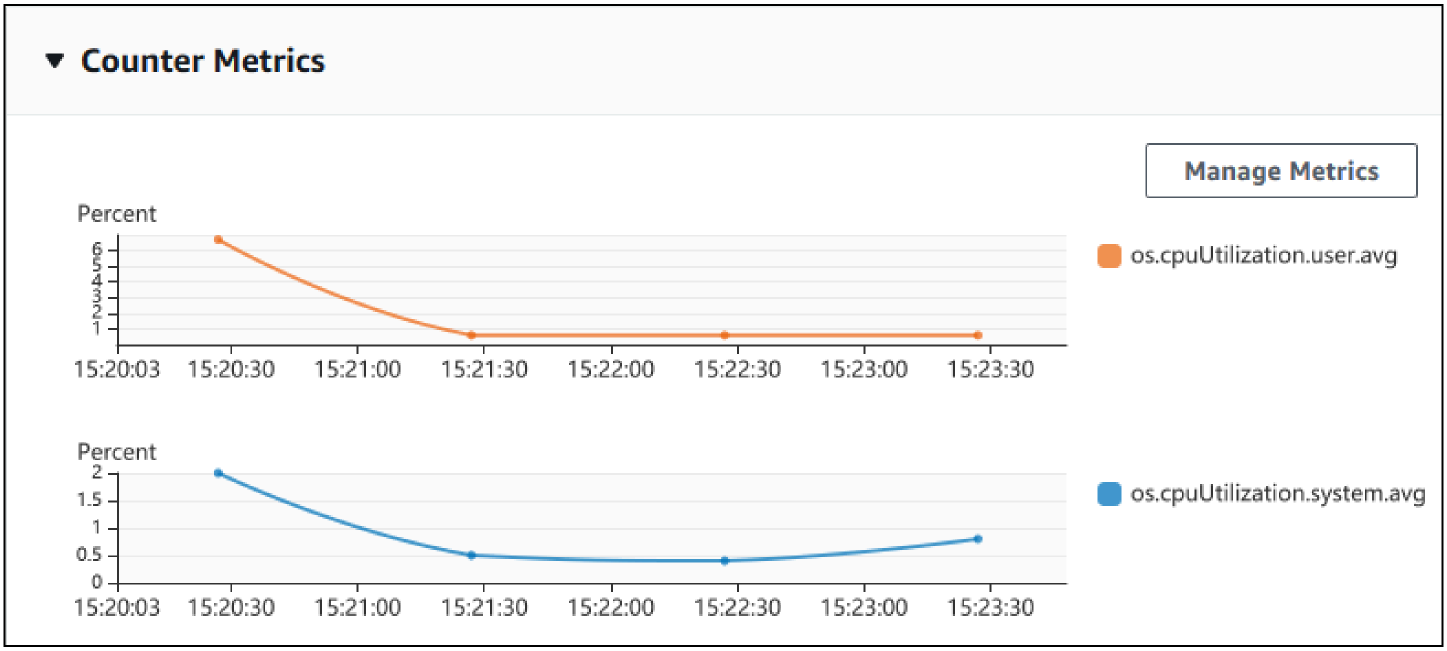
The following examples show how to use the AWS CLI for Performance Insights.

Topics

- Retrieving counter metrics
- Retrieving the DB load average for top wait events
- Retrieving the DB load average for top SQL
- Retrieving the DB load average filtered by SQL
- Retrieving the full text of a SQL statement
- Creating a performance analysis report for a time period
- Retrieving a performance analysis report
- Listing all the performance analysis reports for the DB instance
- Deleting a performance analysis report
- Adding tag to a performance analysis report
- Listing all the tags for a performance analysis report
- Deleting tags from a performance analysis report

Retrieving counter metrics

The following screenshot shows two counter metrics charts in the AWS Management Console.



The following example shows how to gather the same data that the AWS Management Console uses to generate the two counter metric charts.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type RDS \
   --identifier db-ID \
   --start-time 2018-10-30T00:00:00Z \
   --end-time   2018-10-30T01:00:00Z \
   --period-in-seconds 60 \
   --metric-queries '[{"Metric": "os.cpuUtilization.user.avg"  },
                      {"Metric": "os.cpuUtilization.idle.avg"}]'
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type RDS ^
   --identifier db-ID ^
   --start-time 2018-10-30T00:00:00Z ^
   --end-time   2018-10-30T01:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries '[{"Metric": "os.cpuUtilization.user.avg"  },
                      {"Metric": "os.cpuUtilization.idle.avg"}]'
```

You can also make a command easier to read by specifying a file for the `--metrics-query` option. The following example uses a file called query.json for the option. The file has the following contents.

```
[
    {
        "Metric": "os.cpuUtilization.user.avg"
    },
    {
        "Metric": "os.cpuUtilization.idle.avg"
    }
]
```

Run the following command to use the file.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type RDS \
   --identifier db-ID \
   --start-time 2018-10-30T00:00:00Z \
   --end-time   2018-10-30T01:00:00Z \
   --period-in-seconds 60 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type RDS ^
   --identifier db-ID ^
   --start-time 2018-10-30T00:00:00Z ^
   --end-time   2018-10-30T01:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries file://query.json
```

The preceding example specifies the following values for the options:

- `--service-type` – RDS for Amazon RDS
- `--identifier` – The resource ID for the DB instance
- `--start-time` and `--end-time` – The ISO 8601 `DateTime` values for the period to query, with multiple supported formats

It queries for a one-hour time range:

- `--period-in-seconds` – `60` for a per-minute query
- `--metric-queries` – An array of two queries, each just for one metric.

  The metric name uses dots to classify the metric in a useful category, with the final element being a function. In the example, the function is `avg` for each query. As with Amazon CloudWatch, the supported functions are `min`, `max`, `total`, and `avg`.

The response looks similar to the following.

```
{
    "Identifier": "db-XXX",
    "AlignedStartTime": 1540857600.0,
    "AlignedEndTime": 1540861200.0,
    "MetricList": [
        { //A list of key/datapoints
            "Key": {
                "Metric": "os.cpuUtilization.user.avg" //Metric1
            },
            "DataPoints": [
                //Each list of datapoints has the same timestamps and same number of
 items
                {
                    "Timestamp": 1540857660.0, //Minute1
                    "Value": 4.0
                },
                {
                    "Timestamp": 1540857720.0, //Minute2
                    "Value": 4.0
                },
                {
                    "Timestamp": 1540857780.0, //Minute 3
                    "Value": 10.0
                }
                //... 60 datapoints for the os.cpuUtilization.user.avg metric
            ]
        },
        {
            "Key": {
                "Metric": "os.cpuUtilization.idle.avg" //Metric2
            },
            "DataPoints": [
                {
                    "Timestamp": 1540857660.0, //Minute1
                    "Value": 12.0
                },
                {
                    "Timestamp": 1540857720.0, //Minute2
                    "Value": 13.5
                },
                //... 60 datapoints for the os.cpuUtilization.idle.avg metric
            ]
        }
    ] //end of MetricList
} //end of response
```

The response has an `Identifier`, `AlignedStartTime`, and `AlignedEndTime`. B the `--period-in-seconds` value was `60`, the start and end times have been aligned to the minute. If the `--period-in-seconds` was `3600`, the start and end times would have been aligned to the hour.

The `MetricList` in the response has a number of entries, each with a `Key` and a `DataPoints` entry. Each `DataPoint` has a `Timestamp` and a `Value`. Each `Datapoints` list has 60 data points because the queries are for per-minute data over an hour, with `Timestamp1/Minute1`, `Timestamp2/Minute2`, and so on, up to `Timestamp60/Minute60`.

Because the query is for two different counter metrics, there are two elements in the response `MetricList`.

Retrieving the DB load average for top wait events

The following example is the same query that the AWS Management Console uses to generate a stacked area line graph. This example retrieves the `db.load.avg` for the last hour with load divided according to the top seven wait events. The command is the same as the command in Retrieving counter metrics. However, the query.json file has the following contents.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.wait_event", "Limit": 7 }
    }
]
```

Run the following command.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type RDS \
   --identifier db-ID \
   --start-time 2018-10-30T00:00:00Z \
   --end-time   2018-10-30T01:00:00Z \
   --period-in-seconds 60 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type RDS ^
   --identifier db-ID ^
   --start-time 2018-10-30T00:00:00Z ^
   --end-time   2018-10-30T01:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries file://query.json
```

The example specifies the metric of `db.load.avg` and a `GroupBy` of the top seven wait events. For details about valid values for this example, see DimensionGroup in the *Performance Insights API Reference.*

The response looks similar to the following.

```
{
    "Identifier": "db-XXX",
    "AlignedStartTime": 1540857600.0,
    "AlignedEndTime": 1540861200.0,
    "MetricList": [
        { //A list of key/datapoints
            "Key": {
                //A Metric with no dimensions. This is the total db.load.avg
                "Metric": "db.load.avg"
            },
            "DataPoints": [
                //Each list of datapoints has the same timestamps and same number of
 items
                {
                    "Timestamp": 1540857660.0, //Minute1
                    "Value": 0.5166666666666667
                },
                {
                    "Timestamp": 1540857720.0, //Minute2
                    "Value": 0.38333333333333336
                },
                {
                    "Timestamp": 1540857780.0, //Minute 3
                    "Value": 0.26666666666666666
                }
                //... 60 datapoints for the total db.load.avg key
            ]
        },
        {
            "Key": {
                //Another key. This is db.load.avg broken down by CPU
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.name": "CPU",
                    "db.wait_event.type": "CPU"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1540857660.0, //Minute1
                    "Value": 0.35
                },
                {
                    "Timestamp": 1540857720.0, //Minute2
                    "Value": 0.15
                },
                //... 60 datapoints for the CPU key
            ]
        },
        //... In total we have 8 key/datapoints entries, 1) total, 2-8) Top Wait Events
    ] //end of MetricList
} //end of response
```

In this response, there are eight entries in the `MetricList`. There is one entry for the total `db.load.avg`, and seven entries each for the `db.load.avg` divided according to one of the top seven wait events. Unlike in the first example, because there was a grouping dimension, there must be one key for each grouping of the metric. There can't be only one key for each metric, as in the basic counter metric use case.

Retrieving the DB load average for top SQL

The following example groups `db.wait_events` by the top 10 SQL statements. There are two different groups for SQL statements:

- `db.sql` – The full SQL statement, such as `select * from customers where customer_id = 123`
- `db.sql_tokenized` – The tokenized SQL statement, such as `select * from customers where customer_id = ?`

When analyzing database performance, it can be useful to consider SQL statements that only differ by their parameters as one logic item. So, you can use `db.sql_tokenized` when querying. However, especially when you're interested in explain plans, sometimes it's more useful to examine full SQL statements with parameters, and query grouping by `db.sql`. There is a parent-child relationship between tokenized and full SQL, with multiple full SQL (children) grouped under the same tokenized SQL (parent).

The command in this example is the similar to the command in Retrieving the DB load average for top wait events. However, the query.json file has the following contents.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.sql_tokenized", "Limit": 10 }
    }
]
```

The following example uses `db.sql_tokenized`.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type RDS \
   --identifier db-ID \
   --start-time 2018-10-29T00:00:00Z \
   --end-time   2018-10-30T00:00:00Z \
   --period-in-seconds 3600 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type RDS ^
   --identifier db-ID ^
   --start-time 2018-10-29T00:00:00Z ^
   --end-time   2018-10-30T00:00:00Z  ^
   --period-in-seconds 3600 ^
   --metric-queries file://query.json
```

This example queries over 24 hours, with a one hour period-in-seconds.

The example specifies the metric of `db.load.avg` and a `GroupBy` of the top seven wait events. For details about valid values for this example, see DimensionGroup in the *Performance Insights API Reference.*

The response looks similar to the following.

```
{
    "AlignedStartTime": 1540771200.0,
    "AlignedEndTime": 1540857600.0,
    "Identifier": "db-XXX",

    "MetricList": [ //11 entries in the MetricList
        {
            "Key": { //First key is total
                "Metric": "db.load.avg"
            }
            "DataPoints": [ //Each DataPoints list has 24 per-hour Timestamps and a
 value
                {
                    "Value": 1.6964980544747081,
                    "Timestamp": 1540774800.0
                },
                //... 24 datapoints
            ]
        },
        {
            "Key": { //Next key is the top tokenized SQL
                "Dimensions": {
                    "db.sql_tokenized.statement": "INSERT INTO authors (id,name,email)
 VALUES\n( nextval(?)  ,?,?)",
                    "db.sql_tokenized.db_id": "pi-2372568224",
                    "db.sql_tokenized.id": "AKIAIOSFODNN7EXAMPLE"
                },
                "Metric": "db.load.avg"
            },
            "DataPoints": [ //... 24 datapoints
            ]
        },
        // In total 11 entries, 10 Keys of top tokenized SQL, 1 total key
    ] //End of MetricList
} //End of response
```

This response has 11 entries in the `MetricList` (1 total, 10 top tokenized SQL), with each entry having 24 per-hour `DataPoints`.

For tokenized SQL, there are three entries in each dimensions list:

- `db.sql_tokenized.statement` – The tokenized SQL statement.
- `db.sql_tokenized.db_id` – Either the native database ID used to refer to the SQL, or a synthetic ID that Performance Insights generates for you if the native database ID isn't available. This example returns the `pi-2372568224` synthetic ID.

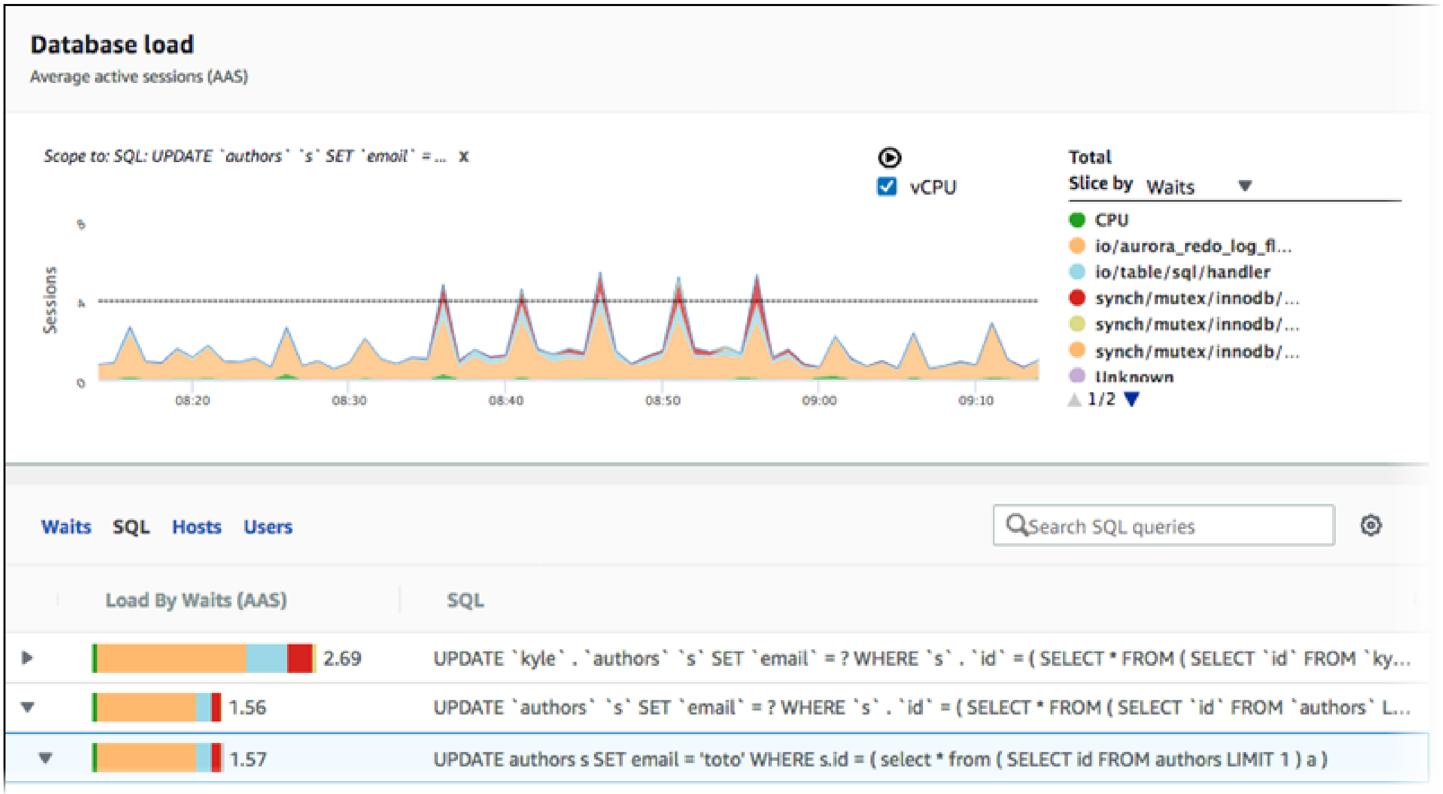
- `db.sql_tokenized.id` – The ID of the query inside Performance Insights.

  In the AWS Management Console, this ID is called the Support ID. It's named this because the ID is data that AWS Support can examine to help you troubleshoot an issue with your database. AWS takes the security and privacy of your data extremely seriously, and almost all data is stored encrypted with your AWS KMS key. Therefore, nobody inside AWS can look at this data. In the example preceding, both the `tokenized.statement` and the `tokenized.db_id` are stored encrypted. If you have an issue with your database, AWS Support can help you by referencing the Support ID.

When querying, it might be convenient to specify a `Group` in `GroupBy`. However, for finer-grained control over the data that's returned, specify the list of dimensions. For example, if all that is needed is the `db.sql_tokenized.statement`, then a `Dimensions` attribute can be added to the query.json file.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": {
            "Group": "db.sql_tokenized",
            "Dimensions":["db.sql_tokenized.statement"],
            "Limit": 10
        }
    }
]
```

Retrieving the DB load average filtered by SQL



	Load By Waits (AAS)	SQL
▶	2.69	UPDATE `kyle` . `authors` `s` SET `email` = ? WHERE `s` . `id` = (SELECT * FROM (SELECT `id` FROM `ky...
▼	1.56	UPDATE `authors` `s` SET `email` = ? WHERE `s` . `id` = (SELECT * FROM (SELECT `id` FROM `authors` L...
▼	1.57	UPDATE authors s SET email = 'toto' WHERE s.id = (select * from (SELECT id FROM authors LIMIT 1) a)

The preceding image shows that a particular query is selected, and the top average active sessions stacked area line graph is scoped to that query. Although the query is still for the top seven overall wait events, the value of the response is filtered. The filter causes it to take into account only sessions that are a match for the particular filter.

The corresponding API query in this example is similar to the command in Retrieving the DB load average for top SQL. However, the query.json file has the following contents.

```
[
 {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.wait_event", "Limit": 5  },
        "Filter": { "db.sql_tokenized.id": "AKIAIOSFODNN7EXAMPLE" }
    }
]
```

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type RDS \
   --identifier db-ID \
   --start-time 2018-10-30T00:00:00Z \
   --end-time   2018-10-30T01:00:00Z \
   --period-in-seconds 60 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type RDS ^
   --identifier db-ID ^
   --start-time 2018-10-30T00:00:00Z ^
   --end-time   2018-10-30T01:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries file://query.json
```

The response looks similar to the following.

```
{
    "Identifier": "db-XXX",
    "AlignedStartTime": 1556215200.0,
    "MetricList": [
        {
            "Key": {
                "Metric": "db.load.avg"
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 1.4878117913832196
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 1.192823803967328
                }
            ]
        },
        {
            "Key": {
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.type": "io",
                    "db.wait_event.name": "wait/io/aurora_redo_log_flush"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 1.1360544217687074
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 1.058051341890315
                }
            ]
        },
        {
            "Key": {
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.type": "io",
                    "db.wait_event.name": "wait/io/table/sql/handler"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 0.16241496598639457
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 0.05163360560093349
                }
            ]
        },
        {
            "Key": {
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.type": "synch",
                    "db.wait_event.name": "wait/synch/mutex/innodb/
aurora_lock_thread_slot_futex"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 0.11479591836734694
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 0.013127187864644107
                }
            ]
        },
        {
            "Key": {
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.type": "CPU",
                    "db.wait_event.name": "CPU"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 0.05215419501133787
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 0.05805134189031505
                }
            ]
        },
        {
            "Key": {
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_event.type": "synch",
                    "db.wait_event.name": "wait/synch/mutex/innodb/lock_wait_mutex"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": 1556218800.0,
                    "Value": 0.017573696145124718
                },
                {
                    "Timestamp": 1556222400.0,
                    "Value": 0.002333722287047841
                }
            ]
        }
    ],
    "AlignedEndTime": 1556222400.0
} //end of response
```

In this response, all values are filtered according to the contribution of tokenized SQL AKIAIOSFODNN7EXAMPLE specified in the query.json file. The keys also might follow a different order than a query without a filter, because it's the top five wait events that affected the filtered SQL.

Retrieving the full text of a SQL statement

The following example retrieves the full text of a SQL statement for DB instance `db-10BCD2EFGHIJ3KL4M5NO6PQRS5`. The `--group` is `db.sql`, and the `--group-identifier` is `db.sql.id`. In this example, *`my-sql-id`* represents a SQL ID retrieved by invoking `pi get-resource-metrics` or `pi describe-dimension-keys`.

Run the following command.

For Linux, macOS, or Unix:

```
aws pi get-dimension-key-details \
   --service-type RDS \
   --identifier db-10BCD2EFGHIJ3KL4M5NO6PQRS5 \
   --group db.sql \
   --group-identifier my-sql-id \
   --requested-dimensions statement
```

For Windows:

```
aws pi get-dimension-key-details ^
   --service-type RDS ^
   --identifier db-10BCD2EFGHIJ3KL4M5NO6PQRS5 ^
   --group db.sql ^
   --group-identifier my-sql-id ^
   --requested-dimensions statement
```

In this example, the dimensions details are available. Thus, Performance Insights retrieves the full text of the SQL statement, without truncating it.

```
{
    "Dimensions":[
    {
        "Value": "SELECT e.last_name, d.department_name FROM employees e, departments d
 WHERE e.department_id=d.department_id",
        "Dimension": "db.sql.statement",
        "Status": "AVAILABLE"
    },
    ...
    ]
}
```

Creating a performance analysis report for a time period

The following example creates a performance analysis report with the `1682969503` start time and `1682979503` end time for the `db-loadtest-0` database.

```
aws pi create-performance-analysis-report \
        --service-type RDS \
        --identifier db-loadtest-0 \
        --start-time 1682969503 \
        --end-time 1682979503 \
        --region us-west-2
```

The response is the unique identifier `report-0234d3ed98e28fb17` for the report.

```
{
   "AnalysisReportId": "report-0234d3ed98e28fb17"
}
```

Retrieving a performance analysis report

The following example retrieves the analysis report details for the `report-0d99cc91c4422ee61` report.

```
aws pi get-performance-analysis-report \
--service-type RDS \
--identifier db-loadtest-0 \
--analysis-report-id report-0d99cc91c4422ee61 \
--region us-west-2
```

The response provides the report status, ID, time details, and insights.

```
        {
    "AnalysisReport": {
        "Status": "Succeeded",
        "ServiceType": "RDS",
        "Identifier": "db-loadtest-0",
        "StartTime": 1680583486.584,
        "AnalysisReportId": "report-0d99cc91c4422ee61",
        "EndTime": 1680587086.584,
        "CreateTime": 1680587087.139,
        "Insights": [
           ... (Condensed for space)
        ]
    }
}
```

Listing all the performance analysis reports for the DB instance

The following example lists all the available performance analysis reports for the `db-loadtest-0` database.

```
aws pi list-performance-analysis-reports \
--service-type RDS \
--identifier db-loadtest-0 \
--region us-west-2
```

The response lists all the reports with the report ID, status, and time period details.

```
{
    "AnalysisReports": [
        {
            "Status": "Succeeded",
            "EndTime": 1680587086.584,
            "CreationTime": 1680587087.139,
            "StartTime": 1680583486.584,
            "AnalysisReportId": "report-0d99cc91c4422ee61"
        },
        {
            "Status": "Succeeded",
            "EndTime": 1681491137.914,
            "CreationTime": 1681491145.973,
            "StartTime": 1681487537.914,
            "AnalysisReportId": "report-002633115cc002233"
        },
        {
            "Status": "Succeeded",
            "EndTime": 1681493499.849,
            "CreationTime": 1681493507.762,
            "StartTime": 1681489899.849,
            "AnalysisReportId": "report-043b1e006b47246f9"
        },
        {
            "Status": "InProgress",
            "EndTime": 1682979503.0,
            "CreationTime": 1682979618.994,
            "StartTime": 1682969503.0,
            "AnalysisReportId": "report-01ad15f9b88bcbd56"
        }
    ]
}
```

Deleting a performance analysis report

The following example deletes the analysis report for the `db-loadtest-0` database.

```
aws pi delete-performance-analysis-report \
--service-type RDS \
--identifier db-loadtest-0 \
--analysis-report-id report-0d99cc91c4422ee61 \
--region us-west-2
```

Adding tag to a performance analysis report

The following example adds a tag with a key `name` and value `test-tag` to the `report-01ad15f9b88bcbd56` report.

```
aws pi tag-resource \
--service-type RDS \
--resource-arn arn:aws:pi:us-west-2:356798100956:perf-reports/RDS/db-loadtest-0/
report-01ad15f9b88bcbd56 \
--tags Key=name,Value=test-tag \
--region us-west-2
```

Listing all the tags for a performance analysis report

The following example lists all the tags for the `report-01ad15f9b88bcbd56` report.

```
aws pi list-tags-for-resource \
--service-type RDS \
--resource-arn arn:aws:pi:us-west-2:356798100956:perf-reports/RDS/db-loadtest-0/
report-01ad15f9b88bcbd56 \
--region us-west-2
```

The response lists the value and key for all the tags added to the report:

```
{
    "Tags": [
        {
            "Value": "test-tag",
            "Key": "name"
        }
    ]
}
```

Deleting tags from a performance analysis report

The following example deletes the `name` tag from the `report-01ad15f9b88bcbd56` report.

```
aws pi untag-resource \
--service-type RDS \
--resource-arn arn:aws:pi:us-west-2:356798100956:perf-reports/RDS/db-loadtest-0/
report-01ad15f9b88bcbd56 \
--tag-keys name \
--region us-west-2
```

After the tag is deleted, calling the `list-tags-for-resource` API doesn't list this tag.

Logging Performance Insights calls using AWS CloudTrail

Performance Insights runs with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Performance Insights. CloudTrail captures all API calls for Performance Insights as events. This capture includes calls from the Amazon RDS console and from code calls to the Performance Insights API operations.

If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Performance Insights. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the data collected by CloudTrail, you can determine certain information. This information includes the request that was made to Performance Insights, the IP address the request was made from, who made the request, and when it was made. It also includes additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Working with Performance Insights information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Performance Insights, that activity is recorded in a CloudTrail event along with other AWS service events in the CloudTrail console in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History in *AWS CloudTrail User Guide.*

For an ongoing record of events in your AWS account, including events for Performance Insights, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all AWS Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following topics in *AWS CloudTrail User Guide:*

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

All Performance Insights operations are logged by CloudTrail and are documented in the Performance Insights API Reference. For example, calls to the `DescribeDimensionKeys` and `GetResourceMetrics` operations generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Performance Insights log file entries

A *trail* is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An *event* represents a single request from any source. Each event includes information about the requested operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `GetResourceMetrics` operation.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
         "principalId": "AKIAIOSFODNN7EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/johndoe",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
        "userName": "johndoe"
    },
    "eventTime": "2019-12-18T19:28:46Z",
    "eventSource": "pi.amazonaws.com",
    "eventName": "GetResourceMetrics",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "72.21.198.67",
    "userAgent": "aws-cli/1.16.240 Python/3.7.4 Darwin/18.7.0 botocore/1.12.230",
    "requestParameters": {
        "identifier": "db-YTDU5J5V66X7CXSCVDFD2V3SZM",
        "metricQueries": [
            {
                "metric": "os.cpuUtilization.user.avg"
            },
            {
                "metric": "os.cpuUtilization.idle.avg"
            }
        ],
        "startTime": "Dec 18, 2019 5:28:46 PM",
        "periodInSeconds": 60,
        "endTime": "Dec 18, 2019 7:28:46 PM",
        "serviceType": "RDS"
    },
    "responseElements": null,
    "requestID": "9ffbe15c-96b5-4fe6-bed9-9fccff1a0525",
    "eventID": "08908de0-2431-4e2e-ba7b-f5424f908433",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

Analyzing Aurora performance anomalies with Amazon DevOps Guru for Amazon RDS

Amazon DevOps Guru is a fully managed operations service that helps developers and operators improve the performance and availability of their applications. DevOps Guru offloads the tasks associated with identifying operational issues so that you can quickly implement recommendations to improve your application. For more information, see What is Amazon DevOps Guru? in the *Amazon DevOps Guru User Guide*.

DevOps Guru detects, analyzes, and makes recommendations for existing operational issues for all Amazon RDS DB engines. DevOps Guru for RDS extends this capability by applying machine learning to Performance Insights metrics for Amazon Aurora databases. These monitoring features allow DevOps Guru for RDS to detect and diagnose performance bottlenecks and recommend specific corrective actions. DevOps Guru for RDS can also detect problematic conditions in your Aurora databases before they occur.

You can now view these recommendations in RDS console. For more information, see Viewing and responding to Amazon Aurora recommendations.

The following video is an overview of DevOps Guru for RDS.

For a deep dive on this subject, see Amazon DevOps Guru for RDS under the hood.

Topics

Benefits of DevOps Guru for RDS

If you're responsible for an Amazon Aurora database, you might not know that an event or regression that is affecting that database is occurring. When you learn about the issue, you might not know why it's occurring or what to do about it. Rather than turning to a database administrator (DBA) for help or relying on third-party tools, you can follow recommendations from DevOps Guru for RDS.

You gain the following advantages from the detailed analysis of DevOps Guru for RDS:

Fast diagnosis

DevOps Guru for RDS continuously monitors and analyzes database telemetry. Performance Insights, Enhanced Monitoring, and Amazon CloudWatch collect telemetry data for your database cluster. DevOps Guru for RDS uses statistical and machine learning techniques to mine this data and detect anomalies. To learn more about telemetry data, see Monitoring DB load with Performance Insights on Amazon Aurora and Monitoring OS metrics with Enhanced Monitoring in the *Amazon Aurora User Guide* .

Fast resolution

Each anomaly identifies the performance issue and suggests avenues of investigation or corrective actions. For example, DevOps Guru for RDS might recommend that you investigate specific wait events. Or it might recommend that you tune your application pool settings to limit the number of database connections. Based on these recommendations, you can resolve performance issues more quickly than by troubleshooting manually.

Proactive insights

DevOps Guru for RDS uses metrics from your resources to detect potentially problematic behavior before it becomes a bigger problem. For example, it can detect when your database is using an increasing number of on-disk temporary tables, which could start to impact performance. DevOps Guru then provides recommendations to help you address issues before they become bigger problems.

Deep knowledge of Amazon engineers and machine learning

To detect performance issues and help you resolve bottlenecks, DevOps Guru for RDS relies on machine learning (ML) and advanced mathematical formulas. Amazon database engineers contributed to the development of the DevOps Guru for RDS findings, which encapsulate many years of managing hundreds of thousands of databases. By drawing on this collective knowledge, DevOps Guru for RDS can teach you best practices.

How DevOps Guru for RDS works

DevOps Guru for RDS collects data about your Aurora databases from Amazon RDS Performance Insights. The most important metric is `DBLoad`. DevOps Guru for RDS consumes the Performance Insights metrics, analyzes them with machine learning, and publishes insights to the dashboard.

An *insight* is a collection of related anomalies that were detected by DevOps Guru.

In DevOps Guru for RDS, an *anomaly* is a pattern that deviates from what is considered normal performance for your Amazon Aurora database.

Proactive insights

A *proactive insight* lets you know about problematic behavior before it occurs. It contains anomalies with recommendations and related metrics to help you address issues in your Amazon Aurora databases before become bigger problems. These insights are published in the DevOps Guru dashboard.

For example, DevOps Guru might detect that your Aurora PostgreSQL database is creating many on-disk temporary tables. If not addressed, this trend might lead to performance issues. Each proactive insight includes recommendations for corrective behavior and links to relevant topics in either Tuning Aurora MySQL with Amazon DevOps Guru proactive insights or Tuning Aurora PostgreSQL with Amazon DevOps Guru proactive insights. For more information, see Working with insights in DevOps Guru in the *Amazon DevOps Guru User Guide*.

Reactive insights

A *reactive insight* identifies anomalous behavior as it occurs. If DevOps Guru for RDS finds performance issues in your Amazon Aurora DB instances, it publishes a reactive insight in the DevOps Guru dashboard. For more information, see Working with insights in DevOps Guru in the *Amazon DevOps Guru User Guide*.

Causal anomalies

A *causal anomaly* is a top-level anomaly within a reactive insight. **Database load (DB load)** is the causal anomaly for DevOps Guru for RDS.

An anomaly measures performance impact by assigning a severity level of **High**, **Medium**, or **Low**. To learn more, see Key concepts for DevOps Guru for RDS in the *Amazon DevOps Guru User Guide*.

If DevOps Guru detects a current anomaly on your DB instance, you're alerted in the **Databases** page of the RDS console. The console also alerts you to anomalies that occurred in the past 24 hours. To go to the anomaly page from the RDS console, choose the link in the alert message. The RDS console also alerts you in the page for your Amazon Aurora DB cluster .

Contextual anomalies

A *contextual anomaly* is a finding within **Database load (DB load)** that is related to a reactive insight. Each contextual anomaly describes a specific Amazon Aurora performance issue that requires investigation. For example, DevOps Guru for RDS might recommend that you consider increasing CPU capacity or investigate wait events that are contributing to DB load.

> ⚠ **Important**
>
> We recommend that you test any changes on a test instance before modifying a production instance. In this way, you understand the impact of the change.

To learn more, see Analyzing anomalies in Amazon RDS in the *Amazon DevOps Guru User Guide*.

Setting up DevOps Guru for RDS

To allow DevOps Guru for Amazon RDS to publish insights for an Amazon Aurora database, complete the following tasks.

Topics

Configuring IAM access policies for DevOps Guru for RDS

To view alerts from DevOps Guru in the RDS console, your AWS Identity and Access Management (IAM) user or role must have either of the following policies:

- The AWS managed policy `AmazonDevOpsGuruConsoleFullAccess`
- The AWS managed policy `AmazonDevOpsGuruConsoleReadOnlyAccess` and either of the following policies:
  - The AWS managed policy `AmazonRDSFullAccess`
  - A customer managed policy that includes `pi:GetResourceMetrics` and `pi:DescribeDimensionKeys`

For more information, see Configuring access policies for Performance Insights.

Turning on Performance Insights for your Aurora DB instances

DevOps Guru for RDS relies on Performance Insights for its data. Without Performance Insights, DevOps Guru publishes anomalies, but doesn't include the detailed analysis and recommendations.

When you create an Aurora DB cluster or modify a cluster instance, you can turn on Performance Insights. For more information, see Turning Performance Insights on and off for Aurora.

Turning on DevOps Guru and specifying resource coverage

You can turn on DevOps Guru to have it monitor your Amazon Aurora databases in either of the following ways.

Topics

- Turning on DevOps Guru in the RDS console
- Adding Aurora resources in the DevOps Guru console
- Adding Aurora resources using AWS CloudFormation

Turning on DevOps Guru in the RDS console

You can take multiple paths in the Amazon RDS console to turn on DevOps Guru.

Topics

- Turning on DevOps Guru when you create an Aurora database
- Turning on DevOps Guru from the notification banner
- Responding to a permissions error when you turn on DevOps Guru

Turning on DevOps Guru when you create an Aurora database

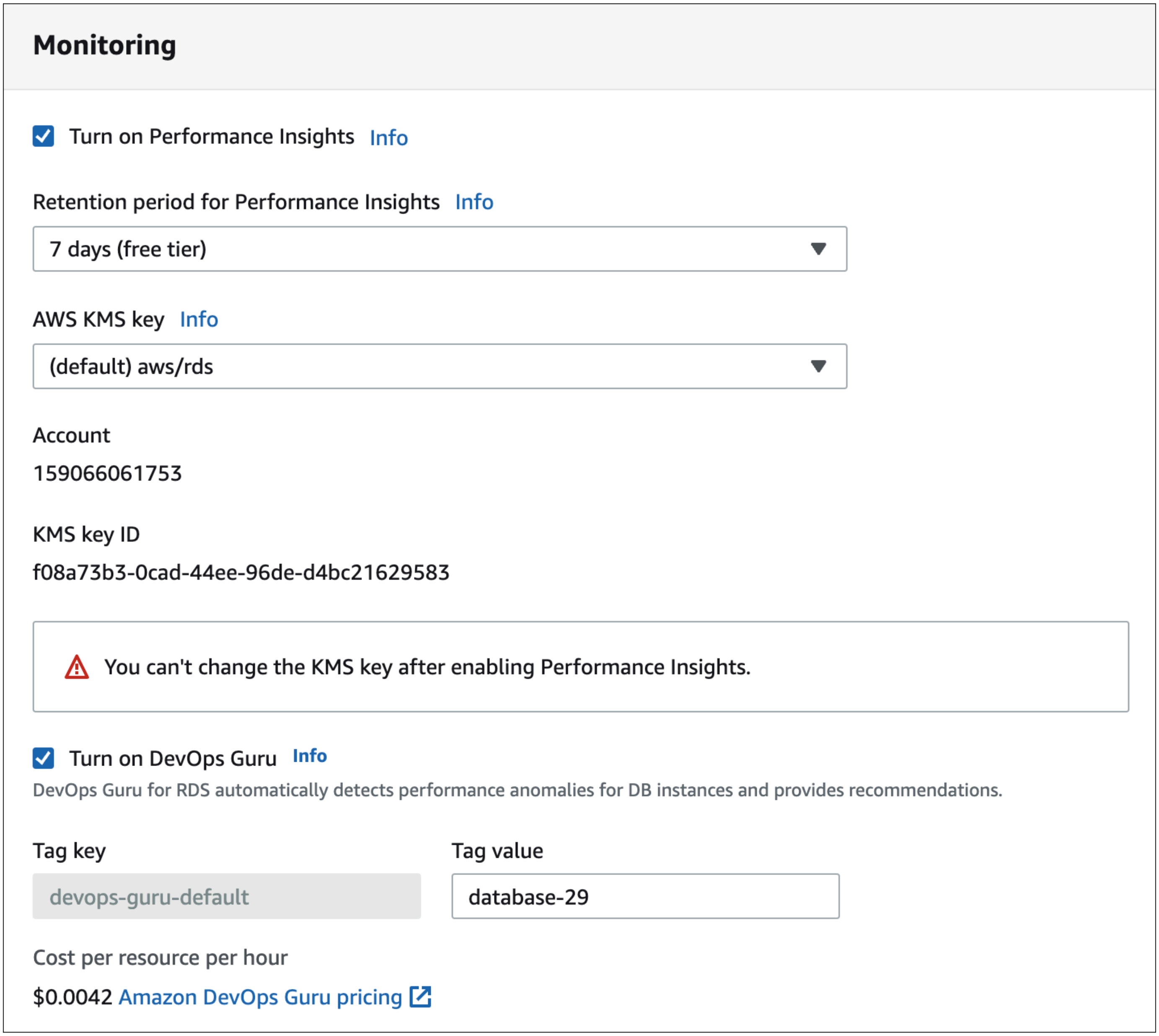
The creation workflow includes a setting that turns on DevOps Guru coverage for your database. This setting is turned on by default when you choose the **Production** template.

To turn on DevOps Guru when you create an Aurora database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Follow the steps in Creating a DB cluster, up to but not including the step where you choose monitoring settings.

3. In **Monitoring**, choose **Turn on Performance Insights**. For DevOps Guru for RDS to provide detailed analysis of performance anomalies, Performance Insights must be turned on.

4. Choose **Turn on DevOps Guru**.



Monitoring

☑ Turn on Performance Insights Info

Retention period for Performance Insights Info

7 days (free tier) ▼

AWS KMS key Info

(default) aws/rds ▼

Account

159066061753

KMS key ID

f08a73b3-0cad-44ee-96de-d4bc21629583

⚠ You can't change the KMS key after enabling Performance Insights.

☑ Turn on DevOps Guru Info

DevOps Guru for RDS automatically detects performance anomalies for DB instances and provides recommendations.

Tag key: devops-guru-default

Tag value: database-29

Cost per resource per hour

$0.0042 Amazon DevOps Guru pricing

5. Create a tag for your database so that DevOps Guru can monitor it. Do the following:

   - In the text field for **Tag key**, enter a name that begins with **`Devops-Guru-`**.
   - In the text field for **Tag value**, enter any value. For example, if you enter **`rds-database-1`** for the name of your Aurora database, you can also enter **`rds-database-1`** as the tag value.

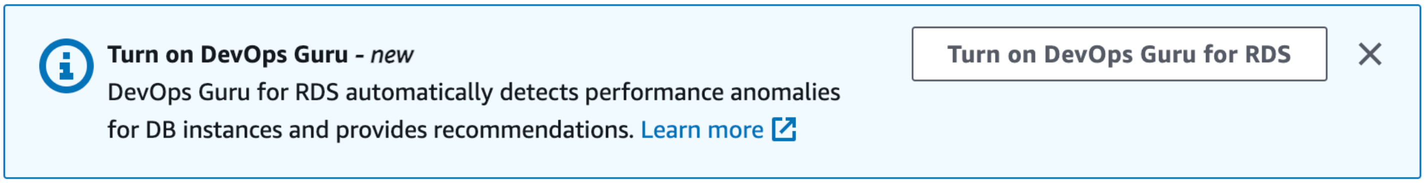
For more information about tags, see "Use tags to identify resources in your DevOps Guru applications" in the *Amazon DevOps Guru User Guide*.

6. Complete the remaining steps in Creating a DB cluster.

Turning on DevOps Guru from the notification banner

If your resources aren't covered by DevOps Guru, Amazon RDS notifies you with a banner in the following locations:

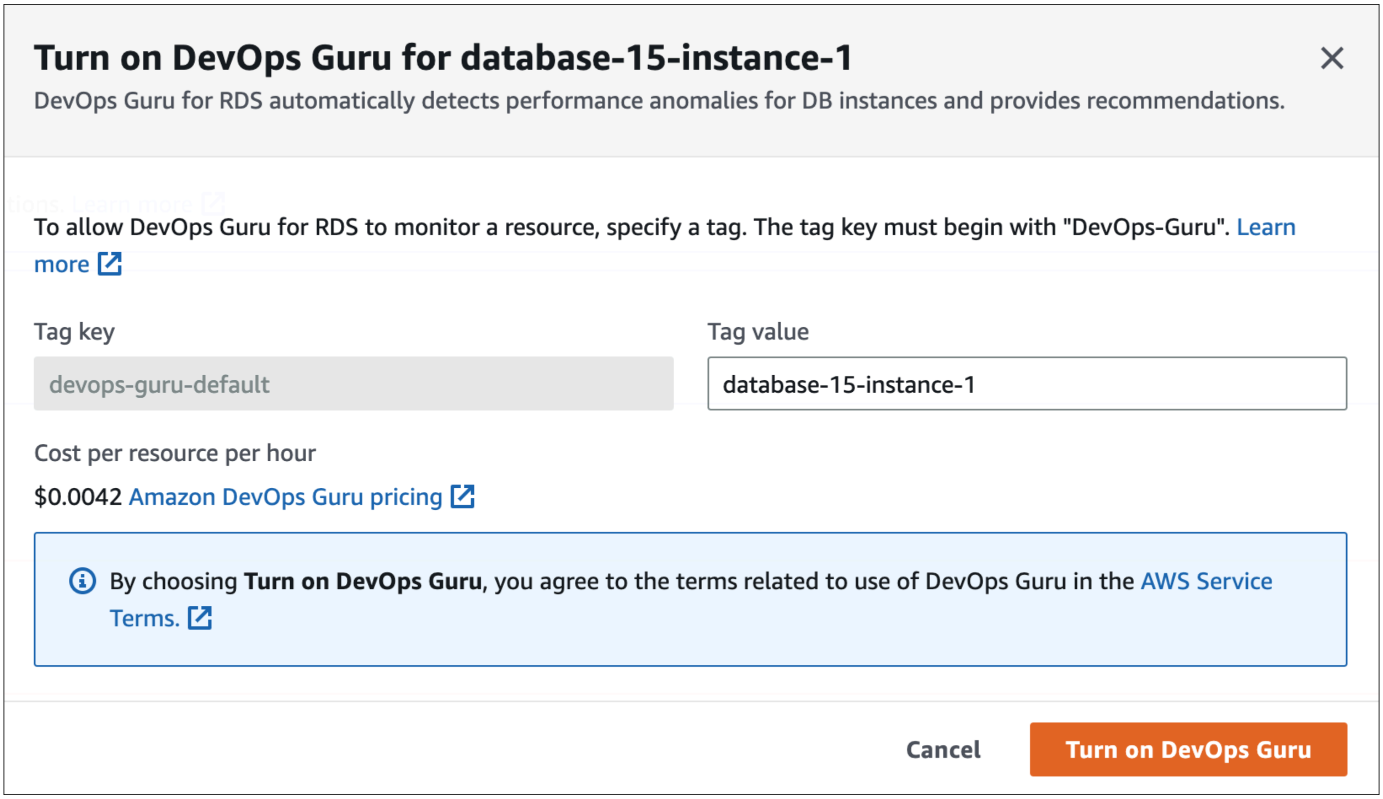
- The **Monitoring** tab of a DB cluster instance
- The Performance Insights dashboard



Turn on DevOps Guru - *new*
DevOps Guru for RDS automatically detects performance anomalies for DB instances and provides recommendations. Learn more

To turn on DevOps Guru for your Aurora database

1. In the banner, choose **Turn on DevOps Guru for RDS**.
2. Enter a tag key name and value. For more information about tags, see "Use tags to identify resources in your DevOps Guru applications" in the *Amazon DevOps Guru User Guide*.



Turn on DevOps Guru for database-15-instance-1

DevOps Guru for RDS automatically detects performance anomalies for DB instances and provides recommendations.

To allow DevOps Guru for RDS to monitor a resource, specify a tag. The tag key must begin with "DevOps-Guru". Learn more

Tag key: devops-guru-default

Tag value: database-15-instance-1

Cost per resource per hour

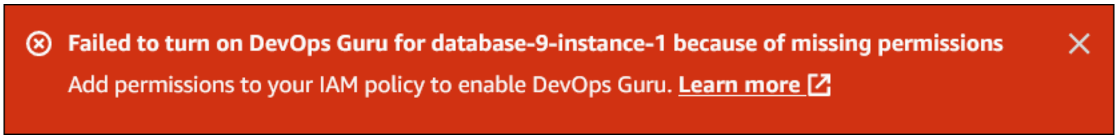
$0.0042 Amazon DevOps Guru pricing

By choosing **Turn on DevOps Guru**, you agree to the terms related to use of DevOps Guru in the AWS Service Terms.

3. Choose **Turn on DevOps Guru**.

Responding to a permissions error when you turn on DevOps Guru

If you turn on DevOps Guru from the RDS console when you create a database, RDS might display the following banner about missing permissions.



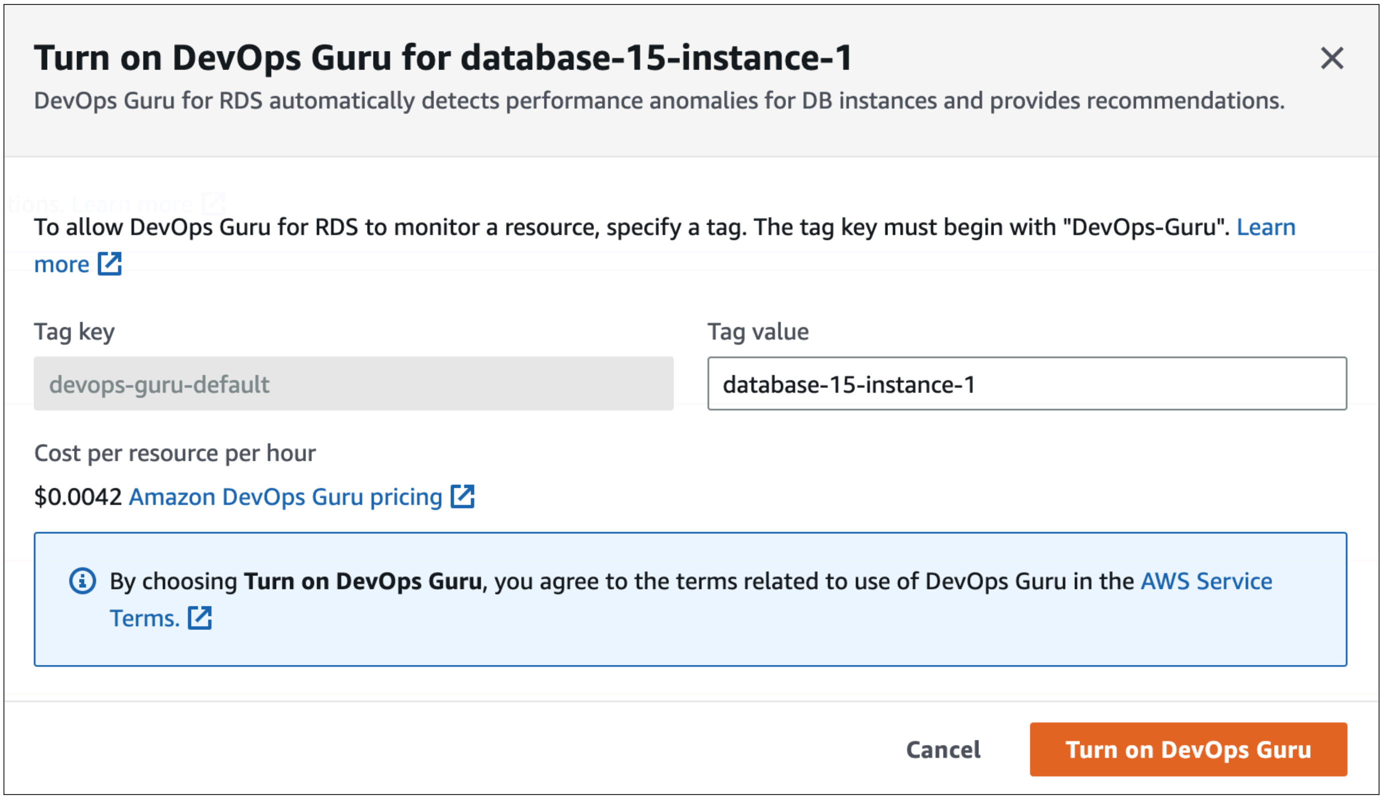
Failed to turn on DevOps Guru for database-9-instance-1 because of missing permissions

Add permissions to your IAM policy to enable DevOps Guru. Learn more

To respond to a permissions error

1. Grant your IAM user or role the user managed role `AmazonDevOpsGuruConsoleFullAccess`. For more information, see Configuring IAM access policies for DevOps Guru for RDS.
2. Open the RDS console.
3. In the navigation pane, choose **Performance Insights**.
4. Choose a DB instance in the cluster that you just created.
5. Choose the switch to turn on **DevOps Guru for RDS**.

6. Choose a tag value. For more information, see "Use tags to identify resources in your DevOps Guru applications" in the *Amazon DevOps Guru User Guide*.



Turn on DevOps Guru for database-15-instance-1

DevOps Guru for RDS automatically detects performance anomalies for DB instances and provides recommendations.

To allow DevOps Guru for RDS to monitor a resource, specify a tag. The tag key must begin with "DevOps-Guru". Learn more

Tag key: devops-guru-default

Tag value: database-15-instance-1

Cost per resource per hour

$0.0042 Amazon DevOps Guru pricing

By choosing **Turn on DevOps Guru**, you agree to the terms related to use of DevOps Guru in the AWS Service Terms.

7. Choose **Turn on DevOps Guru**.

Adding Aurora resources in the DevOps Guru console

You can specify your DevOps Guru resource coverage on the DevOps Guru console. Follow the step described in Specify your DevOps Guru resource coverage in the *Amazon DevOps Guru User Guide*. When you edit your analyzed resources, choose one of the following options:

- Choose **All account resources** to analyze all supported resources, including the Aurora databases, in your AWS account and Region.
- Choose **CloudFormation stacks** to analyze the Aurora databases that are in stacks you choose. For more information, see Use AWS CloudFormation stacks to identify resources in your DevOps Guru applications in the *Amazon DevOps Guru User Guide*.
- Choose **Tags** to analyze the Aurora databases that you have tagged. For more information, see Use tags to identify resources in your DevOps Guru applications in the *Amazon DevOps Guru User Guide*.

For more information, see Enable DevOps Guru in the *Amazon DevOps Guru User Guide*.

Adding Aurora resources using AWS CloudFormation

You can use tags to add coverage for your Aurora resources to your CloudFormation templates. The following procedure assumes that you have a CloudFormation template both for your Aurora DB instance and DevOps Guru stack.

To specify an Aurora DB instance using a CloudFormation tag

1. In the CloudFormation template for your DB instance, define a tag using a key/value pair.

   The following example assigns the value `my-aurora-db-instance1` to `Devops-guru-cfn-default` for an Aurora DB instance.

   ```
   MyAuroraDBInstance1:
     Type: "AWS::RDS::DBInstance"
     Properties:
       DBClusterIdentifier: my-aurora-db-cluster
       DBInstanceIdentifier: my-aurora-db-instance1
       Tags:
         - Key: Devops-guru-cfn-default
           Value: devopsguru-my-aurora-db-instance1
   ```

2. In the CloudFormation template for your DevOps Guru stack, specify the same tag in your resource collection filter.

   The following example configures DevOps Guru to provide coverage for the resource with the tag value `my-aurora-db-instance1`.

   ```
   DevOpsGuruResourceCollection:
     Type: AWS::DevOpsGuru::ResourceCollection
     Properties:
       ResourceCollectionFilter:
         Tags:
           - AppBoundaryKey: "Devops-guru-cfn-default"
             TagValues:
             - "devopsguru-my-aurora-db-instance1"
   ```

   The following example provides coverage for all resources within the application boundary `Devops-guru-cfn-default`.

```
DevOpsGuruResourceCollection:
  Type: AWS::DevOpsGuru::ResourceCollection
  Properties:
    ResourceCollectionFilter:
      Tags:
        - AppBoundaryKey: "Devops-guru-cfn-default"
          TagValues:
          - "*"
```

For more information, see AWS::DevOpsGuru::ResourceCollection and AWS::RDS::DBInstance in the *AWS CloudFormation User Guide*.

Monitoring OS metrics with Enhanced Monitoring

With Enhanced Monitoring, you can monitor the operating system of your DB instance in real time. When you want to see how different processes or threads use the CPU, Enhanced Monitoring metrics are useful.

Topics

- Overview of Enhanced Monitoring
- Setting up and enabling Enhanced Monitoring
- Viewing OS metrics in the RDS console
- Viewing OS metrics using CloudWatch Logs

Overview of Enhanced Monitoring

Amazon RDS provides metrics in real time for the operating system (OS) that your DB instance runs on. You can view all the system metrics and process information for your RDS DB instances on the console. You can manage which metrics you want to monitor for each instance and customize the dashboard according to your requirements. For descriptions of the Enhanced Monitoring metrics, see OS metrics in Enhanced Monitoring.

RDS delivers the metrics from Enhanced Monitoring into your Amazon CloudWatch Logs account. You can create metrics filters in CloudWatch from CloudWatch Logs and display the graphs on the CloudWatch dashboard. You can consume the Enhanced Monitoring JSON output from CloudWatch Logs in a monitoring system of your choice. For more information, see Enhanced Monitoring in the Amazon RDS FAQs.

Topics

- Differences between CloudWatch and Enhanced Monitoring metrics
- Retention of Enhanced Monitoring metrics
- Cost of Enhanced Monitoring

Differences between CloudWatch and Enhanced Monitoring metrics

A *hypervisor* creates and runs virtual machines (VMs). Using a hypervisor, an instance can support multiple guest VMs by virtually sharing memory and CPU. CloudWatch gathers metrics about CPU utilization from the hypervisor for a DB instance. In contrast, Enhanced Monitoring gathers its metrics from an agent on the DB instance.

You might find differences between the CloudWatch and Enhanced Monitoring measurements, because the hypervisor layer performs a small amount of work. The differences can be greater if your DB instances use smaller instance classes. In this scenario, more virtual machines (VMs) are probably managed by the hypervisor layer on a single physical instance.

For descriptions of the Enhanced Monitoring metrics, see OS metrics in Enhanced Monitoring. For more information about CloudWatch metrics, see the *Amazon CloudWatch User Guide*.

Retention of Enhanced Monitoring metrics

By default, Enhanced Monitoring metrics are stored for 30 days in the CloudWatch Logs. This retention period is different from typical CloudWatch metrics.

To modify the amount of time the metrics are stored in the CloudWatch Logs, change the retention for the `RDSOSMetrics` log group in the CloudWatch console. For more information, see Change log data retention in CloudWatch logs in the *Amazon CloudWatch Logs User Guide*.

Cost of Enhanced Monitoring

Enhanced Monitoring metrics are stored in the CloudWatch Logs instead of in CloudWatch metrics. The cost of Enhanced Monitoring depends on the following factors:

- You are charged for Enhanced Monitoring only if you exceed the free tier provided by Amazon CloudWatch Logs. Charges are based on CloudWatch Logs data transfer and storage rates.
- The amount of information transferred for an RDS instance is directly proportional to the defined granularity for the Enhanced Monitoring feature. A smaller monitoring interval results in more frequent reporting of OS metrics and increases your monitoring cost. To manage costs, set different granularities for different instances in your accounts.
- Usage costs for Enhanced Monitoring are applied for each DB instance that Enhanced Monitoring is enabled for. Monitoring a large number of DB instances is more expensive than monitoring only a few.
- DB instances that support a more compute-intensive workload have more OS process activity to report and higher costs for Enhanced Monitoring.

For more information about pricing, see Amazon CloudWatch pricing.

Setting up and enabling Enhanced Monitoring

To use Enhanced Monitoring, you must create an IAM role, and then enable Enhanced Monitoring.

Topics

- Creating an IAM role for Enhanced Monitoring
- Turning Enhanced Monitoring on and off
- Protecting against the confused deputy problem

Creating an IAM role for Enhanced Monitoring

Enhanced Monitoring requires permission to act on your behalf to send OS metric information to CloudWatch Logs. You grant Enhanced Monitoring permissions using an AWS Identity and Access Management (IAM) role. You can either create this role when you enable Enhanced Monitoring or create it beforehand.

Topics

- Creating the IAM role when you enable Enhanced Monitoring
- Creating the IAM role before you enable Enhanced Monitoring

Creating the IAM role when you enable Enhanced Monitoring

When you enable Enhanced Monitoring in the RDS console, Amazon RDS can create the required IAM role for you. The role is named `rds-monitoring-role`. RDS uses this role for the specified DB instance, read replica, or Multi-AZ DB cluster.

To create the IAM role when enabling Enhanced Monitoring

1. Follow the steps in Turning Enhanced Monitoring on and off.
2. Set **Monitoring Role** to **Default** in the step where you choose a role.

Creating the IAM role before you enable Enhanced Monitoring

You can create the required role before you enable Enhanced Monitoring. When you enable Enhanced Monitoring, specify your new role's name. You must create this required role if you enable Enhanced Monitoring using the AWS CLI or the RDS API.

The user that enables Enhanced Monitoring must be granted the `PassRole` permission. For more information, see Example 2 in Granting a user permissions to pass a role to an AWS service in the *IAM User Guide*.

To create an IAM role for Amazon RDS enhanced monitoring

1. Open the IAM console at https://console.aws.amazon.com.
2. In the navigation pane, choose **Roles**.
3. Choose **Create role**.
4. Choose the **AWS service** tab, and then choose **RDS** from the list of services.
5. Choose **RDS - Enhanced Monitoring**, and then choose **Next**.
6. Ensure that the **Permissions policies** shows **AmazonRDSEnhancedMonitoringRole**, and then choose **Next**.
7. For **Role name**, enter a name for your role. For example, enter **`emaccess`**.

   The trusted entity for your role is the AWS service **monitoring.rds.amazonaws.com**.
8. Choose **Create role**.

Turning Enhanced Monitoring on and off

You can turn Enhanced Monitoring on and off using the AWS Management Console, AWS CLI, or RDS API. You choose the RDS DB instances on which you want to turn on Enhanced Monitoring. You can set different granularities for metric collection on each DB instance.

Console

You can turn on Enhanced Monitoring when you create a DB cluster or read replica, or when you modify a DB instance. If you modify a DB instance to turn on Enhanced Monitoring, you don't need to reboot your DB instance for the change to take effect.

You can turn on Enhanced Monitoring in the RDS console when you do one of the following actions in the **Databases** page:

- **Create a DB cluster** – Choose **Create database**.
- **Create a read replica** – Choose **Actions**, then **Create read replica**.
- **Modify a DB instance** – Choose **Modify**.

To turn Enhanced Monitoring on or off in the RDS console

1. Scroll to **Additional configuration**.
2. In **Monitoring**, choose **Enable Enhanced Monitoring** for your DB instance or read replica. To turn Enhanced Monitoring off, choose **Disable Enhanced Monitoring**.
3. Set the **Monitoring Role** property to the IAM role that you created to permit Amazon RDS to communicate with Amazon CloudWatch Logs for you, or choose **Default** to have RDS create a role for you named `rds-monitoring-role`.
4. Set the **Granularity** property to the interval, in seconds, between points when metrics are collected for your DB instance or read replica. The **Granularity** property can be set to one of the following values: `1`, `5`, `10`, `15`, `30`, or `60`.

   The fastest that the RDS console refreshes is every 5 seconds. If you set the granularity to 1 second in the RDS console, you still see updated metrics only every 5 seconds. You can retrieve 1-second metric updates by using CloudWatch Logs.

AWS CLI

To turn on Enhanced Monitoring using the AWS CLI, in the following commands, set the `--monitoring-interval` option to a value other than `0` and set the `--monitoring-role-arn` option to the role you created in Creating an IAM role for Enhanced Monitoring.

- create-db-instance
- create-db-instance-read-replica
- modify-db-instance

The `--monitoring-interval` option specifies the interval, in seconds, between points when Enhanced Monitoring metrics are collected. Valid values for the option are `0`, `1`, `5`, `10`, `15`, `30`, and `60`.

To turn off Enhanced Monitoring using the AWS CLI, set the `--monitoring-interval` option to `0` in these commands.

Example

The following example turns on Enhanced Monitoring for a DB instance:

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier mydbinstance \
    --monitoring-interval 30 \
    --monitoring-role-arn arn:aws:iam::123456789012:role/emaccess
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier mydbinstance ^
    --monitoring-interval 30 ^
    --monitoring-role-arn arn:aws:iam::123456789012:role/emaccess
```

Example

The following example turns on Enhanced Monitoring for a Multi-AZ DB cluster:

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --monitoring-interval 30 \
    --monitoring-role-arn arn:aws:iam::123456789012:role/emaccess
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --monitoring-interval 30 ^
    --monitoring-role-arn arn:aws:iam::123456789012:role/emaccess
```

RDS API

To turn on Enhanced Monitoring using the RDS API, set the `MonitoringInterval` parameter to a value other than `0` and set the `MonitoringRoleArn` parameter to the role you created in Creating an IAM role for Enhanced Monitoring. Set these parameters in the following actions:

- CreateDBInstance
- CreateDBInstanceReadReplica
- ModifyDBInstance

The `MonitoringInterval` parameter specifies the interval, in seconds, between points when Enhanced Monitoring metrics are collected. Valid values are `0`, `1`, `5`, `10`, `15`, `30`, and `60`.

To turn off Enhanced Monitoring using the RDS API, set `MonitoringInterval` to `0`.

Protecting against the confused deputy problem

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account. For more information, see The confused deputy problem.

To limit the permissions to the resource that Amazon RDS can give another service, we recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in a trust policy for your Enhanced Monitoring role. If you use both global condition context keys, they must use the same account ID.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. For Amazon RDS, set `aws:SourceArn` to `arn:aws:rds:`*`Region`*`:`*`my-account-id`*`:db:`*`dbname`*.

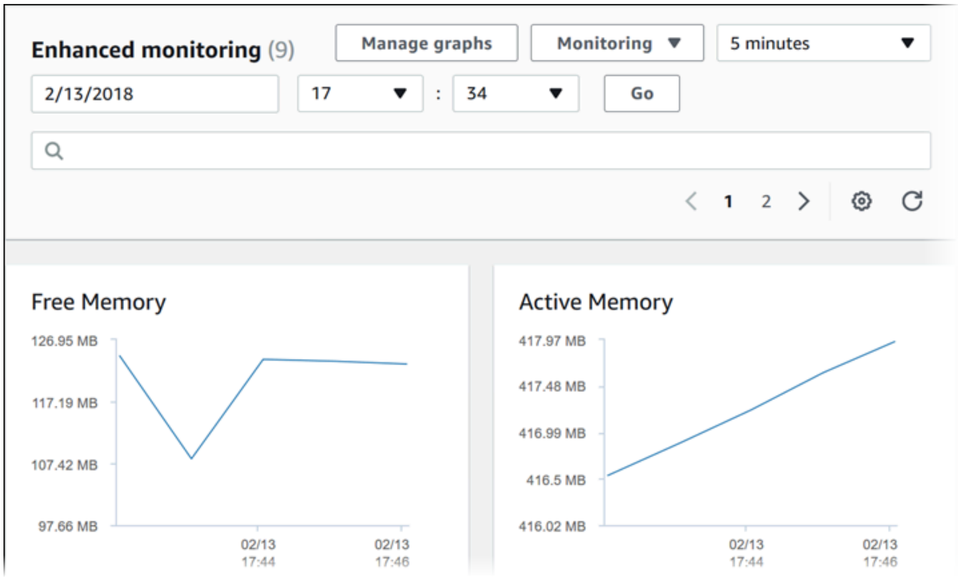
The following example uses the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in a trust policy to prevent the confused deputy problem.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "monitoring.rds.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringLike": {
          "aws:SourceArn": "arn:aws:rds:Region:my-account-id:db:dbname"
        },
        "StringEquals": {
          "aws:SourceAccount": "my-account-id"
        }
      }
    }
  ]
}
```

Viewing OS metrics in the RDS console

You can view OS metrics reported by Enhanced Monitoring in the RDS console by choosing **Enhanced monitoring** for **Monitoring**.

The following example shows the Enhanced Monitoring page. For descriptions of the Enhanced Monitoring metrics, see OS metrics in Enhanced Monitoring.

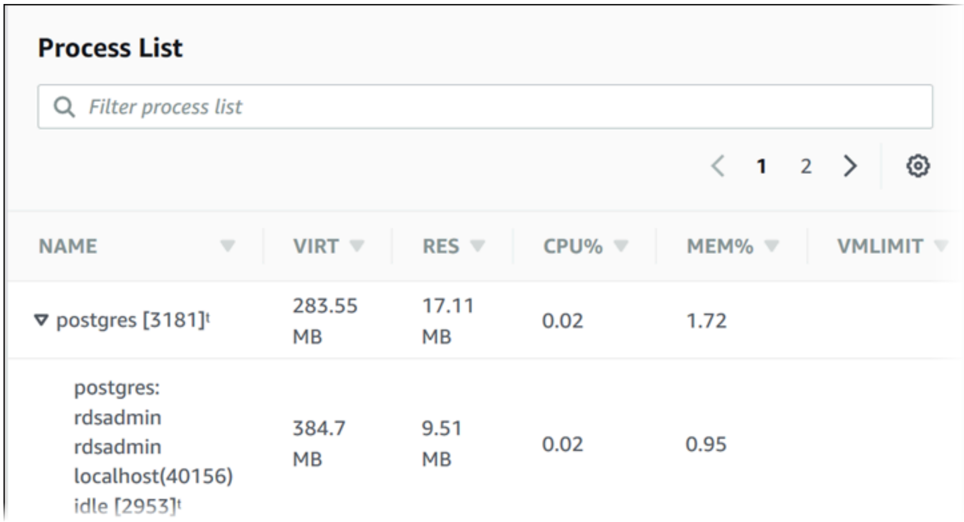

Enhanced monitoring (9)

Monitoring: 5 minutes

2/13/2018 17 : 34 Go

If you want to see details for the processes running on your DB instance, choose **OS process list** for **Monitoring**.

The **Process List** view is shown following.



Process List

NAME	VIRT	RES	CPU%	MEM%	VMLIMIT
▿ postgres [3181]	283.55 MB	17.11 MB	0.02	1.72	
postgres: rdsadmin rdsadmin localhost(40156) idle [2953]	384.7 MB	9.51 MB	0.02	0.95	

The Enhanced Monitoring metrics shown in the **Process list** view are organized as follows:

- **RDS child processes –** Shows a summary of the RDS processes that support the DB instance, for example `aurora` for Amazon Aurora DB clusters. Process threads appear nested beneath the parent process. Process threads show CPU utilization only as other metrics are the same for all threads for the process. The console displays a maximum of 100 processes and threads. The results are a combination of the top CPU consuming and memory consuming processes and threads. If there are more than 50 processes and more than 50 threads, the console displays the top 50 consumers in each category. This display helps you identify which processes are having the greatest impact on performance.
- **RDS processes –** Shows a summary of the resources used by the RDS management agent, diagnostics monitoring processes, and other AWS processes that are required to support RDS DB instances.
- **OS processes –** Shows a summary of the kernel and system processes, which generally have minimal impact on performance.

The items listed for each process are:

- **VIRT –** Displays the virtual size of the process.
- **RES –** Displays the actual physical memory being used by the process.
- **CPU% –** Displays the percentage of the total CPU bandwidth being used by the process.
- **MEM% –** Displays the percentage of the total memory being used by the process.

The monitoring data that is shown in the RDS console is retrieved from Amazon CloudWatch Logs. You can also retrieve the metrics for a DB instance as a log stream from CloudWatch Logs. For more information, see Viewing OS metrics using CloudWatch Logs.

Enhanced Monitoring metrics are not returned during the following:

- A failover of the DB instance.
- Changing the instance class of the DB instance (scale compute).

Enhanced Monitoring metrics are returned during a reboot of a DB instance because only the database engine is rebooted. Metrics for the operating system are still reported.

Viewing OS metrics using CloudWatch Logs

After you have enabled Enhanced Monitoring for your DB cluster, you can view the metrics for it using CloudWatch Logs, with each log stream representing a single DB instance or DB cluster being monitored. The log stream identifier is the resource identifier (`DbiResourceId`) for the DB instance or DB cluster.

To view Enhanced Monitoring log data

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. If necessary, choose the AWS Region that your DB cluster is in. For more information, see Regions and endpoints in the *Amazon Web Services General Reference*.
3. Choose **Logs** in the navigation pane.
4. Choose **RDSOSMetrics** from the list of log groups.
5. Choose the log stream that you want to view from the list of log streams.

Metrics reference for Amazon Aurora

In this reference, you can find descriptions of Amazon Aurora metrics for Amazon CloudWatch, Performance Insights, and Enhanced Monitoring.

Topics

- Amazon CloudWatch metrics for Amazon Aurora
- Amazon CloudWatch dimensions for Aurora
- Availability of Aurora metrics in the Amazon RDS console
- Amazon CloudWatch metrics for Performance Insights
- Performance Insights counter metrics
- SQL statistics for Performance Insights
- OS metrics in Enhanced Monitoring

Amazon CloudWatch metrics for Amazon Aurora

The AWS/RDS namespace includes the following metrics that apply to database entities running on Amazon Aurora. Some metrics apply to either Aurora MySQL, Aurora PostgreSQL, or both. Furthermore, some metrics are specific to a DB cluster, primary DB instance, replica DB instance, or all DB instances.

For Aurora global database metrics, see Amazon CloudWatch metrics for write forwarding in Aurora MySQL and Amazon CloudWatch metrics for write forwarding in Aurora PostgreSQL. For Aurora parallel query metrics, see Monitoring parallel query.

Topics

- Cluster-level metrics for Amazon Aurora
- Instance-level metrics for Amazon Aurora
- Amazon CloudWatch usage metrics for Amazon Aurora

Cluster-level metrics for Amazon Aurora

The following table describes metrics that are specific to Aurora clusters.

Amazon Aurora cluster-level metrics

Metric	Description	Applies to	Units
`AuroraGlobalDBDataTransferBytes`	In an Aurora Global Database, the amount of redo log data transferred from the master AWS Region to a secondary AWS Region. **Note** This metric is available only in secondary AWS Region.	Aurora MySQL and Aurora PostgreSQL	Bytes
`AuroraGlobalDBProgressLag`	In an Aurora Global Database, the measure of how far the secondary cluster is behind the primary cluster for both user transactions and system transactions. **Note** This metric is available only in secondary AWS Region.	Aurora MySQL and Aurora PostgreSQL	Milliseconds
`AuroraGlobalDBReplicatedWriteIO`	In an Aurora Global Database, the number of write I/O operations replicated from the primary	Aurora MySQL and Aurora PostgreSQL	Count

Metric	Description	Applies to	Units
	AWS Region to the cluster volume in a secondary AWS Region. The billing calculati ons for the secondary AWS Regions in a global database use `VolumeWri teIOPs` to account for writes performed within the cluster. The billing calculations for the primary AWS Region in a global database use `VolumeWri teIOPs` to account for the write activity within that cluster, and `AuroraGlo balDBReplicatedWri teIO` to account for cross-Region replication within the global database. **Note** This metric is available only in secondary AWS Region.		

Metric	Description	Applies to	Units
`AuroraGlobalDBReplicationLag`	For an Aurora Global Database, the amount of lag when replicating updates from the primary AWS Region. **Note** This metric is available only in secondary AWS Region.	Aurora MySQL and Aurora PostgreSQL	Milliseconds
`AuroraGlobalDBRPOLag`	In an Aurora Global Database, the recovery point objective (RPO) lag time. This metric measures how far the secondary cluster is behind the primary cluster for user transactions. **Note** This metric is available only in secondary AWS Region.	Aurora MySQL and Aurora PostgreSQL	Milliseconds

Metric	Description	Applies to	Units
`AuroraVolumeBytesLeftTotal`	The remaining available space for the cluster volume. As the cluster volume grows, this value decreases. If it reaches zero, the cluster reports an out-of-space error. If you want to detect whether your Aurora MySQL cluster is approaching the size limit of 128 tebibytes (TiB), this value is simpler and more reliable to monitor than `VolumeBytesUsed` . `AuroraVolumeBytesLeftTotal` takes into account storage used for internal housekeeping and other allocations that don't affect your storage billing.	Aurora MySQL	Bytes
`BacktrackChangeRecordsCreationRate`	The number of backtrack change records created over 5 minutes for your DB cluster.	Aurora MySQL	Count per 5 minutes
`BacktrackChangeRecordsStored`	The number of backtrack change records used by your DB cluster.	Aurora MySQL	Count

Metric	Description	Applies to	Units
`BackupRetentionPeriodStorageUsed`	The total amount of backup storage used to support the point-in-time restore feature within the Aurora DB cluster's backup retention window. This amount is included in the total reported by the `TotalBackupStorageBilled` metric. It is computed separately for each Aurora cluster. For instructions, see [Understanding Amazon Aurora backup storage usage]().	Aurora MySQL and Aurora PostgreSQL	Bytes
`ServerlessDatabaseCapacity`	The current capacity of an Aurora Serverless DB cluster.	Aurora MySQL and Aurora PostgreSQL	Count

Metric	Description	Applies to	Units
`SnapshotStorageUsed`	The total amount of backup storage consumed by all Aurora snapshots for an Aurora DB cluster outside its backup retention window. This amount is included in the total reported by the `TotalBackupStorageBilled` metric. It is computed separately for each Aurora cluster. For instructions, see Understanding Amazon Aurora backup storage usage.	Aurora MySQL and Aurora PostgreSQL	Bytes
`TotalBackupStorageBilled`	The total amount of backup storage in bytes for which you are billed for a given Aurora DB cluster. The metric includes the backup storage measured by the `BackupRetentionPeriodStorageUsed` and `SnapshotStorageUsed` metrics. This metric is computed separately for each Aurora cluster. For instructions, see Understanding Amazon Aurora backup storage usage.	Aurora MySQL and Aurora PostgreSQL	Bytes

Metric	Description	Applies to	Units
`VolumeBytesUsed`	The amount of storage used by your Aurora DB cluster. This value affects the cost of the Aurora DB cluster (for pricing information, see the [Amazon RDS pricing page](https://aws.amazon.com/rds/pricing)). This value doesn't reflect some internal storage allocations that don't affect storage billing. For Aurora MySQL you can anticipate out-of-space issues more accurately by testing whether `AuroraVolumeBytesLeftTotal` is approaching zero instead of comparing `VolumeBytesUsed` against the storage limit of 128 TiB. For clusters that are clones, the value of this metric depends on the amount of data added or changed on the clone. The metric can also increase or decrease when the original cluster is deleted, or as new clones are added or deleted. For details, see [Deleting a source cluster volume](#)	Aurora MySQL and Aurora PostgreSQL	Bytes

Metric	Description	Applies to	Units
`VolumeReadIOPs`	The number of billed read I/O operations from a cluster volume within a 5-minute interval. Billed read operations are calculated at the cluster volume level, aggregated from all instances in the Aurora DB cluster, and then reported at 5-minute intervals. The value is calculated by taking the value of the **Read operations** metric over a 5-minute period. You can determine the amount of billed read operations per second by taking the value of the **Billed read operations** metric and dividing by 300 seconds. For example, if the **Billed read operations** returns 13,686, then the billed read operations per second is 45 (13,686 / 300 = 45.62). You accrue billed read operations for queries that request database pages that aren't in the buffer cache and must be loaded from storage. You might see spikes in billed read	Aurora MySQL and Aurora PostgreSQL	Count per 5 minutes

Metric	Description	Applies to	Units
	operations as query results are read from storage and then loaded into the buffer cache. **Tip** If your Aurora MySQL cluster uses parallel query, you might see an increase in `VolumeReadIOPS` values. Parallel queries don't use the buffer pool. Thus, although the queries are fast, this optimized processing can result in an increase in read operations and associated charges.		
`VolumeWriteIOPs`	The number of write disk I/O operations to the cluster volume, reported at 5-minute intervals. For a detailed description of how billed write operations are calculated, see `VolumeReadIOPs` .	Aurora MySQL and Aurora PostgreSQL	Count per 5 minutes

Instance-level metrics for Amazon Aurora

The following instance-specific CloudWatch metrics apply to all Aurora MySQL and Aurora PostgreSQL instances unless noted otherwise.

Amazon Aurora instance-level metrics

Metric	Description	Applies to	Units
`AbortedClients`	The number of client connections that have not been closed properly.	Aurora MySQL	Count
`ActiveTransactions`	The average number of current transactions executing on an Aurora database instance per second. By default, Aurora doesn't enable this metric. To begin measuring this value, set `innodb_monitor_enable='all'` in the DB parameter group for a specific DB instance.	Aurora MySQL	Count per second
`ACUUtilization`	The value of the `ServerlessDatabaseCapacity` metric divided by the maximum ACU value of the DB cluster. This metric is applicable only for Aurora Serverless v2.	Aurora MySQL and Aurora PostgreSQL	Percentage
`AuroraBinlogReplicaLag`	The amount of time that a binary log replica DB cluster running on Aurora	Primary for Aurora MySQL	Seconds

Metric	Description	Applies to	Units
	MySQL-Compatible Edition lags behind the binary log replication source. A lag means that the source is generating records faster than the replica can apply them. This metric reports different values depending on the engine version: Aurora MySQL version 2 The `Seconds_Behind_Master` field of the MySQL `SHOW SLAVE STATUS` Aurora MySQL version 3 `SHOW REPLICA STATUS` You can use this metric to monitor errors and replica lag in a cluster that acts as a binary log replica. The metric value indicates the following : A high value The replica is lagging the replication source.		

Metric	Description	Applies to	Units

	`0` or a value close to `0`
	The replica process is active and current.
	`-1`
	Aurora can't determine the lag, which can happen during replica setup or when the replica is in an error state.
	Because binary log replication only occurs on the writer instance of the cluster, we recommend using the version of this metric associated with the WRITER role.
	For more information about administering replication, see Replicating Amazon Aurora MySQL DB clusters across AWS Regions. For more information about troubleshooting, see Amazon Aurora MySQL replication issues.		

`AuroraEstimatedSharedMemoryBytes`	The estimated amount of shared buffer or buffer pool memory which was actively used during the last configured polling interval.		Bytes

Metric	Description	Applies to	Units
`AuroraOptimizedReadsCacheHitRatio`	The percentage of requests that are served by the Optimized Reads cache. The value is calculated using the following formula: `orcache_blks_hit/ (orcache_blks_hit + storage_blks_read)` When `AuroraOptimizedReadsCacheHitRatio` is 100%, it means that no pages were read from the Optimized Reads cache and the value will be `0`.	Primary for Aurora PostgreSQL	Percentage
`AuroraReplicaLag`	For an Aurora replica, the amount of lag when replicating updates from the primary instance.	Replica for Aurora MySQL and Aurora PostgreSQL	Milliseconds

Metric	Description	Applies to	Units
`AuroraReplicaLagMaximum`	The maximum amount of lag between the primary instance and any of the Aurora DB instance in the DB cluster. When read replicas are deleted or renamed, there can be a temporary spike in replication lag as the old resource undergoes a recycling process. To obtain an accurate representation of the replication lag during that period, we recommend that you monitor the `AuroraReplicaLag` metric on each read replica instance.	Primary for Aurora MySQL and Aurora PostgreSQL	Milliseconds
`AuroraReplicaLagMinimum`	The minimum amount of lag between the primary instance and any of the Aurora DB instance in the DB cluster.	Primary for Aurora MySQL and Aurora PostgreSQL	Milliseconds
`AuroraSlowConnectionHandleCount`	The number of connections that have waited two seconds or longer to start the handshake. This metric applies only to Aurora MySQL version 3.	Aurora MySQL	Count

Metric	Description	Applies to	Units
`AuroraSlowHandshakeCount`	The number of connections that have taken 50 milliseconds or longer to finish the handshake. This metric applies only to Aurora MySQL version 3.	Aurora MySQL	Count
`BacktrackWindowActual`	The difference between the target backtrack window and the actual backtrack window.	Primary for Aurora MySQL	Minutes
`BacktrackWindowAlert`	The number of times that the actual backtrack window is smaller than the target backtrack window for a given period of time.	Primary for Aurora MySQL	Count
`BlockedTransactions`	The average number of transactions in the database that are blocked per second.	Aurora MySQL	Count per second
`BufferCacheHitRatio`	The percentage of requests that are served by the buffer cache.	Aurora MySQL and Aurora PostgreSQL	Percentage
`CommitLatency`	The average duration taken by the engine and storage to complete the commit operations.	Aurora MySQL and Aurora PostgreSQL	Milliseconds
`CommitThroughput`	The average number of commit operations per second.	Aurora MySQL and Aurora PostgreSQL	Count per second

Metric	Description	Applies to	Units
`ConnectionAttempts`	The number of attempts to connect to an instance, whether successful or not.	Aurora MySQL	Count

Metric	Description	Applies to	Units
`CPUCreditBalance`	The number of CPU credits that an instance has accumulated, reported at 5-minute intervals. You can use this metric to determine how long a DB instance can burst beyond its baseline performance level at a given rate. This metric applies only to these instance classes: • Aurora MySQL: `db.t2.small` , `db.t2.medium` , `db.t3`, and `db.t4g` • Aurora PostgreSQL: `db.t3` and `db.t4g` **Note** We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.	Aurora MySQL and Aurora PostgreSQL	Count

Metric	Description	Applies to	Units
	Launch credits work the same way in Amazon RDS as they do in Amazon EC2. For more information, see Launch credits in the *Amazon Elastic Compute Cloud User Guide for Linux Instances*.		

Metric	Description	Applies to	Units
`CPUCreditUsage`	The number of CPU credits consumed during the specified period, reported at 5-minute intervals. This metric measures the amount of time during which physical CPUs have been used for processing instructi ons by virtual CPUs allocated to the DB instance. This metric applies only to these instance classes: - Aurora MySQL: `db.t2.small` , `db.t2.medium` , `db.t3`, and `db.t4g` - Aurora PostgreSQL: `db.t3` and `db.t4g` **Note** We recommend using the T DB instance classes only for development and test servers, or other non-produ ction servers. For more details on the T instance classes,	Aurora MySQL and Aurora PostgreSQL	Count

Metric	Description	Applies to	Units
	see DB instance class types.		
`CPUSurplusCreditBalance`	The number of surplus credits that have been spent by an unlimited instance when its `CPUCreditBalance` value is zero. The `CPUSurplusCreditBalance` value is paid down by earned CPU credits. If the number of surplus credits exceeds the maximum number of credits that the instance can earn in a 24-hour period, the spent surplus credits above the maximum incur an additional charge. CPU credit metrics are available at a 5-minute frequency only.	Aurora MySQL and Aurora PostgreSQL	Credits (vCPU-minutes)

Metric	Description	Applies to	Units
`CPUSurplusCreditsCharged`	The number of spent surplus credits that are not paid down by earned CPU credits, and which thus incur an additional charge. Spent surplus credits are charged when any of the following occurs: • The spent surplus credits exceed the maximum number of credits that the instance can earn in a 24-hour period. Spent surplus credits above the maximum are charged at the end of the hour. • The instance is stopped or terminated. • The instance is switched from `unlimited` to `standard`. CPU credit metrics are available at a 5-minute frequency only.	Aurora MySQL and Aurora PostgreSQL	Credits (vCPU-minutes)
`CPUUtilization`	The percentage of CPU used by an Aurora DB instance.	Aurora MySQL and Aurora PostgreSQL	Percentage

Metric	Description	Applies to	Units
`DatabaseConnections`	The number of client network connections to the database instance. The number of database sessions can be higher than the metric value because the metric value doesn't include the following: • Sessions that no longer have a network connection but which the database hasn't cleaned up • Sessions created by the database engine for its own purposes • Sessions created by the database engine's parallel execution capabilities • Sessions created by the database engine job scheduler • Amazon Aurora connections	Aurora MySQL and Aurora PostgreSQL	Count
`DDLLatency`	The average duration of requests such as example, create, alter, and drop requests.	Aurora MySQL	Milliseconds
`DDLThroughput`	The average number of DDL requests per second.	Aurora MySQL	Count per second

Metric	Description	Applies to	Units
`Deadlocks`	The average number of deadlocks in the database per second.	Aurora MySQL and Aurora PostgreSQL	Count per second
`DeleteLatency`	The average duration of delete operations.	Aurora MySQL	Milliseconds
`DeleteThroughput`	The average number of delete queries per second.	Aurora MySQL	Count per second
`DiskQueueDepth`	The number of outstanding read/write requests waiting to access the disk.	Aurora MySQL and Aurora PostgreSQL	Count
`DMLLatency`	The average duration of inserts, updates, and deletes.	Aurora MySQL	Milliseconds
`DMLThroughput`	The average number of inserts, updates, and deletes per second.	Aurora MySQL	Count per second
`EngineUptime`	The amount of time that the instance has been running.	Aurora MySQL and Aurora PostgreSQL	Seconds
`FreeableMemory`	The amount of available random access memory.	Aurora MySQL and Aurora PostgreSQL	Bytes
`FreeEphemeralStorage`	The amount of available Ephemeral NVMe storage.	Aurora PostgreSQL	Bytes

Metric	Description	Applies to	Units
`FreeLocalStorage`	The amount of local storage available. Unlike for other DB engines, for Aurora DB instances this metric reports the amount of storage available to each DB instance. This value depends on the DB instance class (for pricing information, see the Amazon RDS pricing page). You can increase the amount of free storage space for an instance by choosing a larger DB instance class for your instance. (This doesn't apply to Aurora Serverless v2.)	Aurora MySQL and Aurora PostgreSQL	Bytes
`InsertLatency`	The average duration of insert operations.	Aurora MySQL	Milliseconds
`InsertThroughput`	The average number of insert operations per second.	Aurora MySQL	Count per second
`LoginFailures`	The average number of failed login attempts per second.	Aurora MySQL	Count per second

Metric	Description	Applies to	Units
`MaximumUsedTransactionIDs`	The age of the oldest unvacuumed transaction ID, in transactions. If this value reaches 2,146,483,648 (2^31 - 1,000,000), the database is forced into read-only mode, to avoid transaction ID wraparound. For more information, see Preventing transaction ID wraparound failures in the PostgreSQL documentation.	Aurora PostgreSQL	Count
`NetworkReceiveThroughput`	The amount of network throughput received from clients by each instance in the Aurora DB cluster. This throughput doesn't include network traffic between instances in the Aurora DB cluster and the cluster volume.	Aurora MySQL and Aurora PostgreSQL	Bytes per second (console shows Megabytes per second)
`NetworkThroughput`	The amount of network throughput both received from and transmitted to clients by each instance in the Aurora DB cluster. This throughput doesn't include network traffic between instances in the Aurora DB cluster and the cluster volume.	Aurora MySQL and Aurora PostgreSQL	Bytes per second

Metric	Description	Applies to	Units
`NetworkTransmitThroughput`	The amount of network throughput sent to clients by each instance in the Aurora DB cluster. This throughput doesn't include network traffic between instances in the DB cluster and the cluster volume.	Aurora MySQL and Aurora PostgreSQL	Bytes per second (console shows Megabytes per second)
`NumBinaryLogFiles`	The number of binlog files generated.	Aurora MySQL	Count
`OldestReplicationSlotLag`	The lagging size of the replica lagging the most in terms of write-ahead log (WAL) data received.	Aurora PostgreSQL	Bytes
`PurgeBoundary`	Transaction number up to which InnoDB purging is allowed. If this metric doesn't advance for extended periods of time, it's a good indication that InnoDB purging is blocked by long-running transactions. To investigate, check the active transactions on your Aurora MySQL DB cluster.	Aurora MySQL version 2, versions 2.11 and higher	Count
`PurgeFinishedPoint`	Transaction number up to which InnoDB purging is performed. This metric can help you examine how fast InnoDB purging is progressing.	Aurora MySQL version 2, versions 2.11 and higher	Count

Metric	Description	Applies to	Units
`Queries`	The average number of queries executed per second.	Aurora MySQL	Count per second
`RDSToAuroraPostgreSQLReplicaLag`	The lag when replicating updates from the primary RDS PostgreSQL instance to other nodes in the cluster.	Replica for Aurora PostgreSQL	Seconds
`ReadIOPS`	The average number of disk I/O operations per second but the reports read and write separately, in 1-minute intervals.	Aurora MySQL and Aurora PostgreSQL	Count per second
`ReadIOPSEphemeralStorage`	The average number of disk read I/O operations to Ephemeral NVMe storage.	Aurora PostgreSQL	Count per second
`ReadLatency`	The average amount of time taken per disk I/O operation.	Aurora MySQL and Aurora PostgreSQL	Seconds
`ReadLatencyEphemeralStorage`	The average amount of time taken per disk read I/O operation for Ephemeral NVMe storage.	Aurora PostgreSQL	Milliseconds
`ReadThroughput`	The average number of bytes read from disk per second.	Aurora MySQL and Aurora PostgreSQL	Bytes per second

Metric	Description	Applies to	Units
`ReadThroughputEphemeralStorage`	The average number of bytes read from disk per second for Ephemeral NVMe storage.	Aurora PostgreSQL	Bytes per second
`ReplicationSlotDiskUsage`	The amount of disk space consumed by replication slot files.	Aurora PostgreSQL	Bytes
`ResultSetCacheHitRatio`	The percentage of requests that are served by the Resultset cache.	Aurora MySQL	Percentage
`RollbackSegmentHistoryListLength`	The undo logs that record committed transactions with delete-marked records. These records are scheduled to be processed by the InnoDB purge operation.	Aurora MySQL	Count
`RowLockTime`	The total time spent acquiring row locks for InnoDB tables.	Aurora MySQL	Milliseconds
`SelectLatency`	The average amount of time for select operations.	Aurora MySQL	Milliseconds
`SelectThroughput`	The average number of select queries per second.	Aurora MySQL	Count per second
`ServerlessDatabaseCapacity`	The current capacity of an Aurora Serverless DB cluster.	Aurora MySQL and Aurora PostgreSQL	Count

Metric	Description	Applies to	Units
`StorageNetworkReceiveThroughput`	The amount of network throughput received from the Aurora storage subsystem by each instance in the DB cluster.	Aurora MySQL and Aurora PostgreSQL	Bytes per second
`StorageNetworkThroughput`	The amount of network throughput received from and sent to the Aurora storage subsystem by each instance in the Aurora DB cluster.	Aurora MySQL and Aurora PostgreSQL	Bytes per second
`StorageNetworkTransmitThroughput`	The amount of network throughput sent to the Aurora storage subsystem by each instance in the Aurora DB cluster.	Aurora MySQL and Aurora PostgreSQL	Bytes per second
`SumBinaryLogSize`	The total size of the binlog files.	Aurora MySQL	Bytes
`SwapUsage`	The amount of swap space used. This metric isn't available for the following DB instance classes: • db.r3.*, db.r4.*, and db.r7g.* (Aurora MySQL) • db.r7g.* (Aurora PostgreSQL)	Aurora MySQL and Aurora PostgreSQL	Bytes

Metric	Description	Applies to	Units
`TempStorageIOPS`	The number of IOPS for both read and writes on local storage attached to the DB instance. This metric represents a count and is measured once per second. This metric is applicable only for Aurora Serverless v2.	Aurora MySQL and Aurora PostgreSQL	Count per second
`TempStorageThroughput`	The amount of data transferred to and from local storage associated with the DB instance. This metric represents bytes and is measured once per second. This metric is applicable only for Aurora Serverless v2.	Aurora MySQL and Aurora PostgreSQL	Bytes per second
`TransactionLogsDiskUsage`	The amount of disk space consumed by transaction logs on the Aurora PostgreSQL DB instance. This metric is generated only when Aurora PostgreSQL is using logical replication or AWS Database Migration Service. By default, Aurora PostgreSQL uses log records, not transaction logs. When transaction logs aren't in use, the value for this metric is `-1`.	Primary for Aurora PostgreSQL	Bytes

Metric	Description	Applies to	Units
`TruncateFinishedPoint`	Transaction identifier up to which undo truncation is performed.	Aurora MySQL version 2, versions 2.11 and higher	Count
`UpdateLatency`	The average amount of time taken for update operations.	Aurora MySQL	Milliseconds
`UpdateThroughput`	The average number of updates per second.	Aurora MySQL	Count per second
`WriteIOPS`	The number of Aurora storage write records generated per second. This is more or less the number of log records generated by the database. These do not correspond to 8K page writes, and do not correspon d to network packets sent.	Aurora MySQL and Aurora PostgreSQL	Count per second
`WriteIOPSEphemeral Storage`	The average number of disk write I/O operations to Ephemeral NVMe storage.	Aurora PostgreSQL	Count per second
`WriteLatency`	The average amount of time taken per disk I/O operation.	Aurora MySQL and Aurora PostgreSQL	Seconds
`WriteLatencyEpheme ralStorage`	The average amount of time taken per disk write I/O operation for Ephemeral NVMe storage.	Aurora PostgreSQL	Milliseconds

Metric	Description	Applies to	Units
`WriteThroughput`	The average number of bytes written to persistent storage every second.	Aurora MySQL and Aurora PostgreSQL	Bytes per second
`WriteThroughputEphemeralStorage`	The average number of bytes written to disk per second for Ephemeral NVMe storage.	Aurora PostgreSQL	Bytes per second

Amazon CloudWatch usage metrics for Amazon Aurora

The `AWS/Usage` namespace in Amazon CloudWatch includes account-level usage metrics for your Amazon RDS service quotas. CloudWatch collects usage metrics automatically for all AWS Regions.

For more information, see CloudWatch usage metrics in the *Amazon CloudWatch User Guide*. For more information about quotas, see Quotas and constraints for Amazon Aurora and Requesting a quota increase in the *Service Quotas User Guide*.

Metric	Description	Units*
`DBClusterParameterGroups`	The number of DB cluster parameter groups in your AWS account. The count excludes default parameter groups.	Count
`DBClusters`	The number of Amazon Aurora DB clusters in your AWS account.	Count
`DBInstances`	The number of DB instances in your AWS account.	Count
`DBParameterGroups`	The number of DB parameter groups in your AWS account. The count excludes the default DB parameter groups.	Count
`DBSubnetGroups`	The number of DB subnet groups in your AWS account. The count excludes the default subnet group.	Count

Metric	Description	Units*
`ManualClusterSnapshots`	The number of manually created DB cluster snapshots in your AWS account. The count excludes invalid snapshots.	Count
`OptionGroups`	The number of option groups in your AWS account. The count excludes the default option groups.	Count
`ReservedDBInstances`	The number of reserved DB instances in your AWS account. The count excludes retired or declined instances.	Count

> **Note**
>
> Amazon RDS doesn't publish units for usage metrics to CloudWatch. The units only appear in the documentation.

Amazon CloudWatch dimensions for Aurora

You can filter Aurora metrics data by using any dimension in the following table.

Dimension	Filters the requested data for . . .
`DBInstanceIdentifier`	A specific DB instance.
`DBClusterIdentifier`	A specific Aurora DB cluster.
`DBClusterIdentifier, Role`	A specific Aurora DB cluster, aggregating the metric by instance role (WRITER/READER). For example, you can aggregate metrics for all READER instances that belong to a cluster.
`DbClusterIdentifier, EngineName`	A specific Aurora DB cluster and engine name combination. For example, you can view the `VolumeReadIOPs` metric for cluster `ams1` and engine `aurora`.
`DatabaseClass`	All instances in a database class. For example, you can aggregate metrics for all instances that belong to the database class `db.r5.large` .

Dimension	Filters the requested data for . . .
`EngineName`	The identified engine name only. For example, you can aggregate metrics for all instances that have the engine name `aurora-postgresql` .
`SourceRegion`	The specified Region only. For example, you can aggregate metrics for all DB instances in the `us-east-1` Region.

Availability of Aurora metrics in the Amazon RDS console

Not all metrics provided by Amazon Aurora are available in the Amazon RDS console. You can view these metrics using tools such as the AWS CLI and CloudWatch API. Also, some metrics in the Amazon RDS console are either shown only for specific instance classes, or with different names and units of measurement.

Topics

- Aurora metrics available in the Last Hour view
- Aurora metrics available in specific cases
- Aurora metrics that aren't available in the console

Aurora metrics available in the Last Hour view

You can view a subset of categorized Aurora metrics in the default Last Hour view in the Amazon RDS console. The following table lists the categories and associated metrics displayed in the Amazon RDS console for an Aurora instance.

Category	Metrics

SQL	`ActiveTransactions`
	`BlockedTransactions`
	`BufferCacheHitRatio`
	`CommitLatency`

Category	Metrics

	`CommitThroughput`
	`DatabaseConnections`
	`DDLLatency`
	`DDLThroughput`
	`Deadlocks`
	`DMLLatency`
	`DMLThroughput`
	`LoginFailures`
	`ResultSetCacheHitRatio`
	`SelectLatency`
	`SelectThroughput`

System	`AuroraReplicaLag`
	`AuroraReplicaLagMaximum`
	`AuroraReplicaLagMinimum`
	`CPUCreditBalance`
	`CPUCreditUsage`
	`CPUUtilization`
	`FreeableMemory`
	`FreeLocalStorage` (This doesn't apply to Aurora Serverless v2.)
	`NetworkReceiveThroughput`

Category	Metrics

Deployment	`AuroraReplicaLag`
	`BufferCacheHitRatio`
	`ResultSetCacheHitRatio`
	`SelectThroughput`

Aurora metrics available in specific cases

In addition, some Aurora metrics are either shown only for specific instance classes, or only for DB instances, or with different names and different units of measurement:

- The `CPUCreditBalance` and `CPUCreditUsage` metrics are displayed only for Aurora MySQL `db.t2` instance classes and for Aurora PostgreSQL `db.t3` instance classes.
- The following metrics that are displayed with different names, as listed:

Metric	Display name
`AuroraReplicaLagMaximum`	Replica lag maximum
`AuroraReplicaLagMinimum`	Replica lag minimum
`DDLThroughput`	DDL
`NetworkReceiveThroughput`	Network throughput
`VolumeBytesUsed`	[Billed] Volume bytes used
`VolumeReadIOPs`	[Billed] Volume read IOPS
`VolumeWriteIOPs`	[Billed] Volume write IOPS

- The following metrics apply to an entire Aurora DB cluster, but are displayed only when viewing DB instances for an Aurora DB cluster in the Amazon RDS console:
  - `VolumeBytesUsed`
  - `VolumeReadIOPs`

  - `VolumeWriteIOPs`
- The following metrics are displayed in megabytes, instead of bytes, in the Amazon RDS console:
  - `FreeableMemory`
  - `FreeLocalStorage`
  - `NetworkReceiveThroughput`
  - `NetworkTransmitThroughput`
- The following metrics apply to an Aurora PostgreSQL DB cluster with Aurora Optimized Reads:
  - `AuroraOptimizedReadsCacheHitRatio`
  - `FreeEphemeralStorage`
  - `ReadIOPSEphemeralStorage`
  - `ReadLatencyEphemeralStorage`
  - `ReadThroughputEphemeralStorage`
  - `WriteIOPSEphemeralStorage`
  - `WriteLatencyEphemeralStorage`
  - `WriteThroughputEphemeralStorage`

Aurora metrics that aren't available in the console

The following Aurora metrics aren't available in the Amazon RDS console:

- `AuroraBinlogReplicaLag`
- `DeleteLatency`
- `DeleteThroughput`
- `EngineUptime`
- `InsertLatency`
- `InsertThroughput`
- `NetworkThroughput`
- `Queries`
- `UpdateLatency`
- `UpdateThroughput`

Amazon CloudWatch metrics for Performance Insights

Performance Insights automatically publishes some metrics to Amazon CloudWatch. The same data can be queried from Performance Insights, but having the metrics in CloudWatch makes it easy to add CloudWatch alarms. It also makes it easy to add the metrics to existing CloudWatch Dashboards.

Metric	Description
DBLoad	The number of active sessions for the DB engine. Typically, you want the data for the average number of active sessions. In Performance Insights, this data is queried as `db.load.avg` .
DBLoadCPU	The number of active sessions where the wait event type is CPU. In Performance Insights, this data is queried as `db.load.avg` , filtered by the wait event type CPU.
DBLoadNonCPU	The number of active sessions where the wait event type is not CPU.

Note

These metrics are published to CloudWatch only if there is load on the DB instance.

You can examine these metrics using the CloudWatch console, the AWS CLI, or the CloudWatch API. You can also examine other Performance Insights counter metrics using a special metric math function. For more information, see Querying other Performance Insights counter metrics in CloudWatch.

For example, you can get the statistics for the `DBLoad` metric by running the get-metric-statistics command.

```
aws cloudwatch get-metric-statistics \
    --region us-west-2 \
    --namespace AWS/RDS \
    --metric-name DBLoad  \
    --period 60 \
    --statistics Average \
    --start-time 1532035185 \
    --end-time 1532036185 \
    --dimensions Name=DBInstanceIdentifier,Value=db-loadtest-0
```

This example generates output similar to the following.

```
{
  "Datapoints": [
  {
  "Timestamp": "2021-07-19T21:30:00Z",
  "Unit": "None",
  "Average": 2.1
  },
  {
  "Timestamp": "2021-07-19T21:34:00Z",
  "Unit": "None",
  "Average": 1.7
  },
  {
  "Timestamp": "2021-07-19T21:35:00Z",
  "Unit": "None",
  "Average": 2.8
  },
  {
  "Timestamp": "2021-07-19T21:31:00Z",
  "Unit": "None",
  "Average": 1.5
  },
  {
  "Timestamp": "2021-07-19T21:32:00Z",
  "Unit": "None",
  "Average": 1.8
  },
  {
  "Timestamp": "2021-07-19T21:29:00Z",
  "Unit": "None",
  "Average": 3.0
  },
{
"Timestamp": "2021-07-19T21:33:00Z",
"Unit": "None",
"Average": 2.4
}
],
"Label": "DBLoad"
}
```

For more information about CloudWatch, see What is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

Querying other Performance Insights counter metrics in CloudWatch

You can query, alarm, and graphs on RDS Performance Insights metrics from CloudWatch. You can access information about your DB cluster by using the `DB_PERF_INSIGHTS` metric math function for CloudWatch. This function allows you to use the Performance Insights metrics that are not directly reported to CloudWatch to create a new time series.

You can use the new Metric Math function by clicking on the **Add Math** drop-down menu in the **Select metric** screen in the CloudWatch console. You can use it to create alarms and graphs on Performance Insights metrics or on combinations of CloudWatch and Performance Insights metrics, including high-resolution alarms for sub-minute metrics. You can also use the function programmatically by including the Metric Math expression in a `get-metric-data` request. For more information, see Metric math syntax and functions and Create an alarm on Performance Insights counter metrics from an AWS database.

Performance Insights counter metrics

Counter metrics are operating system and database performance metrics in the Performance Insights dashboard. To help identify and analyze performance problems, you can correlate counter metrics with DB load. You can add a statistic function to the metric to get the metric values. For example, the supported functions for `os.memory.active` metric are `.avg`, `.min`, `.max`, `.sum`, and `.sample_count`.

The counter metrics are collected one time each minute. The OS metrics collection depends on whether Enhanced Monitoring is turned on or off. If Enhanced Monitoring is turned off, the OS metrics are collected one time each minute. If Enhanced Monitoring is turned on, the OS metrics are collected for the selected time period. For more information about turning Enhanced Monitoring on or off, see Turning Enhanced Monitoring on and off.

Topics

- Performance Insights operating system counters
- Performance Insights counters for Aurora MySQL
- Performance Insights counters for Aurora PostgreSQL

Performance Insights operating system counters

The following operating system counters, which are prefixed with `os`, are available with Performance Insights for Aurora PostgreSQL and Aurora MySQL.

You can use `ListAvailableResourceMetrics` API for the list of available counter metrics for your DB instance. For more information, see ListAvailableResourceMetrics in the Amazon RDS Performance Insights API Reference guide.

Counter	Type	Metric	Description
Active	Memory	os.memory.active	The amount of assigned memory, in kilobytes.
Buffers	Memory	os.memory.buffers	The amount of memory used for buffering I/O requests prior to writing to the storage device, in kilobytes.
Cached	Memory	os.memory.cached	The amount of memory used for caching file system–based I/O, in kilobytes.
DB Cache	Memory	os.memory.db.cache	The amount of memory used for

Counter	Type	Metric	Description
			page cache by database process including tmpfs (shmem), in bytes.
DB Resident Set Size	Memory	os.memory.db.residentSetSize	The amount of memory used for anonymous and swap cache by database process not including tmpfs (shmem), in bytes.
DB Swap	Memory	os.memory.db.swap	The amount of memory used for swap by database process, in bytes.
Dirty	Memory	os.memory.dirty	The amount of memory pages in RAM that have been modified but not written to their related data block in storage, in kilobytes.
Free	Memory	os.memory.free	The amount of unassigned memory, in kilobytes.
Huge Pages Free	Memory	os.memory.hugePagesFree	The number of free huge pages. Huge pages are a feature of the Linux kernel.

Counter	Type	Metric	Description
Huge Pages Rsvd	Memory	os.memory.hugePagesRsvd	The number of committed huge pages.
Huge Pages Size	Memory	os.memory.hugePagesSize	The size for each huge pages unit, in kilobytes.
Huge Pages Surp	Memory	os.memory.hugePagesSurp	The number of available surplus huge pages over the total.
Huge Pages Total	Memory	os.memory.hugePagesTotal	The total number of huge pages.
Inactive	Memory	os.memory.inactive	The amount of least-frequently used memory pages, in kilobytes.
Mapped	Memory	os.memory.mapped	The total amount of file-system contents that is memory mapped inside a process address space, in kilobytes.s
Out of Memory Kill Count	Memory	os.memory.outOfMemoryKillCount	The number of OOM kills that happened over the last collection interval.

Counter	Type	Metric	Description
Page Tables	Memory	os.memory.pageTables	The amount of memory used by page tables, in kilobytes.
Slab	Memory	os.memory.slab	The amount of reusable kernel data structures, in kilobytes.
Total	Memory	os.memory.total	The total amount of memory, in kilobytes.
Writeback	Memory	os.memory.writeback	The amount of dirty pages in RAM that are still being written to the backing storage, in kilobytes.
Guest	Cpu Utilization	os.cpuUtilization.guest	The percentage of CPU in use by guest programs.
Idle	Cpu Utilization	os.cpuUtilization.idle	The percentage of CPU that is idle.
Irq	Cpu Utilization	os.cpuUtilization.irq	The percentage of CPU in use by software interrupts.
Nice	Cpu Utilization	os.cpuUtilization.nice	The percentage of CPU in use by programs running at lowest priority.

Counter	Type	Metric	Description
Steal	Cpu Utilization	os.cpuUtilization.steal	The percentage of CPU in use by other virtual machines.
System	Cpu Utilization	os.cpuUtilization.system	The percentage of CPU in use by the kernel.
Total	Cpu Utilization	os.cpuUtilization.total	The total percentage of the CPU in use. This value includes the nice value.
User	Cpu Utilization	os.cpuUtilization.user	The percentage of CPU in use by user programs.
Wait	Cpu Utilization	os.cpuUtilization.wait	The percentage of CPU unused while waiting for I/O access.
Aurora Storage Aurora Storage Bytes Rx	Disk IO	os.diskIO.auroraStorage.auroraStorageBytesRx	The number of bytes received for aurora storage per second.
Aurora Storage Aurora Storage Bytes Tx	Disk IO	os.diskIO.auroraStorage.auroraStorageBytesTx	The number of bytes uploaded for aurora storage per second.
Aurora Storage Disk Queue Depth	Disk IO	os.diskIO.auroraStorage.diskQueueDepth	The length of aurora storage disk queue.
Aurora Storage Read IOs PS	Disk IO	os.diskIO.auroraStorage.readIOsPS	The number of read operations per second.

Counter	Type	Metric	Description
Aurora Storage Read Latency	Disk IO	os.diskIO.auroraStorage.readLatency	The average latency of a read I/O request to Aurora storage, in milliseconds.
Aurora Storage Read Throughtput	Disk IO	os.diskIO.auroraStorage.readThroughput	The amount of network throughput used by requests to the DB cluster, in bytes per second.
Aurora Storage Write IOs PS	Disk IO	os.diskIO.auroraStorage.writeIOsPS	The number of write operations per second.
Aurora Storage Write Latency	Disk IO	os.diskIO.auroraStorage.writeLatency	The average latency of a write I/O request to Aurora storage, in milliseconds.
Aurora Storage Write Throughput	Disk IO	os.diskIO.auroraStorage.writeThroughput	The amount of network throughput used by responses from the DB cluster, in bytes per second.
Rdstemp Avg Queue Len	Disk IO	os.diskIO.rdstemp.avgQueueLen	The number of requests waiting in the I/O device's queue.
Rdstemp Avg Req Sz	Disk IO	os.diskIO.rdstemp.avgReqSz	The number of requests waiting in the I/O device's queue.

Counter	Type	Metric	Description
Rdstemp Await	Disk IO	os.diskIO.rdstemp.await	The number of milliseconds required to respond to requests, including queue time and service time.
Rdstemp Read IOs PS	Disk IO	os.diskIO.rdstemp.readIOsPS	The number of read operations per second.
Rdstemp Read KB	Disk IO	os.diskIO.rdstemp.readKb	The total number of kilobytes read.
Rdstemp Read KB PS	Disk IO	os.diskIO.rdstemp.readKbPS	The number of kilobytes read per second.
Rdstemp Rrqm PS	Disk IO	os.diskIO.rdstemp.rrqmPS	The number of merged read requests queued per second.
Rdstemp TPS	Disk IO	os.diskIO.rdstemp.tps	The number of I/O transactions per second.
Rdstemp Util	Disk IO	os.diskIO.rdstemp.util	The percentage of CPU time during which requests were issued.
Rdstemp Write IOs PS	Disk IO	os.diskIO.rdstemp.writeIOsPS	The number of write operations per second.
Rdstemp Write KB	Disk IO	os.diskIO.rdstemp.writeKb	The total number of kilobytes written.

Counter	Type	Metric	Description
Rdstemp Write KB PS	Disk IO	os.diskIO.rdstemp.writeKbPS	The number of kilobytes written per second.
Rdstemp Wrqm PS	Disk IO	os.diskIO.rdstemp.wrqmPS	The number of merged write requests queued per second.
Blocked	Tasks	os.tasks.blocked	The number of tasks that are blocked.
Running	Tasks	os.tasks.running	The number of tasks that are running.
Sleeping	Tasks	os.tasks.sleeping	The number of tasks that are sleeping.
Stopped	Tasks	os.tasks.stopped	The number of tasks that are stopped.
Total	Tasks	os.tasks.total	The total number of tasks.
Zombie	Tasks	os.tasks.zombie	The number of child tasks that are inactive with an active parent task.
One	Load Average Minute	os.loadAverageMinute.one	The number of processes requesting CPU time over the last minute.

Counter	Type	Metric	Description
Fifteen	Load Average Minute	os.loadAverageMinute.fifteen	The number of processes requesting CPU time over the last 15 minutes.
Five	Load Average Minute	os.loadAverageMinute.five	The number of processes requesting CPU time over the last 5 minutes.
Cached	Swap	os.swap.cached	The amount of swap memory, in kilobytes, used as cache memory.
Free	Swap	os.swap.free	The amount of swap memory free, in kilobytes.
In	Swap	os.swap.in	The amount of memory, in kilobytes, swapped in from disk.
Out	Swap	os.swap.out	The amount of memory, in kilobytes, swapped out to disk.
Total	Swap	os.swap.total	The total amount of swap memory available in kilobytes.
Max Files	File Sys	os.fileSys.maxFiles	The maximum number of files that can be created for the file system.

Counter	Type	Metric	Description
Used Files	File Sys	os.fileSys.usedFiles	The number of files in the file system.
Used File Percent	File Sys	os.fileSys.usedFilePercent	The percentage of available files in use.
Used Percent	File Sys	os.fileSys.usedPercent	The percentage of the file-system disk space in use.
Used	File Sys	os.fileSys.used	The amount of disk space used by files in the file system, in kilobytes.
Total	File Sys	os.fileSys.total	The total number of disk space available for the file system, in kilobytes.
Rx	Network	os.network.rx	The number of bytes received per second.
Tx	Network	os.network.tx	The number of bytes uploaded per second.
Acu Utilization	General	os.general.acuUtilization	The percentage of current capacity out of the maximum configured capacity.
Max Configured Acu	General	os.general.maxConfiguredAcu	The maximum capacity configured by the user, in ACUs.

Counter	Type	Metric	Description
Min Configured Acu	General	os.general.minConfiguredAcu	The minimum capacity configured by the user, in ACUs.
Num VCPUs	General	os.general.numVCPUs	The number of virtual CPUs for the DB instance.
Serverless Database Capacity	General	os.general.serverlessDatabaseCapacity	The current capacity of the instance, in ACUs.

Performance Insights counters for Aurora MySQL

The following database counters are available with Performance Insights for Aurora MySQL.

Topics

- Native counters for Aurora MySQL
- Non-native counters for Aurora MySQL

Native counters for Aurora MySQL

Native metrics are defined by the database engine and not by Amazon Aurora. You can find definitions for these native metrics in Server status variables in the MySQL documentation.

Counter	Type	Unit	Metric
Com_analyze	SQL	Queries per second	db.SQL.Com_analyze
Com_optimize	SQL	Queries per second	db.SQL.Com_optimize

Counter	Type	Unit	Metric
Com_select	SQL	Queries per second	db.SQL.Com_select
Innodb_rows_deleted	SQL	Rows per second	db.SQL.Innodb_rows_deleted
Innodb_rows_inserted	SQL	Rows per second	db.SQL.Innodb_rows_inserted
Innodb_rows_read	SQL	Rows per second	db.SQL.Innodb_rows_read
Innodb_rows_updated	SQL	Rows per second	db.SQL.Innodb_rows_updated
Queries	SQL	Queries per second	db.SQL.Queries
Questions	SQL	Queries per second	db.SQL.Questions
Select_full_join	SQL	Queries per second	db.SQL.Select_full_join
Select_full_range_join	SQL	Queries per second	db.SQL.Select_full_range_join
Select_range	SQL	Queries per second	db.SQL.Select_range

Counter	Type	Unit	Metric
Select_range_check	SQL	Queries per second	db.SQL.Select_range_check
Select_scan	SQL	Queries per second	db.SQL.Select_scan
Slow_queries	SQL	Queries per second	db.SQL.Slow_queries
Sort_merge_passes	SQL	Queries per second	db.SQL.Sort_merge_passes
Sort_range	SQL	Queries per second	db.SQL.Sort_range
Sort_rows	SQL	Queries per second	db.SQL.Sort_rows
Sort_scan	SQL	Queries per second	db.SQL.Sort_scan
Total_query_time	SQL	Milliseco nds	db.SQL.Total_query_time
Table_locks_immediate	Locks	Requests per second	db.Locks.Table_locks_immediate

Counter	Type	Unit	Metric
Table_locks_waited	Locks	Requests per second	db.Locks.Table_locks_waited
Innodb_row_lock_time	Locks	Milliseconds (average)	db.Locks.Innodb_row_lock_time
Aborted_clients	Users	Connections	db.Users.Aborted_clients
Aborted_connects	Users	Connections	db.Users.Aborted_connects
Connections	Users	Connections	db.Users.Connections
External_threads_connected	Users	Connections	db.Users.External_threads_connected
max_connections	Users	Connections	db.User.max_connections
Threads_connected	Users	Connections	db.Users.Threads_connected
Threads_created	Users	Connections	db.Users.Threads_created
Threads_running	Users	Connections	db.Users.Threads_running
Created_tmp_disk_tables	Temp	Tables per second	db.Temp.Created_tmp_disk_tables
Created_tmp_tables	Temp	Tables per second	db.Temp.Created_tmp_tables

Counter	Type	Unit	Metric
Innodb_buffer_pool_pages_data	Cache	Pages	db.Cache.Innodb_buffer_pool_pages_data
Innodb_buffer_pool_pages_total	Cache	Pages	db.Cache.Innodb_buffer_pool_pages_total
Innodb_buffer_pool_read_requests	Cache	Pages per second	db.Cache.Innodb_buffer_pool_read_requests
Innodb_buffer_pool_reads	Cache	Pages per second	db.Cache.Innodb_buffer_pool_reads
Opened_tables	Cache	Tables	db.Cache.Opened_tables
Opened_table_definitions	Cache	Tables	db.Cache.Opened_table_definitions
Qcache_hits	Cache	Queries	db.Cache.Qcache_hits

Non-native counters for Aurora MySQL

Non-native counter metrics are counters defined by Amazon RDS. A non-native metric can be a metric that you get with a specific query. A non-native metric also can be a derived metric, where two or more native counters are used in calculations for ratios, hit rates, or latencies.

Counter	Type	Metric	Description	Definition
innodb_buffer_pool_hits	Cache	db.Cache.innoDB_buffer_pool_hits	The number of reads that InnoDB could satisfy from the buffer pool.	`innodb_buffer_pool_read_requests - innodb_buffer_pool_reads`
innodb_buffer_pool_hit_rate	Cache	db.Cache.innoDB_buffer_pool_hit_rate	The percentage of reads that InnoDB could satisfy from the buffer pool.	`100 * innodb_buffer_pool_read_requests /`

Counter	Type	Metric	Description	Definition
				`(innodb_buffer_pool_read_requests + innodb_buffer_pool_reads)`
innodb_buffer_pool_usage	Cache	db.Cache.innoDB_buffer_pool_usage	The percentage of the InnoDB buffer pool that contains data (pages). **Note** When using compressed tables, this value can vary. For more information, see the information about `Innodb_buffer_pool_pages_data` and `Innodb_buffer_pool_pages_total` in [Server status sariables]() in the MySQL documentation.	`Innodb_buffer_pool_pages_data / Innodb_buffer_pool_pages_total * 100.0`

Counter	Type	Metric	Description	Definition
query_cache_hit_rate	Cache	db.Cache.query_cache_hit_rate	The hit ratio for the MySQL result set cache (query cache).	`Qcache_hits / (QCache_hits + Com_select) * 100`
innodb_rows_changed	SQL	db.SQL.innodb_rows_changed	The total InnoDB row operations.	`db.SQL.Innodb_rows_inserted + db.SQL.Innodb_rows_deleted + db.SQL.Innodb_rows_updated`
active_transactions	Transactions	db.Transactions.active_transactions	The total active transactions.	`SELECT COUNT(1) AS active_transactions FROM  INFORMATION_SCHEMA.INNODB_TRX`

Counter	Type	Metric	Description	Definition
trx_rseg_history_len	Transactions	db.Transactions.trx_rseg_history_len	A list of the undo log pages for committed transactions that is maintained by the InnoDB transaction system to implement multi-version concurrency control. For more information about undo log records details, see https://dev.mysql.com/doc/refman/8.0/en/innodb-multi-versioning.html in the MySQL documentation.	`SELECT COUNT AS trx_rseg_history_len FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME='trx_rseg_history_len'`
innodb_deadlocks	Locks	db.Locks.innodb_deadlocks	The total number of deadlocks.	`SELECT COUNT AS innodb_deadlocks FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME='lock_deadlocks'`

Counter	Type	Metric	Description	Definition
innodb_lock_timeouts	Locks	db.Locks.innodb_lock_timeouts	The total number of deadlocks that timed out.	`SELECT COUNT AS innodb_lock_timeouts FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME='lock_timeouts'`
innodb_row_lock_waits	Locks	db.Locks.innodb_row_lock_waits	The total number of row locks that resulted in a wait.	`SELECT COUNT AS innodb_row_lock_waits FROM INFORMATION_SCHEMA.INNODB_METRICS WHERE NAME='lock_row_lock_waits'`

Performance Insights counters for Aurora PostgreSQL

The following database counters are available with Performance Insights for Aurora PostgreSQL.

Topics

- Native Counters for Aurora PostgreSQL
- Non-native counters for Aurora PostgreSQL

Native Counters for Aurora PostgreSQL

Native metrics are defined by the database engine and not by Amazon Aurora. You can find definitions for these native metrics in Viewing Statistics in the PostgreSQL documentation.

Counter	Type	Unit	Metric
tup_deleted	SQL	Tuples per second	db.SQL.tup_deleted
tup_fetched	SQL	Tuples per second	db.SQL.tup_fetched
tup_inserted	SQL	Tuples per second	db.SQL.tup_inserted
tup_returned	SQL	Tuples per second	db.SQL.tup_returned
tup_updated	SQL	Tuples per second	db.SQL.tup_updated
blks_hit	Cache	Blocks per second	db.Cache.blks_hit
buffers_alloc	Cache	Blocks per second	db.Cache.buffers_alloc
buffers_checkpoint	Checkpoint	Blocks per second	db.Checkpoint.buffers_checkpoint
checkpoints_req	Checkpoint	Checkpoints per minute	db.Checkpoint.checkpoints_req
checkpoint_sync_time	Checkpoint	Milliseconds per checkpoint	db.Checkpoint.checkpoint_sync_time
checkpoints_timed	Checkpoint	Checkpoints per minute	db.Checkpoint.checkpoints_timed
checkpoint_write_time	Checkpoint	Milliseconds per checkpoint	db.Checkpoint.checkpoint_write_time
maxwritten_clean	Checkpoint	Bgwriter clean stops per minute	db.Checkpoint.maxwritten_clean
deadlocks	Concurrency	Deadlocks per minute	db.Concurrency.deadlocks
blk_read_time	I/O	Milliseconds	db.IO.blk_read_time

Counter	Type	Unit	Metric
blks_read	I/O	Blocks per second	db.IO.blks_read
buffers_backend	I/O	Blocks per second	db.IO.buffers_backend
buffers_backend_fsync	I/O	Blocks per second	db.IO.buffers_backend_fsync
buffers_clean	I/O	Blocks per second	db.IO.buffers_clean
temp_bytes	Temp	Bytes per second	db.Temp.temp_bytes
temp_files	Temp	Files per minute	db.Temp.temp_files
xact_commit	Transactions	Commits per second	db.Transactions.xact_commit
xact_rollback	Transactions	Rollbacks per second	db.Transactions.xact_rollback
numbackends	User	Connections	db.User.numbackends
archived_count	WAL	Files per minute	db.WAL.archived_count

Non-native counters for Aurora PostgreSQL

Non-native counter metrics are counters defined by Amazon Aurora. A non-native metric can be a metric that you get with a specific query. A non-native metric also can be a derived metric, where two or more native counters are used in calculations for ratios, hit rates, or latencies.

Counter	Type	Metric	Description	Definition
checkpoint_sync_latency	Checkpoint	db.Checkpoint.checkpoint_sync_latency	The total amount of time that has been spent in the portion of checkpoin	`checkpoint_sync_time / (checkpoints_timed + checkpoints_req)`

Counter	Type	Metric	Description	Definition
			t processing where files are synchronized to disk.	
checkpoint_write_latency	Checkpoint	db.Checkpoint.checkpoint_write_latency	The total amount of time that has been spent in the portion of checkpoint processing where files are written to disk.	`checkpoint_write_time / (checkpoints_timed + checkpoints_req)`
local_blks_read	I/O	db.IO.local_blks_read	Total number of local blocks read.	–
local_blk_read_time	I/O	db.IO.local_blk_read_time	If `track_io_timing` is enabled, it tracks the total time spent reading local data file blocks, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.	–
orcache_blks_hit	I/O	db.IO.orcache_blks_hit	Total number of shared blocks hits from optimized reads cache.	–
orcache_blk_read_time	I/O	db.IO.orcache_blk_read_time	If `track_io_timing` is enabled, it tracks the total time spent reading data file blocks from Optimized Reads cache, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.	–

Counter	Type	Metric	Description	Definition
read_latency	I/O	db.IO.read_latency	The time spent reading data file blocks by backends in this instance.	`blk_read_time / blks_read`
storage_blks_read	I/O	db.IO.storage_blks_read	Total number of shared blocks read from aurora storage.	–
storage_blk_read_time	I/O	db.IO.storage_blk_read_time	If `track_io_timing` is enabled, it tracks the total time spent reading data file blocks from Aurora storage, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.	–
idle_in_transaction_aborted_count	State	db.state.idle_in_transaction_aborted_count	The number of sessions in the `idle in transaction (aborted)` state.	–
idle_in_transaction_count	State	db.state.idle_in_transaction_count	The number of sessions in the `idle in transaction` state.	–
idle_in_transaction_max_time	State	db.state.idle_in_transaction_max_time	The duration of the longest running transaction in the `idle in transaction` state, in seconds.	–

Counter	Type	Metric	Description	Definition
logical_reads	SQL	db.SQL.logical_reads	The total number of blocks hit and read.	`blks_hit + blks_read`
queries_started	SQL	db.SQL.queries	The number of queries started.	–
queries_finished	SQL	db.SQL.queries	The number of queries finished.	–
total_query_time	SQL	db.SQL.total_query_time	The total time spent executing statements, in milliseconds.	–
active_transactions	Transactions	db.Transactions.active_transactions	The number of active transactions.	–
blocked_transactions	Transactions	db.Transactions.blocked_transactions	The number of blocked transactions.	–
commit_latency	Transactions	db.Transactions.commit_latency	The average duration of commit operations.	`db.Transactions.duration_commits / db.Transactions.xact_commit`
duration_commits	Transactions	db.Transactions.duration_commits	The total transaction time spent in the last minute, in milliseconds.	–

Counter	Type	Metric	Description	Definition
max_used_xact_ids	Transactions	db.Transactions.max_used_xact_ids	The number of transactions that haven't been vacuumed.	–
oldest_inactive_logical_replication_slot_xid_age	Transactions	db.Transactions.oldest_inactive_logical_replication_slot_xid_age	The age of the oldest transaction in an inactive logical replication slot.	–
oldest_active_logical_replication_slot_xid_age	Transactions	db.Transactions.oldest_active_logical_replication_slot_xid_age	The age of the oldest transaction in an active logical replication slot.	–
oldest_reader_feedback_xid_age	Transactions	db.Transactions.oldest_reader_feedback_xid_age	The age of the oldest transaction of a long-running transaction on an Aurora reader instance or Aurora global DB reader instance.	–
oldest_prepared_transaction_xid_age	Transactions	db.Transactions.oldest_prepared_transaction_xid_age	The age of the oldest prepared transaction.	–

Counter	Type	Metric	Description	Definition
oldest_running_transaction_xid_age	Transactions	db.Transactions.oldest_running_transaction_xid_age	The age of the oldest running transaction.	–
max_connections	Users	db.User.max_connections	The maximum number of connections allowed for a database as configured in `max_connections` parameter.	–
total_auth_attempts	Users	db.User.total_auth_attempts	The number of connection attempts to this instance.	–
archive_failed_count	WAL	db.WAL.archive_failed_count	The number of failed attempts for archiving WAL files, in files per minute.	–

SQL statistics for Performance Insights

SQL statistics are performance-related metrics about SQL queries that are collected by Performance Insights. Performance Insights gathers statistics for each second that a query is running and for each SQL call. The SQL statistics are an average for the selected time range.

A SQL digest is a composite of all queries having a given pattern but not necessarily having the same literal values. The digest replaces literal values with a question mark. For example, `SELECT * FROM emp WHERE lname= ?`. This digest might consist of the following child queries:

```
        SELECT * FROM emp WHERE lname = 'Sanchez'
        SELECT * FROM emp WHERE lname = 'Olagappan'
        SELECT * FROM emp WHERE lname = 'Wu'
```

All engines support SQL statistics for digest queries.

For the region, DB engine, and instance class support information for this feature, see Amazon Aurora DB engine, Region, and instance class support for Performance Insights features

Topics

- SQL statistics for Aurora MySQL
- SQL statistics for Aurora PostgreSQL

SQL statistics for Aurora MySQL

Aurora MySQL collect SQL statistics only at the digest level. No statistics are shown at the statement level.

Topics

- Digest statistics for Aurora MySQL
- Per-second statistics for Aurora MySQL
- Per-call statistics for Aurora MySQL

Digest statistics for Aurora MySQL

Performance Insights collects SQL digest statistics from the `events_statements_summary_by_digest` table. The `events_statements_summary_by_digest` table is managed by your database.

The digest table doesn't have an eviction policy. When the table is full, the AWS Management Console shows the following message:

```
Performance Insights is unable to collect SQL Digest statistics on new queries because
 the table events_statements_summary_by_digest is full.
Please truncate events_statements_summary_by_digest table to clear the issue. Check the
 User Guide for more details.
```

In this situation, Aurora MySQL doesn't track SQL queries. To address this issue, Performance Insights automatically truncates the digest table when both of the following conditions are met:

- The table is full.
- Performance Insights manages the Performance Schema automatically.

  For automatic management, the `performance_schema` parameter must be set to `0` and the **Source** must not be set to `user`. If Performance Insights isn't managing the Performance Schema automatically, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

In the AWS CLI, check the source of a parameter value by running the describe-db-parameters command.

Per-second statistics for Aurora MySQL

The following SQL statistics are available for Aurora MySQL DB clusters.

Metric	Unit
db.sql_tokenized.stats.count_star_per_sec	Calls per second
db.sql_tokenized.stats.sum_timer_wait_per_sec	Average active executions per second (AAE)
db.sql_tokenized.stats.sum_select_full_join_per_sec	Select full join per second
db.sql_tokenized.stats.sum_select_range_check_per_sec	Select range check per second
db.sql_tokenized.stats.sum_select_scan_per_sec	Select scan per second
db.sql_tokenized.stats.sum_sort_merge_passes_per_sec	Sort merge passes per second
db.sql_tokenized.stats.sum_sort_scan_per_sec	Sort scans per second
db.sql_tokenized.stats.sum_sort_range_per_sec	Sort ranges per second

Metric	Unit
db.sql_tokenized.stats.sum_sort_rows_per_sec	Sort rows per second
db.sql_tokenized.stats.sum_rows_affected_per_sec	Rows affected per second
db.sql_tokenized.stats.sum_rows_examined_per_sec	Rows examined per second
db.sql_tokenized.stats.sum_rows_sent_per_sec	Rows sent per second
db.sql_tokenized.stats.sum_created_tmp_disk_tables_per_sec	Created temporary disk tables per second
db.sql_tokenized.stats.sum_created_tmp_tables_per_sec	Created temporary tables per second
db.sql_tokenized.stats.sum_lock_time_per_sec	Lock time per second (in ms)

Per-call statistics for Aurora MySQL

The following metrics provide per call statistics for a SQL statement.

Metric	Unit
db.sql_tokenized.stats.sum_timer_wait_per_call	Average latency per call (in ms)
db.sql_tokenized.stats.sum_select_full_join_per_call	Select full joins per call
db.sql_tokenized.stats.sum_select_range_check_per_call	Select range check per call
db.sql_tokenized.stats.sum_select_scan_per_call	Select scans per call

Metric	Unit
db.sql_tokenized.stats.sum_sort_merge_passes_per_call	Sort merge passes per call
db.sql_tokenized.stats.sum_sort_scan_per_call	Sort scans per call
db.sql_tokenized.stats.sum_sort_range_per_call	Sort ranges per call
db.sql_tokenized.stats.sum_sort_rows_per_call	Sort rows per call
db.sql_tokenized.stats.sum_rows_affected_per_call	Rows affected per call
db.sql_tokenized.stats.sum_rows_examined_per_call	Rows examined per call
db.sql_tokenized.stats.sum_rows_sent_per_call	Rows sent per call
db.sql_tokenized.stats.sum_created_tmp_disk_tables_per_call	Created temporary disk tables per call
db.sql_tokenized.stats.sum_created_tmp_tables_per_call	Created temporary tables per call
db.sql_tokenized.stats.sum_lock_time_per_call	Lock time per call (in ms)

SQL statistics for Aurora PostgreSQL

For each SQL call and for each second that a query runs, Performance Insights collects SQL statistics. All Aurora engines collect statistics only at the digest-level.

Following, you can find information about digest-level statistics for Aurora PostgreSQL.

Topics

- Per-call digest statistics for Aurora PostgreSQL

Digest statistics for Aurora PostgreSQL

To view SQL digest statistics, the `pg_stat_statements` library must be loaded. For Aurora PostgreSQL DB clusters that are compatible with PostgreSQL 10, this library is loaded by default. For Aurora PostgreSQL DB clusters that are compatible with PostgreSQL 9.6, you enable this library manually. To enable it manually, add `pg_stat_statements` to `shared_preload_libraries` in the DB parameter group associated with the DB instance. Then reboot your DB instance. For more information, see Working with parameter groups.

Note

Performance Insights can only collect statistics for queries in `pg_stat_activity` that aren't truncated. By default, PostgreSQL databases truncate queries longer than 1,024 bytes. To increase the query size, change the `track_activity_query_size` parameter in the DB parameter group associated with your DB instance. When you change this parameter, a DB instance reboot is required.

Per-second digest statistics for Aurora PostgreSQL

The following SQL digest statistics are available for Aurora PostgreSQL DB instances.

Metric	Unit
db.sql_tokenized.stats.calls_per_sec	Calls per second
db.sql_tokenized.stats.rows_per_sec	Rows per second
db.sql_tokenized.stats.total_time_per_sec	Average active executions per second (AAE)
db.sql_tokenized.stats.shared_blks_hit_per_sec	Block hits per second
db.sql_tokenized.stats.shared_blks_read_per_sec	Block reads per second

Metric	Unit
db.sql_tokenized.stats.shared_blks_dirtied_per_sec	Blocks dirtied per second
db.sql_tokenized.stats.shared_blks_written_per_sec	Block writes per second
db.sql_tokenized.stats.local_blks_hit_per_sec	Local block hits per second
db.sql_tokenized.stats.local_blks_read_per_sec	Local block reads per second
db.sql_tokenized.stats.local_blks_dirtied_per_sec	Local block dirty per second
db.sql_tokenized.stats.local_blks_written_per_sec	Local block writes per second
db.sql_tokenized.stats.temp_blks_written_per_sec	Temporary writes per second
db.sql_tokenized.stats.temp_blks_read_per_sec	Temporary reads per second
db.sql_tokenized.stats.blk_read_time_per_sec	Average concurrent reads per second
db.sql_tokenized.stats.blk_write_time_per_sec	Average concurrent writes per second

Per-call digest statistics for Aurora PostgreSQL

The following metrics provide per call statistics for a SQL statement.

Metric	Unit
db.sql_tokenized.stats.rows_per_call	Rows per call
db.sql_tokenized.stats.avg_latency_per_call	Average latency per call (in ms)

Metric	Unit
db.sql_tokenized.stats.shared_blks_hit_per_call	Block hits per call
db.sql_tokenized.stats.shared_blks_read_per_call	Block reads per call
db.sql_tokenized.stats.shared_blks_written_per_call	Block writes per call
db.sql_tokenized.stats.shared_blks_dirtied_per_call	Blocks dirtied per call
db.sql_tokenized.stats.local_blks_hit_per_call	Local block hits per call
db.sql_tokenized.stats.local_blks_read_per_call	Local block reads per call
db.sql_tokenized.stats.local_blks_dirtied_per_call	Local block dirty per call
db.sql_tokenized.stats.local_blks_written_per_call	Local block writes per call
db.sql_tokenized.stats.temp_blks_written_per_call	Temporary block writes per call
db.sql_tokenized.stats.temp_blks_read_per_call	Temporary block reads per call
db.sql_tokenized.stats.blk_read_time_per_call	Read time per call (in ms)
db.sql_tokenized.stats.blk_write_time_per_call	Write time per call (in ms)

For more information about these metrics, see pg_stat_statements in the PostgreSQL documentation.

OS metrics in Enhanced Monitoring

Amazon Aurora provides metrics in real time for the operating system (OS) that your DB cluster runs on. Aurora delivers the metrics from Enhanced Monitoring to your Amazon CloudWatch Logs account. The following tables list the OS metrics available using Amazon CloudWatch Logs.

Topics

- OS metrics for Aurora

OS metrics for Aurora

Group	Metric	Console name	Description
`General`	`engine`	Not applicable	The database engine for the DB instance.
	`instanceID`	Not applicable	The DB instance identifier.
	`instanceResourceID`	Not applicable	An immutable identifier for the DB instance that is unique to an AWS Region, also used as the log stream identifier.
	`numVCPUs`	Not applicable	The number of virtual CPUs for the DB instance.
	`timestamp`	Not applicable	The time at which the metrics were taken.
	`uptime`	Not applicable	The amount of time that the DB instance has been active.
	`version`	Not applicable	The version of the OS metrics' stream JSON format.
`cpuUtilization`	`guest`	**CPU Guest**	The percentage of CPU in use by guest programs.

Group	Metric	Console name	Description
	`idle`	**CPU Idle**	The percentage of CPU that is idle.
	`irq`	**CPU IRQ**	The percentage of CPU in use by software interrupts.
	`nice`	**CPU Nice**	The percentage of CPU in use by programs running at lowest priority.
	`steal`	**CPU Steal**	The percentage of CPU in use by other virtual machines.
	`system`	**CPU System**	The percentage of CPU in use by the kernel.
	`total`	**CPU Total**	The total percentage of the CPU in use. This value includes the `nice` value.
	`user`	**CPU User**	The percentage of CPU in use by user programs.
	`wait`	**CPU Wait**	The percentage of CPU unused while waiting for I/O access.
`diskIO`	`avgQueueLen`	**Avg Queue Size**	The number of requests waiting in the I/O device's queue.
	`avgReqSz`	**Ave Request Size**	The average request size, in kilobytes.
	`await`	**Disk I/O Await**	The number of milliseconds required to respond to requests, including queue time and service time.
	`device`	Not applicable	The identifier of the disk device in use.
	`readIOsPS`	**Read IO/s**	The number of read operations per second.

Group	Metric	Console name	Description
	`readKb`	**Read Total**	The total number of kilobytes read.
	`readKbPS`	**Read Kb/s**	The number of kilobytes read per second.
	`readLatency`	**Read Latency**	The elapsed time between the submission of a read I/O request and its completion, in milliseconds. This metric is only available for Amazon Aurora.
	`readThroughput`	**Read Throughput**	The amount of network throughput used by requests to the DB cluster, in bytes per second. This metric is only available for Amazon Aurora.
	`rrqmPS`	**Rrqms**	The number of merged read requests queued per second.
	`tps`	**TPS**	The number of I/O transactions per second.
	`util`	**Disk I/O Util**	The percentage of CPU time during which requests were issued.
	`writeIOsPS`	**Write IO/s**	The number of write operations per second.
	`writeKb`	**Write Total**	The total number of kilobytes written.
	`writeKbPS`	**Write Kb/s**	The number of kilobytes written per second.
	`writeLatency`	**Write Latency**	The average elapsed time between the submission of a write I/O request and its completion, in milliseconds. This metric is only available for Amazon Aurora.

Group	Metric	Console name	Description
	`writeThroughput`	**Write Throughput**	The amount of network throughput used by responses from the DB cluster, in bytes per second. This metric is only available for Amazon Aurora.
	`wrqmPS`	**Wrqms**	The number of merged write requests queued per second.
`fileSys`	`maxFiles`	**Max Inodes**	The maximum number of files that can be created for the file system.
	`mountPoint`	Not applicable	The path to the file system.
	`name`	Not applicable	The name of the file system.
	`total`	**Total Filesystem**	The total number of disk space available for the file system, in kilobytes.
	`used`	**Used Filesystem**	The amount of disk space used by files in the file system, in kilobytes.
	`usedFilePercent`	**Used Inodes**	The percentage of available files in use.
	`usedFiles`	**Used%**	The number of files in the file system.
	`usedPercent`	**Used Filesystem**	The percentage of the file-system disk space in use.
`loadAverageMinute`	`fifteen`	**Load Avg 15 min**	The number of processes requesting CPU time over the last 15 minutes.
	`five`	**Load Avg 5 min**	The number of processes requesting CPU time over the last 5 minutes.

Group	Metric	Console name	Description
	one	**Load Avg 1 min**	The number of processes requesting CPU time over the last minute.
memory	active	**Active Memory**	The amount of assigned memory, in kilobytes.
	buffers	**Buffered Memory**	The amount of memory used for buffering I/O requests prior to writing to the storage device, in kilobytes.
	cached	**Cached Memory**	The amount of memory used for caching file system–based I/O.
	dirty	**Dirty Memory**	The amount of memory pages in RAM that have been modified but not written to their related data block in storage, in kilobytes.
	free	**Free Memory**	The amount of unassigned memory, in kilobytes.
	hugePages Free	**Huge Pages Free**	The number of free huge pages. Huge pages are a feature of the Linux kernel.
	hugePages Rsvd	**Huge Pages Rsvd**	The number of committed huge pages.
	hugePages Size	**Huge Pages Size**	The size for each huge pages unit, in kilobytes.
	hugePages Surp	**Huge Pages Surp**	The number of available surplus huge pages over the total.

Group	Metric	Console name	Description
	hugePagesTotal	**Huge Pages Total**	The total number of huge pages.
	inactive	**Inactive Memory**	The amount of least-frequently used memory pages, in kilobytes.
	mapped	**Mapped Memory**	The total amount of file-system contents that is memory mapped inside a process address space, in kilobytes.
	pageTables	**Page Tables**	The amount of memory used by page tables, in kilobytes.
	slab	**Slab Memory**	The amount of reusable kernel data structures, in kilobytes.
	total	**Total Memory**	The total amount of memory, in kilobytes.
	writeback	**Writeback Memory**	The amount of dirty pages in RAM that are still being written to the backing storage, in kilobytes.
network	interface	Not applicable	The identifier for the network interface being used for the DB instance.
	rx	**RX**	The number of bytes received per second.
	tx	**TX**	The number of bytes uploaded per second.
processList	cpuUsedPc	**CPU %**	The percentage of CPU used by the process.
	id	Not applicable	The identifier of the process.

Group	Metric	Console name	Description
	memoryUsedPc	**MEM%**	The percentage of memory used by the process.
	name	Not applicable	The name of the process.
	parentID	Not applicable	The process identifier for the parent process of the process.
	rss	**RES**	The amount of RAM allocated to the process, in kilobytes.
	tgid	Not applicable	The thread group identifier, which is a number representing the process ID to which a thread belongs. This identifier is used to group threads from the same process.
	vss	**VIRT**	The amount of virtual memory allocated to the process, in kilobytes.
swap	swap	**Swap**	The amount of swap memory available, in kilobytes.
	swap in	**Swaps in**	The amount of memory, in kilobytes, swapped in from disk.
	swap out	**Swaps out**	The amount of memory, in kilobytes, swapped out to disk.
	free	**Free Swap**	The amount of swap memory free, in kilobytes.
	committed	**Committed Swap**	The amount of swap memory, in kilobytes, used as cache memory.
tasks	blocked	**Tasks Blocked**	The number of tasks that are blocked.

Group	Metric	Console name	Description
	`running`	**Tasks Running**	The number of tasks that are running.
	`sleeping`	**Tasks Sleeping**	The number of tasks that are sleeping.
	`stopped`	**Tasks Stopped**	The number of tasks that are stopped.
	`total`	**Tasks Total**	The total number of tasks.
	`zombie`	**Tasks Zombie**	The number of child tasks that are inactive with an active parent task.

Monitoring events, logs, and streams in an Amazon Aurora DB cluster

When you monitor your Amazon Aurora databases and your other AWS solutions, your goal is to maintain the following:

- Reliability
- Availability
- Performance
- Security

Monitoring metrics in an Amazon Aurora cluster explains how to monitor your cluster using metrics. A complete solution must also monitor database events, log files, and activity streams. AWS provides you with the following monitoring tools:

- *Amazon EventBridge* is a serverless event bus service that makes it easy to connect your applications with data from a variety of sources. EventBridge delivers a stream of real-time data from your own applications, Software-as-a-Service (SaaS) applications, and AWS services. EventBridge routes that data to targets such as AWS Lambda. This way, you can monitor events that happen in services and build event-driven architectures. For more information, see the Amazon EventBridge User Guide.
- *Amazon CloudWatch Logs* provides a way to monitor, store, and access your log files from Amazon Aurora instances, AWS CloudTrail, and other sources. Amazon CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the Amazon CloudWatch Logs User Guide.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account. CloudTrail delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.
- *Database Activity Streams* is an Amazon Aurora feature that provides a near real-time stream of the activity in your DB cluster. Amazon Aurora pushes activities to an Amazon Kinesis data stream. The Kinesis stream is created automatically. From Kinesis, you can configure AWS

services such as Amazon Data Firehose and AWS Lambda to consume the stream and store the data.

Topics

Viewing logs, events, and streams in the Amazon RDS console

Amazon RDS integrates with AWS services to show information about logs, events, and database activity streams in the RDS console.

The **Logs & events** tab for your Aurora DB cluster shows the following information:

- **Auto scaling policies and activities** – Shows policies and activities relating to the Aurora Auto Scaling feature. This information only appears in the **Logs & events** tab at the cluster level.
- **Amazon CloudWatch alarms** – Shows any metric alarms that you have configured for the DB instance in your Aurora cluster. If you haven't configured alarms, you can create them in the RDS console.
- **Recent events** – Shows a summary of events (environment changes) for your Aurora DB instance or cluster. For more information, see Viewing Amazon RDS events.
- **Logs** – Shows database log files generated by a DB instance in your Aurora cluster. For more information, see Monitoring Amazon Aurora log files.

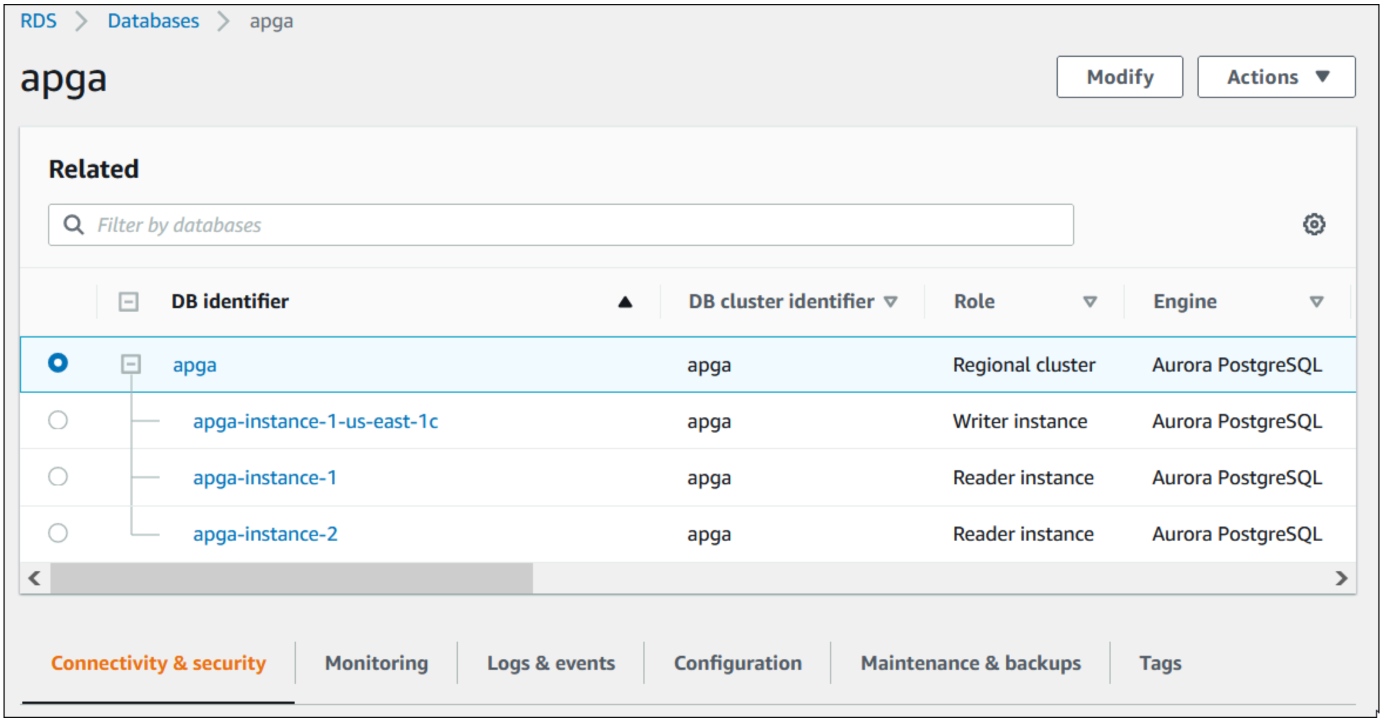
The **Configuration** tab displays information about database activity streams.

To view logs, events, and streams for your Aurora DB cluster in the RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

3. Choose the name of the Aurora DB cluster that you want to monitor.

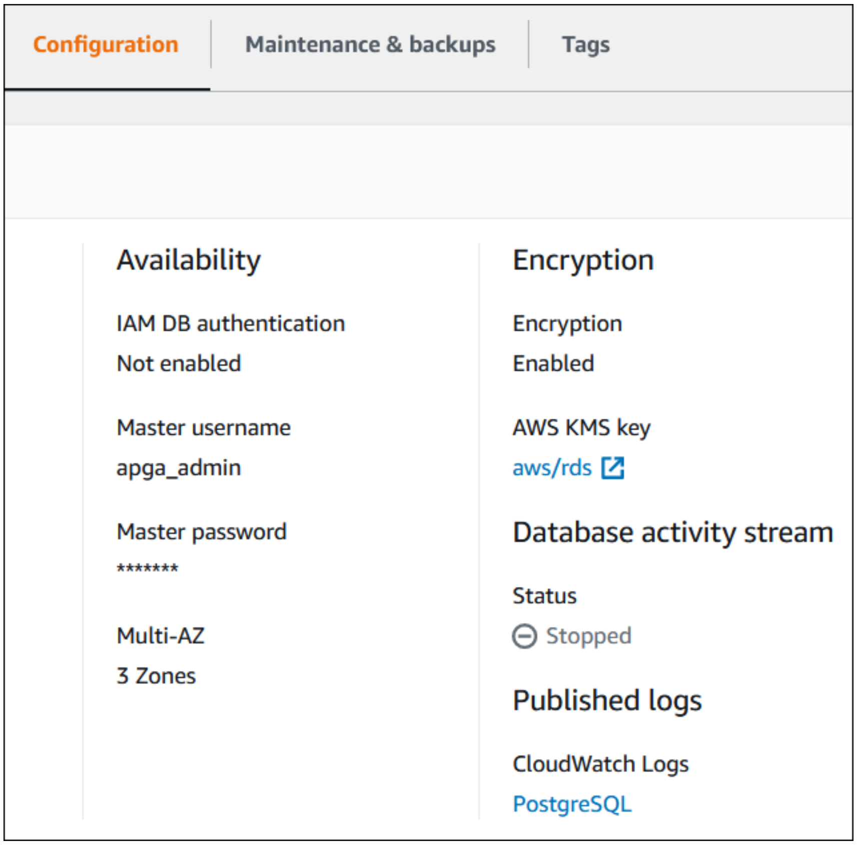
   The database page appears. The following example shows an Amazon Aurora PostgreSQL DB cluster named apga.



	DB identifier ▲	DB cluster identifier ▽	Role ▽	Engine ▽
●	apga	apga	Regional cluster	Aurora PostgreSQL
○	apga-instance-1-us-east-1c	apga	Writer instance	Aurora PostgreSQL
○	apga-instance-1	apga	Reader instance	Aurora PostgreSQL
○	apga-instance-2	apga	Reader instance	Aurora PostgreSQL

4. Scroll down and choose **Configuration**.

   The following example shows the status of the database activity streams for your cluster.



Availability

IAM DB authentication
Not enabled

Master username
apga_admin

Master password
****

Multi-AZ
3 Zones

Encryption

Encryption
Enabled

AWS KMS key
aws/rds

Database activity stream

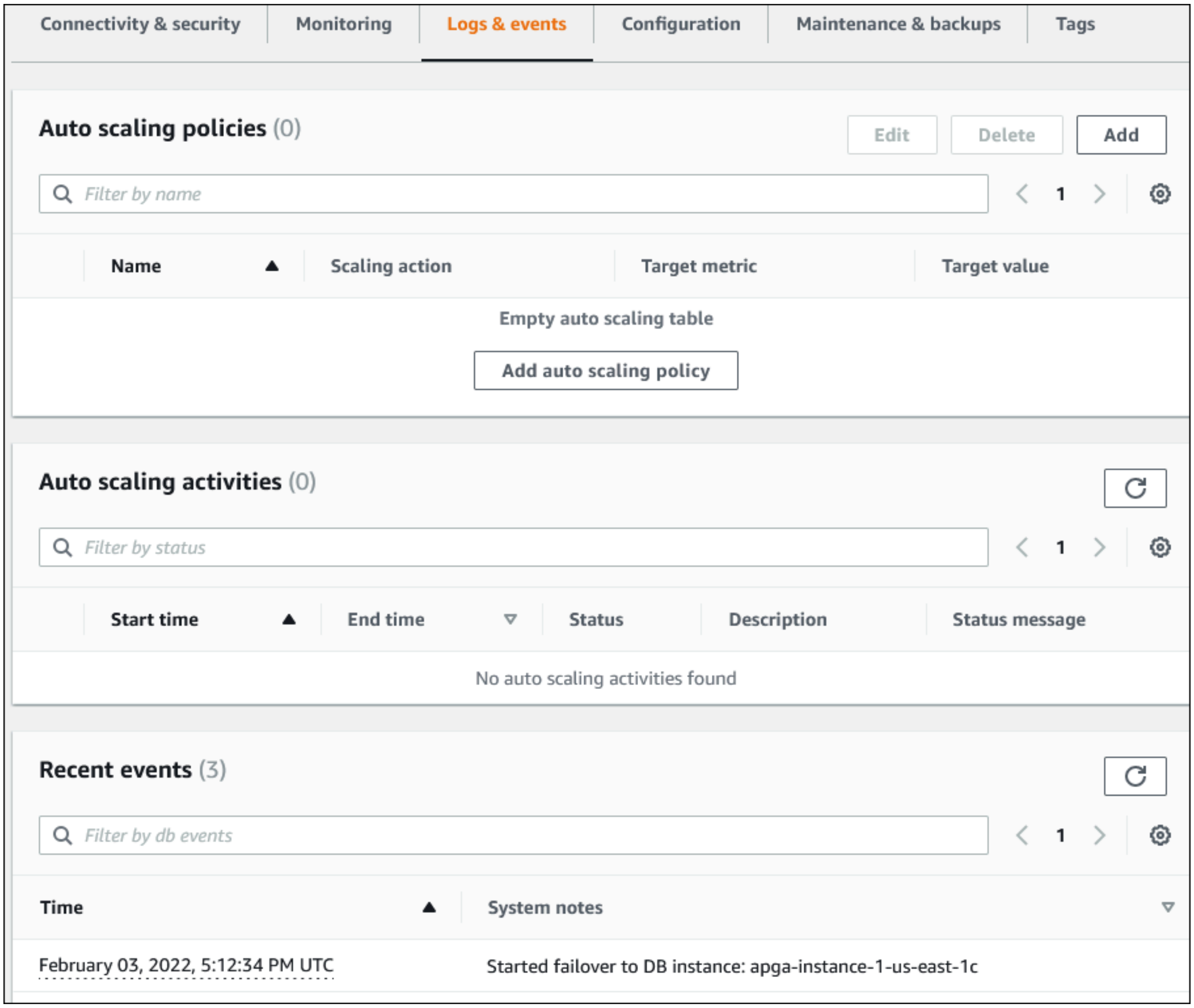
Status
⊖ Stopped

Published logs

CloudWatch Logs
PostgreSQL

5. Choose **Logs & events**.

   The Logs & events section appears.



Auto scaling policies (0)

Name	Scaling action	Target metric	Target value

Empty auto scaling table

Auto scaling activities (0)

Start time	End time	Status	Description	Status message

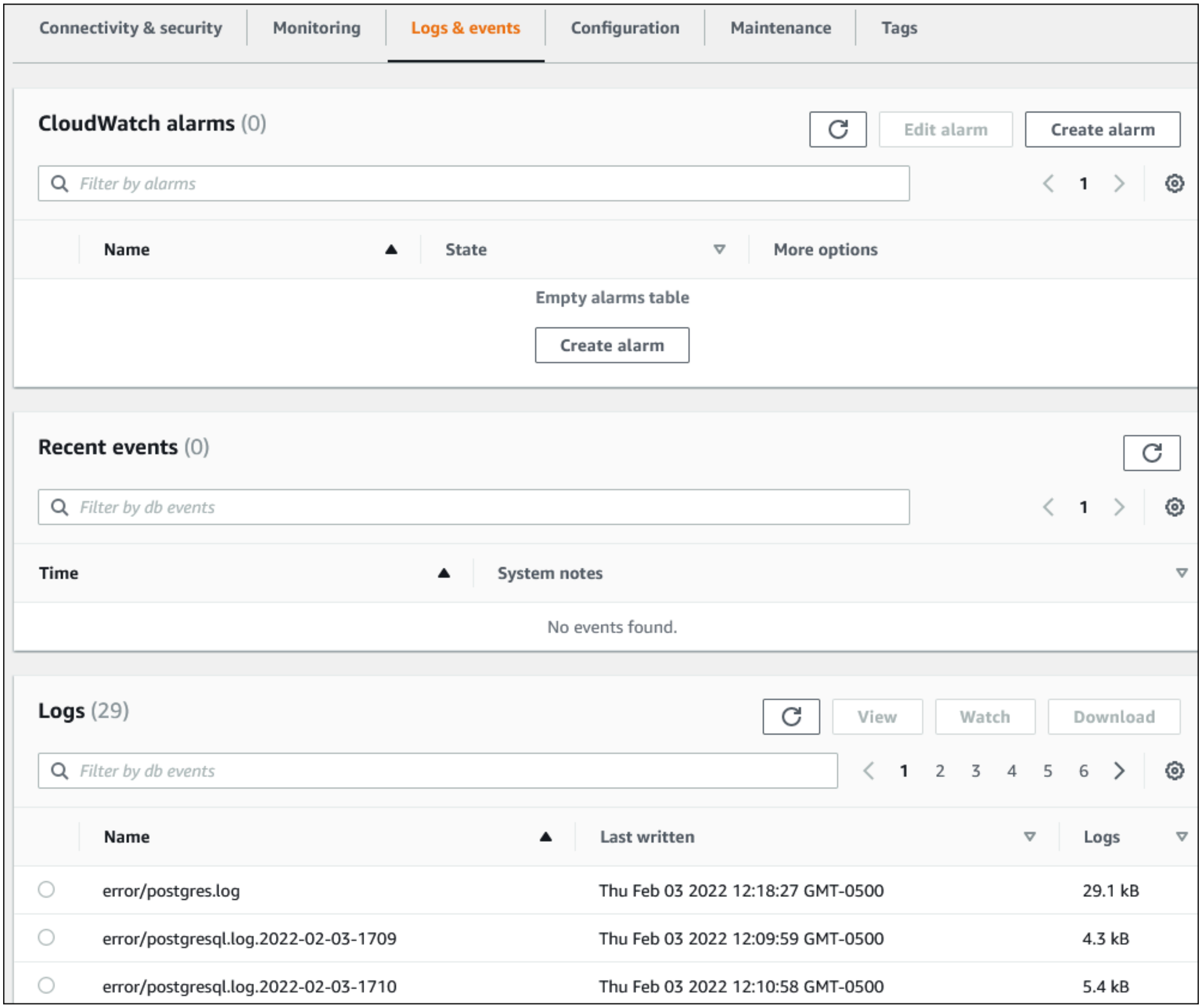
No auto scaling activities found

Recent events (3)

Time	System notes
February 03, 2022, 5:12:34 PM UTC	Started failover to DB instance: apga-instance-1-us-east-1c

6. Choose a DB instance in your Aurora cluster, and then choose **Logs & events** for the instance.

   The following example shows that the contents are different between the DB instance page and the DB cluster page. The DB instance page shows logs and alarms.



CloudWatch alarms (0)

Name ▲	State ▽	More options

Empty alarms table

Recent events (0)

Time ▲	System notes ▽

No events found.

Logs (29)

	Name ▲	Last written ▽	Logs ▽
○	error/postgres.log	Thu Feb 03 2022 12:18:27 GMT-0500	29.1 kB
○	error/postgresql.log.2022-02-03-1709	Thu Feb 03 2022 12:09:59 GMT-0500	4.3 kB
○	error/postgresql.log.2022-02-03-1710	Thu Feb 03 2022 12:10:58 GMT-0500	5.4 kB

Monitoring Amazon Aurora events

An *event* indicates a change in an environment. This can be an AWS environment, an SaaS partner service or application, or a custom application or service. For descriptions of the Aurora events, see Amazon RDS event categories and event messages for Aurora.

Topics

Overview of events for Aurora

An *RDS event* indicates a change in the Aurora environment. For example, Amazon Aurora generates an event when a DB cluster is patched. Amazon Aurora delivers events to EventBridge in near-real time.

> **Note**
>
> Amazon RDS emits events on a best effort basis. We recommend that you avoid writing programs that depend on the order or existence of notification events, because they might be out of sequence or missing.

Amazon RDS records events that relate to the following resources:

- DB clusters

  For a list of cluster events, see DB cluster events.
- DB instances

  For a list of DB instance events, see DB instance events.
- DB parameter groups

  For a list of DB parameter group events, see DB parameter group events.

- DB security groups

  For a list of DB security group events, see DB security group events.
- DB cluster snapshots

  For a list of DB cluster snapshot events, see DB cluster snapshot events.
- RDS Proxy events

  For a list of RDS Proxy events, see RDS Proxy events.
- Blue/green deployment events

  For a list of blue/green deployment events, see Blue/green deployment events.

This information includes the following:

- The date and time of the event
- The source name and source type of the event
- A message associated with the event
- Event notifications include tags from when the message was sent and may not reflect tags at the time when the event occurred

Viewing Amazon RDS events

You can retrieve the following event information for your Amazon Aurora resources:

- Resource name
- Resource type
- Time of the event
- Message summary of the event

Access the events through the AWS Management Console, which shows events from the past 24 hours. You can also retrieve events by using the describe-events AWS CLI command, or the DescribeEvents RDS API operation. If you use the AWS CLI or the RDS API to view events, you can retrieve events for up to the past 14 days.

> **Note**
>
> If you need to store events for longer periods of time, you can send Amazon RDS events to EventBridge. For more information, see Creating a rule that triggers on an Amazon Aurora event

For descriptions of the Amazon Aurora events, see Amazon RDS event categories and event messages for Aurora.

To access detailed information about events using AWS CloudTrail, including request parameters, see CloudTrail events.

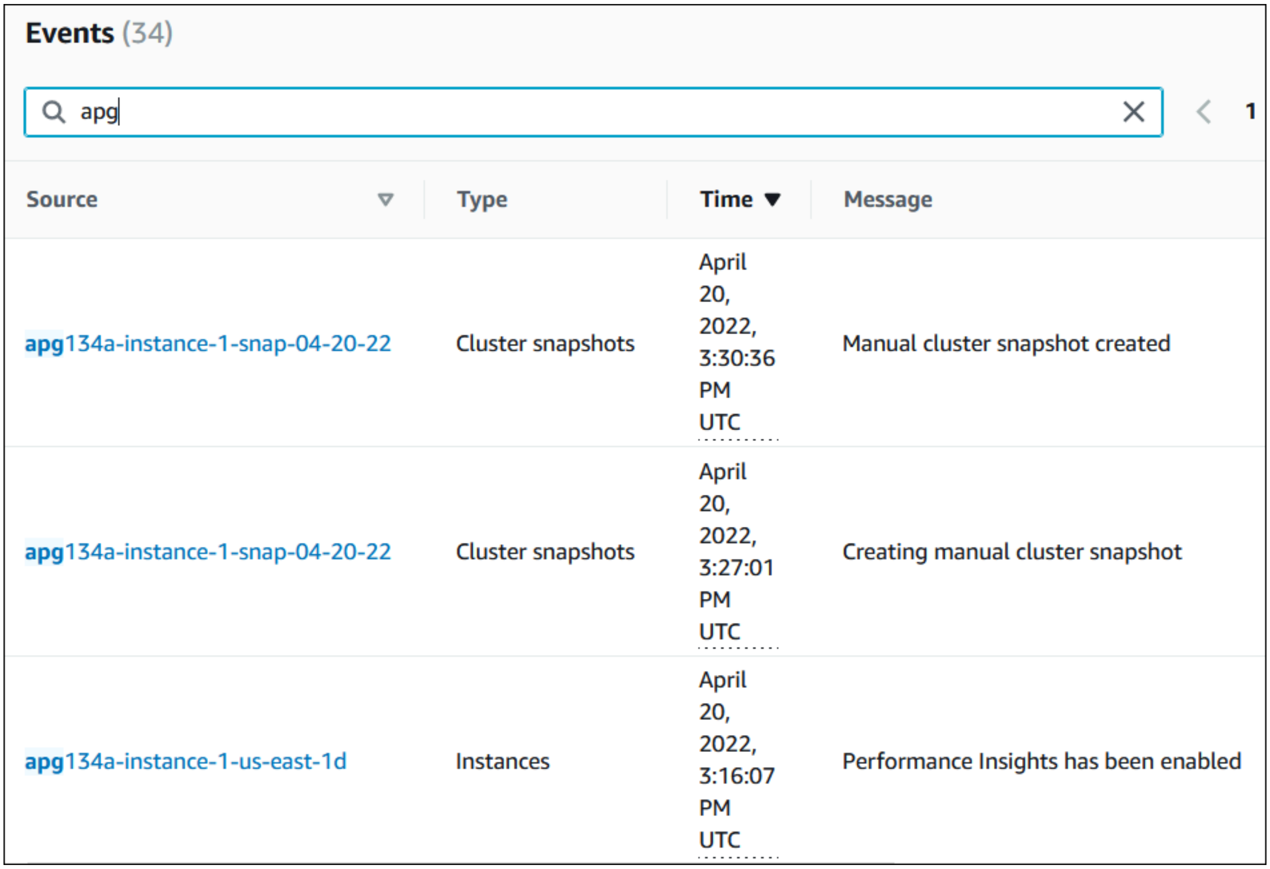
Console

To view all Amazon RDS events for the past 24 hours

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Events**.

   The available events appear in a list.
3. (Optional) Enter a search term to filter your results.

The following example shows a list of events filtered by the characters **apg**.



Events (34)

apg

Source	Type	Time ▼	Message
apg134a-instance-1-snap-04-20-22	Cluster snapshots	April 20, 2022, 3:30:36 PM UTC	Manual cluster snapshot created
apg134a-instance-1-snap-04-20-22	Cluster snapshots	April 20, 2022, 3:27:01 PM UTC	Creating manual cluster snapshot
apg134a-instance-1-us-east-1d	Instances	April 20, 2022, 3:16:07 PM UTC	Performance Insights has been enabled

AWS CLI

To view all events generated in the last hour, call describe-events with no parameters.

```
aws rds describe-events
```

The following sample output shows that a DB cluster instance has started recovery.

```
{
    "Events": [
        {
            "EventCategories": [
                "recovery"
            ],
            "SourceType": "db-instance",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:db:mycluster-instance-1",
            "Date": "2022-04-20T15:02:38.416Z",
```

]]>

```
            "Message": "Recovery of the DB instance has started. Recovery time will
 vary with the amount of data to be recovered.",
            "SourceIdentifier": "mycluster-instance-1"
        }, ...
```

To view all Amazon RDS events for the past 10080 minutes (7 days), call the describe-events AWS CLI command and set the `--duration` parameter to `10080`.

```
aws rds describe-events --duration 10080
```

The following example shows the events in the specified time range for DB instance *test-instance*.

```
aws rds describe-events \
    --source-identifier test-instance \
    --source-type db-instance \
    --start-time 2022-03-13T22:00Z \
    --end-time 2022-03-13T23:59Z
```

The following sample output shows the status of a backup.

```
{
    "Events": [
        {
            "SourceType": "db-instance",
            "SourceIdentifier": "test-instance",
            "EventCategories": [
                "backup"
            ],
            "Message": "Backing up DB instance",
            "Date": "2022-03-13T23:09:23.983Z",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:db:test-instance"
        },
        {
            "SourceType": "db-instance",
            "SourceIdentifier": "test-instance",
            "EventCategories": [
                "backup"
            ],
            "Message": "Finished DB Instance backup",
            "Date": "2022-03-13T23:15:13.049Z",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:db:test-instance"
        }
    ]
}
```

API

You can view all Amazon RDS instance events for the past 14 days by calling the DescribeEvents RDS API operation and setting the `Duration` parameter to `20160`.

Working with Amazon RDS event notification

Amazon RDS uses the Amazon Simple Notification Service (Amazon SNS) to provide notification when an Amazon RDS event occurs. These notifications can be in any notification form supported by Amazon SNS for an AWS Region, such as an email, a text message, or a call to an HTTP endpoint.

Topics

Overview of Amazon RDS event notification

Amazon RDS groups events into categories that you can subscribe to so that you can be notified when an event in that category occurs.

Topics

RDS resources eligible for event subscription

For Amazon Aurora, events occur at both the DB cluster and the DB instance level. You can subscribe to an event category for the following resources:

- DB instance
- DB cluster
- DB cluster snapshot
- DB parameter group
- DB security group
- RDS Proxy
- Custom engine version

For example, if you subscribe to the backup category for a given DB instance, you're notified whenever a backup-related event occurs that affects the DB instance. If you subscribe to a configuration change category for a DB instance, you're notified when the DB instance is changed. You also receive notification when an event notification subscription changes.

You might want to create several different subscriptions. For example, you might create one subscription that receives all event notifications for all DB instances and another subscription that includes only critical events for a subset of the DB instances. For the second subscription, specify one or more DB instances in the filter.

Basic process for subscribing to Amazon RDS event notifications

The process for subscribing to Amazon RDS event notification is as follows:

1. You create an Amazon RDS event notification subscription by using the Amazon RDS console, AWS CLI, or API.

   Amazon RDS uses the ARN of an Amazon SNS topic to identify each subscription. The Amazon RDS console creates the ARN for you when you create the subscription. Create the ARN by using the Amazon SNS console, the AWS CLI, or the Amazon SNS API.
2. Amazon RDS sends an approval email or SMS message to the addresses you submitted with your subscription.
3. You confirm your subscription by choosing the link in the notification you received.

4. The Amazon RDS console updates the **My Event Subscriptions** section with the status of your subscription.
5. Amazon RDS begins sending the notifications to the addresses that you provided when you created the subscription.

To learn about identity and access management when using Amazon SNS, see Identity and access management in Amazon SNS in the *Amazon Simple Notification Service Developer Guide*.

You can use AWS Lambda to process event notifications from a DB instance. For more information, see Using AWS Lambda with Amazon RDS in the *AWS Lambda Developer Guide*.

Delivery of RDS event notifications

Amazon RDS sends notifications to the addresses that you provide when you create the subscription. The notification can include message attributes which provide structured metadata about the message. For more information about message attributes, see Amazon RDS event categories and event messages for Aurora.

Event notifications might take up to five minutes to be delivered.

> ⚠ **Important**
>
> Amazon RDS doesn't guarantee the order of events sent in an event stream. The event order is subject to change.

When Amazon SNS sends a notification to a subscribed HTTP or HTTPS endpoint, the POST message sent to the endpoint has a message body that contains a JSON document. For more information, see Amazon SNS message and JSON formats in the *Amazon Simple Notification Service Developer Guide*.

You can configure SNS to notify you with text messages. For more information, see Mobile text messaging (SMS) in the *Amazon Simple Notification Service Developer Guide*.

To turn off notifications without deleting a subscription, choose **No** for **Enabled** in the Amazon RDS console. Or you can set the `Enabled` parameter to `false` using the AWS CLI or Amazon RDS API.

Billing for Amazon RDS event notifications

Billing for Amazon RDS event notification is through Amazon SNS. Amazon SNS fees apply when using event notification. For more information about Amazon SNS billing, see Amazon Simple Notification Service pricing.

Examples of Aurora events using Amazon EventBridge

The following examples illustrate different types of Aurora events in JSON format. For a tutorial that shows you how to capture and view events in JSON format, see Tutorial: Log DB instance state changes using Amazon EventBridge.

Topics

- Example of a DB cluster event
- Example of a DB parameter group event
- Example of a DB cluster snapshot event

Example of a DB cluster event

The following is an example of a DB cluster event in JSON format. The event shows that the cluster named `my-db-cluster` was patched. The event ID is `RDS-EVENT-0173`.

```
{
  "version": "0",
  "id": "844e2571-85d4-695f-b930-0153b71dcb42",
  "detail-type": "RDS DB Cluster Event",
  "source": "aws.rds",
  "account": "123456789012",
  "time": "2018-10-06T12:26:13Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:rds:us-east-1:123456789012:cluster:my-db-cluster"
  ],
  "detail": {
    "EventCategories": [
      "notification"
    ],
    "SourceType": "CLUSTER",
    "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:my-db-cluster",
    "Date": "2018-10-06T12:26:13.882Z",
    "Message": "Database cluster has been patched",
    "SourceIdentifier": "my-db-cluster",
    "EventID": "RDS-EVENT-0173"
  }
}
```

Example of a DB parameter group event

The following is an example of a DB parameter group event in JSON format. The event shows that the parameter `time_zone` was updated in parameter group `my-db-param-group`. The event ID is RDS-EVENT-0037.

```
{
  "version": "0",
  "id": "844e2571-85d4-695f-b930-0153b71dcb42",
  "detail-type": "RDS DB Parameter Group Event",
  "source": "aws.rds",
  "account": "123456789012",
  "time": "2018-10-06T12:26:13Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:rds:us-east-1:123456789012:pg:my-db-param-group"
  ],
  "detail": {
    "EventCategories": [
      "configuration change"
    ],
    "SourceType": "DB_PARAM",
    "SourceArn": "arn:aws:rds:us-east-1:123456789012:pg:my-db-param-group",
    "Date": "2018-10-06T12:26:13.882Z",
    "Message": "Updated parameter time_zone to UTC with apply method immediate",
    "SourceIdentifier": "my-db-param-group",
    "EventID": "RDS-EVENT-0037"
  }
}
```

Example of a DB cluster snapshot event

The following is an example of a DB cluster snapshot event in JSON format. The event shows the creation of the snapshot named `my-db-cluster-snapshot`. The event ID is RDS-EVENT-0074.

```
{
  "version": "0",
  "id": "844e2571-85d4-695f-b930-0153b71dcb42",
  "detail-type": "RDS DB Cluster Snapshot Event",
  "source": "aws.rds",
  "account": "123456789012",
  "time": "2018-10-06T12:26:13Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:rds:us-east-1:123456789012:cluster-snapshot:rds:my-db-cluster-snapshot"
  ],
  "detail": {
    "EventCategories": [
      "backup"
    ],
    "SourceType": "CLUSTER_SNAPSHOT",
    "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster-snapshot:rds:my-db-
cluster-snapshot",
    "Date": "2018-10-06T12:26:13.882Z",
    "SourceIdentifier": "my-db-cluster-snapshot",
    "Message": "Creating manual cluster snapshot",
    "EventID": "RDS-EVENT-0074"
  }
}
```

Granting permissions to publish notifications to an Amazon SNS topic

To grant Amazon RDS permissions to publish notifications to an Amazon Simple Notification Service (Amazon SNS) topic, attach an AWS Identity and Access Management (IAM) policy to the destination topic. For more information about permissions, see Example cases for Amazon Simple Notification Service access control in the *Amazon Simple Notification Service Developer Guide*.

By default, an Amazon SNS topic has a policy allowing all Amazon RDS resources within the same account to publish notifications to it. You can attach a custom policy to allow cross-account notifications, or to restrict access to certain resources.

The following is an example of an IAM policy that you attach to the destination Amazon SNS topic. It restricts the topic to DB instances with names that match the specified prefix. To use this policy, specify the following values:

- `Resource` – The Amazon Resource Name (ARN) for your Amazon SNS topic
- `SourceARN` – Your RDS resource ARN
- `SourceAccount` – Your AWS account ID

To see a list of resource types and their ARNs, see Resources Defined by Amazon RDS in the *Service Authorization Reference*.

```
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.rds.amazonaws.com"
      },
      "Action": [
        "sns:Publish"
      ],
      "Resource": "arn:aws:sns:us-east-1:123456789012:topic_name",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:rds:us-east-1:123456789012:db:prefix-*"
        },
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
```

```
            }
         }
      }
   ]
}
```

Subscribing to Amazon RDS event notification

The simplest way to create a subscription is with the RDS console. If you choose to create event notification subscriptions using the CLI or API, you must create an Amazon Simple Notification Service topic and subscribe to that topic with the Amazon SNS console or Amazon SNS API. You will also need to retain the Amazon Resource Name (ARN) of the topic because it is used when submitting CLI commands or API operations. For information on creating an SNS topic and subscribing to it, see Getting started with Amazon SNS in the *Amazon Simple Notification Service Developer Guide*.

You can specify the type of source you want to be notified of and the Amazon RDS source that triggers the event:

Source type

> The type of source. For example, **Source type** might be **Instances**. You must choose a source type.

***`Resources`* to include**

> The Amazon RDS resources that are generating the events. For example, you might choose **Select specific instances** and then **myDBInstance1**.

The following table explains the result when you specify or don't specify ***`Resources`* to include**.

Resources to include	Description	Example
Specified	RDS notifies you about all events for the specified resource only.	If your **Source type** is **Instances** and your resource is **myDBInstance1**, RDS notifies you about all events for myDBInstance1 only.
Not specified	RDS notifies you about the events for the specified source type for all your Amazon RDS resources.	If your **Source type** is **Instances**, RDS notifies you about all instance-related events in your account.

An Amazon SNS topic subscriber receives every message published to the topic by default. To receive only a subset of the messages, the subscriber must assign a filter policy to the topic subscription. For more information about SNS message filtering, see Amazon SNS message filtering in the *Amazon Simple Notification Service Developer Guide*

Console

To subscribe to RDS event notification

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In navigation pane, choose **Event subscriptions**.
3. In the **Event subscriptions** pane, choose **Create event subscription**.
4. Enter your subscription details as follows:

   a. For **Name**, enter a name for the event notification subscription.

   b. For **Send notifications to**, do one of the following:

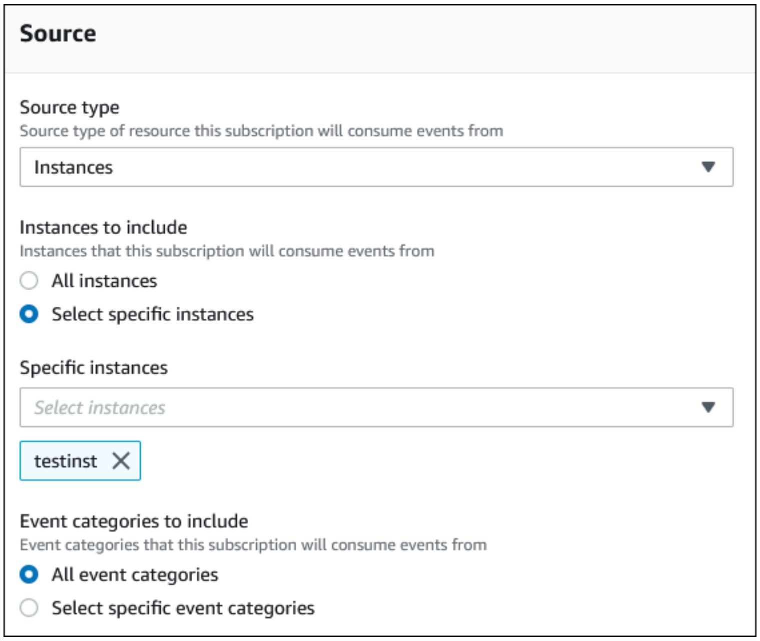
      - Choose **New email topic**. Enter a name for your email topic and a list of recipients. We recommend that you configure the events subscriptions to the same email address as your primary account contact. The recommendations, service events, and personal health messages are sent using different channels. The subscriptions to the same email address ensures that all the messages are consolidated in one location.
      - Choose **Amazon Resource Name (ARN)**. Then choose existing Amazon SNS ARN for an Amazon SNS topic.

        If you want to use a topic that has been enabled for server-side encryption (SSE), grant Amazon RDS the necessary permissions to access the AWS KMS key. For more information, see Enable compatibility between event sources from AWS services and encrypted topics in the *Amazon Simple Notification Service Developer Guide*.

   c. For **Source type**, choose a source type. For example, choose **Clusters** or **Cluster snapshots**.

   d. Choose the event categories and resources that you want to receive event notifications for.

      The following example configures event notifications for the DB instance named `testinst`.



Source

Source type

Source type of resource this subscription will consume events from

Instances ▼

Instances to include

Instances that this subscription will consume events from

○ All instances

● Select specific instances

Specific instances

Select instances ▼

testinst ✕

Event categories to include

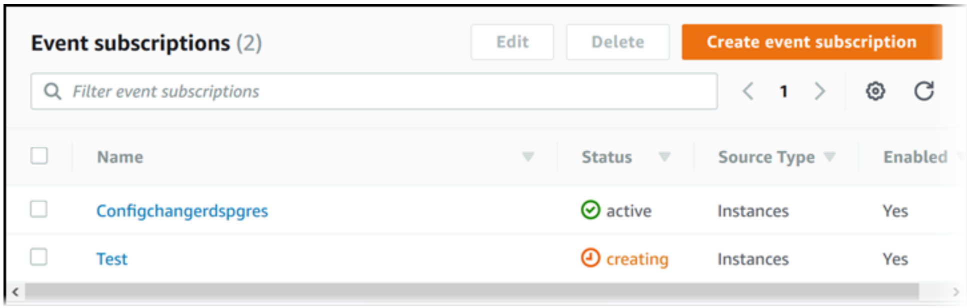
Event categories that this subscription will consume events from

● All event categories

○ Select specific event categories

e. Choose **Create**.

The Amazon RDS console indicates that the subscription is being created.



Event subscriptions (2)

□	Name	Status	Source Type	Enabled
□	Configchangerdspgres	✓ active	Instances	Yes
□	Test	creating	Instances	Yes

AWS CLI

To subscribe to RDS event notification, use the AWS CLI `create-event-subscription` command. Include the following required parameters:

- `--subscription-name`
- `--sns-topic-arn`

Example

For Linux, macOS, or Unix:

```
aws rds create-event-subscription \
    --subscription-name myeventsubscription \
    --sns-topic-arn arn:aws:sns:us-east-1:123456789012:myawsuser-RDS \
    --enabled
```

For Windows:

```
aws rds create-event-subscription ^
    --subscription-name myeventsubscription ^
    --sns-topic-arn arn:aws:sns:us-east-1:123456789012:myawsuser-RDS ^
    --enabled
```

API

To subscribe to Amazon RDS event notification, call the Amazon RDS API function `CreateEventSubscription`. Include the following required parameters:

- `SubscriptionName`
- `SnsTopicArn`

Amazon RDS event notification tags and attributes

When Amazon RDS sends an event notification to Amazon Simple Notification Service (SNS) or Amazon EventBridge, the notification contains message attributes and event tags. RDS sends the message attributes separately along with the message, while the event tags are in the body of the message. Use the message attributes and the Amazon RDS tags to add metadata to your resources. You can modify these tags with your own notations about the DB instances, Aurora clusters, and so on. For more information about tagging Amazon RDS resources, see Tagging Amazon RDS resources.

By default, the Amazon SNS and Amazon EventBridge receives every message sent to them. SNS and EventBridge can filter the message and send the notifications to the preferred communication mode, such as an email, a text message, or a call to an HTTP endpoint.

> **Note**
>
> The notification sent in an email or a text message will not have event tags.

The following table shows the message attributes for RDS events sent to the topic subscriber.

Amazon RDS event attribute	Description
EventID	Identifier for the RDS event message, for example, RDS-EVENT-0006.
Resource	The ARN identifier for the resource emitting the event, for example, `arn:aws:rds:ap-southeast-2:123456789012:db:database-1` .

The RDS tags provide data about the resource that was affected by the service event. RDS adds the current state of the tags in the message body when the notification is sent to SNS or EventBridge.

For more information about filtering message attributes for SNS, see Amazon SNS message filtering in the *Amazon Simple Notification Service Developer Guide*.

For more information about filtering event tags for EventBridge, see Content filtering in Amazon EventBridge event patterns in the *Amazon EventBridge User Guide*.

For more information about filtering payload-based tags for SNS, see https://aws.amazon.com/blogs/compute/introducing-payload-based-message-filtering-for-amazon-sns/

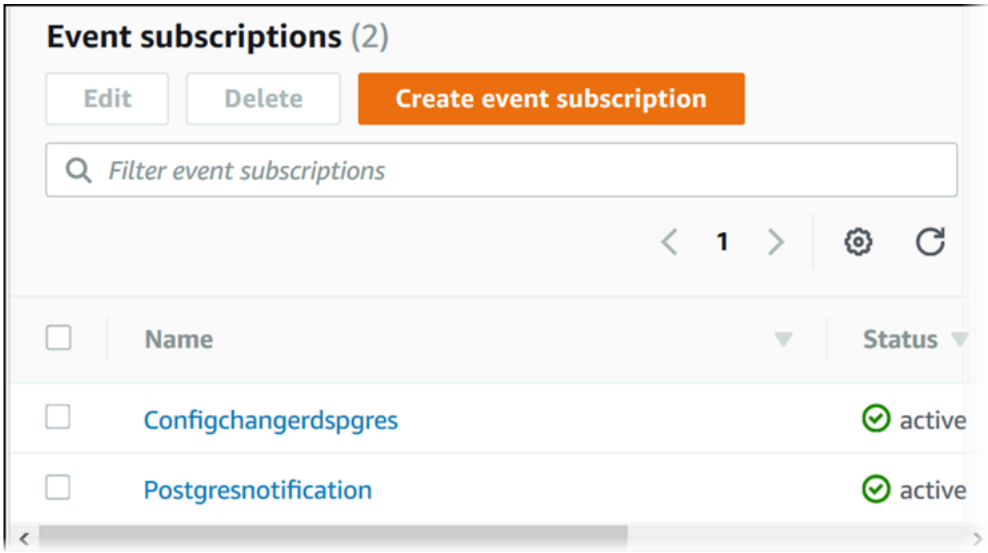
Listing Amazon RDS event notification subscriptions

You can list your current Amazon RDS event notification subscriptions.

Console

To list your current Amazon RDS event notification subscriptions

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Event subscriptions**. The **Event subscriptions** pane shows all your event notification subscriptions.



□	Name	Status
□	Configchangerdspgres	✓ active
□	Postgresnotification	✓ active

AWS CLI

To list your current Amazon RDS event notification subscriptions, use the AWS CLI `describe-event-subscriptions` command.

Example

The following example describes all event subscriptions.

```
aws rds describe-event-subscriptions
```

The following example describes the `myfirsteventsubscription`.

```
aws rds describe-event-subscriptions --subscription-name myfirsteventsubscription
```

API

To list your current Amazon RDS event notification subscriptions, call the Amazon RDS API `DescribeEventSubscriptions` action.

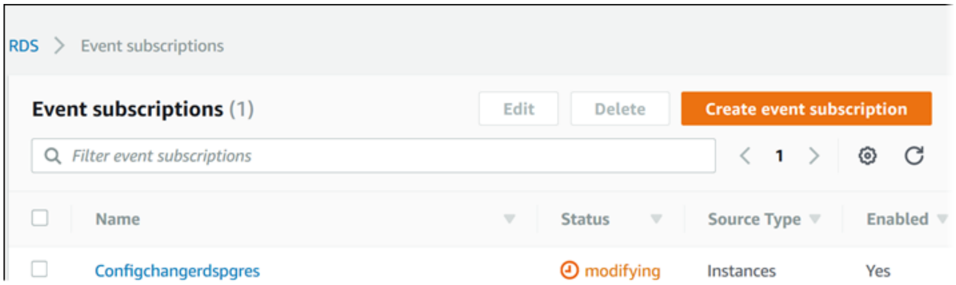
Modifying an Amazon RDS event notification subscription

After you have created a subscription, you can change the subscription name, source identifier, categories, or topic ARN.

Console

To modify an Amazon RDS event notification subscription

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Event subscriptions**.
3. In the **Event subscriptions** pane, choose the subscription that you want to modify and choose **Edit**.
4. Make your changes to the subscription in either the **Target** or **Source** section.
5. Choose **Edit**. The Amazon RDS console indicates that the subscription is being modified.


RDS > Event subscriptions
Event subscriptions (1)
Edit
Delete
Create event subscription
Filter event subscriptions
< 1 >

□	Name	Status	Source Type	Enabled
□	Configchangerdspgres	modifying	Instances	Yes

AWS CLI

To modify an Amazon RDS event notification subscription, use the AWS CLI `modify-event-subscription` command. Include the following required parameter:

- `--subscription-name`

Example

The following code enables `myeventsubscription`.

For Linux, macOS, or Unix:

```
aws rds modify-event-subscription \
    --subscription-name myeventsubscription \
    --enabled
```

For Windows:

```
aws rds modify-event-subscription ^
    --subscription-name myeventsubscription ^
    --enabled
```

API

To modify an Amazon RDS event, call the Amazon RDS API operation `ModifyEventSubscription`. Include the following required parameter:

- `SubscriptionName`

Adding a source identifier to an Amazon RDS event notification subscription

You can add a source identifier (the Amazon RDS source generating the event) to an existing subscription.

Console

You can easily add or remove source identifiers using the Amazon RDS console by selecting or deselecting them when modifying a subscription. For more information, see Modifying an Amazon RDS event notification subscription.

AWS CLI

To add a source identifier to an Amazon RDS event notification subscription, use the AWS CLI `add-source-identifier-to-subscription` command. Include the following required parameters:

- `--subscription-name`
- `--source-identifier`

Example

The following example adds the source identifier `mysqldb` to the `myrdseventsubscription` subscription.

For Linux, macOS, or Unix:

```
aws rds add-source-identifier-to-subscription \
    --subscription-name myrdseventsubscription \
    --source-identifier mysqldb
```

For Windows:

```
aws rds add-source-identifier-to-subscription ^
    --subscription-name myrdseventsubscription ^
    --source-identifier mysqldb
```

API

To add a source identifier to an Amazon RDS event notification subscription, call the Amazon RDS API `AddSourceIdentifierToSubscription`. Include the following required parameters:

- `SubscriptionName`
- `SourceIdentifier`

Removing a source identifier from an Amazon RDS event notification subscription

You can remove a source identifier (the Amazon RDS source generating the event) from a subscription if you no longer want to be notified of events for that source.

Console

You can easily add or remove source identifiers using the Amazon RDS console by selecting or deselecting them when modifying a subscription. For more information, see Modifying an Amazon RDS event notification subscription.

AWS CLI

To remove a source identifier from an Amazon RDS event notification subscription, use the AWS CLI `remove-source-identifier-from-subscription` command. Include the following required parameters:

- `--subscription-name`
- `--source-identifier`

Example

The following example removes the source identifier `mysqldb` from the `myrdseventsubscription` subscription.

For Linux, macOS, or Unix:

```
aws rds remove-source-identifier-from-subscription \
    --subscription-name myrdseventsubscription \
    --source-identifier mysqldb
```

For Windows:

```
aws rds remove-source-identifier-from-subscription ^
    --subscription-name myrdseventsubscription ^
    --source-identifier mysqldb
```

API

To remove a source identifier from an Amazon RDS event notification subscription, use the Amazon RDS API `RemoveSourceIdentifierFromSubscription` command. Include the following required parameters:

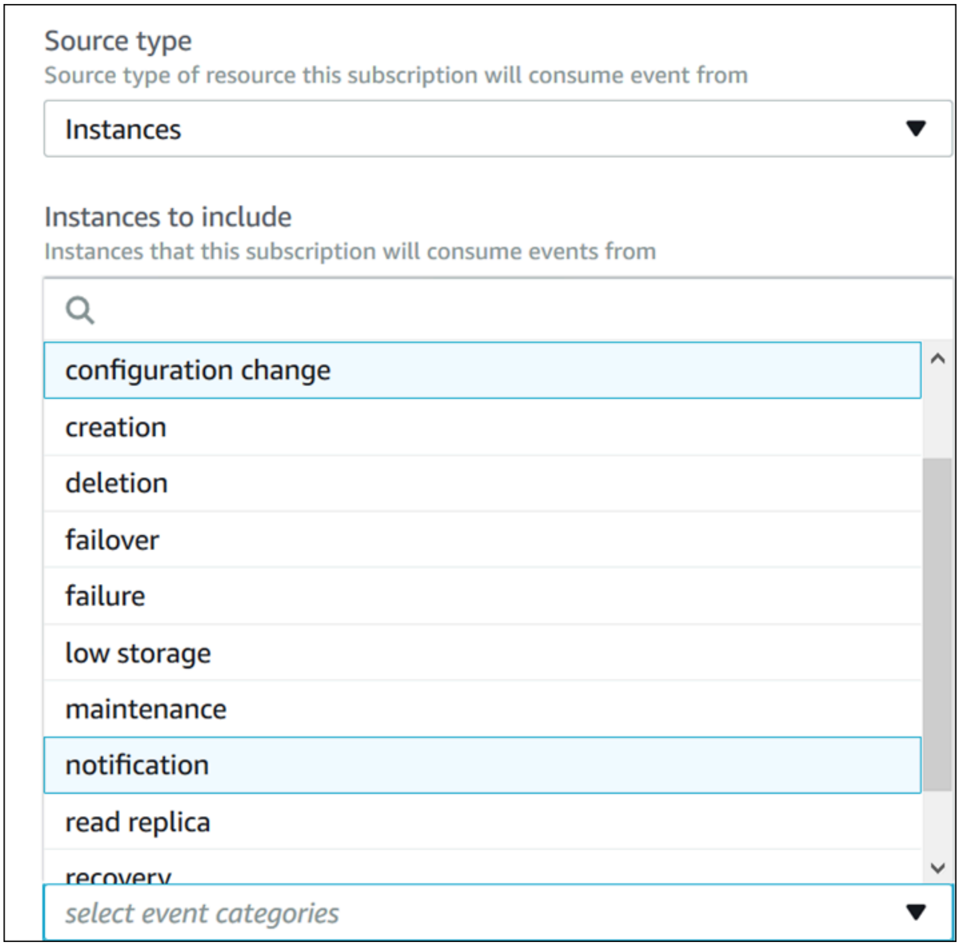
- `SubscriptionName`
- `SourceIdentifier`

Listing the Amazon RDS event notification categories

All events for a resource type are grouped into categories. To view the list of categories available, use the following procedures.

Console

When you create or modify an event notification subscription, the event categories are displayed in the Amazon RDS console. For more information, see Modifying an Amazon RDS event notification subscription.



Source type
Source type of resource this subscription will consume event from
Instances

Instances to include
Instances that this subscription will consume events from

AWS CLI

To list the Amazon RDS event notification categories, use the AWS CLI `describe-event-categories` command. This command has no required parameters.

Example

```
aws rds describe-event-categories
```

API

To list the Amazon RDS event notification categories, use the Amazon RDS API `DescribeEventCategories` command. This command has no required parameters.

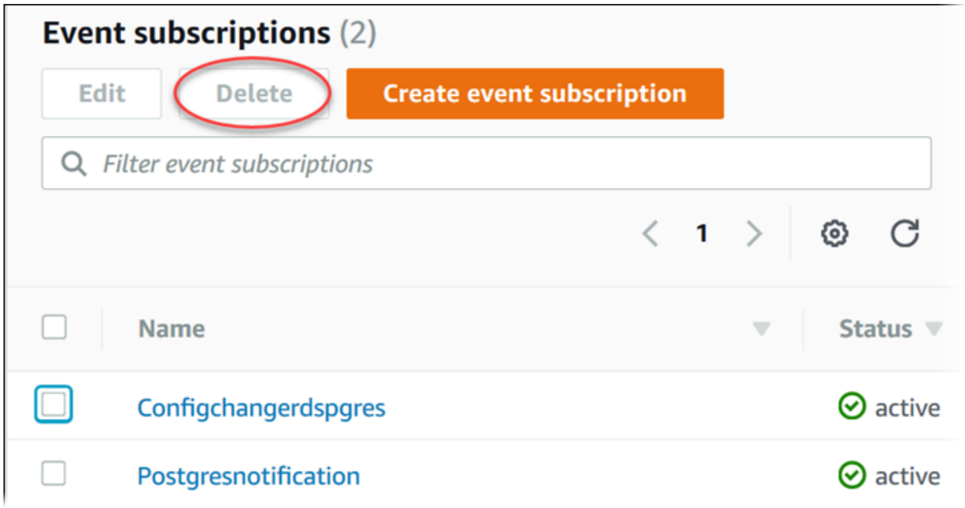
Deleting an Amazon RDS event notification subscription

You can delete a subscription when you no longer need it. All subscribers to the topic will no longer receive event notifications specified by the subscription.

Console

To delete an Amazon RDS event notification subscription

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **DB Event Subscriptions**.
3. In the **My DB Event Subscriptions** pane, choose the subscription that you want to delete.
4. Choose **Delete**.
5. The Amazon RDS console indicates that the subscription is being deleted.



Event subscriptions (2)

□	Name	Status
□	Configchangerdspgres	✓ active
□	Postgresnotification	✓ active

AWS CLI

To delete an Amazon RDS event notification subscription, use the AWS CLI [`delete-event-subscription`]() command. Include the following required parameter:

- `--subscription-name`

Example

The following example deletes the subscription `myrdssubscription`.

```
aws rds delete-event-subscription --subscription-name myrdssubscription
```

API

To delete an Amazon RDS event notification subscription, use the RDS API `DeleteEventSubscription` command. Include the following required parameter:

- `SubscriptionName`

Creating a rule that triggers on an Amazon Aurora event

Using Amazon EventBridge, you can automate AWS services and respond to system events such as application availability issues or resource changes.

Topics

- Tutorial: Log DB instance state changes using Amazon EventBridge

Tutorial: Log DB instance state changes using Amazon EventBridge

In this tutorial, you create an AWS Lambda function that logs the state changes for an instance. You then create a rule that runs the function whenever there is a state change of an existing RDS DB instance. The tutorial assumes that you have a small running test instance that you can shut down temporarily.

> ⚠ **Important**
>
> Don't perform this tutorial on a running production DB instance.

Topics

- Step 1: Create an AWS Lambda function
- Step 2: Create a rule
- Step 3: Test the rule

Step 1: Create an AWS Lambda function

Create a Lambda function to log the state change events. You specify this function when you create your rule.

To create a Lambda function

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. If you're new to Lambda, you see a welcome page. Choose **Get Started Now**. Otherwise, choose **Create function**.
3. Choose **Author from scratch**.
4. On the **Create function** page, do the following:

   a. Enter a name and description for the Lambda function. For example, name the function `RDSInstanceStateChange`.

   b. In **Runtime**, select **Node.js 16x**.

   c. For **Architecture**, choose **x86_64**.

   d. For **Execution role**, do either of the following:

      - Choose **Create a new role with basic Lambda permissions**.
      - For **Existing role**, choose **Use an existing role**. Choose the role that you want to use.

   e. Choose **Create function**.

5. On the **RDSInstanceStateChange** page, do the following:

   a. In **Code source**, select **index.js**.

   b. In the **index.js** pane, delete the existing code.

   c. Enter the following code:

```
console.log('Loading function');

exports.handler = async (event, context) => {
    console.log('Received event:', JSON.stringify(event));
};
```

   d. Choose **Deploy**.

Step 2: Create a rule

Create a rule to run your Lambda function whenever you launch an Amazon RDS instance.

To create the EventBridge rule

1. Open the Amazon EventBridge console at https://console.aws.amazon.com/events/.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule. For example, enter `RDSInstanceStateChangeRule`.
5. Choose **Rule with an event pattern**, and then choose **Next**.
6. For **Event source**, choose **AWS events or EventBridge partner events**.

7. Scroll down to the **Event pattern** section.
8. For **Event source**, choose **AWS services**.
9. For **AWS service**, choose **Relational Database Service (RDS)**.
10. For **Event type**, choose **RDS DB Instance Event**.
11. Leave the default event pattern. Then choose **Next**.
12. For **Target types**, choose **AWS service**.
13. For **Select a target**, choose **Lambda function**.
14. For **Function**, choose the Lambda function that you created. Then choose **Next**.
15. In **Configure tags**, choose **Next**.
16. Review the steps in your rule. Then choose **Create rule**.

Step 3: Test the rule

To test your rule, shut down an RDS DB instance. After waiting a few minutes for the instance to shut down, verify that your Lambda function was invoked.

To test your rule by stopping a DB instance

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Stop an RDS DB instance.
3. Open the Amazon EventBridge console at https://console.aws.amazon.com/events/.
4. In the navigation pane, choose **Rules**, choose the name of the rule that you created.
5. In **Rule details**, choose **Monitoring**.

   You are redirected to the Amazon CloudWatch console. If you are not redirected, click **View the metrics in CloudWatch**.
6. In **All metrics**, choose the name of the rule that you created.

   The graph should indicate that the rule was invoked.
7. In the navigation pane, choose **Log groups**.
8. Choose the name of the log group for your Lambda function (**/aws/lambda/*function-name***).
9. Choose the name of the log stream to view the data provided by the function for the instance that you launched. You should see a received event similar to the following:

```
{
    "version": "0",
    "id": "12a345b6-78c9-01d2-34e5-123f4ghi5j6k",
    "detail-type": "RDS DB Instance Event",
    "source": "aws.rds",
    "account": "111111111111",
    "time": "2021-03-19T19:34:09Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:rds:us-east-1:111111111111:db:testdb"
    ],
    "detail": {
        "EventCategories": [
            "notification"
        ],
        "SourceType": "DB_INSTANCE",
        "SourceArn": "arn:aws:rds:us-east-1:111111111111:db:testdb",
        "Date": "2021-03-19T19:34:09.293Z",
        "Message": "DB instance stopped",
        "SourceIdentifier": "testdb",
        "EventID": "RDS-EVENT-0087"
    }
}
```

For more examples of RDS events in JSON format, see Overview of events for Aurora.

10. (Optional) When you're finished, you can open the Amazon RDS console and start the instance that you stopped.

Amazon RDS event categories and event messages for Aurora

Amazon RDS generates a significant number of events in categories that you can subscribe to using the Amazon RDS Console, AWS CLI, or the API.

Topics

DB cluster events

The following table shows the event category and a list of events when a DB cluster is the source type.

> **Note**
>
> No event category exists for Aurora Serverless in the DB cluster event type. The Aurora Serverless events range from RDS-EVENT-0141 to RDS-EVENT-0149.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0016	Reset master credentials.	
configuration change	RDS-EVENT-0179	Database Activity Streams is started on your database cluster.	For more information see Monitoring Amazon Aurora with Database Activity Streams.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0180	Database Activity Streams is stopped on your database cluster.	For more information see Monitoring Amazon Aurora with Database Activity Streams.
creation	RDS-EVENT-0170	DB cluster created.	
deletion	RDS-EVENT-0171	DB cluster deleted.	
failover	RDS-EVENT-0069	Cluster failover failed, check the health of your cluster instances and try again.	
failover	RDS-EVENT-0070	Promoting previous primary again: *name*.	
failover	RDS-EVENT-0071	Completed failover to DB instance: *name*.	
failover	RDS-EVENT-0072	Started same AZ failover to DB instance: *name*.	
failover	RDS-EVENT-0073	Started cross AZ failover to DB instance: *name*.	

Category	RDS event ID	Message	Notes
failure	RDS-EVENT-0083	Amazon RDS has been unable to create credentials to access your Amazon S3 Bucket for your DB cluster *name*. This is due to the S3 snapshot ingestion IAM role not being configured correctly in your account or the specified Amazon S3 bucket cannot be found. Please refer to the troubleshooting section in the Amazon RDS documentation for further details.	For more information, see Physical migration from MySQL by using Percona XtraBackup and Amazon S3 .
failure	RDS-EVENT-0143	The DB cluster failed to scale from *units* to *units* for this reason: *reason*.	Scaling failed for the Aurora Serverless DB cluster.
failure	RDS-EVENT-0354	You can't create the DB cluster because of incompatible resources. *message*.	The *message* includes details about the failure.
failure	RDS-EVENT-0355	The DB cluster can't be created because of insufficient resource limits. *message*.	The *message* includes details about the failure.

Category	RDS event ID	Message	Notes
global failover	RDS-EVENT-0181	Global switchover to DB cluster *name* in Region *name* started.	This event is for a switchover operation (previously called "managed planned failover"). The process can be delayed because other operation s are running on the DB cluster.
global failover	RDS-EVENT-0182	Old primary DB cluster *name* in Region *name* successfully shut down.	This event is for a switchover operation (previously called "managed planned failover"). The old primary instance in the global database isn't accepting writes. All volumes are synchronized.
global failover	RDS-EVENT-0183	Waiting for data synchroni zation across global cluster members. Current lags behind primary DB cluster: *reason*.	This event is for a switchover operation (previously called "managed planned failover"). A replication lag is occurring during the synchronization phase of the global database failover.

Category	RDS event ID	Message	Notes
global failover	RDS-EVENT-0184	New primary DB cluster *name* in Region *name* was successfully promoted.	This event is for a switchover operation (previously called "managed planned failover"). The volume topology of the global database is reestablished with the new primary volume.
global failover	RDS-EVENT-0185	Global switchover to DB cluster *name* in Region *name* finished.	This event is for a switchover operation (previously called "managed planned failover"). The global database switchover is finished on the primary DB cluster. Replicas might take long to come online after the failover completes.
global failover	RDS-EVENT-0186	Global switchover to DB cluster *name* in Region *name* is cancelled.	This event is for a switchover operation (previously called "managed planned failover").
global failover	RDS-EVENT-0187	Global switchover to DB cluster *name* in Region *name* failed.	This event is for a switchover operation (previously called "managed planned failover").

Category	RDS event ID	Message	Notes
global failover	RDS-EVENT-0238	Global failover to DB cluster *name* in Region *name* completed.	
global failover	RDS-EVENT-0239	Global failover to DB cluster *name* in Region *name* failed.	
global failover	RDS-EVENT-0240	Started resynchronizing members of DB cluster *name* in Region *name* after global failover.	
global failover	RDS-EVENT-0241	Finished resynchronizing members of DB cluster *name* in Region *name* after global failover.	
maintenance	RDS-EVENT-0156	The DB cluster has a DB engine minor version upgrade available.	
maintenance	RDS-EVENT-0173	Database cluster engine version has been upgraded.	Patching of the DB cluster has completed.
maintenance	RDS-EVENT-0176	Database cluster engine major version has been upgraded.	
maintenance	RDS-EVENT-0286	Database cluster engine version upgrade started.	
maintenance	RDS-EVENT-0287	Operating system upgrade requirement detected.	

Category	RDS event ID	Message	Notes
maintenance	RDS-EVENT-0288	Cluster operating system upgrade starting.	
maintenance	RDS-EVENT-0289	Cluster operating system upgrade completed.	
maintenance	RDS-EVENT-0363	Upgrade preparation in progress: *cluster_name*	Upgrade prechecks have started for the DB cluster.
notification	RDS-EVENT-0076	Failed to migrate from *name* to *name*. Reason: *reason*.	Migration to an Aurora DB cluster failed.
notification	RDS-EVENT-0077	Failed to convert *name*.*name* to InnoDB. Reason: *reason*.	An attempt to convert a table from the source database to InnoDB failed during the migration to an Aurora DB cluster.
notification	RDS-EVENT-0085	Unable to upgrade DB cluster *name* because the instance *name* has a status of *name*. Resolve the issue or delete the instance and try again.	An error occurred while attempting to patch the Aurora DB cluster. Check your instance status, resolve the issue, and try again. For more informati on see Maintaining an Amazon Aurora DB cluster.
notification	RDS-EVENT-0141	Scaling DB cluster from *units* to *units* for this reason: *reason*.	Scaling initiated for the Aurora Serverless DB cluster.
notification	RDS-EVENT-0142	The DB cluster has scaled from *units* to *units*.	Scaling completed for the Aurora Serverless DB cluster.

Category	RDS event ID	Message	Notes
notification	RDS-EVENT-0144	The DB cluster is being paused.	An automatic pause was initiated for the Aurora Serverless DB cluster.
notification	RDS-EVENT-0145	The DB cluster is paused.	The Aurora Serverless DB cluster has been paused.
notification	RDS-EVENT-0146	Pause was canceled for the DB cluster.	The pause was canceled for the Aurora Serverless DB cluster.
notification	RDS-EVENT-0147	The DB cluster is being resumed.	A resume operation was initiated for the Aurora Serverless DB cluster.
notification	RDS-EVENT-0148	The DB cluster is resumed.	The resume operation completed for the Aurora Serverless DB cluster.
notification	RDS-EVENT-0149	The DB cluster has scaled from *units* to *units*, but scaling wasn't seamless for this reason: *reason*.	Seamless scaling completed with the force option for the Aurora Serverless DB cluster. Connections might have been interrupted as required.
notification	RDS-EVENT-0150	DB cluster stopped.	
notification	RDS-EVENT-0151	DB cluster started.	
notification	RDS-EVENT-0152	DB cluster stop failed.	
notification	RDS-EVENT-0153	DB cluster is being started due to it exceeding the maximum allowed time being stopped.	

Category	RDS event ID	Message	Notes
notification	RDS-EVENT-0172	Renamed cluster from *name* to *name*.	
notification	RDS-EVENT-0234	Export task failed.	The DB cluster export task failed.
notification	RDS-EVENT-0235	Export task canceled.	The DB cluster export task was canceled.
notification	RDS-EVENT-0236	Export task completed.	The DB cluster export task completed.

DB instance events

The following table shows the event category and a list of events when a DB instance is the source type.

Category	RDS event ID	Message	Notes
availability	RDS-EVENT-0004	DB instance shutdown.	
availability	RDS-EVENT-0006	DB instance restarted.	
availability	RDS-EVENT-0022	Error restarting mysql: *message*.	An error has occurred while restarting Aurora MySQL or RDS for MariaDB.
backtrack	RDS-EVENT-0131	The actual Backtrack window is smaller than the target Backtrack window you specified. Consider reducing the number of hours in your target Backtrack window.	For more information about backtracking, see Backtracking an Aurora DB cluster.

Category	RDS event ID	Message	Notes
backtrack	RDS-EVENT-0132	The actual Backtrack window is the same as the target Backtrack window.	
configuration change	RDS-EVENT-0011	Updated to use DBParameterGroup *name*.	
configuration change	RDS-EVENT-0012	Applying modification to database instance class.	
configuration change	RDS-EVENT-0014	Finished applying modification to DB instance class.	
configuration change	RDS-EVENT-0017	Finished applying modification to allocated storage.	
configuration change	RDS-EVENT-0025	Finished applying modification to convert to a Multi-AZ DB instance.	
configuration change	RDS-EVENT-0029	Finished applying modification to convert to a standard (Single-AZ) DB instance.	
configuration change	RDS-EVENT-0033	There are *number* users matching the master username; only resetting the one not tied to a specific host.	
configuration change	RDS-EVENT-0067	Unable to reset your password. Error information: *message*.	

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0078	Monitoring Interval changed to *number*.	The Enhanced Monitoring configuration has been changed.
configuration change	RDS-EVENT-0092	Finished updating DB parameter group.	
creation	RDS-EVENT-0005	DB instance created.	
deletion	RDS-EVENT-0003	DB instance deleted.	
failure	RDS-EVENT-0035	Database instance put into *state*. *message*.	The DB instance has invalid parameters. For example, if the DB instance could not start because a memory-related parameter is set too high for this instance class, your action would be to modify the memory parameter and reboot the DB instance.
failure	RDS-EVENT-0036	Database instance in *state*. *message*.	The DB instance is in an incompatible network. Some of the specified subnet IDs are invalid or do not exist.

Category	RDS event ID	Message	Notes
failure	RDS-EVENT-0079	Amazon RDS has been unable to create credentials for enhanced monitoring and this feature has been disabled. This is likely due to the rds-monitoring-role not being present and configured correctly in your account. Please refer to the troubleshooting section in the Amazon RDS documentation for further details.	Enhanced Monitoring can't be enabled without the Enhanced Monitoring IAM role. For information about creating the IAM role, see To create an IAM role for Amazon RDS enhanced monitoring.
failure	RDS-EVENT-0080	Amazon RDS has been unable to configure enhanced monitoring on your instance: *name* and this feature has been disabled. This is likely due to the rds-monitoring-role not being present and configured correctly in your account. Please refer to the troubleshooting section in the Amazon RDS documentation for further details.	Enhanced Monitoring was disabled because an error occurred during the configuration change. It is likely that the Enhanced Monitoring IAM role is configured incorrectly. For information about creating the enhanced monitoring IAM role, see To create an IAM role for Amazon RDS enhanced monitoring.

Category	RDS event ID	Message	Notes
failure	RDS-EVENT-0082	Amazon RDS has been unable to create credentials to access your Amazon S3 Bucket for your DB instance *name*. This is due to the S3 snapshot ingestion IAM role not being configured correctly in your account or the specified Amazon S3 bucket cannot be found. Please refer to the troubleshooting section in the Amazon RDS documentation for further details.	Aurora was unable to copy backup data from an Amazon S3 bucket. It is likely that the permissions for Aurora to access the Amazon S3 bucket are configured incorrectly. For more information, see Physical migration from MySQL by using Percona XtraBackup and Amazon S3 .
failure	RDS-EVENT-0254	Underlying storage quota for this customer account has exceeded the limit. Please increase the allowed storage quota to let the scaling go through on the instance.	
failure	RDS-EVENT-0353	The DB instance can't be created because of insufficient resource limits. *message*.	The *message* includes details about the failure.

Category	RDS event ID	Message	Notes
low storage	RDS-EVENT-0007	Allocated storage has been exhausted. Allocate additional storage to resolve.	The allocated storage for the DB instance has been consumed. To resolve this issue, allocate additional storage for the DB instance. For more information, see the RDS FAQ. You can monitor the storage space for a DB instance using the **Free Storage Space** metric.
low storage	RDS-EVENT-0089	The free storage capacity for DB instance: *name* is low at *percentage* of the provisioned storage [Provisioned Storage: *size*, Free Storage: *size*]. You may want to increase the provisioned storage to address this issue.	The DB instance has consumed more than 90% of its allocated storage. You can monitor the storage space for a DB instance using the **Free Storage Space** metric.
low storage	RDS-EVENT-0227	Your Aurora cluster's storage is dangerously low with only *amount* terabytes remaining. Please take measures to reduce the storage load on your cluster.	The Aurora storage subsystem is running low on space.
maintenance	RDS-EVENT-0026	Applying off-line patches to DB instance.	Offline maintenance of the DB instance is taking place. The DB instance is currently unavailable.

Category	RDS event ID	Message	Notes
maintenance	RDS-EVENT-0027	Finished applying off-line patches to DB instance.	Offline maintenance of the DB instance is complete. The DB instance is now available.
maintenance	RDS-EVENT-0047	Database instance patched.	
maintenance	RDS-EVENT-0155	The DB instance has a DB engine minor version upgrade available.	
notification	RDS-EVENT-0044	*message*	This is an operator-issued notification. For more information, see the event message.
notification	RDS-EVENT-0048	Delaying database engine upgrade since this instance has read replicas that need to be upgraded first.	Patching of the DB instance has been delayed.
notification	RDS-EVENT-0087	DB instance stopped.	
notification	RDS-EVENT-0088	DB instance started.	
read replica	RDS-EVENT-0045	Replication has stopped.	Replication on your DB instance has been stopped due to insufficient storage. Scale storage or reduce the maximum size of your redo logs to let replication continue. To accommodate redo logs of size *amount* MiB you need at least *amount* MiB free storage.

Category	RDS event ID	Message	Notes
read replica	RDS-EVENT-0046	Replication for the Read Replica resumed.	This message appears when you first create a read replica, or as a monitoring message confirming that replication is functioning properly. If this message follows an `RDS-EVENT-0045` notification, then replication has resumed following an error or after replication was stopped.
read replica	RDS-EVENT-0057	Replication streaming has been terminated.	
recovery	RDS-EVENT-0020	Recovery of the DB instance has started. Recovery time will vary with the amount of data to be recovered.	
recovery	RDS-EVENT-0021	Recovery of the DB instance is complete.	
recovery	RDS-EVENT-0023	Emergent Snapshot Request: *message*.	A manual backup has been requested but Amazon RDS is currently in the process of creating a DB snapshot. Submit the request again after Amazon RDS has completed the DB snapshot.

Category	RDS event ID	Message	Notes
recovery	RDS-EVENT-0052	Multi-AZ instance recovery started.	Recovery time will vary with the amount of data to be recovered.
recovery	RDS-EVENT-0053	Multi-AZ instance recovery completed. Pending failover or activation.	
recovery	RDS-EVENT-0361	Recovery of standby DB instance has started.	The standby DB instance is rebuilt during the recovery process. Database performance is impacted during the recovery process.
recovery	RDS-EVENT-0362	Recovery of standby DB instance has completed.	The standby DB instance is rebuilt during the recovery process. Database performance is impacted during the recovery process.
restoration	RDS-EVENT-0019	Restored from DB instance *name* to *name*.	The DB instance has been restored from a point-in-time backup.
security patching	RDS-EVENT-0230	A system update is available for your DB instance. For information about applying updates, see 'Maintaining a DB instance' in the RDS User Guide.	A new Operating System patch is available. A new, minor version, operating system update is available for your DB instance. For informati on about applying updates, see Working with operating system updates.

DB parameter group events

The following table shows the event category and a list of events when a DB parameter group is the source type.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0037	Updated parameter *name* to *value* with apply method *method*.	

DB security group events

The following table shows the event category and a list of events when a DB security group is the source type.

> **Note**
>
> DB security groups are resources for EC2-Classic. EC2-Classic was retired on August 15, 2022. If you haven't migrated from EC2-Classic to a VPC, we recommend that you migrate as soon as possible. For more information, see Migrate from EC2-Classic to a VPC in the *Amazon EC2 User Guide* and the blog EC2-Classic Networking is Retiring – Here's How to Prepare.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0038	Applied change to security group.	
failure	RDS-EVENT-0039	Revoking authorization as *user*.	The security group owned by *user* doesn't exist. The authorization for the security group has been revoked because it is invalid.

DB cluster snapshot events

The following table shows the event category and a list of events when a DB cluster snapshot is the source type.

Category	RDS event ID	Message	Notes
backup	RDS-EVENT-0074	Creating manual cluster snapshot.	
backup	RDS-EVENT-0075	Manual cluster snapshot created.	
notification	RDS-EVENT-0162	The cluster snapshot export task failed.	
notification	RDS-EVENT-0163	The cluster snapshot export task was canceled.	
notification	RDS-EVENT-0164	The cluster snapshot export task completed.	
backup	RDS-EVENT-0168	Creating automated cluster snapshot.	
backup	RDS-EVENT-0169	Automated cluster snapshot created.	

RDS Proxy events

The following table shows the event category and a list of events when an RDS Proxy is the source type.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0204	RDS modified DB proxy *name*.	

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0207	RDS modified the end point of the DB proxy *name*.	
configuration change	RDS-EVENT-0213	RDS detected the addition of the DB instance and automatically added it to the target group of the DB proxy *name*.	
configuration change	RDS-EVENT-0213	RDS detected creation of DB instance *name* and automatically added it to target group *name* of DB proxy *name*.	
configuration change	RDS-EVENT-0214	RDS detected deletion of DB instance *name* and automatically removed it from target group *name* of DB proxy *name*.	
configuration change	RDS-EVENT-0215	RDS detected deletion of DB cluster *name* and automatically removed it from target group *name* of DB proxy *name*.	
creation	RDS-EVENT-0203	RDS created DB proxy *name*.	
creation	RDS-EVENT-0206	RDS created endpoint *name* for DB proxy *name*.	
deletion	RDS-EVENT-0205	RDS deleted DB proxy *name*.	

Category	RDS event ID	Message	Notes
deletion	RDS-EVENT-0208	RDS deleted endpoint *name* for DB proxy *name*.	
failure	RDS-EVENT-0243	RDS failed to provision capacity for proxy *name* because there aren't enough IP addresses available in your subnets: *name*. To fix the issue, make sure that your subnets have the minimum number of unused IP addresses as recommend ed in the RDS Proxy documentation.	To determine the recommended number for your instance class, see Planning for IP address capacity.
failure	RDS-EVENT-0275	RDS throttled some connections to DB proxy *name*. The number of simultaneous connection requests from the client to the proxy has exceeded the limit.	

Blue/green deployment events

The following table shows the event category and a list of events when a blue/green deployment is the source type.

For more information about blue/green deployments, see Using Amazon RDS Blue/Green Deployments for database updates.

Category	Amazon RDS event ID	Message	Notes
creation	RDS-EVENT-0244	Blue/green deployment tasks completed. You can make more modifications to the green environment databases or switch over the deployment.	
failure	RDS-EVENT-0245	Creation of blue/green deployment failed because the (source/target) DB (instance/cluster) wasn't found.	
deletion	RDS-EVENT-0246	Blue/green deployment deleted.	
notification	RDS-EVENT-0247	Switchover from *`blue`* to *`green`* started.	
notification	RDS-EVENT-0248	Switchover completed on blue/green deployment.	
failure	RDS-EVENT-0249	Switchover canceled on blue/green deployment.	
notification	RDS-EVENT-0259	Switchover from DB cluster *`blue`* to *`green`* started.	
notification	RDS-EVENT-0260	Switchover from DB cluster *`blue`* to *`green`* completed. Renamed *`blue`* to *`blue-old`* and *`green`* to *`blue`*.	

Category	Amazon RDS event ID	Message	Notes
failure	RDS-EVENT-0261	Switchover from DB cluster *blue* to *green* was canceled due to *reason*.	
notification	RDS-EVENT-0311	Sequence sync for switchover of DB cluster *blue* to *green* has initiated. Switchover when using sequences may lead to extended downtime.	
notification	RDS-EVENT-0312	Sequence sync for switchover of DB cluster *blue* to *green* has completed.	
failure	RDS-EVENT-0314	Sequence sync for switchover of DB cluster *blue* to *green* was cancelled because sequences failed to sync.	

Monitoring Amazon Aurora log files

Every RDS database engine generates logs that you can access for auditing and troubleshooting. The type of logs depends on your database engine.

You can access database logs using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the Amazon RDS API. You can't view, watch, or download transaction logs.

> **Note**
>
> In some cases, logs contain hidden data. Therefore, the AWS Management Console might show content in a log file, but the log file might be empty when you download it.

Topics

Viewing and listing database log files

You can view database log files for your Amazon Aurora DB engine by using the AWS Management Console. You can list what log files are available for download or monitoring by using the AWS CLI or Amazon RDS API.

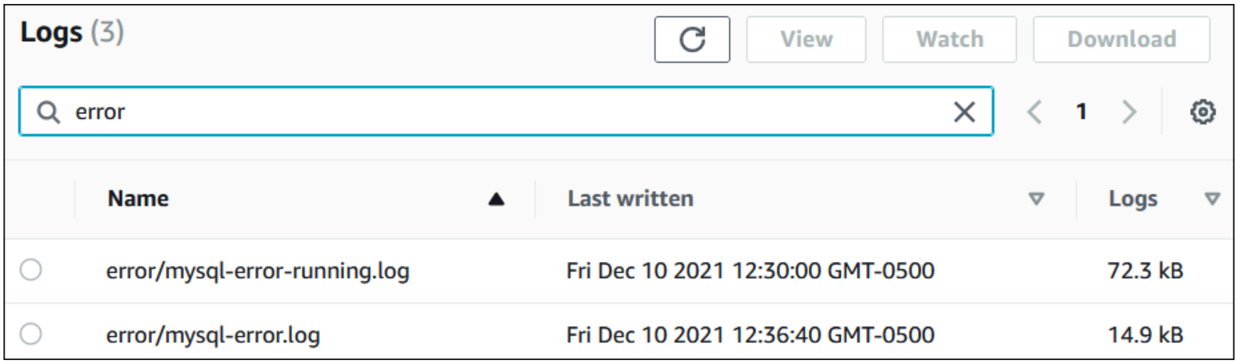
> **Note**
>
> You can't view the log files for Aurora Serverless v1 DB clusters in the RDS console. However, you can view them in the Amazon CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

Console

To view a database log file

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the DB instance that has the log file that you want to view.
4. Choose the **Logs & events** tab.
5. Scroll down to the **Logs** section.
6. (Optional) Enter a search term to filter your results.

   The following example lists logs filtered by the text **`error`**.



Name	Last written	Logs
○ error/mysql-error-running.log	Fri Dec 10 2021 12:30:00 GMT-0500	72.3 kB
○ error/mysql-error.log	Fri Dec 10 2021 12:36:40 GMT-0500	14.9 kB

7. Choose the log that you want to view, and then choose **View**.

AWS CLI

To list the available database log files for a DB instance, use the AWS CLI `describe-db-log-files` command.

The following example returns a list of log files for a DB instance named `my-db-instance`.

Example

```
aws rds describe-db-log-files --db-instance-identifier my-db-instance
```

RDS API

To list the available database log files for a DB instance, use the Amazon RDS API `DescribeDBLogFiles` action.

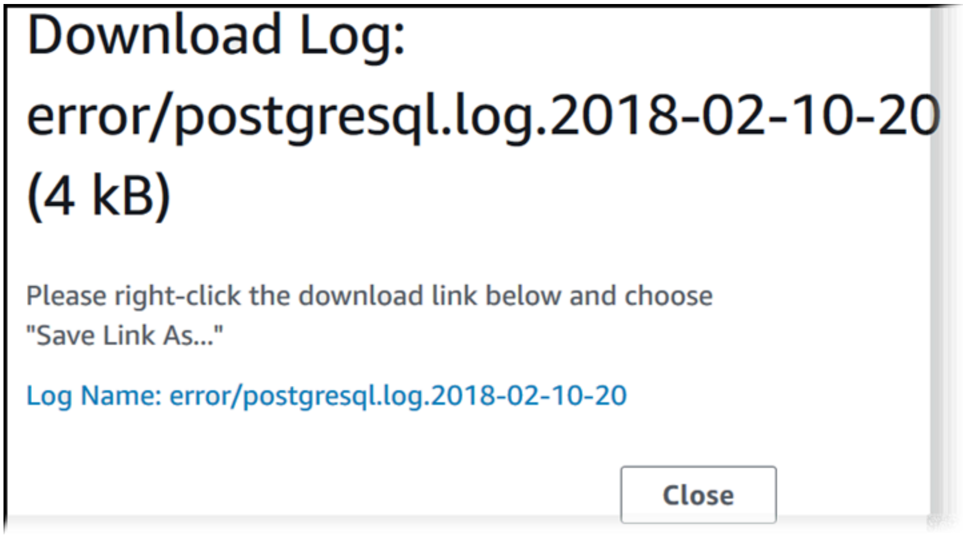
Downloading a database log file

You can use the AWS Management Console, AWS CLI, or API to download a database log file.

Console

To download a database log file

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the DB instance that has the log file that you want to view.
4. Choose the **Logs & events** tab.
5. Scroll down to the **Logs** section.
6. In the **Logs** section, choose the button next to the log that you want to download, and then choose **Download**.
7. Open the context (right-click) menu for the link provided, and then choose **Save Link As**. Enter the location where you want the log file to be saved, and then choose **Save**.



Download Log: error/postgresql.log.2018-02-10-20 (4 kB)

Please right-click the download link below and choose "Save Link As..."

Log Name: error/postgresql.log.2018-02-10-20

AWS CLI

To download a database log file, use the AWS CLI command `download-db-log-file-portion`. By default, this command downloads only the latest portion of a log file. However, you can download an entire file by specifying the parameter `--starting-token 0`.

The following example shows how to download the entire contents of a log file called *log/ERROR.4* and store it in a local file called *errorlog.txt*.

Example

For Linux, macOS, or Unix:

```
aws rds download-db-log-file-portion \
    --db-instance-identifier myexampledb \
    --starting-token 0 --output text \
    --log-file-name log/ERROR.4 > errorlog.txt
```

For Windows:

```
aws rds download-db-log-file-portion ^
    --db-instance-identifier myexampledb ^
    --starting-token 0 --output text ^
    --log-file-name log/ERROR.4 > errorlog.txt
```

RDS API

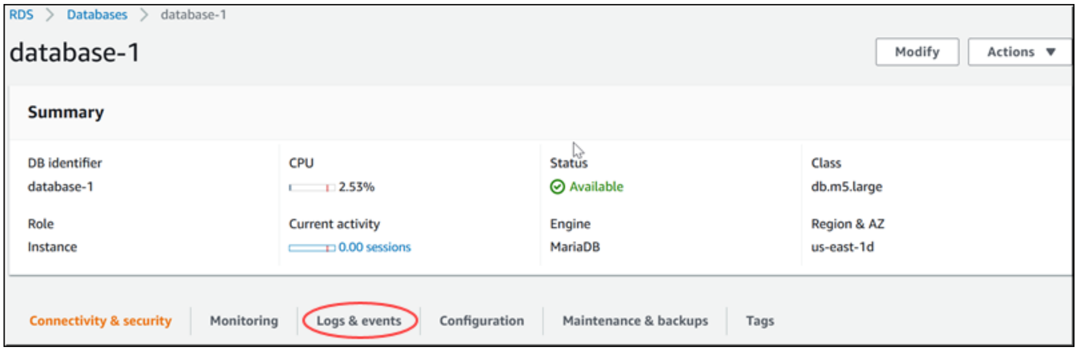
To download a database log file, use the Amazon RDS API DownloadDBLogFilePortion action.

Watching a database log file

Watching a database log file is equivalent to tailing the file on a UNIX or Linux system. You can watch a log file by using the AWS Management Console. RDS refreshes the tail of the log every 5 seconds.

To watch a database log file

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the DB instance that has the log file that you want to view.
4. Choose the **Logs & events** tab.

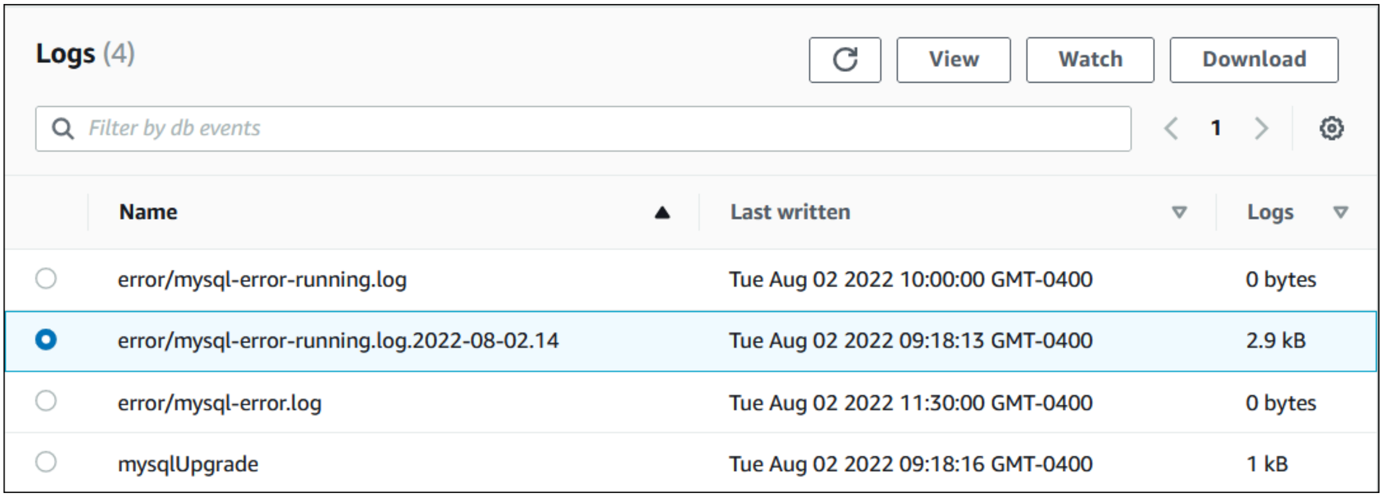


database-1

Summary

DB identifier database-1	CPU 2.53%	Status ✓ Available	Class db.m5.large
Role Instance	Current activity 0.00 sessions	Engine MariaDB	Region & AZ us-east-1d

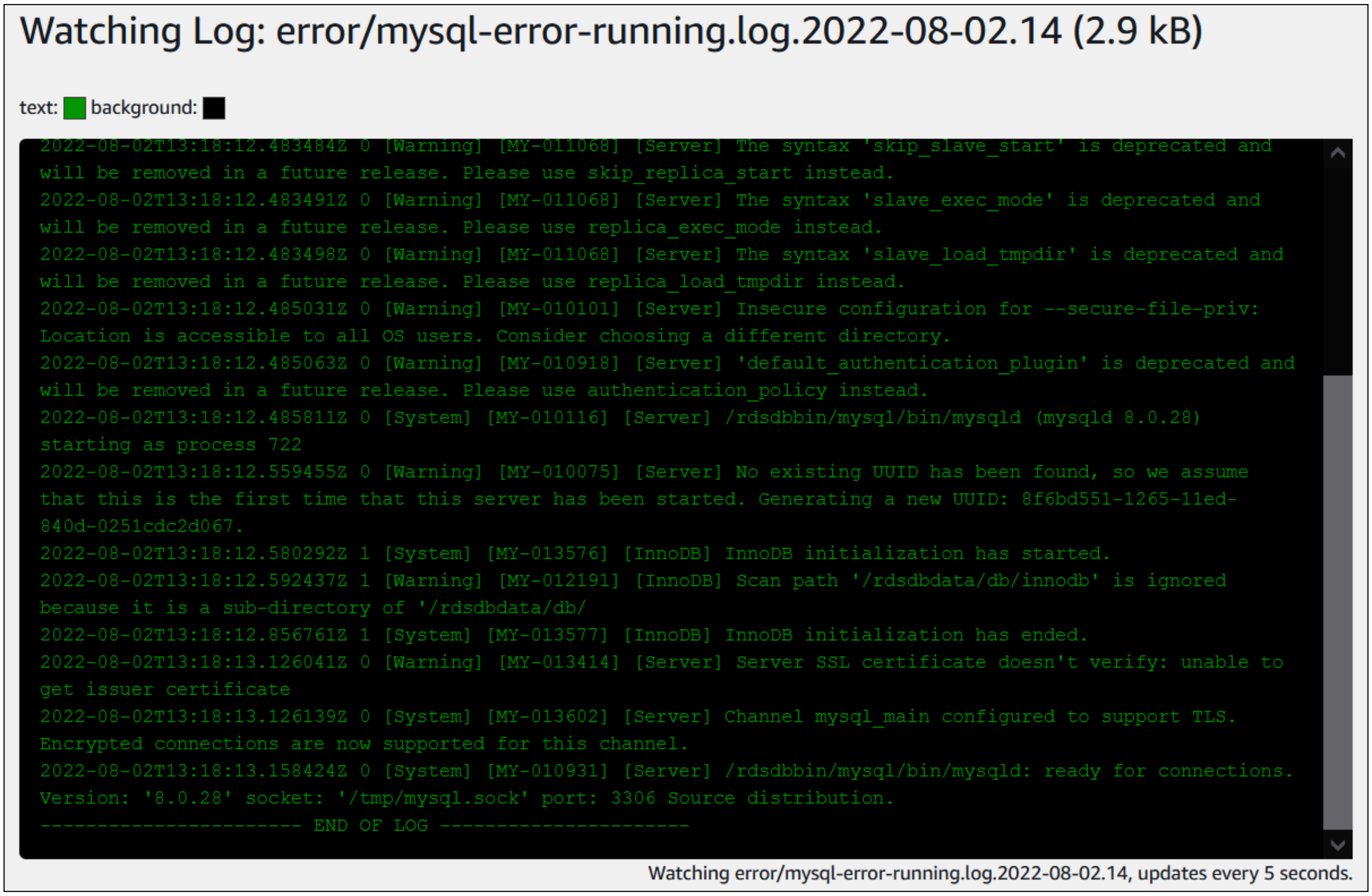
5. In the **Logs** section, choose a log file, and then choose **Watch**.



Logs (4)

	Name	Last written	Logs
○	error/mysql-error-running.log	Tue Aug 02 2022 10:00:00 GMT-0400	0 bytes
●	error/mysql-error-running.log.2022-08-02.14	Tue Aug 02 2022 09:18:13 GMT-0400	2.9 kB
○	error/mysql-error.log	Tue Aug 02 2022 11:30:00 GMT-0400	0 bytes
○	mysqlUpgrade	Tue Aug 02 2022 09:18:16 GMT-0400	1 kB

RDS shows the tail of the log, as in the following MySQL example.



Watching Log: error/mysql-error-running.log.2022-08-02.14 (2.9 kB)

text: 🟩 background: ■

```
2022-08-02T13:18:12.483484Z 0 [Warning] [MY-011068] [Server] The syntax 'skip_slave_start' is deprecated and
will be removed in a future release. Please use skip_replica_start instead.
2022-08-02T13:18:12.483491Z 0 [Warning] [MY-011068] [Server] The syntax 'slave_exec_mode' is deprecated and
will be removed in a future release. Please use replica_exec_mode instead.
2022-08-02T13:18:12.483498Z 0 [Warning] [MY-011068] [Server] The syntax 'slave_load_tmpdir' is deprecated and
will be removed in a future release. Please use replica_load_tmpdir instead.
2022-08-02T13:18:12.485031Z 0 [Warning] [MY-010101] [Server] Insecure configuration for --secure-file-priv:
Location is accessible to all OS users. Consider choosing a different directory.
2022-08-02T13:18:12.485063Z 0 [Warning] [MY-010918] [Server] 'default_authentication_plugin' is deprecated and
will be removed in a future release. Please use authentication_policy instead.
2022-08-02T13:18:12.485811Z 0 [System] [MY-010116] [Server] /rdsdbbin/mysql/bin/mysqld (mysqld 8.0.28)
starting as process 722
2022-08-02T13:18:12.559455Z 0 [Warning] [MY-010075] [Server] No existing UUID has been found, so we assume
that this is the first time that this server has been started. Generating a new UUID: 8f6bd551-1265-11ed-
840d-0251cdc2d067.
2022-08-02T13:18:12.580292Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2022-08-02T13:18:12.592437Z 1 [Warning] [MY-012191] [InnoDB] Scan path '/rdsdbdata/db/innodb' is ignored
because it is a sub-directory of '/rdsdbdata/db/
2022-08-02T13:18:12.856761Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2022-08-02T13:18:13.126041Z 0 [Warning] [MY-013414] [Server] Server SSL certificate doesn't verify: unable to
get issuer certificate
2022-08-02T13:18:13.126139Z 0 [System] [MY-013602] [Server] Channel mysql_main configured to support TLS.
Encrypted connections are now supported for this channel.
2022-08-02T13:18:13.158424Z 0 [System] [MY-010931] [Server] /rdsdbbin/mysql/bin/mysqld: ready for connections.
Version: '8.0.28' socket: '/tmp/mysql.sock' port: 3306 Source distribution.
----------------------- END OF LOG ----------------------
```

Watching error/mysql-error-running.log.2022-08-02.14, updates every 5 seconds.

Publishing database logs to Amazon CloudWatch Logs

In an on-premises database, the database logs reside on the file system. Amazon RDS doesn't provide host access to the database logs on the file system of your DB cluster. For this reason, Amazon RDS lets you export database logs to Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data. You can also store the data in highly durable storage and manage the data with the CloudWatch Logs Agent.

Topics

Overview of RDS integration with CloudWatch Logs

In CloudWatch Logs, a *log stream* is a sequence of log events that share the same source. Each separate source of logs in CloudWatch Logs makes up a separate log stream. A *log group* is a group of log streams that share the same retention, monitoring, and access control settings.

Amazon Aurora continuously streams your DB cluster log records to a log group. For example, you have a log group `/aws/rds/cluster/`*`cluster_name`*`/`*`log_type`* for each type of log that you publish. This log group is in the same AWS Region as the database instance that generates the log.

AWS retains log data published to CloudWatch Logs for an indefinite time period unless you specify a retention period. For more information, see Change log data retention in CloudWatch Logs.

Deciding which logs to publish to CloudWatch Logs

Each RDS database engine supports its own set of logs. To learn about the options for your database engine, review the following topics:

- the section called “Publishing Aurora MySQL logs to CloudWatch Logs”
- the section called “Publishing Aurora PostgreSQL logs to CloudWatch Logs”

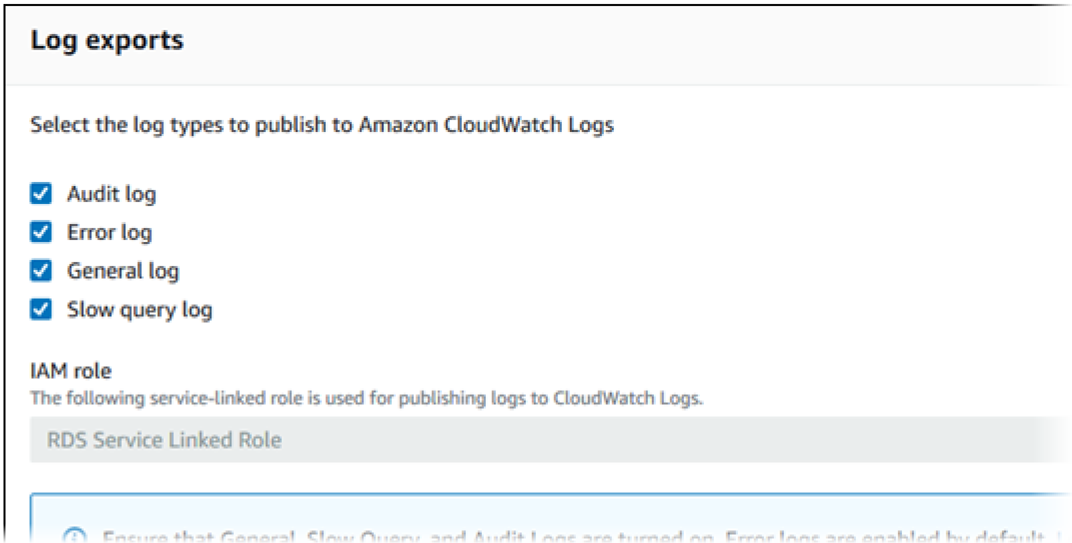
Specifying the logs to publish to CloudWatch Logs

You specify which logs to publish in the console. Make sure that you have a service-linked role in AWS Identity and Access Management (IAM). For more information about service-linked roles, see Using service-linked roles for Amazon Aurora.

To specify the logs to publish

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Do either of the following:
   - Choose **Create database**.
   - Choose a database from the list, and then choose **Modify**.
4. In **Logs exports**, choose which logs to publish.

   The following example specifies the audit log, error logs, general log, and slow query log.



Log exports

Select the log types to publish to Amazon CloudWatch Logs

☑ Audit log
☑ Error log
☑ General log
☑ Slow query log

IAM role
The following service-linked role is used for publishing logs to CloudWatch Logs.
RDS Service Linked Role

Ensure that General, Slow Query, and Audit Logs are turned on. Error logs are enabled by default.

Searching and filtering your logs in CloudWatch Logs

You can search for log entries that meet a specified criteria using the CloudWatch Logs console. You can access the logs either through the RDS console, which leads you to the CloudWatch Logs console, or from the CloudWatch Logs console directly.

To search your RDS logs using the RDS console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose a DB cluster or a DB instance.
4. Choose **Configuration**.
5. Under **Published logs**, choose the database log that you want to view.

To search your RDS logs using the CloudWatch Logs console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Log groups**.
3. In the filter box, enter **`/aws/rds`**.
4. For **Log Groups**, choose the name of the log group containing the log stream to search.
5. For **Log Streams**, choose the name of the log stream to search.
6. Under **Log events**, enter the filter syntax to use.

For more information, see Searching and filtering log data in the *Amazon CloudWatch Logs User Guide*. For a blog tutorial explaining how to monitor RDS logs, see Build proactive database monitoring for Amazon RDS with Amazon CloudWatch Logs, AWS Lambda, and Amazon SNS.

Reading log file contents using REST

Amazon RDS provides a REST endpoint that allows access to DB instance log files. This is useful if you need to write an application to stream Amazon RDS log file contents.

The syntax is:

```
GET /v13/downloadCompleteLogFile/DBInstanceIdentifier/LogFileName HTTP/1.1
Content-type: application/json
host: rds.region.amazonaws.com
```

The following parameters are required:

- *`DBInstanceIdentifier`*—the name of the DB instance that contains the log file you want to download.

- *LogFileName*—the name of the log file to be downloaded.

The response contains the contents of the requested log file, as a stream.

The following example downloads the log file named *log/ERROR.6* for the DB instance named *sample-sql* in the *us-west-2* region.

```
GET /v13/downloadCompleteLogFile/sample-sql/log/ERROR.6 HTTP/1.1
host: rds.us-west-2.amazonaws.com
X-Amz-Security-Token: AQoDYXdzEIH//////////
wEa0AIXLhngC5zp9CyB1R6abwKrXHVR5efnAVN3XvR7IwqKYalFSn6UyJuEFTft9nObglx4QJ+GXV9cpACkETq=
X-Amz-Date: 20140903T233749Z
X-Amz-Algorithm: AWS4-HMAC-SHA256
X-Amz-Credential: AKIADQKE4SARGYLE/20140903/us-west-2/rds/aws4_request
X-Amz-SignedHeaders: host
X-Amz-Content-SHA256: e3b0c44298fc1c229afbf4c8996fb92427ae41e4649b934de495991b7852b855
X-Amz-Expires: 86400
X-Amz-Signature: 353a4f14b3f250142d9afc34f9f9948154d46ce7d4ec091d0cdabbcf8b40c558
```

If you specify a nonexistent DB instance, the response consists of the following error:

- DBInstanceNotFound—*DBInstanceIdentifier* does not refer to an existing DB instance. (HTTP status code: 404)

Aurora MySQL database log files

You can monitor the Aurora MySQL logs directly through the Amazon RDS console, Amazon RDS API, AWS CLI, or AWS SDKs. You can also access MySQL logs by directing the logs to a database table in the main database and querying that table. You can use the mysqlbinlog utility to download a binary log.

For more information about viewing, downloading, and watching file-based database logs, see Monitoring Amazon Aurora log files.

Topics

- Overview of Aurora MySQL database logs
- Publishing Aurora MySQL logs to Amazon CloudWatch Logs
- Managing table-based Aurora MySQL logs
- Configuring Aurora MySQL binary logging
- Accessing MySQL binary logs

Overview of Aurora MySQL database logs

You can monitor the following types of Aurora MySQL log files:

- Error log
- Slow query log
- General log
- Audit log

The Aurora MySQL error log is generated by default. You can generate the slow query and general logs by setting parameters in your DB parameter group.

Topics

- Aurora MySQL error logs
- Aurora MySQL slow query and general logs
- Aurora MySQL audit log
- Log rotation and retention for Aurora MySQL

Aurora MySQL error logs

Aurora MySQL writes errors in the `mysql-error.log` file. Each log file has the hour it was generated (in UTC) appended to its name. The log files also have a timestamp that helps you determine when the log entries were written.

Aurora MySQL writes to the error log only on startup, shutdown, and when it encounters errors. A DB instance can go hours or days without new entries being written to the error log. If you see no recent entries, it's because the server didn't encounter an error that would result in a log entry.

By design, the error logs are filtered so that only unexpected events such as errors are shown. However, the error logs also contain some additional database information, for example query progress, which isn't shown. Therefore, even without any actual errors the size of the error logs might increase because of ongoing database activities. And while you might see a certain size in bytes or kilobytes for the error logs in the AWS Management Console, they might have 0 bytes when you download them.

Aurora MySQL writes `mysql-error.log` to disk every 5 minutes. It appends the contents of the log to `mysql-error-running.log`.

Aurora MySQL rotates the `mysql-error-running.log` file every hour.

> **Note**
>
> The log retention period is different between Amazon RDS and Aurora.

Aurora MySQL slow query and general logs

You can write the Aurora MySQL slow query log and the general log to a file or a database table. To do so, set parameters in your DB parameter group. For information about creating and modifying a DB parameter group, see Working with parameter groups. You must set these parameters before you can view the slow query log or general log in the Amazon RDS console or by using the Amazon RDS API, Amazon RDS CLI, or AWS SDKs.

You can control Aurora MySQL logging by using the parameters in this list:

- `slow_query_log`: To create the slow query log, set to 1. The default is 0.
- `general_log`: To create the general log, set to 1. The default is 0.

- `long_query_time`: To prevent fast-running queries from being logged in the slow query log, specify a value for the shortest query runtime to be logged, in seconds. The default is 10 seconds; the minimum is 0. If log_output = FILE, you can specify a floating point value that goes to microsecond resolution. If log_output = TABLE, you must specify an integer value with second resolution. Only queries whose runtime exceeds the `long_query_time` value are logged. For example, setting `long_query_time` to 0.1 prevents any query that runs for less than 100 milliseconds from being logged.
- `log_queries_not_using_indexes`: To log all queries that do not use an index to the slow query log, set to 1. Queries that don't use an index are logged even if their runtime is less than the value of the `long_query_time` parameter. The default is 0.
- `log_output` *`option`*: You can specify one of the following options for the `log_output` parameter.
  - **TABLE** – Write general queries to the `mysql.general_log` table, and slow queries to the `mysql.slow_log` table.
  - **FILE** – Write both general and slow query logs to the file system.
  - **NONE** – Disable logging.

  For Aurora MySQL version 2, the default for `log_output` is `FILE`.

For more information about the slow query and general logs, go to the following topics in the MySQL documentation:

- The slow query log
- The general query log

Aurora MySQL audit log

Audit logging for Aurora MySQL is called Advanced Auditing. To turn on Advanced Auditing, you set certain DB cluster parameters. For more information, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

Log rotation and retention for Aurora MySQL

When logging is enabled, Amazon Aurora rotates or deletes log files at regular intervals. This measure is a precaution to reduce the possibility of a large log file either blocking database use or affecting performance. Aurora MySQL handles rotation and deletion as follows:

- The Aurora MySQL error log file sizes are constrained to no more than 15 percent of the local storage for a DB instance. To maintain this threshold, logs are automatically rotated every hour. Aurora MySQL removes logs after 30 days or when 15% of disk space is reached. If the combined log file size exceeds the threshold after removing old log files, then the oldest log files are deleted until the log file size no longer exceeds the threshold.
- Aurora MySQL removes the audit, general, and slow query logs after either 24 hours or when 15% of storage has been consumed.
- When `FILE` logging is enabled, general log and slow query log files are examined every hour and log files more than 24 hours old are deleted. In some cases, the remaining combined log file size after the deletion might exceed the threshold of 15 percent of a DB instance's local space. In these cases, the oldest log files are deleted until the log file size no longer exceeds the threshold.
- When `TABLE` logging is enabled, log tables aren't rotated or deleted. Log tables are truncated when the size of all logs combined is too large. You can subscribe to the `low_free_storage` event to be notified when log tables should be manually rotated or deleted to free up space. For more information, see Working with Amazon RDS event notification.

  You can rotate the `mysql.general_log` table manually by calling the `mysql.rds_rotate_general_log` procedure. You can rotate the `mysql.slow_log` table by calling the `mysql.rds_rotate_slow_log` procedure.

  When you rotate log tables manually, the current log table is copied to a backup log table and the entries in the current log table are removed. If the backup log table already exists, then it is deleted before the current log table is copied to the backup. You can query the backup log table if needed. The backup log table for the `mysql.general_log` table is named `mysql.general_log_backup`. The backup log table for the `mysql.slow_log` table is named `mysql.slow_log_backup`.
- The Aurora MySQL audit logs are rotated when the file size reaches 100 MB, and removed after 24 hours.

To work with the logs from the Amazon RDS console, Amazon RDS API, Amazon RDS CLI, or AWS SDKs, set the `log_output` parameter to FILE. Like the Aurora MySQL error log, these log files are rotated hourly. The log files that were generated during the previous 24 hours are retained. Note that the retention period is different between Amazon RDS and Aurora.

Publishing Aurora MySQL logs to Amazon CloudWatch Logs

You can configure your Aurora MySQL DB cluster to publish log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, and use CloudWatch to create alarms and view metrics. You can use CloudWatch Logs to store your log records in highly durable storage. For more information, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.

Managing table-based Aurora MySQL logs

You can direct the general and slow query logs to tables on the DB instance by creating a DB parameter group and setting the `log_output` server parameter to `TABLE`. General queries are then logged to the `mysql.general_log` table, and slow queries are logged to the `mysql.slow_log` table. You can query the tables to access the log information. Enabling this logging increases the amount of data written to the database, which can degrade performance.

Both the general log and the slow query logs are disabled by default. In order to enable logging to tables, you must also set the `general_log` and `slow_query_log` server parameters to `1`.

Log tables keep growing until the respective logging activities are turned off by resetting the appropriate parameter to `0`. A large amount of data often accumulates over time, which can use up a considerable percentage of your allocated storage space. Amazon Aurora doesn't allow you to truncate the log tables, but you can move their contents. Rotating a table saves its contents to a backup table and then creates a new empty log table. You can manually rotate the log tables with the following command line procedures, where the command prompt is indicated by `PROMPT>`:

```
PROMPT> CALL mysql.rds_rotate_slow_log;
PROMPT> CALL mysql.rds_rotate_general_log;
```

To completely remove the old data and reclaim the disk space, call the appropriate procedure twice in succession.

Configuring Aurora MySQL binary logging

The *binary log* is a set of log files that contain information about data modifications made to an Aurora MySQL server instance. The binary log contains information such as the following:

- Events that describe database changes such as table creation or row modifications
- Information about the duration of each statement that updated data

- Events for statements that could have updated data but didn't

The binary log records statements that are sent during replication. It is also required for some recovery operations. For more information, see The Binary Log and Binary Log Overview in the MySQL documentation.

Binary logs are accessible only from the primary DB instance, not from the replicas.

MySQL on Amazon Aurora supports the *row-based*, *statement-based*, and *mixed* binary logging formats. We recommend mixed unless you need a specific binlog format. For details on the different Aurora MySQL binary log formats, see Binary logging formats in the MySQL documentation.

If you plan to use replication, the binary logging format is important because it determines the record of data changes that is recorded in the source and sent to the replication targets. For information about the advantages and disadvantages of different binary logging formats for replication, see Advantages and disadvantages of statement-based and row-based replication in the MySQL documentation.

> ⚠ **Important**
>
> Setting the binary logging format to row-based can result in very large binary log files. Large binary log files reduce the amount of storage available for a DB cluster and can increase the amount of time to perform a restore operation of a DB cluster.
> Statement-based replication can cause inconsistencies between the source DB cluster and a read replica. For more information, see Determination of safe and unsafe statements in binary logging in the MySQL documentation.
> Enabling binary logging increases the number of write disk I/O operations to the DB cluster. You can monitor IOPS usage with the `VolumeWriteIOPs` CloudWatch metric.

To set the MySQL binary logging format

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. Choose the DB cluster parameter group, associated with the DB cluster, that you want to modify.

You can't modify a default parameter group. If the DB cluster is using a default parameter group, create a new parameter group and associate it with the DB cluster.

For more information on parameter groups, see Working with parameter groups.

4. From **Actions**, choose **Edit**.
5. Set the `binlog_format` parameter to the binary logging format of your choice (`ROW`, `STATEMENT`, or `MIXED`). You can also use the value `OFF` to turn off binary logging.

   > **Note**
   >
   > Setting `binlog_format` to `OFF` in the DB cluster parameter group disables the `log_bin` session variable. This disables binary logging on the Aurora MySQL DB cluster, which in turn resets the `binlog_format` session variable to the default value of `ROW` in the database.

6. Choose **Save changes** to save the updates to the DB cluster parameter group.

After you perform these steps, you must reboot the writer instance in the DB cluster for your changes to apply. In Aurora MySQL version 2.09 and lower, when you reboot the writer instance, all of the reader instances in the DB cluster are also rebooted. In Aurora MySQL version 2.10 and higher, you must reboot all of the reader instances manually. For more information, see Rebooting an Amazon Aurora DB cluster or Amazon Aurora DB instance.

> **Important**
>
> Changing a DB cluster parameter group affects all DB clusters that use that parameter group. If you want to specify different binary logging formats for different Aurora MySQL DB clusters in an AWS Region, the DB clusters must use different DB cluster parameter groups. These parameter groups identify different logging formats. Assign the appropriate DB cluster parameter group to each DB clusters. For more information about Aurora MySQL parameters, see Aurora MySQL configuration parameters.

Accessing MySQL binary logs

You can use the mysqlbinlog utility to download or stream binary logs from RDS for MySQL DB instances. The binary log is downloaded to your local computer, where you can perform actions such as replaying the log using the mysql utility. For more information about using the mysqlbinlog utility, see Using mysqlbinlog to back up binary log files in the MySQL documentation.

To run the mysqlbinlog utility against an Amazon RDS instance, use the following options:

- `--read-from-remote-server` – Required.
- `--host` – The DNS name from the endpoint of the instance.
- `--port` – The port used by the instance.
- `--user` – A MySQL user that has been granted the `REPLICATION SLAVE` permission.
- `--password` – The password for the MySQL user, or omit a password value so that the utility prompts you for a password.
- `--raw` – Download the file in binary format.
- `--result-file` – The local file to receive the raw output.
- `--stop-never` – Stream the binary log files.
- `--verbose` – When you use the `ROW` binlog format, include this option to see the row events as pseudo-SQL statements. For more information on the `--verbose` option, see mysqlbinlog row event display in the MySQL documentation.
- Specify the names of one or more binary log files. To get a list of the available logs, use the SQL command `SHOW BINARY LOGS`.

For more information about mysqlbinlog options, see mysqlbinlog — Utility for processing binary log files in the MySQL documentation.

The following examples show how to use the mysqlbinlog utility.

For Linux, macOS, or Unix:

```
mysqlbinlog \
    --read-from-remote-server \
    --host=MySQLInstance1.cg034hpkmmjt.region.rds.amazonaws.com \
    --port=3306  \
    --user ReplUser \
    --password \
    --raw \
    --verbose \
    --result-file=/tmp/ \
    binlog.00098
```

For Windows:

```
mysqlbinlog ^
    --read-from-remote-server ^
    --host=MySQLInstance1.cg034hpkmmjt.region.rds.amazonaws.com ^
    --port=3306  ^
    --user ReplUser ^
    --password ^
    --raw ^
    --verbose ^
    --result-file=/tmp/ ^
    binlog.00098
```

Amazon RDS normally purges a binary log as soon as possible, but the binary log must still be available on the instance to be accessed by mysqlbinlog. To specify the number of hours for RDS to retain binary logs, use the mysql.rds_set_configuration stored procedure and specify a period with enough time for you to download the logs. After you set the retention period, monitor storage usage for the DB instance to ensure that the retained binary logs don't take up too much storage.

The following example sets the retention period to 1 day.

```
call mysql.rds_set_configuration('binlog retention hours', 24);
```

To display the current setting, use the mysql.rds_show_configuration stored procedure.

```
call mysql.rds_show_configuration;
```

Aurora PostgreSQL database log files

Aurora PostgreSQL logs database activities to the default PostgreSQL log file. For an on-premises PostgreSQL DB instance, these messages are stored locally in `log/postgresql.log`. For an Aurora PostgreSQL DB cluster, the log file is available on the Aurora cluster. Also, you must use the Amazon RDS Console to view or download its contents. The default logging level captures login failures, fatal server errors, deadlocks, and query failures.

For more information about how you can view, download, and watch file-based database logs, see Monitoring Amazon Aurora log files. To learn more about PostgreSQL logs, see Working with Amazon RDS and Aurora PostgreSQL logs: Part 1 and Working with Amazon RDS and Aurora PostgreSQL logs: Part 2.

In addition to the standard PostgreSQL logs discussed in this topic, Aurora PostgreSQL also supports the PostgreSQL Audit extension (`pgAudit`). Most regulated industries and government agencies need to maintain an audit log or audit trail of changes made to data to comply with legal requirements. For information about installing and using pgAudit, see Using pgAudit to log database activity.

Topics

Parameters that affect logging behavior

You can customize the logging behavior for your Aurora PostgreSQL DB cluster by modifying various parameters. In the following table you can find the parameters that affect how long the logs are stored, when to rotate the log, and whether to output the log as a CSV (comma-separated value) format. You can also find the text output sent to STDERR, among other settings. To change settings for the parameters that are modifiable, use a custom DB cluster parameter group for your Aurora PostgreSQL DB cluster. For more information, see Working with parameter groups. As noted in the table, the `log_line_prefix` can't be changed.

Parameter	Default	Description
log_destination	stderr	Sets the output format for the log. The default is `stderr` but you can also specify comma-

Parameter	Default	Description
		separated value (CSV) by adding `csvlog` to the setting. For more information, see [Setting the log destination (`stderr`, `csvlog`)]()
log_filename	postgresql.log.%Y-%m-%d-%H%M	Specifies the pattern for the log file name. In addition to the default, this parameter supports `postgresql.log.%Y-%m-%d` and `postgresql.log.%Y-%m-%d-%H` for the filename pattern.
log_line_prefix	%t:%r:%u@%d:[%p]:	Defines the prefix for each log line that gets written to `stderr`, to note the time (%t), remote host (%r), user (%u), database (%d), and process ID (%p). You can't modify this parameter.
log_rotation_age	60	Minutes after which log file is automatically rotated. You can change this value within the range of 1 and 1440 minutes. For more information, see [Setting log file rotation]().
log_rotation_size	–	The size (kB) at which the log is automatically rotated. You can change this value within the range of 50,000 to 1,000,000 kilobytes. To learn more, see [Setting log file rotation]().
rds.log_retention_period	4320	PostgreSQL logs that are older than the specified number of minutes are deleted. The default value of 4320 minutes deletes log files after 3 days. For more information, see [Setting the log retention period]().

To identify application issues, you can look for query failures, login failures, deadlocks, and fatal server errors in the log. For example, suppose that you converted a legacy application from Oracle to Aurora PostgreSQL, but not all queries converted correctly. These incorrectly formatted queries generate error messages that you can find in the logs to help identify problems. For more information about logging queries, see Turning on query logging for your Aurora PostgreSQL DB cluster.

In the following topics, you can find information about how to set various parameters that control the basic details for your PostgreSQL logs.

Topics

- Setting the log retention period
- Setting log file rotation
- Setting the log destination (stderr, csvlog)
- Understanding the log_line_prefix parameter

Setting the log retention period

The `rds.log_retention_period` parameter specifies how long your Aurora PostgreSQL DB cluster keeps its log files. The default setting is 3 days (4,320 minutes), but you can set this value to anywhere from 1 day (1,440 minutes) to 7 days (10,080 minutes). Be sure that your Aurora PostgreSQL DB cluster has sufficient storage to hold the log files for the period of time.

We recommend that you have your logs routinely published to Amazon CloudWatch Logs so that you can view and analyze system data long after the logs have been removed from your Aurora PostgreSQL DB cluster. For more information, see Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs. After you set up CloudWatch publishing, Aurora doesn't delete a log until after it's published to CloudWatch Logs.

Amazon Aurora compresses older PostgreSQL logs when storage for the DB instance reaches a threshold. Aurora compresses the files using the gzip compression utility. For more information, see the gzip website.

When storage for the DB instance is low and all available logs are compressed, you get a warning such as the following:

```
Warning: local storage for PostgreSQL log files is critically low for
this Aurora PostgreSQL instance, and could lead to a database outage.
```

If there's not enough storage, Aurora might delete compressed PostgreSQL logs before the end of a specified retention period. If that happens, you see a message similar to the following:

```
The oldest PostgreSQL log files were deleted due to local storage constraints.
```

Setting log file rotation

Aurora creates new log files every hour by default. The timing is controlled by the `log_rotation_age` parameter. This parameter has a default value of 60 (minutes), but you can set it to anywhere from 1 minute to 24 hours (1,440 minutes). When it's time for rotation, a new distinct log file is created. The file is named according to the pattern specified by the `log_filename` parameter.

Log files can also be rotated according to their size, as specified in the `log_rotation_size` parameter. This parameter specifies that the log should be rotated when it reaches the specified size (in kilobytes). The default `log_rotation_size` is 100000 kB (kilobytes) for an Aurora PostgreSQL DB cluster, but you can set this value to anywhere from 50,000 to 1,000,000 kilobytes.

The log file names are based on the file name pattern specified in the `log_filename` parameter. The available settings for this parameter are as follows:

- `postgresql.log.%Y-%m-%d` – Default format for the log file name. Includes the year, month, and date in the name of the log file.
- `postgresql.log.%Y-%m-%d-%H` – Includes the hour in the log file name format.
- `postgresql.log.%Y-%m-%d-%H%M` – Includes hour:minute in the log file name format.

If you set `log_rotation_age` parameter to less than 60 minutes, set the `log_filename` parameter to the minute format.

For more information, see [`log_rotation_age`] and [`log_rotation_size`] in the PostgreSQL documentation.

Setting the log destination (`stderr`, `csvlog`)

By default, Aurora PostgreSQL generates logs in standard error (stderr) format. This format is the default setting for the `log_destination` parameter. Each message is prefixed using the pattern specified in the `log_line_prefix` parameter. For more information, see Understanding the log_line_prefix parameter.

Aurora PostgreSQL can also generate the logs in `csvlog` format. The `csvlog` is useful for analyzing the log data as comma-separated values (CSV) data. For example, suppose that you use the log_fdw extension to work with your logs as foreign tables. The foreign table created on stderr log files contains a single column with log event data. By adding csvlog to the log_destination parameter, you get the log file in the CSV format with demarcations for the multiple columns of the foreign table. You can now sort and analyze your logs more easily.

If you specify csvlog for this parameter, be aware that both stderr and csvlog files are generated. Be sure to monitor the storage consumed by the logs, taking into account the rds.log_retention_period and other settings that affect log storage and turnover. Using stderr and csvlog more than doubles the storage consumed by the logs.

If you add csvlog to log_destination and you want to revert to the stderr alone, you need to reset the parameter. To do so, open the Amazon RDS Console and then open the custom DB cluster parameter group for your instance. Choose the log_destination parameter, choose **Edit parameter**, and then choose **Reset**.

For more information about configuring logging, see Working with Amazon RDS and Aurora PostgreSQL logs: Part 1.

Understanding the log_line_prefix parameter

The stderr log format prefixes each log message with the details specified by the log_line_prefix parameter, as follows.

```
%t:%r:%u@%d:[%p]:t
```

You can't change this setting. Each log entry sent to stderr includes the following information.

- %t – Time of log entry
- %r – Remote host address
- %u@%d – User name @ database name
- [%p] – Process ID if available

Turning on query logging for your Aurora PostgreSQL DB cluster

You can collect more detailed information about your database activities, including queries, queries waiting for locks, checkpoints, and many other details by setting some of the parameters listed in the following table. This topic focuses on logging queries.

Parameter	Default	Description
log_connections	–	Logs each successful connection. To learn how to use this parameter with `log_disconnections` to detect connection churn, see Managing Aurora PostgreSQL connection churn with pooling.
log_disconnections	–	Logs the end of each session and its duration. To learn how to use this parameter with `log_connections` to detect connection churn, see Managing Aurora PostgreSQL connection churn with pooling.
log_checkpoints	1	Logs each checkpoint.
log_lock_waits	–	Logs long lock waits. By default, this parameter isn't set.
log_min_duration_sample	–	(ms) Sets the minimum execution time above which a sample of statements is logged. Sample size is set using the `log_statement_sample_rate` parameter.
log_min_duration_statement	–	Any SQL statement that runs atleast for the specified amount of time or longer gets logged. By default, this parameter isn't set. Turning on this parameter can help you find unoptimized queries.
log_statement	–	Sets the type of statements logged. By default, this parameter isn't set, but you can change it to `all`, `ddl`, or `mod` to specify the types of SQL statements that you want logged. If you specify anything other than none for this parameter, you should also take additional steps to prevent the exposure of passwords in the log files. For more informati

Parameter	Default	Description
		on, see Mitigating risk of password exposure when using query logging.
log_statement_sample_rate	–	The percentage of statements exceeding the time specified in `log_min_duration_sample` to be logged, expressed as a floating point value between 0.0 and 1.0.
log_statement_stats	–	Writes cumulative performance statistics to the server log.

Using logging to find slow performing queries

You can log SQL statements and queries to help find slow performing queries. You turn on this capability by modifying the settings in the `log_statement` and `log_min_duration` parameters as outlined in this section. Before turning on query logging for your Aurora PostgreSQL DB cluster, you should be aware of possible password exposure in the logs and how to mitigate the risks. For more information, see Mitigating risk of password exposure when using query logging.

Following, you can find reference information about the `log_statement` and `log_min_duration` parameters.

log_statement

This parameter specifies the type of SQL statements that should get sent to the log. The default value is none. If you change this parameter to `all`, `ddl`, or `mod`, be sure to apply recommended actions to mitigate the risk of exposing passwords in the logs. For more information, see Mitigating risk of password exposure when using query logging.

all

Logs all statements. This setting is recommended for debugging purposes.

ddl

Logs all data definition language (DDL) statements, such as CREATE, ALTER, DROP, and so on.

mod

Logs all DDL statements and data manipulation language (DML) statements, such as INSERT, UPDATE, and DELETE, which modify the data.

none

No SQL statements get logged. We recommend this setting to avoid the risk of exposing passwords in the logs.

log_min_duration_statement

Any SQL statement that runs atleast for the specified amount of time or longer gets logged. By default, this parameter isn't set. Turning on this parameter can help you find unoptimized queries.

–1–2147483647

The number of milliseconds (ms) of runtime over which a statement gets logged.

To set up query logging

These steps assume that your Aurora PostgreSQL DB cluster uses a custom DB cluster parameter group.

1. Set the `log_statement` parameter to `all`. The following example shows the information that is written to the `postgresql.log` file with this parameter setting.

```
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:LOG: statement:
 SELECT feedback, s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:LOG: QUERY
 STATISTICS
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:DETAIL: ! system
 usage stats:
! 0.017355 s user, 0.000000 s system, 0.168593 s elapsed
! [0.025146 s user, 0.000000 s system total]
! 36644 kB max resident size
! 0/8 [0/8] filesystem blocks in/out
! 0/733 [0/1364] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 19/0 [27/0] voluntary/involuntary context switches
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:STATEMENT: SELECT
 feedback, s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;
2022-10-05 22:05:56 UTC:52.95.4.1(11335):postgres@labdb:[3639]:ERROR: syntax error
 at or near "ORDER" at character 1
2022-10-05 22:05:56 UTC:52.95.4.1(11335):postgres@labdb:[3639]:STATEMENT: ORDER BY
 s.confidence DESC;
----------------------- END OF LOG ----------------------
```

2. Set the `log_min_duration_statement` parameter. The following example shows the information that is written to the `postgresql.log` file when the parameter is set to 1.

   Queries that exceed the duration specified in the `log_min_duration_statement` parameter are logged. The following shows an example. You can view the log file for your Aurora PostgreSQL DB cluster in the Amazon RDS Console.

```
2022-10-05 19:05:19 UTC:52.95.4.1(6461):postgres@labdb:[6144]:LOG: statement: DROP
 table comments;
2022-10-05 19:05:19 UTC:52.95.4.1(6461):postgres@labdb:[6144]:LOG: duration:
 167.754 ms
2022-10-05 19:08:07 UTC::@:[355]:LOG: checkpoint starting: time
2022-10-05 19:08:08 UTC::@:[355]:LOG: checkpoint complete: wrote 11 buffers
 (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=1.013 s, sync=0.006 s,
 total=1.033 s; sync files=8, longest=0.004 s, average=0.001 s; distance=131028 kB,
 estimate=131028 kB
----------------------- END OF LOG ----------------------
```

Mitigating risk of password exposure when using query logging

We recommend that you keep `log_statement` set to `none` to avoid exposing passwords. If you set `log_statement` to `all`, `ddl`, or `mod`, we recommend that you take one or more of the following steps.

- For the client, encrypt sensitive information. For more information, see Encryption Options in the PostgreSQL documentation. Use the `ENCRYPTED` (and `UNENCRYPTED`) options of the `CREATE` and `ALTER` statements. For more information, see CREATE USER in the PostgreSQL documentation.

- For your Aurora PostgreSQL DB cluster, set up and use the PostgreSQL Auditing (pgAudit) extension. This extension redacts sensitive information in CREATE and ALTER statements sent to the log. For more information, see Using pgAudit to log database activity.
- Restrict access to the CloudWatch logs.
- Use stronger authentication mechanisms such as IAM.

Monitoring Amazon Aurora API calls in AWS CloudTrail

AWS CloudTrail is an AWS service that helps you audit your AWS account. AWS CloudTrail is turned on for your AWS account when you create it. For more information about CloudTrail, see the AWS CloudTrail User Guide.

Topics

- CloudTrail integration with Amazon Aurora
- Amazon Aurora log file entries

CloudTrail integration with Amazon Aurora

All Amazon Aurora actions are logged by CloudTrail. CloudTrail provides a record of actions taken by a user, role, or an AWS service in Amazon Aurora.

CloudTrail events

CloudTrail captures API calls for Amazon Aurora as events. An *event* represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. Events include calls from the Amazon RDS console and from code calls to the Amazon RDS API operations.

Amazon Aurora activity is recorded in a CloudTrail event in **Event history**. You can use the CloudTrail console to view the last 90 days of recorded API activity and events in an AWS Region. For more information, see Viewing events with CloudTrail event history.

CloudTrail trails

For an ongoing record of events in your AWS account, including events for Amazon Aurora, create a *trail*. A trail is a configuration that enables delivery of events to a specified Amazon S3 bucket. CloudTrail typically delivers log files within 15 minutes of account activity.

Note

If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**.

You can create two types of trails for an AWS account: a trail that applies to all Regions, or a trail that applies to one Region. By default, when you create a trail in the console, the trail applies to all Regions.

Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple Regions and Receiving CloudTrail log files from multiple accounts

Amazon Aurora log file entries

CloudTrail log files contain one or more log entries. CloudTrail log files are not an ordered stack trace of the public API calls, so they do not appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateDBInstance` action.

```
{
    "eventVersion": "1.04",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AKIAIOSFODNN7EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/johndoe",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
        "userName": "johndoe"
    },
    "eventTime": "2018-07-30T22:14:06Z",
    "eventSource": "rds.amazonaws.com",
    "eventName": "CreateDBInstance",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "aws-cli/1.15.42 Python/3.6.1 Darwin/17.7.0 botocore/1.10.42",
    "requestParameters": {
        "enableCloudwatchLogsExports": [
            "audit",
            "error",
            "general",
            "slowquery"
        ],
        "dBInstanceIdentifier": "test-instance",
        "engine": "mysql",
        "masterUsername": "myawsuser",
        "allocatedStorage": 20,
        "dBInstanceClass": "db.m1.small",
        "masterUserPassword": "****"
    },
    "responseElements": {
        "dBInstanceArn": "arn:aws:rds:us-east-1:123456789012:db:test-instance",
        "storageEncrypted": false,
        "preferredBackupWindow": "10:27-10:57",
        "preferredMaintenanceWindow": "sat:05:47-sat:06:17",
        "backupRetentionPeriod": 1,
        "allocatedStorage": 20,
        "storageType": "standard",
        "engineVersion": "8.0.28",
        "dbInstancePort": 0,
        "optionGroupMemberships": [
            {
                "status": "in-sync",
                "optionGroupName": "default:mysql-8-0"
            }
        ],
        "dBParameterGroups": [
            {
                "dBParameterGroupName": "default.mysql8.0",
                "parameterApplyStatus": "in-sync"
            }
        ],
        "monitoringInterval": 0,
        "dBInstanceClass": "db.m1.small",
        "readReplicaDBInstanceIdentifiers": [],
        "dBSubnetGroup": {
            "dBSubnetGroupName": "default",
            "dBSubnetGroupDescription": "default",
            "subnets": [
                {
                    "subnetAvailabilityZone": {"name": "us-east-1b"},
                    "subnetIdentifier": "subnet-cbfff283",
                    "subnetStatus": "Active"
                },
                {
                    "subnetAvailabilityZone": {"name": "us-east-1e"},
                    "subnetIdentifier": "subnet-d7c825e8",
                    "subnetStatus": "Active"
                },
                {
                    "subnetAvailabilityZone": {"name": "us-east-1f"},
                    "subnetIdentifier": "subnet-6746046b",
                    "subnetStatus": "Active"
                },
                {
                    "subnetAvailabilityZone": {"name": "us-east-1c"},
                    "subnetIdentifier": "subnet-bac383e0",
                    "subnetStatus": "Active"
                },
                {
                    "subnetAvailabilityZone": {"name": "us-east-1d"},
                    "subnetIdentifier": "subnet-42599426",
                    "subnetStatus": "Active"
                },
                {
                    "subnetAvailabilityZone": {"name": "us-east-1a"},
                    "subnetIdentifier": "subnet-da327bf6",
                    "subnetStatus": "Active"
                }
            ],
            "vpcId": "vpc-136a4c6a",
            "subnetGroupStatus": "Complete"
        },
        "masterUsername": "myawsuser",
        "multiAZ": false,
        "autoMinorVersionUpgrade": true,
        "engine": "mysql",
        "cACertificateIdentifier": "rds-ca-2015",
        "dbiResourceId": "db-ETDZIIXHEWY5N7GXVC4SH7H5IA",
        "dBSecurityGroups": [],
        "pendingModifiedValues": {
            "masterUserPassword": "****",
            "pendingCloudwatchLogsExports": {
                "logTypesToEnable": [
                    "audit",
                    "error",
                    "general",
                    "slowquery"
                ]
            }
        },
        "dBInstanceStatus": "creating",
        "publiclyAccessible": true,
        "domainMemberships": [],
        "copyTagsToSnapshot": false,
        "dBInstanceIdentifier": "test-instance",
        "licenseModel": "general-public-license",
        "iAMDatabaseAuthenticationEnabled": false,
        "performanceInsightsEnabled": false,
        "vpcSecurityGroups": [
            {
                "status": "active",
                "vpcSecurityGroupId": "sg-f839b688"
            }
        ]
    },
    "requestID": "daf2e3f5-96a3-4df7-a026-863f96db793e",
    "eventID": "797163d3-5726-441d-80a7-6eeb7464acd4",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

As shown in the `userIdentity` element in the preceding example, every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information about the `userIdentity`, see the CloudTrail userIdentity element. For more information about `CreateDBInstance` and other Amazon Aurora actions, see the Amazon RDS API Reference.

Monitoring Amazon Aurora with Database Activity Streams

By using Database Activity Streams, you can monitor near real-time streams of database activity.

Topics

Overview of Database Activity Streams

As an Amazon Aurora database administrator, you need to safeguard your database and meet compliance and regulatory requirements. One strategy is to integrate database activity streams with your monitoring tools. In this way, you monitor and set alarms for auditing activity in your Amazon Aurora cluster .

Security threats are both external and internal. To protect against internal threats, you can control administrator access to data streams by configuring the Database Activity Streams feature. DBAs don't have access to the collection, transmission, storage, and processing of the streams.

Topics

How database activity streams work

In Amazon Aurora, you start a database activity stream at the cluster level. All DB instances within your cluster have database activity streams enabled.

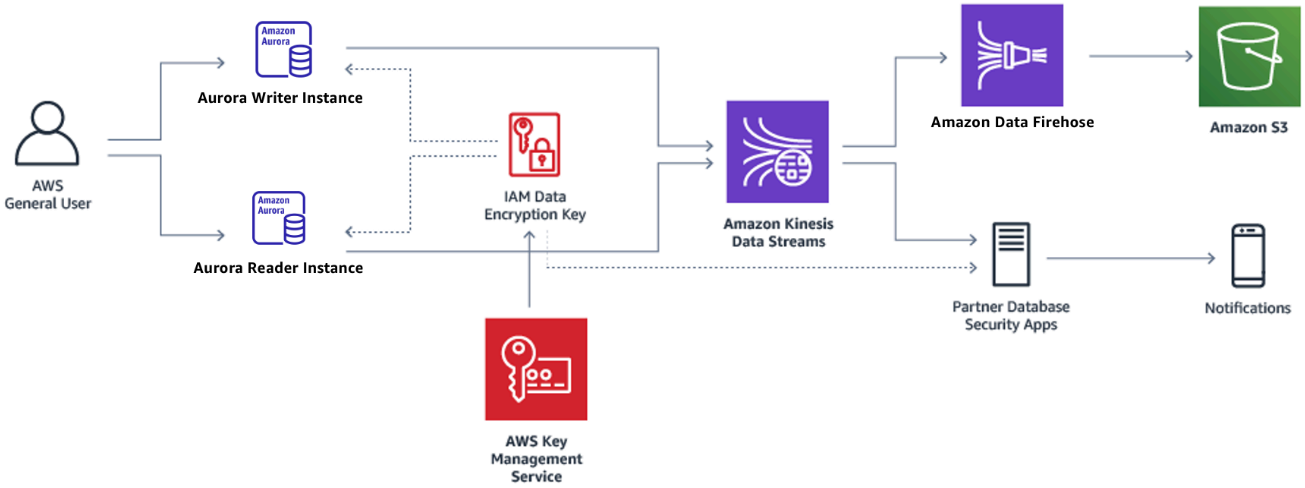
Your Aurora DB cluster pushes activities to an Amazon Kinesis data stream in near real time. The Kinesis stream is created automatically. From Kinesis, you can configure AWS services such as Amazon Data Firehose and AWS Lambda to consume the stream and store the data.

> **Important**
>
> Use of the database activity streams feature in Amazon Aurora is free, but Amazon Kinesis charges for a data stream. For more information, see Amazon Kinesis Data Streams pricing.

If you use an Aurora global database, start a database activity stream on each DB cluster separately. Each cluster delivers audit data to its own Kinesis stream within its own AWS Region. The activity streams don't operate differently during a failover. They continue to audit your global database as usual.

You can configure applications for compliance management to consume database activity streams. For Aurora PostgreSQL, compliance applications include IBM's Security Guardium and Imperva's SecureSphere Database Audit and Protection. These applications can use the stream to generate alerts and audit activity on your Aurora DB cluster.

The following graphic shows an Aurora DB cluster configured with Amazon Data Firehose.



Asynchronous and synchronous mode for database activity streams

You can choose to have the database session handle database activity events in either of the following modes:

- **Asynchronous mode** – When a database session generates an activity stream event, the session returns to normal activities immediately. In the background, the activity stream event is made a durable record. If an error occurs in the background task, an RDS event is sent. This event indicates the beginning and end of any time windows where activity stream event records might have been lost.

  Asynchronous mode favors database performance over the accuracy of the activity stream.

  > ⓘ **Note**
  >
  > Asynchronous mode is available for both Aurora PostgreSQL and Aurora MySQL.

- **Synchronous mode** – When a database session generates an activity stream event, the session blocks other activities until the event is made durable. If the event can't be made durable for some reason, the database session returns to normal activities. However, an RDS event is sent indicating that activity stream records might be lost for some time. A second RDS event is sent after the system is back to a healthy state.

  The synchronous mode favors the accuracy of the activity stream over database performance.

  > ⓘ **Note**
  >
  > Synchronous mode is available for Aurora PostgreSQL. You can't use synchronous mode with Aurora MySQL.

Requirements and limitations for database activity streams

In Aurora, database activity streams have the following requirements and limitations:

- Amazon Kinesis is required for database activity streams.
- AWS Key Management Service (AWS KMS) is required for database activity streams because they are always encrypted.
- Applying additional encryption to your Amazon Kinesis data stream is incompatible with database activity streams, which are already encrypted with your AWS KMS key.
- Start your database activity stream at the DB cluster level. If you add a DB instance to your cluster, you don't need to start an activity stream on the instance: it is audited automatically.

- In an Aurora global database, make sure to start an activity stream on each DB cluster separately. Each cluster delivers audit data to its own Kinesis stream within its own AWS Region.
- In Aurora PostgreSQL, make sure to stop database activity stream before an upgrade. You can start the database activity stream after the upgrade completes.

Region and version availability

Feature availability and support varies across specific versions of each Aurora database engine, and across AWS Regions. For more information on version and Region availability with Aurora and database activity streams, see Supported Regions and Aurora DB engines for database activity streams.

Supported DB instance classes for database activity streams

For Aurora MySQL, you can use database activity streams with the following DB instance classes:

- db.r7g.*large
- db.r6g.*large
- db.r6i.*large
- db.r5.*large
- db.x2g.*

For Aurora PostgreSQL, you can use database activity streams with the following DB instance classes:

- db.r7g.*large
- db.r6g.*large
- db.r6i.*large
- db.r6id.*large
- db.r5.*large
- db.r4.*large
- db.x2g.*

Network prerequisites for Aurora MySQL database activity streams

In the following section, you can find how to configure your virtual private cloud (VPC) for use with database activity streams.

> **Note**
>
> Aurora MySQL network prerequisites are applicable to the following engine versions:
>
> - Aurora MySQL version 2, up to 2.11.3
> - Aurora MySQL version 2.12.0
> - Aurora MySQL version 3, up to 3.04.2

Topics

Prerequisites for AWS KMS endpoints

Instances in an Aurora MySQL cluster that use activity streams must be able to access AWS KMS endpoints. Make sure this requirement is satisfied before enabling database activity streams for your Aurora MySQL cluster. If the Aurora cluster is publicly available, this requirement is satisfied automatically.

> **Important**
>
> If the Aurora MySQL DB cluster can't access the AWS KMS endpoint, the activity stream stops. In that case, Aurora notifies you about this issue using RDS Events.

Prerequisites for public availability

For an Aurora DB cluster to be public, it must meet the following requirements:

- **Publicly Accessible** is **Yes** in the AWS Management Console cluster details page.

- The DB cluster is in an Amazon VPC public subnet. For more information about publicly accessible DB instances, see Working with a DB cluster in a VPC. For more information about public Amazon VPC subnets, see Your VPC and Subnets.

Prerequisites for private availability

If your Aurora DB cluster is in a VPC public subnet and isn't publicly accessible, it's private. To keep your cluster private and use it with database activity streams, you have the following options:

- Configure Network Address Translation (NAT) in your VPC. For more information, see NAT Gateways.
- Create an AWS KMS endpoint in your VPC. This option is recommended because it's easier to configure.

To create an AWS KMS endpoint in your VPC

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create Endpoint**.

   The **Create Endpoint** page appears.
4. Do the following:
   - In **Service category**, choose **AWS services**.
   - In **Service Name**, choose **com.amazonaws.*region*.kms**, where *region* is the AWS Region where your cluster is located.
   - For **VPC**, choose the VPC where your cluster is located.
5. Choose **Create Endpoint**.

For more information about configuring VPC endpoints, see VPC Endpoints.

Starting a database activity stream

To monitor database activity for all instances in your Aurora DB cluster, start an activity stream at the cluster level. Any DB instances that you add to the cluster are also automatically monitored. If you use an Aurora global database, start a database activity stream on each DB cluster separately. Each cluster delivers audit data to its own Kinesis stream within its own AWS Region.

When you start an activity stream, each database activity event that you configured in the audit policy generates an activity stream event. SQL commands such as `CONNECT` and `SELECT` generate access events. SQL commands such as `CREATE` and `INSERT` generate change events.

Console

To start a database activity stream

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster on which you want to start an activity stream.
4. For **Actions**, choose **Start activity stream**.

   The **Start database activity stream:** *name* window appears, where *name* is your DB cluster.
5. Enter the following settings:

   - For **AWS KMS key**, choose a key from the list of AWS KMS keys.

     > **Note**
     >
     > If your Aurora MySQL cluster can't access KMS keys, follow the instructions in Network prerequisites for Aurora MySQL database activity streams to enable such access first.

     Aurora uses the KMS key to encrypt the key that in turn encrypts database activity. Choose a KMS key other than the default key. For more information about encryption keys and AWS KMS, see What is AWS Key Management Service? in the *AWS Key Management Service Developer Guide.*
   - For **Database activity stream mode**, choose **Asynchronous** or **Synchronous**.

     > **Note**
     >
     > This choice applies only to Aurora PostgreSQL. For Aurora MySQL, you can use only asynchronous mode.

- Choose **Immediately**.

  When you choose **Immediately**, the DB cluster restarts right away. If you choose **During the next maintenance window**, the DB cluster doesn't restart right away. In this case, the database activity stream doesn't start until the next maintenance window.

6. Choose **Start database activity stream**.

   The status for the DB cluster shows that the activity stream is starting.

> **Note**
>
> If you get the error `You can't start a database activity stream in this configuration`, check Supported DB instance classes for database activity streams to see whether your DB cluster is using a supported instance class.

AWS CLI

To start database activity streams for a DB cluster , configure the DB cluster using the start-activity-stream AWS CLI command.

- `--resource-arn` *`arn`* – Specifies the Amazon Resource Name (ARN) of the DB cluster.
- `--mode` *`sync-or-async`* – Specifies either synchronous (`sync`) or asynchronous (`async`) mode. For Aurora PostgreSQL, you can choose either value. For Aurora MySQL, specify `async`.
- `--kms-key-id` *`key`* – Specifies the KMS key identifier for encrypting messages in the database activity stream. The AWS KMS key identifier is the key ARN, key ID, alias ARN, or alias name for the AWS KMS key.

The following example starts a database activity stream for a DB cluster in asynchronous mode.

For Linux, macOS, or Unix:

```
aws rds start-activity-stream \
    --mode async \
    --kms-key-id my-kms-key-arn \
    --resource-arn my-cluster-arn \
    --apply-immediately
```

For Windows:

```
aws rds start-activity-stream ^
    --mode async ^
    --kms-key-id my-kms-key-arn ^
    --resource-arn my-cluster-arn ^
    --apply-immediately
```

RDS API

To start database activity streams for a DB cluster, configure the cluster using the StartActivityStream operation.

Call the action with the parameters below:

- `Region`
- `KmsKeyId`
- `ResourceArn`
- `Mode`

Getting the status of a database activity stream

You can get the status of an activity stream using the console or AWS CLI.

Console

To get the status of a database activity stream

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster link.
3. Choose the **Configuration** tab, and check **Database activity stream** for status.

AWS CLI

You can get the activity stream configuration for a DB cluster as the response to a describe-db-clusters CLI request.

The following example describes *`my-cluster`*.

```
aws rds --region my-region describe-db-clusters --db-cluster-identifier my-cluster
```

The following example shows a JSON response. The following fields are shown:

- `ActivityStreamKinesisStreamName`
- `ActivityStreamKmsKeyId`
- `ActivityStreamStatus`
- `ActivityStreamMode`
-

These fields are the same for Aurora PostgreSQL and Aurora MySQL, except that `ActivityStreamMode` is always `async` for Aurora MySQL, while for Aurora PostgreSQL it might be `sync` or `async`.

```
{
    "DBClusters": [
        {
      "DBClusterIdentifier": "my-cluster",
            ...
            "ActivityStreamKinesisStreamName": "aws-rds-das-cluster-
A6TSYXITZCZXJHIRVFUBZ5LTWY",
            "ActivityStreamStatus": "starting",
            "ActivityStreamKmsKeyId": "12345678-abcd-efgh-ijkl-bd041f170262",
            "ActivityStreamMode": "async",
            "DbClusterResourceId": "cluster-ABCD123456"
            ...
        }
    ]
}
```

RDS API

You can get the activity stream configuration for a DB cluster as the response to a DescribeDBClusters operation.

Stopping a database activity stream

You can stop an activity stream using the console or AWS CLI.

If you delete your DB cluster, the activity stream is stopped and the underlying Amazon Kinesis stream is deleted automatically.

Console

To turn off an activity stream

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose a DB cluster that you want to stop the database activity stream for.
4. For **Actions**, choose **Stop activity stream**. The **Database Activity Stream** window appears.

    a. Choose **Immediately**.

       When you choose **Immediately**, the DB cluster restarts right away. If you choose **During the next maintenance window**, the DB cluster doesn't restart right away. In this case, the database activity stream doesn't stop until the next maintenance window.

    b. Choose **Continue**.

AWS CLI

To stop database activity streams for your DB cluster, configure the DB cluster using the AWS CLI command stop-activity-stream. Identify the AWS Region for the DB cluster using the `--region` parameter. The `--apply-immediately` parameter is optional.

For Linux, macOS, or Unix:

```
aws rds --region MY_REGION \
    stop-activity-stream \
    --resource-arn MY_CLUSTER_ARN \
    --apply-immediately
```

For Windows:

```
aws rds --region MY_REGION ^
    stop-activity-stream ^
    --resource-arn MY_CLUSTER_ARN ^
    --apply-immediately
```

RDS API

To stop database activity streams for your DB cluster, configure the cluster using the StopActivityStream operation. Identify the AWS Region for the DB cluster using the `Region` parameter. The `ApplyImmediately` parameter is optional.

Monitoring database activity streams

Database activity streams monitor and report activities. The stream of activity is collected and transmitted to Amazon Kinesis. From Kinesis, you can monitor the activity stream, or other services and applications can consume the activity stream for further analysis. You can find the underlying Kinesis stream name by using the AWS CLI command `describe-db-clusters` or the RDS API `DescribeDBClusters` operation.

Aurora manages the Kinesis stream for you as follows:

- Aurora creates the Kinesis stream automatically with a 24-hour retention period.
- Aurora scales the Kinesis stream if necessary.
- If you stop the database activity stream or delete the DB cluster, Aurora deletes the Kinesis stream.

The following categories of activity are monitored and put in the activity stream audit log:

- **SQL commands** – All SQL commands are audited, and also prepared statements, built-in functions, and functions in PL/SQL. Calls to stored procedures are audited. Any SQL statements issued inside stored procedures or functions are also audited.
- **Other database information** – Activity monitored includes the full SQL statement, the row count of affected rows from DML commands, accessed objects, and the unique database name. For Aurora PostgreSQL, database activity streams also monitor the bind variables and stored procedure parameters.

> ⚠ **Important**
>
> The full SQL text of each statement is visible in the activity stream audit log, including any sensitive data. However, database user passwords are redacted if Aurora can determine them from the context, such as in the following SQL statement.

```
ALTER ROLE role-name WITH password
```

- **Connection information** – Activity monitored includes session and network information, the server process ID, and exit codes.

If an activity stream has a failure while monitoring your DB instance, you are notified through RDS events.

Topics

- Accessing an activity stream from Kinesis
- Audit log contents and examples
- databaseActivityEventList JSON array
- Processing a database activity stream using the AWS SDK

Accessing an activity stream from Kinesis

When you enable an activity stream for a DB cluster, a Kinesis stream is created for you. From Kinesis, you can monitor your database activity in real time. To further analyze database activity, you can connect your Kinesis stream to consumer applications. You can also connect the stream to compliance management applications such as IBM's Security Guardium or Imperva's SecureSphere Database Audit and Protection.

You can access your Kinesis stream either from the RDS console or the Kinesis console.

To access an activity stream from Kinesis using the RDS console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster on which you started an activity stream.
4. Choose **Configuration**.
5. Under **Database activity stream**, choose the link under **Kinesis stream**.
6. In the Kinesis console, choose **Monitoring** to begin observing the database activity.

To access an activity stream from Kinesis using the Kinesis console

1. Open the Kinesis console at https://console.aws.amazon.com/kinesis.
2. Choose your activity stream from the list of Kinesis streams.

   An activity stream's name includes the prefix `aws-rds-das-cluster-` followed by the resource ID of the DB cluster. The following is an example.

   ```
   aws-rds-das-cluster-NHVOV4PCLWHGF52NP
   ```

   To use the Amazon RDS console to find the resource ID for the DB cluster, choose your DB cluster from the list of databases, and then choose the **Configuration** tab.

   To use the AWS CLI to find the full Kinesis stream name for an activity stream, use a describe-db-clusters CLI request and note the value of `ActivityStreamKinesisStreamName` in the response.
3. Choose **Monitoring** to begin observing the database activity.

For more information about using Amazon Kinesis, see What Is Amazon Kinesis Data Streams?.

Audit log contents and examples

Monitored events are represented in the database activity stream as JSON strings. The structure consists of a JSON object containing a `DatabaseActivityMonitoringRecord`, which in turn contains a `databaseActivityEventList` array of activity events.

Topics

- Examples of an audit log for an activity stream
- DatabaseActivityMonitoringRecords JSON object
- databaseActivityEvents JSON Object

Examples of an audit log for an activity stream

Following are sample decrypted JSON audit logs of activity event records.

Example Activity event record of an Aurora PostgreSQL CONNECT SQL statement

The following activity event record shows a login with the use of a `CONNECT` SQL statement (`command`) by a psql client (`clientApplication`).

```
{
  "type":"DatabaseActivityMonitoringRecords",
  "version":"1.1",
  "databaseActivityEvents":
    {
      "type":"DatabaseActivityMonitoringRecord",
      "clusterId":"cluster-4HNY5V4RRNPKKYB7ICFKE5JBQQ",
      "instanceId":"db-FZJTMYKCXQBUUZ6VLU7NW3ITCM",
      "databaseActivityEventList":[
        {
          "startTime": "2019-10-30 00:39:49.940668+00",
          "logTime": "2019-10-30 00:39:49.990579+00",
          "statementId": 1,
          "substatementId": 1,
          "objectType": null,
          "command": "CONNECT",
          "objectName": null,
          "databaseName": "postgres",
          "dbUserName": "rdsadmin",
          "remoteHost": "172.31.3.195",
          "remotePort": "49804",
          "sessionId": "5ce5f7f0.474b",
          "rowCount": null,
          "commandText": null,
          "paramList": [],
          "pid": 18251,
          "clientApplication": "psql",
          "exitCode": null,
          "class": "MISC",
          "serverVersion": "2.3.1",
          "serverType": "PostgreSQL",
          "serviceName": "Amazon Aurora PostgreSQL-Compatible edition",
          "serverHost": "172.31.3.192",
          "netProtocol": "TCP",
          "dbProtocol": "Postgres 3.0",
          "type": "record",
          "errorMessage": null
        }
      ]
    },
   "key":"decryption-key"
}
```

Example Activity event record of an Aurora MySQL CONNECT SQL statement

The following activity event record shows a logon with the use of a `CONNECT` SQL statement (command) by a mysql client (`clientApplication`).

```
{
  "type":"DatabaseActivityMonitoringRecord",
  "clusterId":"cluster-some_id",
  "instanceId":"db-some_id",
  "databaseActivityEventList":[
    {
      "logTime":"2020-05-22 18:07:13.267214+00",
      "type":"record",
      "clientApplication":null,
      "pid":2830,
      "dbUserName":"rdsadmin",
      "databaseName":"",
      "remoteHost":"localhost",
      "remotePort":"11053",
      "command":"CONNECT",
      "commandText":"",
      "paramList":null,
      "objectType":"TABLE",
      "objectName":"",
      "statementId":0,
      "substatementId":1,
      "exitCode":"0",
      "sessionId":"725121",
      "rowCount":0,
      "serverHost":"master",
      "serverType":"MySQL",
      "serviceName":"Amazon Aurora MySQL",
      "serverVersion":"MySQL 5.7.12",
      "startTime":"2020-05-22 18:07:13.267207+00",
      "endTime":"2020-05-22 18:07:13.267213+00",
      "transactionId":"0",
      "dbProtocol":"MySQL",
      "netProtocol":"TCP",
      "errorMessage":"",
      "class":"MAIN"
    }
  ]
}
```

Example Activity event record of an Aurora PostgreSQL CREATE TABLE statement

The following example shows a `CREATE TABLE` event for Aurora PostgreSQL.

```
{
  "type":"DatabaseActivityMonitoringRecords",
  "version":"1.1",
  "databaseActivityEvents":
    {
      "type":"DatabaseActivityMonitoringRecord",
      "clusterId":"cluster-4HNY5V4RRNPKKYB7ICFKE5JBQQ",
      "instanceId":"db-FZJTMYKCXQBUUZ6VLU7NW3ITCM",
      "databaseActivityEventList":[
        {
          "startTime": "2019-05-24 00:36:54.403455+00",
          "logTime": "2019-05-24 00:36:54.494235+00",
          "statementId": 2,
          "substatementId": 1,
          "objectType": null,
          "command": "CREATE TABLE",
          "objectName": null,
          "databaseName": "postgres",
          "dbUserName": "rdsadmin",
          "remoteHost": "172.31.3.195",
          "remotePort": "34534",
          "sessionId": "5ce73c6f.7e64",
          "rowCount": null,
          "commandText": "create table my_table (id serial primary key, name
 varchar(32));",
          "paramList": [],
          "pid": 32356,
          "clientApplication": "psql",
          "exitCode": null,
          "class": "DDL",
          "serverVersion": "2.3.1",
          "serverType": "PostgreSQL",
          "serviceName": "Amazon Aurora PostgreSQL-Compatible edition",
          "serverHost": "172.31.3.192",
          "netProtocol": "TCP",
          "dbProtocol": "Postgres 3.0",
          "type": "record",
          "errorMessage": null
        }
      ]
    },
   "key":"decryption-key"
}
```

Example Activity event record of an Aurora MySQL CREATE TABLE statement

The following example shows a `CREATE TABLE` statement for Aurora MySQL. The operation is represented as two separate event records. One event has `"class":"MAIN"`. The other event has `"class":"AUX"`. The messages might arrive in any order. The `logTime` field of the `MAIN` event is always earlier than the `logTime` fields of any corresponding `AUX` events.

The following example shows the event with a `class` value of `MAIN`.

```
{
  "type":"DatabaseActivityMonitoringRecord",
  "clusterId":"cluster-some_id",
  "instanceId":"db-some_id",
  "databaseActivityEventList":[
    {
      "logTime":"2020-05-22 18:07:12.250221+00",
      "type":"record",
      "clientApplication":null,
      "pid":2830,
      "dbUserName":"master",
      "databaseName":"test",
      "remoteHost":"localhost",
      "remotePort":"11054",
      "command":"QUERY",
      "commandText":"CREATE TABLE test1 (id INT)",
      "paramList":null,
      "objectType":"TABLE",
      "objectName":"test1",
      "statementId":65459278,
      "substatementId":1,
      "exitCode":"0",
      "sessionId":"725118",
      "rowCount":0,
      "serverHost":"master",
      "serverType":"MySQL",
      "serviceName":"Amazon Aurora MySQL",
      "serverVersion":"MySQL 5.7.12",
      "startTime":"2020-05-22 18:07:12.226384+00",
      "endTime":"2020-05-22 18:07:12.250222+00",
      "transactionId":"0",
      "dbProtocol":"MySQL",
      "netProtocol":"TCP",
      "errorMessage":"",
      "class":"MAIN"
    }
  ]
}
```

The following example shows the corresponding event with a `class` value of `AUX`.

```
{
  "type":"DatabaseActivityMonitoringRecord",
  "clusterId":"cluster-some_id",
  "instanceId":"db-some_id",
  "databaseActivityEventList":[
    {
      "logTime":"2020-05-22 18:07:12.247182+00",
      "type":"record",
      "clientApplication":null,
      "pid":2830,
      "dbUserName":"master",
      "databaseName":"test",
      "remoteHost":"localhost",
      "remotePort":"11054",
      "command":"CREATE",
      "commandText":"test1",
      "paramList":null,
      "objectType":"TABLE",
      "objectName":"test1",
      "statementId":65459278,
      "substatementId":2,
      "exitCode":"",
      "sessionId":"725118",
      "rowCount":0,
      "serverHost":"master",
      "serverType":"MySQL",
      "serviceName":"Amazon Aurora MySQL",
      "serverVersion":"MySQL 5.7.12",
      "startTime":"2020-05-22 18:07:12.226384+00",
      "endTime":"2020-05-22 18:07:12.247182+00",
      "transactionId":"0",
      "dbProtocol":"MySQL",
      "netProtocol":"TCP",
      "errorMessage":"",
      "class":"AUX"
    }
  ]
}
```

Example Activity event record of an Aurora PostgreSQL SELECT statement

The following example shows a `SELECT` event .

```
{
  "type":"DatabaseActivityMonitoringRecords",
  "version":"1.1",
  "databaseActivityEvents":
    {
      "type":"DatabaseActivityMonitoringRecord",
      "clusterId":"cluster-4HNY5V4RRNPKKYB7ICFKE5JBQQ",
      "instanceId":"db-FZJTMYKCXQBUUZ6VLU7NW3ITCM",
      "databaseActivityEventList":[
        {
          "startTime": "2019-05-24 00:39:49.920564+00",
          "logTime": "2019-05-24 00:39:49.940668+00",
          "statementId": 6,
          "substatementId": 1,
          "objectType": "TABLE",
          "command": "SELECT",
          "objectName": "public.my_table",
          "databaseName": "postgres",
          "dbUserName": "rdsadmin",
          "remoteHost": "172.31.3.195",
          "remotePort": "34534",
          "sessionId": "5ce73c6f.7e64",
          "rowCount": 10,
          "commandText": "select * from my_table;",
          "paramList": [],
          "pid": 32356,
          "clientApplication": "psql",
          "exitCode": null,
          "class": "READ",
          "serverVersion": "2.3.1",
          "serverType": "PostgreSQL",
          "serviceName": "Amazon Aurora PostgreSQL-Compatible edition",
          "serverHost": "172.31.3.192",
          "netProtocol": "TCP",
          "dbProtocol": "Postgres 3.0",
          "type": "record",
          "errorMessage": null
        }
      ]
    },
   "key":"decryption-key"
}
```

```
{
    "type": "DatabaseActivityMonitoringRecord",
    "clusterId": "",
    "instanceId": "db-4JCWQLUZVFYP7DIWP6JVQ77O3Q",
    "databaseActivityEventList": [
        {
            "class": "TABLE",
            "clientApplication": "Microsoft SQL Server Management Studio - Query",
            "command": "SELECT",
            "commandText": "select * from [testDB].[dbo].[TestTable]",
            "databaseName": "testDB",
            "dbProtocol": "SQLSERVER",
            "dbUserName": "test",
            "endTime": null,
            "errorMessage": null,
            "exitCode": 1,
            "logTime": "2022-10-06 21:24:59.9422268+00",
            "netProtocol": null,
            "objectName": "TestTable",
            "objectType": "TABLE",
            "paramList": null,
            "pid": null,
            "remoteHost": "local machine",
            "remotePort": null,
            "rowCount": 0,
            "serverHost": "172.31.30.159",
            "serverType": "SQLSERVER",
            "serverVersion": "15.00.4073.23.v1.R1",
            "serviceName": "sqlserver-ee",
            "sessionId": 62,
            "startTime": null,
            "statementId": "0x03baed90412f564fad640ebe51f89b99",
            "substatementId": 1,
            "transactionId": "4532935",
            "type": "record",
            "engineNativeAuditFields": {
                "target_database_principal_id": 0,
                "target_server_principal_id": 0,
                "target_database_principal_name": "",
                "server_principal_id": 2,
                "user_defined_information": "",
                "response_rows": 0,
                "database_principal_name": "dbo",
                "target_server_principal_name": "",
                "schema_name": "dbo",
                "is_column_permission": true,
                "object_id": 581577110,
                "server_instance_name": "EC2AMAZ-NFUJJNO",
                "target_server_principal_sid": null,
                "additional_information": "",
                "duration_milliseconds": 0,
                "permission_bitmask": "0x00000000000000000000000000000001",
                "data_sensitivity_information": "",
                "session_server_principal_name": "test",
                "connection_id": "AD3A5084-FB83-45C1-8334-E923459A8109",
                "audit_schema_version": 1,
                "database_principal_id": 1,
                "server_principal_sid":
 "0x010500000000000515000000bdc2795e2d0717901ba6998cf4010000",
                "user_defined_event_id": 0,
                "host_name": "EC2AMAZ-NFUJJNO"
            }
        }
    ]
}
```

Example Activity event record of an Aurora MySQL SELECT statement

The following example shows a SELECT event.

The following example shows the event with a class value of MAIN.

```
{
  "type":"DatabaseActivityMonitoringRecord",
  "clusterId":"cluster-some_id",
  "instanceId":"db-some_id",
  "databaseActivityEventList":[
    {
      "logTime":"2020-05-22 18:29:57.986467+00",
      "type":"record",
      "clientApplication":null,
      "pid":2830,
      "dbUserName":"master",
      "databaseName":"test",
      "remoteHost":"localhost",
      "remotePort":"11054",
      "command":"QUERY",
      "commandText":"SELECT * FROM test1 WHERE id < 28",
      "paramList":null,
      "objectType":"TABLE",
      "objectName":"test1",
      "statementId":65469218,
      "substatementId":1,
      "exitCode":"0",
      "sessionId":"726571",
      "rowCount":2,
      "serverHost":"master",
      "serverType":"MySQL",
      "serviceName":"Amazon Aurora MySQL",
      "serverVersion":"MySQL 5.7.12",
      "startTime":"2020-05-22 18:29:57.986364+00",
      "endTime":"2020-05-22 18:29:57.986467+00",
      "transactionId":"0",
      "dbProtocol":"MySQL",
      "netProtocol":"TCP",
      "errorMessage":"",
      "class":"MAIN"
    }
  ]
}
```

The following example shows the corresponding event with a `class` value of AUX.

```
{
  "type":"DatabaseActivityMonitoringRecord",
  "instanceId":"db-some_id",
  "databaseActivityEventList":[
    {
      "logTime":"2020-05-22 18:29:57.986399+00",
      "type":"record",
      "clientApplication":null,
      "pid":2830,
      "dbUserName":"master",
      "databaseName":"test",
      "remoteHost":"localhost",
      "remotePort":"11054",
      "command":"READ",
      "commandText":"test1",
      "paramList":null,
      "objectType":"TABLE",
      "objectName":"test1",
      "statementId":65469218,
      "substatementId":2,
      "exitCode":"",
      "sessionId":"726571",
      "rowCount":0,
      "serverHost":"master",
      "serverType":"MySQL",
      "serviceName":"Amazon Aurora MySQL",
      "serverVersion":"MySQL 5.7.12",
      "startTime":"2020-05-22 18:29:57.986364+00",
      "endTime":"2020-05-22 18:29:57.986399+00",
      "transactionId":"0",
      "dbProtocol":"MySQL",
      "netProtocol":"TCP",
      "errorMessage":"",
      "class":"AUX"
    }
  ]
}
```

DatabaseActivityMonitoringRecords JSON object

The database activity event records are in a JSON object that contains the following information.

JSON Field	Data Type	Description
`type`	string	The type of JSON record. The value is `DatabaseActivityMonitoringRecords` .
`version`	string	The version of the database activity monitoring records. The version of the generated database activity records depends on the engine version of the DB cluster: - Version 1.1 database activity records are generated for Aurora PostgreSQL DB clusters running the engine versions 10.10 and later minor versions and engine versions 11.5 and later. - Version 1.0 database activity records are generated for Aurora PostgreSQL DB clusters running the engine versions 10.7 and 11.4. All of the following fields are in both version 1.0 and version 1.1 except where specifically noted.
databaseActivityEvents	string	A JSON object that contains the activity events.
key	string	An encryption key that you use to decrypt the databaseActivityEventList

databaseActivityEvents JSON Object

The `databaseActivityEvents` JSON object contains the following information.

Top-level fields in JSON record

Each event in the audit log is wrapped inside a record in JSON format. This record contains the following fields.

type

This field always has the value `DatabaseActivityMonitoringRecords`.

version

This field represents the version of the database activity stream data protocol or contract. It defines which fields are available.

Version 1.0 represents the original data activity streams support for Aurora PostgreSQL versions 10.7 and 11.4. Version 1.1 represents the data activity streams support for Aurora PostgreSQL versions 10.10 and higher and Aurora PostgreSQL 11.5 and higher. Version 1.1 includes the additional fields `errorMessage` and `startTime`. Version 1.2 represents the data activity streams support for Aurora MySQL 2.08 and higher. Version 1.2 includes the additional fields `endTime` and `transactionId`.

databaseActivityEvents

An encrypted string representing one or more activity events. It's represented as a base64 byte array. When you decrypt the string, the result is a record in JSON format with fields as shown in the examples in this section.

key

The encrypted data key used to encrypt the `databaseActivityEvents` string. This is the same AWS KMS key that you provided when you started the database activity stream.

The following example shows the format of this record.

```
{
  "type":"DatabaseActivityMonitoringRecords",
  "version":"1.1",
  "databaseActivityEvents":"encrypted audit records",
  "key":"encrypted key"
}
```

Take the following steps to decrypt the contents of the `databaseActivityEvents` field:

1. Decrypt the value in the `key` JSON field using the KMS key you provided when starting database activity stream. Doing so returns the data encryption key in clear text.
2. Base64-decode the value in the `databaseActivityEvents` JSON field to obtain the ciphertext, in binary format, of the audit payload.
3. Decrypt the binary ciphertext with the data encryption key that you decoded in the first step.
4. Decompress the decrypted payload.
   - The encrypted payload is in the `databaseActivityEvents` field.
   - The `databaseActivityEventList` field contains an array of audit records. The `type` fields in the array can be `record` or `heartbeat`.

The audit log activity event record is a JSON object that contains the following information.

JSON Field	Data Type	Description
`type`	string	The type of JSON record. The value is `DatabaseActivityMonitoringRecord` .
`clusterId`	string	The DB cluster resource identifier. It corresponds to the DB cluster attribute `DbClusterResourceId` .
`instanceId`	string	The DB instance resource identifier. It corresponds to the DB instance attribute `DbiResourceId` .
databaseActivityEventList	string	An array of activity audit records or heartbeat messages.

databaseActivityEventList JSON array

The audit log payload is an encrypted `databaseActivityEventList` JSON array. The following tables lists alphabetically the fields for each activity event in the decrypted `DatabaseActivityEventList` array of an audit log. The fields differ depending on whether you use Aurora PostgreSQL or Aurora MySQL. Consult the table that applies to your database engine.

⚠ **Important**

The event structure is subject to change. Aurora might add new fields to activity events in the future. In applications that parse the JSON data, make sure that your code can ignore or take appropriate actions for unknown field names.

databaseActivityEventList fields for Aurora PostgreSQL

Field	Data Type	Description
`class`	string	The class of activity event. Valid values for Aurora PostgreSQL are the following: • `ALL` • `CONNECT` – A connect or disconnect event. • `DDL` – A DDL statement that is not included in the list of statements for the `ROLE` class. • `FUNCTION` – A function call or a `DO` block. • `MISC` – A miscellaneous command such as `DISCARD`, `FETCH`, `CHECKPOINT` , or `VACUUM`. • `NONE` • `READ` – A `SELECT` or `COPY` statement when the source is a relation or a query. • `ROLE` – A statement related to roles and privileges including `GRANT`, `REVOKE`, and `CREATE/ALTER/DROP ROLE`. • `WRITE` – An `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, or `COPY` statement when the destination is a relation.
`clientApplication`	string	The application the client used to connect as reported by the client. The client doesn't have to provide this information, so the value can be null.
`command`	string	The name of the SQL command without any command details.

Field	Data Type	Description
`commandText`	string	The actual SQL statement passed in by the user. For Aurora PostgreSQL, the value is identical to the original SQL statement. This field is used for all types of records except for connect or disconnect records, in which case the value is null. **Important** The full SQL text of each statement is visible in the activity stream audit log, including any sensitive data. However, database user passwords are redacted if Aurora can determine them from the context, such as in the following SQL statement. `ALTER ROLE role-name WITH password`
`databaseName`	string	The database to which the user connected.
`dbProtocol`	string	The database protocol, for example `Postgres 3.0`.
`dbUserName`	string	The database user with which the client authenticated.

Field	Data Type	Description
`errorMessage` (version 1.1 database activity records only)	string	If there was any error, this field is populated with the error message that would've been generated by the DB server. The `errorMessage` value is null for normal statements that didn't result in an error. An error is defined as any activity that would produce a client-visible PostgreSQL error log event at a severity level of `ERROR` or greater. For more information, see PostgreSQL Message Severity Levels. For example, syntax errors and query cancellations generate an error message. Internal PostgreSQL server errors such as background checkpointer process errors do not generate an error message. However, records for such events are still emitted regardless of the setting of the log severity level. This prevents attackers from turning off logging to attempt avoiding detection. See also the `exitCode` field.
`exitCode`	int	A value used for a session exit record. On a clean exit, this contains the exit code. An exit code can't always be obtained in some failure scenarios. Examples are if PostgreSQL does an `exit()` or if an operator performs a command such as `kill -9`. If there was any error, the `exitCode` field shows the SQL error code, `SQLSTATE`, as listed in PostgreSQL Error Codes. See also the `errorMessage` field.
`logTime`	string	A timestamp as recorded in the auditing code path. This represents the SQL statement execution end time. See also the `startTime` field.
`netProtocol`	string	The network communication protocol.

Field	Data Type	Description
`objectName`	string	The name of the database object if the SQL statement is operating on one. This field is used only where the SQL statement operates on a database object. If the SQL statement is not operating on an object, this value is null.
`objectType`	string	The database object type such as table, index, view, and so on. This field is used only where the SQL statement operates on a database object. If the SQL statement is not operating on an object, this value is null. Valid values include the following: • `COMPOSITE TYPE` • `FOREIGN TABLE` • `FUNCTION` • `INDEX` • `MATERIALIZED VIEW` • `SEQUENCE` • `TABLE` • `TOAST TABLE` • `VIEW` • `UNKNOWN`
`paramList`	string	An array of comma-separated parameters passed to the SQL statement. If the SQL statement has no parameters, this value is an empty array.
`pid`	int	The process ID of the backend process that is allocated for serving the client connection.
`remoteHost`	string	Either the client IP address or hostname. For Aurora PostgreSQL, which one is used depends on the database's `log_hostname` parameter setting.
`remotePort`	string	The client port number.

Field	Data Type	Description
`rowCount`	int	The number of rows returned by the SQL statement. For example, if a SELECT statement returns 10 rows, rowCount is 10. For INSERT or UPDATE statements, rowCount is 0.
`serverHost`	string	The database server host IP address.
`serverType`	string	The database server type, for example `PostgreSQL` .
`serverVersion`	string	The database server version, for example `2.3.1` for Aurora PostgreSQL.
`serviceName`	string	The name of the service, for example `Amazon Aurora PostgreSQL-Compatible edition` .
`sessionId`	int	A pseudo-unique session identifier.
`sessionId`	int	A pseudo-unique session identifier.
`startTime` (version 1.1 database activity records only)	string	The time when execution began for the SQL statement. To calculate the approximate execution time of the SQL statement, use `logTime - startTime` . See also the `logTime` field.
`statementId`	int	An identifier for the client's SQL statement. The counter is at the session level and increments with each SQL statement entered by the client.
`substatementId`	int	An identifier for a SQL substatement. This value counts the contained substatements for each SQL statement identified by the `statementId` field.
`type`	string	The event type. Valid values are `record` or `heartbeat` .

databaseActivityEventList fields for Aurora MySQL

Field	Data Type	Description
`class`	string	The class of activity event. Valid values for Aurora MySQL are the following: • `MAIN` – The primary event representing a SQL statement. • `AUX` – A supplemental event containing additional details. For example, a statement that renames an object might have an event with class `AUX` that reflects the new name. To find `MAIN` and `AUX` events corresponding to the same statement, check for different events that have the same values for the `pid` field and for the `statementId` field.
`clientApplication`	string	The application the client used to connect as reported by the client. The client doesn't have to provide this information, so the value can be null.
`command`	string	The general category of the SQL statement. The values for this field depend on the value of `class`. The values when `class` is `MAIN` include the following: • `CONNECT` – When a client session is connected. • `QUERY` – A SQL statement. Accompanied by one or more events with a `class` value of `AUX`. • `DISCONNECT` – When a client session is disconnected. • `FAILED_CONNECT` – When a client attempts to connect but isn't able to. • `CHANGEUSER` – A state change that's part of the MySQL network protocol, not from a statement that you issue. The values when `class` is `AUX` include the following:

Field	Data Type	Description
		- READ – A SELECT or COPY statement when the source is a relation or a query. - WRITE – An INSERT, UPDATE, DELETE, TRUNCATE, or COPY statement when the destination is a relation. - DROP – Deleting an object. - CREATE – Creating an object. - RENAME – Renaming an object. - ALTER – Changing the properties of an object.

Field	Data Type	Description
`commandText`	string	For events with a `class` value of `MAIN`, this field represents the actual SQL statement passed in by the user. This field is used for all types of records except for connect or disconnect records, in which case the value is null. For events with a `class` value of `AUX`, this field contains supplemental information about the objects involved in the event. For Aurora MySQL, characters such as quotation marks are preceded by a backslash, representing an escape character. **Important** The full SQL text of each statement is visible in the audit log, including any sensitive data. However, database user passwords are redacted if Aurora can determine them from the context, such as in the following SQL statement. `mysql> SET PASSWORD = 'my-password ';` **Note** Specify a password other than the prompt shown here as a security best practice.
`databaseName`	string	The database to which the user connected.
`dbProtocol`	string	The database protocol. Currently, this value is always `MySQL` for Aurora MySQL.
`dbUserName`	string	The database user with which the client authenticated.

Field	Data Type	Description
endTime (version 1.2 database activity records only)	string	The time when execution ended for the SQL statement. It is represented in Coordinated Universal Time (UTC) format. To calculate the execution time of the SQL statement, use endTime - startTime . See also the startTime field.
errorMessage (version 1.1 database activity records only)	string	If there was any error, this field is populated with the error message that would've been generated by the DB server. The errorMessage value is null for normal statements that didn't result in an error. An error is defined as any activity that would produce a client-visible MySQL error log event at a severity level of ERROR or greater. For more information, see The Error Log in the *MySQL Reference Manual*. For example, syntax errors and query cancellations generate an error message. Internal MySQL server errors such as background checkpointer process errors do not generate an error message. However, records for such events are still emitted regardless of the setting of the log severity level. This prevents attackers from turning off logging to attempt avoiding detection. See also the exitCode field.
exitCode	int	A value used for a session exit record. On a clean exit, this contains the exit code. An exit code can't always be obtained in some failure scenarios. In such cases, this value might be zero or might be blank.
logTime	string	A timestamp as recorded in the auditing code path. It is represented in Coordinated Universal Time (UTC) format. For the most accurate way to calculate statement duration, see the startTime and endTime fields.

Field	Data Type	Description
`netProtocol`	string	The network communication protocol. Currently, this value is always TCP for Aurora MySQL.
`objectName`	string	The name of the database object if the SQL statement is operating on one. This field is used only where the SQL statement operates on a database object. If the SQL statement isn't operating on an object, this value is blank. To construct the fully qualified name of the object, combine `databaseName` and `objectName` . If the query involves multiple objects, this field can be a comma-separated list of names.
`objectType`	string	The database object type such as table, index, and so on. This field is used only where the SQL statement operates on a database object. If the SQL statement is not operating on an object, this value is null. Valid values for Aurora MySQL include the following: • `INDEX` • `TABLE` • `UNKNOWN`
`paramList`	string	This field isn't used for Aurora MySQL and is always null.
`pid`	int	The process ID of the backend process that is allocated for serving the client connection. When the database server is restarted, the `pid` changes and the counter for the `statementId` field starts over.
`remoteHost`	string	Either the IP address or hostname of the client that issued the SQL statement. For Aurora MySQL, which one is used depends on the database's `skip_name_resolve` parameter setting. The value `localhost` indicates activity from the `rdsadmin` special user.

Field	Data Type	Description
`remotePort`	string	The client port number.
`rowCount`	int	The number of table rows affected or retrieved by the SQL statement. This field is used only for SQL statements that are data manipulation language (DML) statements. If the SQL statement is not a DML statement, this value is null.
`serverHost`	string	The database server instance identifier. This value is represented differently for Aurora MySQL than for Aurora PostgreSQL. Aurora PostgreSQL uses an IP address instead of an identifier.
`serverType`	string	The database server type, for example `MySQL`.
`serverVersion`	string	The database server version. Currently, this value is always `MySQL 5.7.12` for Aurora MySQL.
`serviceName`	string	The name of the service. Currently, this value is always `Amazon Aurora MySQL` for Aurora MySQL.
`sessionId`	int	A pseudo-unique session identifier.
`startTime` (version 1.1 database activity records only)	string	The time when execution began for the SQL statement. It is represented in Coordinated Universal Time (UTC) format. To calculate the execution time of the SQL statement, use `endTime - startTime` . See also the `endTime` field.
`statementId`	int	An identifier for the client's SQL statement. The counter increments with each SQL statement entered by the client. The counter is reset when the DB instance is restarted.
`substatementId`	int	An identifier for a SQL substatement. This value is 1 for events with class `MAIN` and 2 for events with class AUX. Use the `statementId` field to identify all the events generated by the same statement.

Field	Data Type	Description
transactionId (version 1.2 database activity records only)	int	An identifier for a transaction.
type	string	The event type. Valid values are record or heartbeat .

Processing a database activity stream using the AWS SDK

You can programmatically process an activity stream by using the AWS SDK. The following are fully functioning Java and Python examples of how you might process the Kinesis data stream.

Java

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.InetAddress;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.Security;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.zip.GZIPInputStream;

import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.SecretKeySpec;

import com.amazonaws.auth.AWSStaticCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import
 com.amazonaws.services.kinesis.clientlibrary.exceptions.InvalidStateException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ShutdownException;
import com.amazonaws.services.kinesis.clientlibrary.exceptions.ThrottlingException;
import com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessor;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorCheckpointer;
import
 com.amazonaws.services.kinesis.clientlibrary.interfaces.IRecordProcessorFactory;
import
 com.amazonaws.services.kinesis.clientlibrary.lib.worker.InitialPositionInStream;
import
 com.amazonaws.services.kinesis.clientlibrary.lib.worker.KinesisClientLibConfiguration;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.ShutdownReason;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker;
import com.amazonaws.services.kinesis.clientlibrary.lib.worker.Worker.Builder;
import com.amazonaws.services.kinesis.model.Record;
import com.amazonaws.services.kms.AWSKMS;
import com.amazonaws.services.kms.AWSKMSClientBuilder;
import com.amazonaws.services.kms.model.DecryptRequest;
import com.amazonaws.services.kms.model.DecryptResult;
import com.amazonaws.util.Base64;
import com.amazonaws.util.IOUtils;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.annotations.SerializedName;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class DemoConsumer {

    private static final String STREAM_NAME = "aws-rds-das-[cluster-external-
resource-id]";
    private static final String APPLICATION_NAME = "AnyApplication"; //unique
 application name for dynamo table generation that holds kinesis shard tracking
    private static final String AWS_ACCESS_KEY =
 "[AWS_ACCESS_KEY_TO_ACCESS_KINESIS]";
    private static final String AWS_SECRET_KEY =
 "[AWS_SECRET_KEY_TO_ACCESS_KINESIS]";
    private static final String DBC_RESOURCE_ID = "[cluster-external-resource-id]";
    private static final String REGION_NAME = "[region-name]"; //us-east-1, us-
east-2...
    private static final BasicAWSCredentials CREDENTIALS = new
 BasicAWSCredentials(AWS_ACCESS_KEY, AWS_SECRET_KEY);
```

```
    private static final AWSStaticCredentialsProvider CREDENTIALS_PROVIDER = new 
AWSStaticCredentialsProvider(CREDENTIALS);

    private static final AwsCrypto CRYPTO = new AwsCrypto();
    private static final AWSKMS KMS = AWSKMSClientBuilder.standard()
            .withRegion(REGION_NAME)
            .withCredentials(CREDENTIALS_PROVIDER).build();

    class Activity {
        String type;
        String version;
        String databaseActivityEvents;
        String key;
    }

    class ActivityEvent {
        @SerializedName("class") String _class;
        String clientApplication;
        String command;
        String commandText;
        String databaseName;
        String dbProtocol;
        String dbUserName;
        String endTime;
        String errorMessage;
        String exitCode;
        String logTime;
        String netProtocol;
        String objectName;
        String objectType;
        List<String> paramList;
        String pid;
        String remoteHost;
        String remotePort;
        String rowCount;
        String serverHost;
        String serverType;
        String serverVersion;
        String serviceName;
        String sessionId;
        String startTime;
        String statementId;
        String substatementId;
        String transactionId;
```

```
       String type;
   }

   class ActivityRecords {
       String type;
       String clusterId;
       String instanceId;
       List<ActivityEvent> databaseActivityEventList;
   }

   static class RecordProcessorFactory implements IRecordProcessorFactory {
       @Override
       public IRecordProcessor createProcessor() {
           return new RecordProcessor();
       }
   }

   static class RecordProcessor implements IRecordProcessor {

       private static final long BACKOFF_TIME_IN_MILLIS = 3000L;
       private static final int PROCESSING_RETRIES_MAX = 10;
       private static final long CHECKPOINT_INTERVAL_MILLIS = 60000L;
       private static final Gson GSON = new
GsonBuilder().serializeNulls().create();

       private static final Cipher CIPHER;
       static {
           Security.insertProviderAt(new BouncyCastleProvider(), 1);
           try {
               CIPHER = Cipher.getInstance("AES/GCM/NoPadding", "BC");
           } catch (NoSuchAlgorithmException | NoSuchPaddingException |
NoSuchProviderException e) {
               throw new ExceptionInInitializerError(e);
           }
       }

       private long nextCheckpointTimeInMillis;

       @Override
       public void initialize(String shardId) {
       }

       @Override
       public void processRecords(final List<Record> records, final
IRecordProcessorCheckpointer checkpointer) {
           for (final Record record : records) {
               processSingleBlob(record.getData());
           }

           if (System.currentTimeMillis() > nextCheckpointTimeInMillis) {
               checkpoint(checkpointer);
               nextCheckpointTimeInMillis = System.currentTimeMillis() +
CHECKPOINT_INTERVAL_MILLIS;
           }
       }

       @Override
       public void shutdown(IRecordProcessorCheckpointer checkpointer,
ShutdownReason reason) {
           if (reason == ShutdownReason.TERMINATE) {
               checkpoint(checkpointer);
           }
       }

       private void processSingleBlob(final ByteBuffer bytes) {
           try {
               // JSON $Activity
               final Activity activity = GSON.fromJson(new String(bytes.array(),
StandardCharsets.UTF_8), Activity.class);

               // Base64.Decode
               final byte[] decoded =
Base64.decode(activity.databaseActivityEvents);
               final byte[] decodedDataKey = Base64.decode(activity.key);

               Map<String, String> context = new HashMap<>();
               context.put("aws:rds:dbc-id", DBC_RESOURCE_ID);

               // Decrypt
               final DecryptRequest decryptRequest = new DecryptRequest()

.withCiphertextBlob(ByteBuffer.wrap(decodedDataKey)).withEncryptionContext(context);
               final DecryptResult decryptResult = KMS.decrypt(decryptRequest);
               final byte[] decrypted = decrypt(decoded,
getByteArray(decryptResult.getPlaintext()));

               // GZip Decompress
               final byte[] decompressed = decompress(decrypted);
               // JSON $ActivityRecords
               final ActivityRecords activityRecords = GSON.fromJson(new 
String(decompressed, StandardCharsets.UTF_8), ActivityRecords.class);

               // Iterate throught $ActivityEvents
               for (final ActivityEvent event : 
activityRecords.databaseActivityEventList) {
                   System.out.println(GSON.toJson(event));
               }
           } catch (Exception e) {
               // Handle error.
               e.printStackTrace();
           }
       }

       private static byte[] decompress(final byte[] src) throws IOException {
           ByteArrayInputStream byteArrayInputStream = new 
ByteArrayInputStream(src);
           GZIPInputStream gzipInputStream = new 
GZIPInputStream(byteArrayInputStream);
           return IOUtils.toByteArray(gzipInputStream);
       }

       private void checkpoint(IRecordProcessorCheckpointer checkpointer) {
           for (int i = 0; i < PROCESSING_RETRIES_MAX; i++) {
               try {
                   checkpointer.checkpoint();
                   break;
               } catch (ShutdownException se) {
                   // Ignore checkpoint if the processor instance has been shutdown 
(fail over).
                   System.out.println("Caught shutdown exception, skipping 
checkpoint." + se);
                   break;
               } catch (ThrottlingException e) {
                   // Backoff and re-attempt checkpoint upon transient failures
                   if (i >= (PROCESSING_RETRIES_MAX - 1)) {
                       System.out.println("Checkpoint failed after " + (i + 1) + 
"attempts." + e);
                       break;
                   } else {
                       System.out.println("Transient issue when checkpointing - 
attempt " + (i + 1) + " of " + PROCESSING_RETRIES_MAX + e);
```

```
                   }
               } catch (InvalidStateException e) {
                   // This indicates an issue with the DynamoDB table (check for 
table, provisioned IOPS).
                   System.out.println("Cannot save checkpoint to the DynamoDB table 
used by the Amazon Kinesis Client Library." + e);
                   break;
               }
               try {
                   Thread.sleep(BACKOFF_TIME_IN_MILLIS);
               } catch (InterruptedException e) {
                   System.out.println("Interrupted sleep" + e);
               }
           }
       }
   }

   private static byte[] decrypt(final byte[] decoded, final byte[] decodedDataKey) 
throws IOException {
       // Create a JCE master key provider using the random key and an AES-GCM 
encryption algorithm
       final JceMasterKey masterKey = JceMasterKey.getInstance(new 
SecretKeySpec(decodedDataKey, "AES"),
               "BC", "DataKey", "AES/GCM/NoPadding");
       try (final CryptoInputStream<JceMasterKey> decryptingStream = 
CRYPTO.createDecryptingStream(masterKey, new ByteArrayInputStream(decoded));
            final ByteArrayOutputStream out = new ByteArrayOutputStream()) {
           IOUtils.copy(decryptingStream, out);
           return out.toByteArray();
       }
   }

   public static void main(String[] args) throws Exception {
       final String workerId = InetAddress.getLocalHost().getCanonicalHostName() + 
":" + UUID.randomUUID();
       final KinesisClientLibConfiguration kinesisClientLibConfiguration =
               new KinesisClientLibConfiguration(APPLICATION_NAME, STREAM_NAME, 
CREDENTIALS_PROVIDER, workerId);
       
kinesisClientLibConfiguration.withInitialPositionInStream(InitialPositionInStream.LATEST);
       kinesisClientLibConfiguration.withRegionName(REGION_NAME);
       final Worker worker = new Builder()
               .recordProcessorFactory(new RecordProcessorFactory())
               .config(kinesisClientLibConfiguration)
```

```
                .build();

        System.out.printf("Running %s to process stream %s as worker %s...\n",
 APPLICATION_NAME, STREAM_NAME, workerId);

        try {
            worker.run();
        } catch (Throwable t) {
            System.err.println("Caught throwable while processing data.");
            t.printStackTrace();
            System.exit(1);
        }
        System.exit(0);
    }

    private static byte[] getByteArray(final ByteBuffer b) {
        byte[] byteArray = new byte[b.remaining()];
        b.get(byteArray);
        return byteArray;
    }
}
```

Python

```
import base64
import json
import zlib
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy
from aws_encryption_sdk.internal.crypto import WrappingKey
from aws_encryption_sdk.key_providers.raw import RawMasterKeyProvider
from aws_encryption_sdk.identifiers import WrappingAlgorithm, EncryptionKeyType
import boto3

REGION_NAME = '<region>'                    # us-east-1
RESOURCE_ID = '<external-resource-id>'      # cluster-ABCD123456
STREAM_NAME = 'aws-rds-das-' + RESOURCE_ID  # aws-rds-das-cluster-ABCD123456

enc_client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_AL

class MyRawMasterKeyProvider(RawMasterKeyProvider):
    provider_id = "BC"
    def __new__(cls, *args, **kwargs):
        obj = super(RawMasterKeyProvider, cls).__new__(cls)
        return obj

    def __init__(self, plain_key):
        RawMasterKeyProvider.__init__(self)
        self.wrapping_key =
 WrappingKey(wrapping_algorithm=WrappingAlgorithm.AES_256_GCM_IV12_TAG16_NO_PADDING,
                                        wrapping_key=plain_key,
 wrapping_key_type=EncryptionKeyType.SYMMETRIC)

    def _get_raw_key(self, key_id):
        return self.wrapping_key


def decrypt_payload(payload, data_key):
    my_key_provider = MyRawMasterKeyProvider(data_key)
    my_key_provider.add_master_key("DataKey")
    decrypted_plaintext, header = enc_client.decrypt(
        source=payload,

 materials_manager=aws_encryption_sdk.materials_managers.default.DefaultCryptoMaterialsManag
    return decrypted_plaintext


def decrypt_decompress(payload, key):
    decrypted = decrypt_payload(payload, key)
    return zlib.decompress(decrypted, zlib.MAX_WBITS + 16)


def main():
    session = boto3.session.Session()
    kms = session.client('kms', region_name=REGION_NAME)
    kinesis = session.client('kinesis', region_name=REGION_NAME)

    response = kinesis.describe_stream(StreamName=STREAM_NAME)
    shard_iters = []
    for shard in response['StreamDescription']['Shards']:
        shard_iter_response = kinesis.get_shard_iterator(StreamName=STREAM_NAME,
 ShardId=shard['ShardId'],

 ShardIteratorType='LATEST')
        shard_iters.append(shard_iter_response['ShardIterator'])
    while len(shard_iters) > 0:
        next_shard_iters = []
        for shard_iter in shard_iters:
            response = kinesis.get_records(ShardIterator=shard_iter, Limit=10000)
            for record in response['Records']:
                record_data = record['Data']
                record_data = json.loads(record_data)
                payload_decoded =
 base64.b64decode(record_data['databaseActivityEvents'])
                data_key_decoded = base64.b64decode(record_data['key'])
                data_key_decrypt_result =
 kms.decrypt(CiphertextBlob=data_key_decoded,

 EncryptionContext={'aws:rds:dbc-id': RESOURCE_ID})
                print (decrypt_decompress(payload_decoded,
 data_key_decrypt_result['Plaintext']))
            if 'NextShardIterator' in response:
                next_shard_iters.append(response['NextShardIterator'])
        shard_iters = next_shard_iters


if __name__ == '__main__':
    main()
```

Managing access to database activity streams

Any user with appropriate AWS Identity and Access Management (IAM) role privileges for database activity streams can create, start, stop, and modify the activity stream settings for a DB cluster. These actions are included in the audit log of the stream. For best compliance practices, we recommend that you don't provide these privileges to DBAs.

You set access to database activity streams using IAM policies. For more information about Aurora authentication, see Identity and access management for Amazon Aurora. For more information about creating IAM policies, see Creating and using an IAM policy for IAM database access.

Example Policy to allow configuring database activity streams

To give users fine-grained access to modify activity streams, use the service-specific operation context keys `rds:StartActivityStream` and `rds:StopActivityStream` in an IAM policy. The following IAM policy example allows a user or role to configure activity streams.

```
{
   "Version":"2012-10-17",
   "Statement":[
        {
            "Sid":"ConfigureActivityStreams",
            "Effect":"Allow",
            "Action": [
                "rds:StartActivityStream",
                "rds:StopActivityStream"
            ],
            "Resource":"*",
          }
     ]
}
```

Example Policy to allow starting database activity streams

The following IAM policy example allows a user or role to start activity streams.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"AllowStartActivityStreams",
            "Effect":"Allow",
            "Action":"rds:StartActivityStream",
            "Resource":"*"
        }
    ]
}
```

Example Policy to allow stopping database activity streams

The following IAM policy example allows a user or role to stop activity streams.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"AllowStopActivityStreams",
            "Effect":"Allow",
            "Action":"rds:StopActivityStream",
            "Resource":"*"
        }
     ]
}
```

Example Policy to deny starting database activity streams

The following IAM policy example prevents a user or role from starting activity streams.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"DenyStartActivityStreams",
            "Effect":"Deny",
            "Action":"rds:StartActivityStream",
            "Resource":"*"
        }
     ]
}
```

Example Policy to deny stopping database activity streams

The following IAM policy example prevents a user or role from stopping activity streams.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"DenyStopActivityStreams",
            "Effect":"Deny",
            "Action":"rds:StopActivityStream",
            "Resource":"*"
        }
    ]
}
```

Monitoring threats with Amazon GuardDuty RDS Protection

Amazon GuardDuty is a threat detection service that helps protect your accounts, containers, workloads, and the data within your AWS environment. Using machine learning (ML) models, and anomaly and threat detection capabilities, GuardDuty continuously monitors different log sources and runtime activity to identify and prioritize potential security risks and malicious activities in your environment.

GuardDuty RDS Protection analyzes and profiles login events for potential access threats to your Amazon Aurora databases. When you turn on RDS Protection, GuardDuty consumes RDS login events from your Aurora databases. RDS Protection monitors these events and profiles them for potential insider threats or external actors.

For more information about enabling GuardDuty RDS Protection, see GuardDuty RDS Protection in the *Amazon GuardDuty User Guide*.

When RDS Protection detects a potential threat, such as an unusual pattern in successful or failed login attempts, GuardDuty generates a new finding with details about the potentially compromised database. You can view the finding details in the finding summary section in the Amazon GuardDuty console. The finding details vary based on the finding type. The primary details, resource type and resource role, determine the kind of information available for any finding. For more information about the commonly available details for findings and the finding types, see Finding details and GuardDuty RDS Protection finding types respectively in the *Amazon GuardDuty User Guide*.

You can turn the RDS Protection feature on or off for any AWS account in any AWS Region where this feature is available. When RDS Protection isn't enabled, GuardDuty doesn't detect potentially compromised Aurora databases or provide details of the compromise.

An existing GuardDuty account can enable RDS Protection with a 30-day trial period. For a new GuardDuty account, RDS Protection is already enabled and included in the 30-day free trial period. For more information, see Estimating GuardDuty cost in the *Amazon GuardDuty User Guide*.

For information about the AWS Regions where GuardDuty doesn't yet support RDS Protection, see Region-specific feature availability in the *Amazon GuardDuty User Guide*.

The following table provides the Aurora database versions that GuardDuty RDS Protection supports:

Amazon Aurora DB engine	Supported engine versions
Aurora MySQL	• 2.10.2 or later • 3.02.1 or later
Aurora PostgreSQL	• 10.17 or later • 11.12 or later • 12.7 or later • 13.3 or later • 14.3 or later • 15.2 or later • 16.1 or later

Working with Amazon Aurora MySQL

Amazon Aurora MySQL is a fully managed, MySQL-compatible, relational database engine that combines the speed and reliability of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. Aurora MySQL is a drop-in replacement for MySQL and makes it simple and cost-effective to set up, operate, and scale your new and existing MySQL deployments, thus freeing you to focus on your business and applications. Amazon RDS provides administration for Aurora by handling routine database tasks such as provisioning, patching, backup, recovery, failure detection, and repair. Amazon RDS also provides push-button migration tools to convert your existing Amazon RDS for MySQL applications to Aurora MySQL.

Topics

Overview of Amazon Aurora MySQL

The following sections provide an overview of Amazon Aurora MySQL.

Topics

Amazon Aurora MySQL performance enhancements

Amazon Aurora includes performance enhancements to support the diverse needs of high-end commercial databases.

Fast insert

Fast insert accelerates parallel inserts sorted by primary key and applies specifically to `LOAD DATA` and `INSERT INTO ... SELECT ...` statements. Fast insert caches the position of a cursor in an index traversal while executing the statement. This avoids unnecessarily traversing the index again.

Fast insert is enabled only for regular InnoDB tables in Aurora MySQL version 3.03.2 and higher. This optimization doesn’t work for InnoDB temporary tables. It's disabled in Aurora MySQL version 2 for all 2.11 and 2.12 versions. Fast insert optimization works only if Adaptive Hash Index optimization is disabled.

You can monitor the following metrics to determine the effectiveness of fast insert for your DB cluster:

- `aurora_fast_insert_cache_hits`: A counter that is incremented when the cached cursor is successfully retrieved and verified.
- `aurora_fast_insert_cache_misses`: A counter that is incremented when the cached cursor is no longer valid and Aurora performs a normal index traversal.

You can retrieve the current value of the fast insert metrics using the following command:

```
mysql> show global status like 'Aurora_fast_insert%';
```

You will get output similar to the following:

```
+---------------------------------+-----------+
| Variable_name                   | Value     |
+---------------------------------+-----------+
| Aurora_fast_insert_cache_hits   | 3598300   |
| Aurora_fast_insert_cache_misses | 436401336 |
+---------------------------------+-----------+
```

Amazon Aurora MySQL and spatial data

The following list summarizes the main Aurora MySQL spatial features and explains how they correspond to spatial features in MySQL:

- Aurora MySQL version 2 supports the same spatial data types and spatial relation functions as MySQL 5.7. For more information about these data types and functions, see Spatial Data Types and Spatial Relation Functions in the MySQL 5.7 documentation.
- Aurora MySQL version 3 supports the same spatial data types and spatial relation functions as MySQL 8.0. For more information about these data types and functions, see Spatial Data Types and Spatial Relation Functions in the MySQL 8.0 documentation.
- Aurora MySQL supports spatial indexing on InnoDB tables. Spatial indexing improves query performance on large datasets for queries on spatial data. In MySQL, spatial indexing for InnoDB tables is available in MySQL 5.7 and 8.0.

  Aurora MySQL uses a different spatial indexing strategy from MySQL for high performance with spatial queries. The Aurora spatial index implementation uses a space-filling curve on a B-tree, which is intended to provide higher performance for spatial range scans than an R-tree.

  > **Note**
  >
  > In Aurora MySQL, a transaction on a table with a spatial index defined on a column with a spatial reference identifier (SRID) can't insert into an area selected for update by another transaction.

The following data definition language (DDL) statements are supported for creating indexes on columns that use spatial data types.

CREATE TABLE

You can use the SPATIAL INDEX keywords in a CREATE TABLE statement to add a spatial index to a column in a new table. Following is an example.

```
CREATE TABLE test (shape POLYGON NOT NULL, SPATIAL INDEX(shape));
```

ALTER TABLE

You can use the SPATIAL INDEX keywords in an ALTER TABLE statement to add a spatial index to a column in an existing table. Following is an example.

```
ALTER TABLE test ADD SPATIAL INDEX(shape);
```

CREATE INDEX

You can use the SPATIAL keyword in a CREATE INDEX statement to add a spatial index to a column in an existing table. Following is an example.

```
CREATE SPATIAL INDEX shape_index ON test (shape);
```

Aurora MySQL version 3 compatible with MySQL 8.0

You can use Aurora MySQL version 3 to get the latest MySQL-compatible features, performance enhancements, and bug fixes. Following, you can learn about Aurora MySQL version 3, with MySQL 8.0 compatibility. You can learn how to upgrade your clusters and applications to Aurora MySQL version 3.

Some Aurora features, such as Aurora Serverless v2, require Aurora MySQL version 3.

Topics

- Comparison of Aurora MySQL version 3 and MySQL 8.0 Community Edition
- Upgrading to Aurora MySQL version 3

Features from MySQL 8.0 Community Edition

The initial release of Aurora MySQL version 3 is compatible with MySQL 8.0.23 Community Edition. MySQL 8.0 introduces several new features, including the following:

- JSON functions. For usage information, see JSON Functions in the *MySQL Reference Manual*.
- Window functions. For usage information, see Window Functions in the *MySQL Reference Manual*.
- Common table expressions (CTEs), using the `WITH` clause. For usage information, see WITH (Common Table Expressions) in the *MySQL Reference Manual*.
- Optimized `ADD COLUMN` and `RENAME COLUMN` clauses for the `ALTER TABLE` statement. These optimizations are called "instant DDL." Aurora MySQL version 3 is compatible with the community MySQL instant DDL feature. The former Aurora fast DDL feature isn't used. For usage information for instant DDL, see Instant DDL (Aurora MySQL version 3).
- Descending, functional, and invisible indexes. For usage information, see Invisible Indexes, Descending Indexes, and CREATE INDEX Statement in the *MySQL Reference Manual*.
- Role-based privileges controlled through SQL statements. For more information on changes to the privilege model, see Role-based privilege model.
- `NOWAIT` and `SKIP LOCKED` clauses with the `SELECT ... FOR SHARE` statement. These clauses avoid waiting for other transactions to release row locks. For usage information, see Locking Reads in the *MySQL Reference Manual*.
- Improvements to binary log (binlog) replication. For the Aurora MySQL details, see Binary log replication. In particular, you can perform filtered replication. For usage information about filtered replication, see How Servers Evaluate Replication Filtering Rules in the *MySQL Reference Manual*.
- Hints. Some of the MySQL 8.0–compatible hints were already backported to Aurora MySQL version 2. For information about using hints with Aurora MySQL, see Aurora MySQL hints. For the full list of hints in community MySQL 8.0, see Optimizer Hints in the *MySQL Reference Manual*.

For the full list of features added to MySQL 8.0 community edition, see the blog post The complete list of new features in MySQL 8.0.

Aurora MySQL version 3 also includes changes to keywords for inclusive language, backported from community MySQL 8.0.26. For details about those changes, see Inclusive language changes for Aurora MySQL version 3.

Aurora MySQL version 3 prerequisite for Aurora MySQL Serverless v2

Aurora MySQL version 3 is a prerequisite for all DB instances in an Aurora MySQL Serverless v2 cluster. Aurora MySQL Serverless v2 includes support for reader instances in a DB cluster, and other Aurora features that aren't available for Aurora MySQL Serverless v1. It also has faster and more granular scaling than Aurora MySQL Serverless v1.

Release notes for Aurora MySQL version 3

For the release notes for all Aurora MySQL version 3 releases, see Database engine updates for Amazon Aurora MySQL version 3 in the *Release Notes for Aurora MySQL*.

New parallel query optimizations

The Aurora parallel query optimization now applies to more SQL operations:

- Parallel query now applies to tables containing the data types `TEXT`, `BLOB`, `JSON`, `GEOMETRY`, and `VARCHAR` and `CHAR` longer than 768 bytes.
- Parallel query can optimize queries involving partitioned tables.
- Parallel query can optimize queries involving aggregate function calls in the select list and the `HAVING` clause.

For more information about these enhancements, see Upgrading parallel query clusters to Aurora MySQL version 3. For general information about Aurora parallel query, see Working with parallel query for Amazon Aurora MySQL.

Optimizations to reduce database restart time

Your Aurora MySQL DB cluster must be highly available during both planned and unplanned outages.

Database administrators need to perform occasional database maintenance. This maintenance includes database patching, upgrades, database parameter modifications requiring a manual reboot, performing a failover to reduce the time it takes for instance class changes, and so on. These planned actions require downtime.

However, downtime can also be caused by unplanned actions, such as an unexpected failover due to an underlying hardware fault or database resource throttling. All of these planned and unplanned actions result in a database restart.

In Aurora MySQL version 3.05 and higher, we've introduced optimizations that reduce the database restart time. These optimizations provide up to 65% less downtime than without optimizations, and fewer disruptions to your database workloads, after a restart.

During database startup, many internal memory components are initialized. The largest of these is the InnoDB buffer pool, which in Aurora MySQL is 75% of the instance memory size by default. Our testing has found that the initialization time is proportional to the size of InnoDB buffer pool, and therefore scales with the DB instance class size. During this initialization phase, the database can't accept connections, which causes longer downtime during restarts. The first phase of Aurora MySQL fast restart optimizes the buffer pool initialization, which reduces the time for database initialization and thereby reduces the overall restart time.

For more details, see the blog Reduce downtime with Amazon Aurora MySQL database restart time optimizations.

New temporary table behavior in Aurora MySQL version 3

Aurora MySQL version 3 handles temporary tables differently from earlier Aurora MySQL versions. This new behavior is inherited from MySQL 8.0 Community Edition. There are two types of temporary tables that can be created with Aurora MySQL version 3:

- Internal (or *implicit*) temporary tables – Created by the Aurora MySQL engine to handle operations such as sorting aggregation, derived tables, or common table expressions (CTEs).
- User-created (or *explicit*) temporary tables – Created by the Aurora MySQL engine when you use the `CREATE TEMPORARY TABLE` statement.

There are additional considerations for both internal and user-created temporary tables on Aurora reader DB instances. We discuss these changes in the following sections.

Topics

- User-created (explicit) temporary tables on reader DB instances
- Temporary table creation errors and mitigation

Storage engine for internal (implicit) temporary tables

When generating intermediate result sets, Aurora MySQL initially attempts to write to in-memory temporary tables. This might be unsuccessful, because of either incompatible data types or configured limits. If so, the temporary table is converted to an on-disk temporary table rather than being held in memory. More information on this can be found in the Internal Temporary Table Use in MySQL in the MySQL documentation.

In Aurora MySQL version 3, the way internal temporary tables work is different from earlier Aurora MySQL versions. Instead of choosing between the InnoDB and MyISAM storage engines for such temporary tables, now you choose between the `TempTable` and InnoDB storage engines.

With the `TempTable` storage engine, you can make an additional choice for how to handle certain data. The data affected overflows the memory pool that holds all the internal temporary tables for the DB instance.

Those choices can influence the performance for queries that generate high volumes of temporary data, for example while performing aggregations such as `GROUP BY` on large tables.

> **Tip**
>
> If your workload includes queries that generate internal temporary tables, confirm how your application performs with this change by running benchmarks and monitoring performance-related metrics.
> In some cases, the amount of temporary data fits within the `TempTable` memory pool or only overflows the memory pool by a small amount. In these cases, we recommend using the `TempTable` setting for internal temporary tables and memory-mapped files to hold any overflow data. This setting is the default.

The `TempTable` storage engine is the default. `TempTable` uses a common memory pool for all temporary tables that use this engine, instead of a maximum memory limit per table. The size of this memory pool is specified by the temptable_max_ram parameter. It defaults to 1 GiB on DB instances with 16 or more GiB of memory, and 16 MB on DB instances with less than 16 GiB of memory. The size of the memory pool influences session-level memory consumption.

In some cases when you use the TempTable storage engine, the temporary data might exceed the size of the memory pool. If so, Aurora MySQL stores the overflow data using a secondary mechanism.

You can set the temptable_max_mmap parameter to choose whether the data overflows to memory-mapped temporary files or to InnoDB internal temporary tables on disk. The different data formats and overflow criteria of these overflow mechanisms can affect query performance. They do so by influencing the amount of data written to disk and the demand on disk storage throughput.

Aurora MySQL stores the overflow data differently depending on your choice of data overflow destination and whether the query runs on a writer or reader DB instance:

- On the writer instance, data that overflows to InnoDB internal temporary tables is stored in the Aurora cluster volume.
- On the writer instance, data that overflows to memory-mapped temporary files resides on local storage on the Aurora MySQL version 3 instance.
- On reader instances, overflow data always resides on memory-mapped temporary files on local storage. That's because read-only instances can't store any data on the Aurora cluster volume.

The configuration parameters related to internal temporary tables apply differently to the writer and reader instances in your cluster:

- On reader instances, Aurora MySQL always uses the TempTable storage engine.
- The size for temptable_max_mmap defaults to 1 GiB for both writer and reader instances, regardless of the DB instance memory size. You can adjust this value on both writer and reader instances.
- Setting temptable_max_mmap to 0 turns off the use of memory-mapped temporary files on writer instances.
- You can't set temptable_max_mmap to 0 on reader instances.

Note

We don't recommend using the temptable_use_mmap parameter. It has been deprecated, and support for it is expected to be removed in a future MySQL release.

Limiting the size of internal, in-memory temporary tables

As discussed in Storage engine for internal (implicit) temporary tables, you can control temporary table resources globally by using the temptable_max_ram and temptable_max_mmap settings.

You can also limit the size of any individual internal, in-memory temporary table by using the tmp_table_size DB parameter. This limit is intended to prevent individual queries from consuming an inordinate amount of global temporary table resources, which can affect the performance of concurrent queries that require these resources.

The `tmp_table_size` parameter defines the maximum size of temporary tables created by the `MEMORY` storage engine in Aurora MySQL version 3.

In Aurora MySQL version 3.04 and higher, `tmp_table_size` also defines the maximum size of temporary tables created by the `TempTable` storage engine when the `aurora_tmptable_enable_per_table_limit` DB parameter is set to `ON`. This behavior is disabled by default (`OFF`), which is the same behavior as in Aurora MySQL version 3.03 and lower versions.

- When `aurora_tmptable_enable_per_table_limit` is `OFF`, `tmp_table_size` isn't considered for internal, in-memory temporary tables created by the `TempTable` storage engine.

  However, the global `TempTable` resources limit still applies. Aurora MySQL has the following behavior when the global `TempTable` resources limit is reached:

  - Writer DB instances – Aurora MySQL automatically converts the in-memory temporary table to an InnoDB on-disk temporary table.
  - Reader DB instances – The query ends with an error.

  ```
  ERROR 1114 (HY000): The table '/rdsdbdata/tmp/#sqlxx_xxx' is full
  ```

- When `aurora_tmptable_enable_per_table_limit` is `ON`, Aurora MySQL has the following behavior when the `tmp_table_size` limit is reached:

  - Writer DB instances – Aurora MySQL automatically converts the in-memory temporary table to an InnoDB on-disk temporary table.
  - Reader DB instances – The query ends with an error.

  ```
  ERROR 1114 (HY000): The table '/rdsdbdata/tmp/#sqlxx_xxx' is full
  ```

  Both the global `TempTable` resources limit and the per-table limit apply in this case.

Note

The `aurora_tmptable_enable_per_table_limit` parameter has no effect when internal_tmp_mem_storage_engine is set to `MEMORY`. In this case, the maximum size of an in-memory temporary table is defined by the tmp_table_size or max_heap_table_size value, whichever is smaller.

The following examples show the behavior of the `aurora_tmptable_enable_per_table_limit` parameter for writer and reader DB instances.

Example of writer DB instance with `aurora_tmptable_enable_per_table_limit` set to `OFF`

The in-memory temporary table isn't converted to an InnoDB on-disk temporary table.

```
mysql> set aurora_tmptable_enable_per_table_limit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> select
 @@innodb_read_only,@@aurora_version,@@aurora_tmptable_enable_per_table_limit,@@temptable_max_r
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
| @@innodb_read_only | @@aurora_version | @@aurora_tmptable_enable_per_table_limit |
 @@temptable_max_ram | @@temptable_max_mmap |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
|                  0 | 3.04.0           |                                        0 |
       1073741824 |           1073741824 |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
1 row in set (0.00 sec)

mysql> show status like '%created_tmp_disk%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
+-------------------------+-------+
1 row in set (0.00 sec)

mysql> set cte_max_recursion_depth=4294967295;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n <
 60000000) SELECT max(n) FROM cte;
+----------+
| max(n)   |
+----------+
| 60000000 |
+----------+
1 row in set (13.99 sec)

mysql> show status like '%created_tmp_disk%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
+-------------------------+-------+
1 row in set (0.00 sec)
```

Example of writer DB instance with `aurora_tmptable_enable_per_table_limit` set to ON

The in-memory temporary table is converted to an InnoDB on-disk temporary table.

```
mysql> set aurora_tmptable_enable_per_table_limit=1;
Query OK, 0 rows affected (0.00 sec)

mysql> select
 @@innodb_read_only,@@aurora_version,@@aurora_tmptable_enable_per_table_limit,@@tmp_table_size;
+--------------------+------------------+------------------------------------------
+------------------+
| @@innodb_read_only | @@aurora_version | @@aurora_tmptable_enable_per_table_limit |
 @@tmp_table_size |
+--------------------+------------------+------------------------------------------
+------------------+
|                  0 | 3.04.0           |                                        1 |
      16777216 |
+--------------------+------------------+------------------------------------------
+------------------+
1 row in set (0.00 sec)

mysql> set cte_max_recursion_depth=4294967295;
Query OK, 0 rows affected (0.00 sec)

mysql> show status like '%created_tmp_disk%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 0     |
+-------------------------+-------+
1 row in set (0.00 sec)

mysql> WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n <
 6000000) SELECT max(n) FROM cte;
+---------+
| max(n)  |
+---------+
| 6000000 |
+---------+
1 row in set (4.10 sec)

mysql> show status like '%created_tmp_disk%';
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| Created_tmp_disk_tables | 1     |
+-------------------------+-------+
1 row in set (0.00 sec)
```

Example of reader DB instance with `aurora_tmptable_enable_per_table_limit` set to OFF

The query finishes without an error because `tmp_table_size` doesn't apply, and the global `TempTable` resources limit hasn't been reached.

```
mysql> set aurora_tmptable_enable_per_table_limit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> select
 @@innodb_read_only,@@aurora_version,@@aurora_tmptable_enable_per_table_limit,@@temptable_max_r
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
| @@innodb_read_only | @@aurora_version | @@aurora_tmptable_enable_per_table_limit |
 @@temptable_max_ram | @@temptable_max_mmap |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
|                  1 | 3.04.0           |                                        0 |
       1073741824 |           1073741824 |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
1 row in set (0.00 sec)

mysql> set cte_max_recursion_depth=4294967295;
Query OK, 0 rows affected (0.00 sec)

mysql> WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n <
 60000000) SELECT max(n) FROM cte;
+----------+
| max(n)   |
+----------+
| 60000000 |
+----------+
1 row in set (14.05 sec)
```

Example of reader DB instance with `aurora_tmptable_enable_per_table_limit` set to OFF

This query reaches the global TempTable resources limit with `aurora_tmptable_enable_per_table_limit` set to OFF. The query ends with an error on reader instances.

```
mysql> set aurora_tmptable_enable_per_table_limit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> select
 @@innodb_read_only,@@aurora_version,@@aurora_tmptable_enable_per_table_limit,@@temptable_max_r
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
| @@innodb_read_only | @@aurora_version | @@aurora_tmptable_enable_per_table_limit |
 @@temptable_max_ram | @@temptable_max_mmap |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
|                  1 | 3.04.0           |                                        0 |
       1073741824 |           1073741824 |
+--------------------+------------------+------------------------------------------
+---------------------+----------------------+
1 row in set (0.00 sec)

mysql> set cte_max_recursion_depth=4294967295;
Query OK, 0 rows affected (0.01 sec)

mysql> WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n <
 120000000) SELECT max(n) FROM cte;
ERROR 1114 (HY000): The table '/rdsdbdata/tmp/#sqlfd_1586_2' is full
```

Example of reader DB instance with `aurora_tmptable_enable_per_table_limit` set to ON

The query ends with an error when the `tmp_table_size` limit is reached.

```
mysql> set aurora_tmptable_enable_per_table_limit=1;
Query OK, 0 rows affected (0.00 sec)

mysql> select
 @@innodb_read_only,@@aurora_version,@@aurora_tmptable_enable_per_table_limit,@@tmp_table_size;
+--------------------+------------------+------------------------------------------
+------------------+
| @@innodb_read_only | @@aurora_version | @@aurora_tmptable_enable_per_table_limit |
 @@tmp_table_size |
+--------------------+------------------+------------------------------------------
+------------------+
|                  1 | 3.04.0           |                                        1 |
      16777216 |
+--------------------+------------------+------------------------------------------
+------------------+
1 row in set (0.00 sec)

mysql> set cte_max_recursion_depth=4294967295;
Query OK, 0 rows affected (0.00 sec)

mysql> WITH RECURSIVE cte (n) AS (SELECT 1 UNION ALL SELECT n + 1 FROM cte WHERE n <
 6000000) SELECT max(n) FROM cte;
ERROR 1114 (HY000): The table '/rdsdbdata/tmp/#sqlfd_8_2' is full
```

Mitigating fullness issues for internal temporary tables on Aurora Replicas

To avoid size limitation issues for temporary tables, set the `temptable_max_ram` and `temptable_max_mmap` parameters to a combined value that can fit the requirements of your workload.

Be careful when setting the value of the `temptable_max_ram` parameter. Setting the value too high reduces the available memory on the database instance, which can cause an out-of-memory condition. Monitor the average freeable memory on the DB instance. Then determine an appropriate value for `temptable_max_ram` so that you will still have a reasonable amount of free memory left on the instance. For more information, see Freeable memory issues in Amazon Aurora.

It is also important to monitor the size of the local storage and the temporary table space consumption. For more information on monitoring local storage on an instance, see the AWS

Knowledge Center article What is stored in Aurora MySQL-compatible local storage, and how can I troubleshoot local storage issues?.

> **Note**
>
> This procedure doesn't work when the `aurora_tmptable_enable_per_table_limit` parameter is set to `ON`. For more information, see Limiting the size of internal, in-memory temporary tables.

Example 1

You know that your temporary tables grow to a cumulative size of 20 GiB. You want to set in-memory temporary tables to 2 GiB and to grow to a maximum of 20 GiB on disk.

Set `temptable_max_ram` to **`2,147,483,648`** and `temptable_max_mmap` to **`21,474,836,480`**. These values are in bytes.

These parameter settings make sure that your temporary tables can grow to a cumulative total of 22 GiB.

Example 2

Your current instance size is 16xlarge or larger. You don't know the total size of the temporary tables that you might need. You want to be able to use up to 4 GiB in memory and up to the maximum available storage size on disk.

Set `temptable_max_ram` to **`4,294,967,296`** and `temptable_max_mmap` to **`1,099,511,627,776`**. These values are in bytes.

Here you're setting `temptable_max_mmap` to 1 TiB, which is less than the maximum local storage of 1.2 TiB on a 16xlarge Aurora DB instance.

On a smaller instance size, adjust the value of `temptable_max_mmap` so that it doesn't fill up the available local storage. For example, a 2xlarge instance has only 160 GiB of local storage available. Hence, we recommend setting the value to less than 160 GiB. For more information on the available local storage for DB instance sizes, see Temporary storage limits for Aurora MySQL.

User-created (explicit) temporary tables on reader DB instances

You can create explicit temporary tables using the `TEMPORARY` keyword in your `CREATE TABLE` statement. Explicit temporary tables are supported on the writer DB instance in an Aurora DB cluster. You can also use explicit temporary tables on reader DB instances, but the tables can't enforce the use of the InnoDB storage engine.

To avoid errors while creating explicit temporary tables on Aurora MySQL reader DB instances, make sure that you run all `CREATE TEMPORARY TABLE` statements in either or both of the following ways:

- Don't specify the `ENGINE=InnoDB` clause.
- Don't set the SQL mode to `NO_ENGINE_SUBSTITUTION`.

Temporary table creation errors and mitigation

The error that you receive is different depending on whether you use a plain `CREATE TEMPORARY TABLE` statement or the variation `CREATE TEMPORARY TABLE AS SELECT`. The following examples show the different kinds of errors.

This temporary table behavior only applies to read-only instances. This first example confirms that's the kind of instance the session is connected to.

```
mysql> select @@innodb_read_only;
+--------------------+
| @@innodb_read_only |
+--------------------+
|                  1 |
+--------------------+
```

For plain `CREATE TEMPORARY TABLE` statements, the statement fails when the `NO_ENGINE_SUBSTITUTION` SQL mode is turned on. When `NO_ENGINE_SUBSTITUTION` is turned off (default), the appropriate engine substitution is made, and the temporary table creation succeeds.

```
mysql> set sql_mode = 'NO_ENGINE_SUBSTITUTION';

mysql>  CREATE TEMPORARY TABLE tt2 (id int) ENGINE=InnoDB;
ERROR 3161 (HY000): Storage engine InnoDB is disabled (Table creation is disallowed).

mysql> SET sql_mode = '';

mysql> CREATE TEMPORARY TABLE tt4 (id int) ENGINE=InnoDB;
mysql> SHOW CREATE TABLE tt4\G
*************************** 1. row ***************************
       Table: tt4
Create Table: CREATE TEMPORARY TABLE `tt4` (
  `id` int DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

For `CREATE TEMPORARY TABLE AS SELECT` statements, the statement fails when the `NO_ENGINE_SUBSTITUTION` SQL mode is turned on. When `NO_ENGINE_SUBSTITUTION` is turned off (default), the appropriate engine substitution is made, and the temporary table creation succeeds.

```
mysql> set sql_mode = 'NO_ENGINE_SUBSTITUTION';

mysql> CREATE TEMPORARY TABLE tt1 ENGINE=InnoDB AS SELECT * FROM t1;
ERROR 3161 (HY000): Storage engine InnoDB is disabled (Table creation is disallowed).

mysql> SET sql_mode = '';

mysql> show create table tt3;
+-------+----------------------------------------------------------+
| Table | Create Table                                             |
+-------+----------------------------------------------------------+
| tt3   | CREATE TEMPORARY TABLE `tt3` (
  `id` int DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+-------+----------------------------------------------------------+
1 row in set (0.00 sec)
```

For more information about the storage aspects and performance implications of temporary tables in Aurora MySQL version 3, see the blog post Use the TempTable storage engine on Amazon RDS for MySQL and Amazon Aurora MySQL.

Comparison of Aurora MySQL version 2 and Aurora MySQL version 3

Use the following to learn about changes to be aware of when you upgrade your Aurora MySQL version 2 cluster to version 3.

Topics

- Parameter changes for Aurora MySQL version 3
- Status variables
- Inclusive language changes for Aurora MySQL version 3
- AUTO_INCREMENT values
- Binary log replication

Feature differences between Aurora MySQL version 2 and 3

The following Amazon Aurora MySQL features are supported in Aurora MySQL for MySQL 5.7, but these features aren't supported in Aurora MySQL for MySQL 8.0:

- You can't use Aurora MySQL version 3 for Aurora Serverless v1 clusters. Aurora MySQL version 3 works with Aurora Serverless v2.
- Lab mode doesn't apply to Aurora MySQL version 3. There aren't any lab mode features in Aurora MySQL version 3. Instant DDL supersedes the fast online DDL feature that was formerly available in lab mode. For an example, see Instant DDL (Aurora MySQL version 3).
- The query cache is removed from community MySQL 8.0 and also from Aurora MySQL version 3.
- Aurora MySQL version 3 is compatible with the community MySQL hash join feature. The Aurora-specific implementation of hash joins in Aurora MySQL version 2 isn't used. For information about using hash joins with Aurora parallel query, see Turning on hash join for parallel query clusters and Aurora MySQL hints. For general usage information about hash joins, see Hash Join Optimization in the *MySQL Reference Manual*.
- The `mysql.lambda_async` stored procedure that was deprecated in Aurora MySQL version 2 is removed in version 3. For version 3, use the asynchronous function `lambda_async` instead.
- The default character set in Aurora MySQL version 3 is `utf8mb4`. In Aurora MySQL version 2, the default character set was `latin1`. For information about this character set, see The utf8mb4 Character Set (4-Byte UTF-8 Unicode Encoding) in the *MySQL Reference Manual*.

Some Aurora MySQL features are available for certain combinations of AWS Region and DB engine version. For details, see Supported features in Amazon Aurora by AWS Region and Aurora DB engine.

Instance class support

Aurora MySQL version 3 supports a different set of instance classes from Aurora MySQL version 2:

- For larger instances, you can use the modern instance classes such as `db.r5`, `db.r6g`, and `db.x2g`.
- For smaller instances, you can use the modern instance classes such as `db.t3` and `db.t4g`.

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see Using T instance classes for development and testing.

The following instance classes from Aurora MySQL version 2 aren't available for Aurora MySQL version 3:

- `db.r4`
- `db.r3`
- `db.t3.small`
- `db.t2`

Check your administration scripts for any CLI statements that create Aurora MySQL DB instances. Hardcode instance class names that aren't available for Aurora MySQL version 3. If necessary, modify the instance class names to ones that Aurora MySQL version 3 supports.

> **Tip**
>
> To check the instance classes that you can use for a specific combination of Aurora MySQL version and AWS Region, use the `describe-orderable-db-instance-options` AWS CLI command.

For full details about Aurora instance classes, see Aurora DB instance classes.

Parameter changes for Aurora MySQL version 3

Aurora MySQL version 3 includes new cluster-level and instance-level configuration parameters. Aurora MySQL version 3 also removes some parameters that were present in Aurora MySQL version 2. Some parameter names are changed as a result of the initiative for inclusive language. For backward compatibility, you can still retrieve the parameter values using either the old names or the new names. However, you must use the new names to specify parameter values in a custom parameter group.

In Aurora MySQL version 3, the value of the `lower_case_table_names` parameter is set permanently at the time the cluster is created. If you use a nondefault value for this option, set up your Aurora MySQL version 3 custom parameter group before upgrading. Then specify the parameter group during the create cluster or snapshot restore operation.

> **Note**
>
> With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. Use the snapshot restore method instead.

In Aurora MySQL version 3, the `init_connect` and `read_only` parameters don't apply for users who have the `CONNECTION_ADMIN` privilege. This includes the Aurora master user. For more information, see Role-based privilege model.

For the full list of Aurora MySQL cluster parameters, see Cluster-level parameters. The table covers all the parameters from Aurora MySQL version 2 and 3. The table includes notes showing which parameters are new in Aurora MySQL version 3 or were removed from Aurora MySQL version 3.

For the full list of Aurora MySQL instance parameters, see Instance-level parameters. The table covers all the parameters from Aurora MySQL version 2 and 3. The table includes notes showing which parameters are new in Aurora MySQL version 3 and which parameters were removed from Aurora MySQL version 3. It also includes notes showing which parameters were modifiable in earlier versions but not Aurora MySQL version 3.

For information about parameter names that changed, see Inclusive language changes for Aurora MySQL version 3.

Status variables

For information about status variables that aren't applicable to Aurora MySQL, see MySQL status variables that don't apply to Aurora MySQL.

Inclusive language changes for Aurora MySQL version 3

Aurora MySQL version 3 is compatible with version 8.0.23 from the MySQL community edition. Aurora MySQL version 3 also includes changes from MySQL 8.0.26 related to keywords and system schemas for inclusive language. For example, the `SHOW REPLICA STATUS` command is now preferred instead of `SHOW SLAVE STATUS`.

The following Amazon CloudWatch metrics have new names in Aurora MySQL version 3.

In Aurora MySQL version 3, only the new metric names are available. Make sure to update any alarms or other automation that relies on metric names when you upgrade to Aurora MySQL version 3.

Old name	New name
`ForwardingMasterDMLLatency`	`ForwardingWriterDMLLatency`
`ForwardingMasterOpenSessions`	`ForwardingWriterOpenSessions`
`AuroraDMLRejectedMasterFull`	`AuroraDMLRejectedWriterFull`
`ForwardingMasterDMLThroughput`	`ForwardingWriterDMLThroughput`

The following status variables have new names in Aurora MySQL version 3.

For compatibility, you can use either name in the initial Aurora MySQL version 3 release. The old status variable names are to be removed in a future release.

Name to be removed	New or preferred name
`Aurora_fwd_master_dml_stmt_duration`	`Aurora_fwd_writer_dml_stmt_duration`
`Aurora_fwd_master_dml_stmt_count`	`Aurora_fwd_writer_dml_stmt_count`

Name to be removed	New or preferred name
`Aurora_fwd_master_select_stmt_duration`	`Aurora_fwd_writer_select_stmt_duration`
`Aurora_fwd_master_select_stmt_count`	`Aurora_fwd_writer_select_stmt_count`
`Aurora_fwd_master_errors_session_timeout`	`Aurora_fwd_writer_errors_session_timeout`
`Aurora_fwd_master_open_sessions`	`Aurora_fwd_writer_open_sessions`
`Aurora_fwd_master_errors_session_limit`	`Aurora_fwd_writer_errors_session_limit`
`Aurora_fwd_master_errors_rpc_timeout`	`Aurora_fwd_writer_errors_rpc_timeout`

The following configuration parameters have new names in Aurora MySQL version 3.

For compatibility, you can check the parameter values in the `mysql` client by using either name in the initial Aurora MySQL version 3 release. You can use only the new names when modifying values in a custom parameter group. The old parameter names are to be removed in a future release.

Name to be removed	New or preferred name
`aurora_fwd_master_idle_timeout`	`aurora_fwd_writer_idle_timeout`
`aurora_fwd_master_max_connections_pct`	`aurora_fwd_writer_max_connections_pct`

Name to be removed	New or preferred name	
`master_verify_checksum`	`source_verify_checksum`	
`sync_master_info`	`sync_source_info`	
`init_slave`	`init_replica`	
`rpl_stop_slave_timeout`	`rpl_stop_replica_timeout`	
`log_slow_slave_statements`	`log_slow_replica_statements`	
`slave_max_allowed_packet`	`replica_max_allowed_packet`	
`slave_compressed_protocol`	`replica_compressed_protocol`	
`slave_exec_mode`	`replica_exec_mode`	
`slave_type_conversions`	`replica_type_conversions`	
`slave_sql_verify_checksum`	`replica_sql_verify_checksum`	
`slave_parallel_type`	`replica_parallel_type`	
`slave_preserve_commit_order`	`replica_preserve_commit_order`	
`log_slave_updates`	`log_replica_updates`	
`slave_allow_batching`	`replica_allow_batching`	

Name to be removed	New or preferred name
`slave_load_tmpdir`	`replica_load_tmpdir`
`slave_net_timeout`	`replica_net_timeout`
`sql_slave_skip_counter`	`sql_replica_skip_counter`
`slave_skip_errors`	`replica_skip_errors`
`slave_checkpoint_period`	`replica_checkpoint_period`
`slave_checkpoint_group`	`replica_checkpoint_group`
`slave_transaction_retries`	`replica_transaction_retries`
`slave_parallel_workers`	`replica_parallel_workers`
`slave_pending_jobs_size_max`	`replica_pending_jobs_size_max`
`pseudo_slave_mode`	`pseudo_replica_mode`

The following stored procedures have new names in Aurora MySQL version 3.

For compatibility, you can use either name in the initial Aurora MySQL version 3 release. The old procedure names are to be removed in a future release.

Name to be removed	New or preferred name
`mysql.rds_set_master_auto_position`	`mysql.rds_set_source_auto_position`

Name to be removed	New or preferred name
`mysql.rds_set_external_master`	`mysql.rds_set_external_source`
`mysql.rds_set_external_master_with_auto_position`	`mysql.rds_set_external_source_with_auto_position`
`mysql.rds_reset_external_master`	`mysql.rds_reset_external_source`
`mysql.rds_next_master_log`	`mysql.rds_next_source_log`

AUTO_INCREMENT values

In Aurora MySQL version 3, Aurora preserves the `AUTO_INCREMENT` value for each table when it restarts each DB instance. In Aurora MySQL version 2, the `AUTO_INCREMENT` value wasn't preserved after a restart.

The `AUTO_INCREMENT` value isn't preserved when you set up a new cluster by restoring from a snapshot, performing a point-in-time recovery, and cloning a cluster. In these cases, the `AUTO_INCREMENT` value is initialized to the value based on the largest column value in the table at the time the snapshot was created. This behavior is different than in RDS for MySQL 8.0, where the `AUTO_INCREMENT` value is preserved during these operations.

Binary log replication

In MySQL 8.0 community edition, binary log replication is turned on by default. In Aurora MySQL version 3, binary log replication is turned off by default.

> **Tip**
>
> If your high availability requirements are fulfilled by the Aurora built-in replication features, you can leave binary log replication turned off. That way, you can avoid the performance overhead of binary log replication. You can also avoid the associated monitoring and troubleshooting that are needed to manage binary log replication.

Aurora supports binary log replication from a MySQL 5.7–compatible source to Aurora MySQL version 3. The source system can be an Aurora MySQL DB cluster, an RDS for MySQL DB instance, or an on-premises MySQL instance.

As does community MySQL, Aurora MySQL supports replication from a source running a specific version to a target running the same major version or one major version higher. For example, replication from a MySQL 5.6–compatible system to Aurora MySQL version 3 isn't supported. Replicating from Aurora MySQL version 3 to a MySQL 5.7–compatible or MySQL 5.6–compatible system isn't supported. For details about using binary log replication, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

Aurora MySQL version 3 includes improvements to binary log replication in community MySQL 8.0, such as filtered replication. For details about the community MySQL 8.0 improvements, see How Servers Evaluate Replication Filtering Rules in the *MySQL Reference Manual*.

Transaction compression for binary log replication

For usage information about binary log compression, see Binary Log Transaction Compression in the MySQL Reference Manual.

The following limitations apply to binary log compression in Aurora MySQL version 3:

- Transactions whose binary log data is larger than the maximum allowed packet size aren't compressed. This is true regardless of whether the Aurora MySQL binary log compression setting is turned on. Such transactions are replicated without being compressed.
- If you use a connector for change data capture (CDC) that doesn't support MySQL 8.0 yet, you can't use this feature. We recommend that you test any third-party connectors thoroughly with binary log compression. Also, we recommend that you do so before turning on binlog compression on systems that use binlog replication for CDC.

Comparison of Aurora MySQL version 3 and MySQL 8.0 Community Edition

You can use the following information to learn about the changes to be aware of when you convert from a different MySQL 8.0–compatible system to Aurora MySQL version 3.

In general, Aurora MySQL version 3 supports the feature set of community MySQL 8.0.23. Some new features from MySQL 8.0 community edition don't apply to Aurora MySQL. Some of those features aren't compatible with some aspect of Aurora, such as the Aurora storage architecture.

Other features aren't needed because the Amazon RDS management service provides equivalent functionality. The following features in community MySQL 8.0 aren't supported or work differently in Aurora MySQL version 3.

For release notes for all Aurora MySQL version 3 releases, see Database engine updates for Amazon Aurora MySQL version 3 in the *Release Notes for Aurora MySQL*.

Topics

- MySQL 8.0 features not available in Aurora MySQL version 3
- Role-based privilege model
- Authentication

MySQL 8.0 features not available in Aurora MySQL version 3

The following features from community MySQL 8.0 aren't available or work differently in Aurora MySQL version 3.

- Resource groups and associated SQL statements aren't supported in Aurora MySQL.
- Aurora MySQL doesn't support user-defined undo tablespaces and associated SQL statements, such as `CREATE UNDO TABLESPACE`, `ALTER UNDO TABLESPACE ... SET INACTIVE`, and `DROP UNDO TABLESPACE`.
- Aurora MySQL doesn't support undo tablespace truncation for Aurora MySQL versions lower than 3.06. In Aurora MySQL version 3.06 and higher, automated undo tablespace truncation is supported.
- You can't modify the settings of any MySQL plugins.
- The X plugin isn't supported.
- Multisource replication isn't supported.

Role-based privilege model

With Aurora MySQL version 3, you can't modify the tables in the `mysql` database directly. In particular, you can't set up users by inserting into the `mysql.user` table. Instead, you use SQL statements to grant role-based privileges. You also can't create other kinds of objects such as stored procedures in the `mysql` database. You can still query the `mysql` tables. If you use binary log replication, changes made directly to the `mysql` tables on the source cluster aren't replicated to the target cluster.

In some cases, your application might use shortcuts to create users or other objects by inserting into the mysql tables. If so, change your application code to use the corresponding statements such as CREATE USER. If your application creates stored procedures or other objects in the mysql database, use a different database instead.

To export metadata for database users during the migration from an external MySQL database, you can use a MySQL Shell command instead of mysqldump. For more information, see Instance Dump Utility, Schema Dump Utility, and Table Dump Utility.

To simplify managing permissions for many users or applications, you can use the CREATE ROLE statement to create a role that has a set of permissions. Then you can use the GRANT and SET ROLE statements and the current_role function to assign roles to users or applications, switch the current role, and check which roles are in effect. For more information on the role-based permission system in MySQL 8.0, see Using Roles in the MySQL Reference Manual.

⚠ **Important**

We strongly recommend that you do not use the master user directly in your applications. Instead, adhere to the best practice of using a database user created with the minimal privileges required for your application.

Aurora MySQL version 3 includes a special role that has all of the following privileges. This role is named rds_superuser_role. The primary administrative user for each cluster already has this role granted. The rds_superuser_role role includes the following privileges for all database objects:

- ALTER
- APPLICATION_PASSWORD_ADMIN
- ALTER ROUTINE
- CONNECTION_ADMIN
- CREATE
- CREATE ROLE
- CREATE ROUTINE
- CREATE TEMPORARY TABLES
- CREATE USER

- `CREATE VIEW`
- `DELETE`
- `DROP`
- `DROP ROLE`
- `EVENT`
- `EXECUTE`
- `INDEX`
- `INSERT`
- `LOCK TABLES`
- `PROCESS`
- `REFERENCES`
- `RELOAD`
- `REPLICATION CLIENT`
- `REPLICATION SLAVE`
- `ROLE_ADMIN`
- `SET_USER_ID`
- `SELECT`
- `SHOW DATABASES`
- `SHOW_ROUTINE` (Aurora MySQL version 3.04 and higher)
- `SHOW VIEW`
- `TRIGGER`
- `UPDATE`
- `XA_RECOVER_ADMIN`

The role definition also includes `WITH GRANT OPTION` so that an administrative user can grant that role to other users. In particular, the administrator must grant any privileges needed to perform binary log replication with the Aurora MySQL cluster as the target.

Tip

To see the full details of the permissions, enter the following statements.

```
SHOW GRANTS FOR rds_superuser_role@'%';
SHOW GRANTS FOR name_of_administrative_user_for_your_cluster@'%';
```

Aurora MySQL version 3 also includes roles that you can use to access other AWS services. You can set these roles as an alternative to GRANT statements. For example, you specify GRANT AWS_LAMBDA_ACCESS TO *user* instead of GRANT INVOKE LAMBDA ON *.* TO *user*. For the procedures to access other AWS services, see Integrating Amazon Aurora MySQL with other AWS services. Aurora MySQL version 3 includes the following roles related to accessing other AWS services:

- AWS_LAMBDA_ACCESS role, as an alternative to the INVOKE LAMBDA privilege. For usage information, Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.
- AWS_LOAD_S3_ACCESS role, as an alternative to the LOAD FROM S3 privilege. For usage information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.
- AWS_SELECT_S3_ACCESS role, as an alternative to the SELECT INTO S3 privilege. For usage information, see Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket.
- AWS_SAGEMAKER_ACCESS role, as an alternative to the INVOKE SAGEMAKER privilege. For usage information, see Using Amazon Aurora machine learning with Aurora MySQL.
- AWS_COMPREHEND_ACCESS role, as an alternative to the INVOKE COMPREHEND privilege. For usage information, see Using Amazon Aurora machine learning with Aurora MySQL.

When you grant access by using roles in Aurora MySQL version 3, you also activate the role by using the SET ROLE *role_name* or SET ROLE ALL statement. The following example shows how. Substitute the appropriate role name for AWS_SELECT_S3_ACCESS.

```
# Grant role to user.
mysql> GRANT AWS_SELECT_S3_ACCESS TO 'user'@'domain-or-ip-address'

# Check the current roles for your user. In this case, the AWS_SELECT_S3_ACCESS role
 has not been activated.
# Only the rds_superuser_role is currently in effect.
mysql> SELECT CURRENT_ROLE();
+--------------------------+
| CURRENT_ROLE()            |
+--------------------------+
| `rds_superuser_role`@`%` |
+--------------------------+
1 row in set (0.00 sec)

# Activate all roles associated with this user using SET ROLE.
# You can activate specific roles or all roles.
# In this case, the user only has 2 roles, so we specify ALL.
mysql> SET ROLE ALL;
Query OK, 0 rows affected (0.00 sec)

# Verify role is now active
mysql> SELECT CURRENT_ROLE();
+-----------------------------------------------------+
| CURRENT_ROLE()                                      |
+-----------------------------------------------------+
| `AWS_SELECT_S3_ACCESS`@`%`,`rds_superuser_role`@`%` |
+-----------------------------------------------------+
```

Authentication

In community MySQL 8.0, the default authentication plugin is `caching_sha2_password`. Aurora MySQL version 3 still uses the `mysql_native_password` plugin. You can't change the `default_authentication_plugin` setting.

Upgrading to Aurora MySQL version 3

For information on upgrading your database from Aurora MySQL version 2 to version 3, see Upgrading the major version of an Amazon Aurora MySQL DB cluster.

Aurora MySQL version 2 compatible with MySQL 5.7

This topic describes the differences between Aurora MySQL version 2 and MySQL 5.7 Community Edition.

Features not supported in Aurora MySQL version 2

The following features are supported in MySQL 5.7, but are currently not supported in Aurora MySQL version 2:

- `CREATE TABLESPACE` SQL statement

- Group replication plugin
- Increased page size
- InnoDB buffer pool loading at startup
- InnoDB full-text parser plugin
- Multisource replication
- Online buffer pool resizing
- Password validation plugin – You can install the plugin, but it isn't supported. You can't customize the plugin.
- Query rewrite plugins
- Replication filtering
- X Protocol

For more information about these features, see the MySQL 5.7 documentation.

Temporary tablespace behavior in Aurora MySQL version 2

In MySQL 5.7, the temporary tablespace is autoextending and increases in size as necessary to accommodate on-disk temporary tables. When temporary tables are dropped, freed space can be reused for new temporary tables, but the temporary tablespace remains at the extended size and doesn't shrink. The temporary tablespace is dropped and re-created when engine is restarted.

In Aurora MySQL version 2, the following behavior applies:

- For new Aurora MySQL DB clusters created with version 2.10 and higher, the temporary tablespace is removed and re-created when you restart the database. This allows the dynamic resizing feature to reclaim the storage space.
- For existing Aurora MySQL DB clusters upgraded to:
  - Version 2.10 or higher – The temporary tablespace is removed and re-created when you restart the database. This allows the dynamic resizing feature to reclaim the storage space.
  - Version 2.09 – Temporary table space isn't removed when you restart the database.

You can check the size of the temporary tablespace on your Aurora MySQL version 2 DB cluster by using the following query:

```
SELECT
```

```
    FILE_NAME,
    TABLESPACE_NAME,
    ROUND((TOTAL_EXTENTS * EXTENT_SIZE) / 1024 / 1024 / 1024, 4) AS SIZE
FROM
    INFORMATION_SCHEMA.FILES
WHERE
    TABLESPACE_NAME = 'innodb_temporary';
```

For more information, see The Temporary Tablespace in the MySQL documentation.

Storage engine for on-disk temporary tables

Aurora MySQL version 2 uses different storage engines for on-disk internal temporary tables depending on the role of the instance.

- On the writer instance, on-disk temporary tables use the InnoDB storage engine by default. They're stored in the temporary tablespace in the Aurora cluster volume.

  You can change this behavior on the writer instance by modifying the value for the DB parameter `internal_tmp_disk_storage_engine`. For more information, see Instance-level parameters.
- On reader instances, on-disk temporary tables use the MyISAM storage engine, which uses local storage. That's because read-only instances can't store any data on the Aurora cluster volume.

Security with Amazon Aurora MySQL

Security for Amazon Aurora MySQL is managed at three levels:

- To control who can perform Amazon RDS management actions on Aurora MySQL DB clusters and DB instances, you use AWS Identity and Access Management (IAM). When you connect to AWS using IAM credentials, your AWS account must have IAM policies that grant the permissions required to perform Amazon RDS management operations. For more information, see Identity and access management for Amazon Aurora

  If you are using IAM to access the Amazon RDS console, make sure to first sign in to the AWS Management Console with your IAM user credentials. Then go to the Amazon RDS console at https://console.aws.amazon.com/rds/.
- Make sure to create Aurora MySQL DB clusters in a virtual public cloud (VPC) based on the Amazon VPC service. To control which devices and Amazon EC2 instances can open connections to the endpoint and port of the DB instance for Aurora MySQL DB clusters in a VPC, use a VPC security group. You can make these endpoint and port connections by using Transport Layer Security (TLS). In addition, firewall rules at your company can control whether devices running at your company can open connections to a DB instance. For more information on VPCs, see Amazon VPC VPCs and Amazon Aurora.

  The supported VPC tenancy depends on the DB instance class used by your Aurora MySQL DB clusters. With `default` VPC tenancy, the VPC runs on shared hardware. With `dedicated` VPC tenancy, the VPC runs on a dedicated hardware instance. The burstable performance DB instance classes support default VPC tenancy only. The burstable performance DB instance classes include the db.t2, db.t3, and db.t4g DB instance classes. All other Aurora MySQL DB instance classes support both default and dedicated VPC tenancy.

  > **Note**
  >
  > We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see Using T instance classes for development and testing.

  For more information about instance classes, see Aurora DB instance classes. For more information about `default` and `dedicated` VPC tenancy, see Dedicated instances in the *Amazon Elastic Compute Cloud User Guide*.

- To authenticate login and permissions for an Amazon Aurora MySQL DB cluster, you can take either of the following approaches, or a combination of them:
  - You can take the same approach as with a standalone instance of MySQL.

    Commands such as `CREATE USER`, `RENAME USER`, `GRANT`, `REVOKE`, and `SET PASSWORD` work just as they do in on-premises databases, as does directly modifying database schema tables. For more information, see Access control and account management in the MySQL documentation.
  - You can also use IAM database authentication.

    With IAM database authentication, you authenticate to your DB cluster by using an IAM user or IAM role and an authentication token. An *authentication token* is a unique value that is generated using the Signature Version 4 signing process. By using IAM database authentication, you can use the same credentials to control access to your AWS resources and your databases. For more information, see IAM database authentication.

> **Note**
>
> For more information, see Security in Amazon Aurora.

Master user privileges with Amazon Aurora MySQL

When you create an Amazon Aurora MySQL DB instance, the master user has the default privileges listed in Master user account privileges.

To provide management services for each DB cluster, the `admin` and `rdsadmin` users are created when the DB cluster is created. Attempting to drop, rename, change the password, or change privileges for the `rdsadmin` account results in an error.

In Aurora MySQL version 2 DB clusters, the `admin` and `rdsadmin` users are created when the DB cluster is created. In Aurora MySQL version 3 DB clusters, the `admin`, `rdsadmin`, and `rds_superuser_role` users are created.

Important

We strongly recommend that you do not use the master user directly in your applications. Instead, adhere to the best practice of using a database user created with the minimal privileges required for your application.

For management of the Aurora MySQL DB cluster, the standard `kill` and `kill_query` commands have been restricted. Instead, use the Amazon RDS commands `rds_kill` and `rds_kill_query` to terminate user sessions or queries on Aurora MySQL DB instances.

Note

Encryption of a database instance and snapshots is not supported for the China (Ningxia) region.

Using TLS with Aurora MySQL DB clusters

Amazon Aurora MySQL DB clusters support Transport Layer Security (TLS) connections from applications using the same process and public key as RDS for MySQL DB instances.

Amazon RDS creates an TLS certificate and installs the certificate on the DB instance when Amazon RDS provisions the instance. These certificates are signed by a certificate authority. The TLS certificate includes the DB instance endpoint as the Common Name (CN) for the TLS certificate to guard against spoofing attacks. As a result, you can only use the DB cluster endpoint to connect to a DB cluster using TLS if your client supports Subject Alternative Names (SAN). Otherwise, you must use the instance endpoint of a writer instance.

For information about downloading certificates, see Using SSL/TLS to encrypt a connection to a DB cluster.

We recommend the AWS JDBC Driver as a client that supports SAN with TLS. For more information about the AWS JDBC Driver and complete instructions for using it, see the Amazon Web Services (AWS) JDBC Driver GitHub repository.

Topics

- Requiring a TLS connection to an Aurora MySQL DB cluster

- TLS versions for Aurora MySQL
- Configuring cipher suites for connections to Aurora MySQL DB clusters
- Encrypting connections to an Aurora MySQL DB cluster

Requiring a TLS connection to an Aurora MySQL DB cluster

You can require that all user connections to your Aurora MySQL DB cluster use TLS by using the `require_secure_transport` DB cluster parameter. By default, the `require_secure_transport` parameter is set to `OFF`. You can set the `require_secure_transport` parameter to `ON` to require TLS for connections to your DB cluster.

You can set the `require_secure_transport` parameter value by updating the DB cluster parameter group for your DB cluster. You don't need to reboot your DB cluster for the change to take effect. For more information on parameter groups, see Working with parameter groups.

> **Note**
>
> The `require_secure_transport` parameter is available for Aurora MySQL version 2 and 3. You can set this parameter in a custom DB cluster parameter group. The parameter isn't available in DB instance parameter groups.

When the `require_secure_transport` parameter is set to `ON` for a DB cluster, a database client can connect to it if it can establish an encrypted connection. Otherwise, an error message similar to the following is returned to the client:

```
MySQL Error 3159 (HY000): Connections using insecure transport are prohibited while --
require_secure_transport=ON.
```

TLS versions for Aurora MySQL

Aurora MySQL supports Transport Layer Security (TLS) versions 1.0, 1.1, 1.2, and 1.3. Starting in Aurora MySQL version 3.04.0 and higher, you can use the TLS 1.3 protocol to secure your connections. The following table shows the TLS support for Aurora MySQL versions.

Aurora MySQL version	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3	Default
Aurora MySQL version 2	Supported	Supported	Supported	Not supported	All supported TLS versions
Aurora MySQL version 3 (below 3.04.0)	Supported	Supported	Supported	Not supported	All supported TLS versions
Aurora MySQL version 3 (3.04.0 and above)	Not supported	Not supported	Supported	Supported	All supported TLS versions

⚠ **Important**

If you are using custom parameter groups for your Aurora MySQL clusters with version 2 and versions lower than 3.04.0, we recommend using TLS 1.2 because TLS 1.0 and 1.1 are less secure. The community edition of MySQL 8.0.26 and Aurora MySQL 3.03 and its minor versions deprecated support for TLS versions 1.1 and 1.0.

The community edition of MySQL 8.0.28 and compatible Aurora MySQL versions 3.04.0 and higher do not support TLS 1.1 and TLS 1.0. If you are using Aurora MySQL versions 3.04.0 and higher, do not set the TLS protocol to 1.0 and 1.1 in your custom parameter group.

For Aurora MySQL versions 3.04.0 and higher, the default setting is TLS 1.3 and TLS 1.2.

You can use the `tls_version` DB cluster parameter to indicate the permitted protocol versions. Similar client parameters exist for most client tools or database drivers. Some older clients might not support newer TLS versions. By default, the DB cluster attempts to use the highest TLS protocol version allowed by both the server and client configuration.

Set the `tls_version` DB cluster parameter to one of the following values:

- `TLSv1.3`
- `TLSv1.2`
- `TLSv1.1`
- `TLSv1`

You can also set the `tls_version` parameter as a string of comma-separated list. If you want to use both TLS 1.2 and TLS 1.0 protocols, the `tls_version` parameter must include all protocols from the lowest to the highest protocol. In this case, `tls_version` is set as:

```
tls_version=TLSv1,TLSv1.1,TLSv1.2
```

For information about modifying parameters in a DB cluster parameter group, see Modifying parameters in a DB cluster parameter group. If you use the AWS CLI to modify the `tls_version` DB cluster parameter, the `ApplyMethod` must be set to `pending-reboot`. When the application method is `pending-reboot`, changes to parameters are applied after you stop and restart the DB clusters associated with the parameter group.

Configuring cipher suites for connections to Aurora MySQL DB clusters

By using configurable cipher suites, you can have more control over the security of your database connections. You can specify a list of cipher suites that you want to allow to secure client TLS connections to your database. With configurable cipher suites, you can control the connection encryption that your database server accepts. Doing this prevents the use of insecure or deprecated ciphers.

Configurable cipher suites are supported in Aurora MySQL version 3 and Aurora MySQL version 2. To specify the list of permissible TLS 1.2, TLS 1.1, TLS 1.0 ciphers for encrypting connections, modify the `ssl_cipher` cluster parameter. Set the `ssl_cipher` parameter in a cluster parameter group using the AWS Management Console, the AWS CLI, or the RDS API.

Set the `ssl_cipher` parameter to a string of comma-separated cipher values for your TLS version. For the client application, you can specify the ciphers to use for encrypted connections by using the `--ssl-cipher` option when connecting to the database. For more about connecting to your database, see Connecting to an Amazon Aurora MySQL DB cluster.

Starting in Aurora MySQL version 3.04.0 and higher, you can specify TLS 1.3 cipher suites. To specify the permissible TLS 1.3 cipher suites, modify the `tls_ciphersuites` parameter in your parameter group. TLS 1.3 has reduced the number of available cipher suites due to changes in the naming convention that removes the key exchange mechanism and certificate used. Set the `tls_ciphersuites` to a string of comma-separated cipher values for TLS 1.3.

The following table shows the supported ciphers along with the TLS encryption protocol and valid Aurora MySQL engine versions for each cipher.

Cipher	Encryption protocol	Supported Aurora MySQL versions
`DHE-RSA-AES128-SHA`	TLS 1.0	3.01.0 and higher, all below 2.11.0
`DHE-RSA-AES128-SHA256`	TLS 1.2	3.01.0 and higher, all below 2.11.0
`DHE-RSA-AES128-GCM-SHA256`	TLS 1.2	3.01.0 and higher, all below 2.11.0
`DHE-RSA-AES256-SHA`	TLS 1.0	3.03.0 and lower, all below 2.11.0
`DHE-RSA-AES256-SHA256`	TLS 1.2	3.01.0 and higher, all below 2.11.0
`DHE-RSA-AES256-GCM-SHA384`	TLS 1.2	3.01.0 and higher, all below 2.11.0
`ECDHE-RSA-AES128-SHA`	TLS 1.0	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher
`ECDHE-RSA-AES128-SHA256`	TLS 1.2	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher
`ECDHE-RSA-AES128-GCM-SHA256`	TLS 1.2	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher

Cipher	Encryption protocol	Supported Aurora MySQL versions
ECDHE-RSA-AES256-SHA	TLS 1.0	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher
ECDHE-RSA-AES256-SHA384	TLS 1.2	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher
ECDHE-RSA-AES256-GCM-SHA384	TLS 1.2	3.01.0 and higher, 2.09.3 and higher, 2.10.2 and higher
TLS_AES_128_GCM_SHA256	TLS 1.3	3.04.0 and higher
TLS_AES_256_GCM_SHA384	TLS 1.3	3.04.0 and higher
TLS_CHACHA20_POLY1305_SHA256	TLS 1.3	3.04.0 and higher

Note

DHE-RSA ciphers are only supported by Aurora MySQL versions before 2.11.0. Versions 2.11.0 and higher support only ECDHE ciphers.

For information about modifying parameters in a DB cluster parameter group, see Modifying parameters in a DB cluster parameter group. If you use the CLI to modify the `ssl_cipher` DB cluster parameter, make sure to set the `ApplyMethod` to `pending-reboot`. When the application method is `pending-reboot`, changes to parameters are applied after you stop and restart the DB clusters associated with the parameter group.

You can also use the describe-engine-default-cluster-parameters CLI command to determine which cipher suites are currently supported for a specific parameter group family. The following example shows how to get the allowed values for the `ssl_cipher` cluster parameter for Aurora MySQL version 2.

```
aws rds describe-engine-default-cluster-parameters --db-parameter-group-family aurora-
mysql5.7

        ...some output truncated...
 {
  "ParameterName": "ssl_cipher",
  "ParameterValue": "DHE-RSA-AES128-SHA,DHE-RSA-AES128-SHA256,DHE-RSA-AES128-GCM-
SHA256,DHE-RSA-AES256-SHA,DHE-RSA-AES256-SHA256,DHE-RSA-AES256-GCM-SHA384,ECDHE-RSA-
AES128-SHA,ECDHE-RSA-AES128-SHA256,ECDHE-RSA-AES128-GCM-SHA256,ECDHE-RSA-AES256-
SHA,ECDHE-RSA-AES256-SHA384,ECDHE-RSA-AES256-GCM-SHA384",
  "Description": "The list of permissible ciphers for connection encryption.",
  "Source": "system",
  "ApplyType": "static",
  "DataType": "list",
  "AllowedValues": "DHE-RSA-AES128-SHA,DHE-RSA-AES128-SHA256,DHE-RSA-AES128-GCM-
SHA256,DHE-RSA-AES256-SHA,DHE-RSA-AES256-SHA256,DHE-RSA-AES256-GCM-SHA384,ECDHE-
RSA-AES128-SHA,ECDHE-RSA-AES128-SHA256,ECDHE-RSA-AES128-GCM-SHA256,ECDHE-RSA-AES256-
SHA,ECDHE-RSA-AES256-SHA384,ECDHE-RSA-AES256-GCM-SHA384",
  "IsModifiable": true,
  "SupportedEngineModes": [
   "provisioned"
  ]
 },
       ...some output truncated...
```

For more information about ciphers, see the ssl_cipher variable in the MySQL documentation. For more information about cipher suite formats, see the openssl-ciphers list format and openssl-ciphers string format documentation on the OpenSSL website.

Encrypting connections to an Aurora MySQL DB cluster

To encrypt connections using the default `mysql` client, launch the mysql client using the `--ssl-ca` parameter to reference the public key, for example:

For MySQL 5.7 and 8.0:

```
mysql -h myinstance.123456789012.rds-us-east-1.amazonaws.com
--ssl-ca=full_path_to_CA_certificate --ssl-mode=VERIFY_IDENTITY
```

For MySQL 5.6:

```
mysql -h myinstance.123456789012.rds-us-east-1.amazonaws.com
```

```
--ssl-ca=full_path_to_CA_certificate --ssl-verify-server-cert
```

Replace *full_path_to_CA_certificate* with the full path to your Certificate Authority (CA) certificate. For information about downloading a certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

You can require TLS connections for specific users accounts. For example, you can use one of the following statements, depending on your MySQL version, to require TLS connections on the user account `encrypted_user`.

For MySQL 5.7 and 8.0:

```
ALTER USER 'encrypted_user'@'%' REQUIRE SSL;
```

For MySQL 5.6:

```
GRANT USAGE ON *.* TO 'encrypted_user'@'%' REQUIRE SSL;
```

When you use an RDS Proxy, you connect to the proxy endpoint instead of the usual cluster endpoint. You can make SSL/TLS required or optional for connections to the proxy, in the same way as for connections directly to the Aurora DB cluster. For information about using RDS Proxy, see Using Amazon RDS Proxy for Aurora.

Note

For more information on TLS connections with MySQL, see the MySQL documentation.

Updating applications to connect to Aurora MySQL DB clusters using new TLS certificates

As of January 13, 2023, Amazon RDS has published new Certificate Authority (CA) certificates for connecting to your Aurora DB clusters using Transport Layer Security (TLS). Following, you can find information about updating your applications to use the new certificates.

This topic can help you to determine whether any client applications use TLS to connect to your DB clusters. If they do, you can further check whether those applications require certificate verification to connect.

> **Note**
>
> Some applications are configured to connect to Aurora MySQL DB clusters only if they can successfully verify the certificate on the server.
> For such applications, you must update your client application trust stores to include the new CA certificates.

After you update your CA certificates in the client application trust stores, you can rotate the certificates on your DB clusters. We strongly recommend testing these procedures in a development or staging environment before implementing them in your production environments.

For more information about certificate rotation, see Rotating your SSL/TLS certificate. For more information about downloading certificates, see Using SSL/TLS to encrypt a connection to a DB cluster. For information about using TLS with Aurora MySQL DB clusters, see Using TLS with Aurora MySQL DB clusters.

Topics

Determining whether any applications are connecting to your Aurora MySQL DB cluster using TLS

If you are using Aurora MySQL version 2 (compatible with MySQL 5.7) and the Performance Schema is enabled, run the following query to check if connections are using TLS. For information about enabling the Performance Schema, see Performance Schema quick start in the MySQL documentation.

```
mysql> SELECT id, user, host, connection_type
       FROM performance_schema.threads pst
       INNER JOIN information_schema.processlist isp
       ON pst.processlist_id = isp.id;
```

In this sample output, you can see both your own session (`admin`) and an application logged in as `webapp1` are using TLS.

```
+----+-----------------+------------------+-----------------+
| id | user            | host             | connection_type |
+----+-----------------+------------------+-----------------+
|  8 | admin           | 10.0.4.249:42590 | SSL/TLS         |
|  4 | event_scheduler | localhost        | NULL            |
| 10 | webapp1         | 159.28.1.1:42189 | SSL/TLS         |
+----+-----------------+------------------+-----------------+
3 rows in set (0.00 sec)
```

Determining whether a client requires certificate verification to connect

You can check whether JDBC clients and MySQL clients require certificate verification to connect.

JDBC

The following example with MySQL Connector/J 8.0 shows one way to check an application's JDBC connection properties to determine whether successful connections require a valid certificate. For more information on all of the JDBC connection options for MySQL, see Configuration properties in the MySQL documentation.

When using the MySQL Connector/J 8.0, an TLS connection requires verification against the server CA certificate if your connection properties have `sslMode` set to `VERIFY_CA` or `VERIFY_IDENTITY`, as in the following example.

```
Properties properties = new Properties();
properties.setProperty("sslMode", "VERIFY_IDENTITY");
properties.put("user", DB_USER);
properties.put("password", DB_PASSWORD);
```

> **Note**
>
> If you use either the MySQL Java Connector v5.1.38 or later, or the MySQL Java Connector v8.0.9 or later to connect to your databases, even if you haven't explicitly configured your applications to use TLS when connecting to your databases, these client drivers default to using TLS. In addition, when using TLS, they perform partial certificate verification and fail to connect if the database server certificate is expired.

MySQL

The following examples with the MySQL Client show two ways to check a script's MySQL connection to determine whether successful connections require a valid certificate. For more information on all of the connection options with the MySQL Client, see Client-side configuration for encrypted connections in the MySQL documentation.

When using the MySQL 5.7 or MySQL 8.0 Client, an TLS connection requires verification against the server CA certificate if for the `--ssl-mode` option you specify `VERIFY_CA` or `VERIFY_IDENTITY`, as in the following example.

```
mysql -h mysql-database.rds.amazonaws.com -uadmin -ppassword --ssl-ca=/tmp/ssl-cert.pem
 --ssl-mode=VERIFY_CA
```

When using the MySQL 5.6 Client, an SSL connection requires verification against the server CA certificate if you specify the `--ssl-verify-server-cert` option, as in the following example.

```
mysql -h mysql-database.rds.amazonaws.com -uadmin -ppassword --ssl-ca=/tmp/ssl-cert.pem
 --ssl-verify-server-cert
```

Updating your application trust store

For information about updating the trust store for MySQL applications, see Installing SSL certificates in the MySQL documentation.

Note

When you update the trust store, you can retain older certificates in addition to adding the new certificates.

Updating your application trust store for JDBC

You can update the trust store for applications that use JDBC for TLS connections.

For information about downloading the root certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

For sample scripts that import certificates, see Sample script for importing certificates into your trust store.

If you are using the mysql JDBC driver in an application, set the following properties in the application.

```
System.setProperty("javax.net.ssl.trustStore", certs);
System.setProperty("javax.net.ssl.trustStorePassword", "password");
```

Note

Specify a password other than the prompt shown here as a security best practice.

When you start the application, set the following properties.

```
java -Djavax.net.ssl.trustStore=/path_to_truststore/MyTruststore.jks -
Djavax.net.ssl.trustStorePassword=my_truststore_password com.companyName.MyApplication
```

Example Java code for establishing TLS connections

The following code example shows how to set up the SSL connection that validates the server certificate using JDBC.

```
public class MySQLSSLTest {
        private static final String DB_USER = "user name";
        private static final String DB_PASSWORD = "password";
        // This key store has only the prod root ca.
        private static final String KEY_STORE_FILE_PATH = "file-path-to-keystore";
        private static final String KEY_STORE_PASS = "keystore-password";

    public static void test(String[] args) throws Exception {
        Class.forName("com.mysql.jdbc.Driver");


        System.setProperty("javax.net.ssl.trustStore", KEY_STORE_FILE_PATH);
        System.setProperty("javax.net.ssl.trustStorePassword", KEY_STORE_PASS);

        Properties properties = new Properties();
        properties.setProperty("sslMode", "VERIFY_IDENTITY");
        properties.put("user", DB_USER);
        properties.put("password", DB_PASSWORD);

        Connection connection = DriverManager.getConnection("jdbc:mysql://jagdeeps-ssl-
test.cni62e2e7kwh.us-east-1.rds.amazonaws.com:3306",properties);
        Statement stmt=connection.createStatement();

        ResultSet rs=stmt.executeQuery("SELECT 1 from dual");

        return;
    }
}
```

⚠ **Important**

After you have determined that your database connections use TLS and have updated your application trust store, you can update your database to use the rds-ca-rsa2048-g1 certificates. For instructions, see step 3 in Updating your CA certificate by modifying your DB instance .

Using Kerberos authentication for Aurora MySQL

You can use Kerberos authentication to authenticate users when they connect to your Aurora MySQL DB cluster. To do so, configure your DB cluster to use AWS Directory Service for Microsoft Active Directory for Kerberos authentication. AWS Directory Service for Microsoft Active Directory is also called AWS Managed Microsoft AD. It's a feature available with AWS Directory Service. To learn more, see What is AWS Directory Service? in the *AWS Directory Service Administration Guide*.

To start, create an AWS Managed Microsoft AD directory to store user credentials. Then, provide the Active Directory's domain and other information to your Aurora MySQL DB cluster. When users authenticate with the Aurora MySQL DB cluster, authentication requests are forwarded to the AWS Managed Microsoft AD directory.

Keeping all of your credentials in the same directory can save you time and effort. With this approach, you have a centralized location for storing and managing credentials for multiple DB clusters. Using a directory can also improve your overall security profile.

In addition, you can access credentials from your own on-premises Microsoft Active Directory. To do so, create a trusting domain relationship so that the AWS Managed Microsoft AD directory trusts your on-premises Microsoft Active Directory. In this way, your users can access your Aurora MySQL DB clusters with the same Windows single sign-on (SSO) experience as when they access workloads in your on-premises network.

A database can use Kerberos, AWS Identity and Access Management (IAM), or both Kerberos and IAM authentication. However, because Kerberos and IAM authentication provide different authentication methods, a specific user can log in to a database using only one or the other authentication method, but not both. For more information about IAM authentication, see IAM database authentication.

Contents

Overview of Kerberos authentication for Aurora MySQL DB clusters

To set up Kerberos authentication for an Aurora MySQL DB cluster, complete the following general steps. These steps are described in more detail later.

1. Use AWS Managed Microsoft AD to create an AWS Managed Microsoft AD directory. You can use the AWS Management Console, the AWS CLI, or the AWS Directory Service to create the directory. For detailed instructions, see Create your AWS Managed Microsoft AD directory in the *AWS Directory Service Administration Guide*.
2. Create an AWS Identity and Access Management (IAM) role that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`. The role allows Amazon Aurora to make calls to your directory.

   For the role to allow access, the AWS Security Token Service (AWS STS) endpoint must be activated in the AWS Region for your AWS account. AWS STS endpoints are active by default in all AWS Regions, and you can use them without any further action. For more information, see Activating and deactivating AWS STS in an AWS Region in the *IAM User Guide*.
3. Create and configure users in the AWS Managed Microsoft AD directory using the Microsoft Active Directory tools. For more information about creating users in your Active Directory,

see Manage users and groups in AWS managed Microsoft AD in the *AWS Directory Service Administration Guide*.

4. Create or modify an Aurora MySQL DB cluster. If you use either the CLI or RDS API in the create request, specify a domain identifier with the `Domain` parameter. Use the `d-*` identifier that was generated when you created your directory and the name of the IAM role that you created.

   If you modify an existing Aurora MySQL DB cluster to use Kerberos authentication, set the domain and IAM role parameters for the DB cluster. Locate the DB cluster in the same VPC as the domain directory.

5. Use the Amazon RDS primary user credentials to connect to the Aurora MySQL DB cluster. Create the database user in Aurora MySQL by using the instructions in Step 6: Create Aurora MySQL users that use Kerberos authentication.

   Users that you create this way can log in to the Aurora MySQL DB cluster using Kerberos authentication. For more information, see Connecting to Aurora MySQL with Kerberos authentication.

To use Kerberos authentication with an on-premises or self-hosted Microsoft Active Directory, create a *forest trust*. A forest trust is a trust relationship between two groups of domains. The trust can be one-way or two-way. For more information about setting up forest trusts using AWS Directory Service, see When to create a trust relationship in the *AWS Directory Service Administration Guide*.

Limitations of Kerberos authentication for Aurora MySQL

The following limitations apply to Kerberos authentication for Aurora MySQL:

- Kerberos authentication is supported for Aurora MySQL version 3.03 and higher.

  For information about AWS Region support, see Kerberos authentication with Aurora MySQL.

- To use Kerberos authentication with Aurora MySQL, your MySQL client or connector must use version 8.0.26 or higher on Unix platforms, 8.0.27 or higher on Windows. Otherwise, the client-side `authentication_kerberos_client` plugin isn't available and you can't authenticate.

- Only AWS Managed Microsoft AD is supported on Aurora MySQL. However, you can join Aurora MySQL DB clusters to shared Managed Microsoft AD domains owned by different accounts in the same AWS Region.

You can also use your own on-premises Active Directory. For more information, see Step 2: (Optional) Create a trust for an on-premises Active Directory

- When using Kerberos to authenticate a user connecting to the Aurora MySQL cluster from MySQL clients or from drivers on the Windows operating system, by default the character case of the database username must match the case of the user in the Active Directory. For example, if the user in the Active Directory appears as `Admin`, the database username must be `Admin`.

  However, you can now use case-insensitive username comparison with the `authentication_kerberos` plugin. For more information, see Step 8: (Optional) Configure case-insensitive username comparison.
- You must reboot the reader DB instances after turning on the feature to install the `authentication_kerberos` plugin.
- Replicating to DB instances that don't support the `authentication_kerberos` plugin can lead to replication failure.
- For Aurora global databases to use Kerberos authentication, you must configure it for every DB cluster in the global database.
- The domain name must be less than 62 characters long.
- Don't modify the DB cluster port after turning on Kerberos authentication. If you modify the port, then Kerberos authentication will no longer work.

Setting up Kerberos authentication for Aurora MySQL DB clusters

Use AWS Managed Microsoft AD to set up Kerberos authentication for an Aurora MySQL DB cluster. To set up Kerberos authentication, take the following steps.

Topics

- Step 8: (Optional) Configure case-insensitive username comparison

Step 1: Create a directory using AWS Managed Microsoft AD

AWS Directory Service creates a fully managed Active Directory in the AWS Cloud. When you create an AWS Managed Microsoft AD directory, AWS Directory Service creates two domain controllers and Domain Name System (DNS) servers on your behalf. The directory servers are created in different subnets in a VPC. This redundancy helps make sure that your directory remains accessible even if a failure occurs.

When you create an AWS Managed Microsoft AD directory, AWS Directory Service performs the following tasks on your behalf:

- Sets up an Active Directory within the VPC.
- Creates a directory administrator account with the username `Admin` and the specified password. You use this account to manage your directory.

> **Note**
>
> Be sure to save this password. AWS Directory Service doesn't store it. You can reset it, but you can't retrieve it.

- Creates a security group for the directory controllers.

When you launch an AWS Managed Microsoft AD, AWS creates an Organizational Unit (OU) that contains all of your directory's objects. This OU has the NetBIOS name that you entered when you created your directory. It is located in the domain root, which is owned and managed by AWS.

The `Admin` account that was created with your AWS Managed Microsoft AD directory has permissions for the most common administrative activities for your OU, including:

- Create, update, or delete users
- Add resources to your domain, such as file or print servers, and then assign permissions for those resources to users in your OU
- Create additional OUs and containers
- Delegate authority
- Restore deleted objects from the Active Directory Recycle Bin

- Run AD and DNS Windows PowerShell modules on the Active Directory Web Service

The `Admin` account also has rights to perform the following domain-wide activities:

- Manage DNS configurations (add, remove, or update records, zones, and forwarders)
- View DNS event logs
- View security event logs

To create a directory with AWS Managed Microsoft AD

1. Sign in to the AWS Management Console and open the AWS Directory Service console at https://console.aws.amazon.com/directoryservicev2/.
2. In the navigation pane, choose **Directories** and choose **Set up Directory**.
3. Choose **AWS Managed Microsoft AD**. AWS Managed Microsoft AD is the only option that you can currently use with Amazon RDS.
4. Enter the following information:

   Directory DNS name

      The fully qualified name for the directory, such as **`corp.example.com`**.

   Directory NetBIOS name

      The short name for the directory, such as **`CORP`**.

   Directory description

      (Optional) A description for the directory.

   Admin password

      The password for the directory administrator. The directory creation process creates an administrator account with the username Admin and this password.

      The directory administrator password and can't include the word "admin." The password is case-sensitive and must be 8–64 characters in length. It must also contain at least one character from three of the following four categories:

      - Lowercase letters (a–z)
      - Uppercase letters (A–Z)

    - Numbers (0–9)
    - Non-alphanumeric characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/)

   Confirm password

   The administrator password re-entered.

5. Choose **Next**.
6. Enter the following information in the **Networking** section and then choose **Next**:

   VPC

   The VPC for the directory. Create the Aurora MySQL DB cluster in this same VPC.

   Subnets

   Subnets for the directory servers. The two subnets must be in different Availability Zones.
7. Review the directory information and make any necessary changes. When the information is correct, choose **Create directory**.

It takes several minutes to create the directory. When it has been successfully created, the **Status** value changes to **Active**.

To see information about your directory, choose the directory name in the directory listing. Note the **Directory ID** value because you need this value when you create or modify your Aurora MySQL DB cluster.

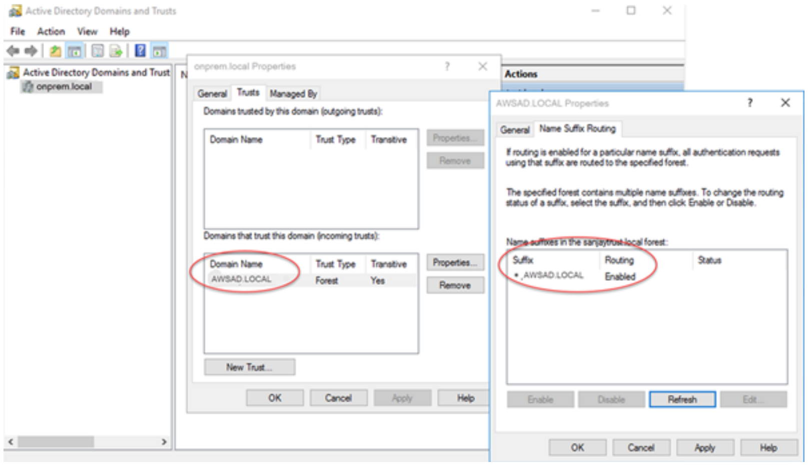
Step 2: (Optional) Create a trust for an on-premises Active Directory

If you don't plan to use your own on-premises Microsoft Active Directory, skip to Step 3: Create an IAM role for use by Amazon Aurora.

To use Kerberos authentication with your on-premises Active Directory, you need to create a trusting domain relationship using a forest trust between your on-premises Microsoft Active Directory and the AWS Managed Microsoft AD directory (created in Step 1: Create a directory using AWS Managed Microsoft AD). The trust can be one-way, where the AWS Managed Microsoft AD directory trusts the on-premises Microsoft Active Directory. The trust can also be two-way, where both Active Directories trust each other. For more information about setting up trusts using AWS Directory Service, see When to create a trust relationship in the *AWS Directory Service Administration Guide*.

> **Note**
>
> If you use an on-premises Microsoft Active Directory:
>
> - Windows clients must connect using the domain name of the AWS Directory Service in the endpoint rather than rds.amazonaws.com. For more information, see Connecting to Aurora MySQL with Kerberos authentication.
> - Windows clients can't connect using Aurora custom endpoints. To learn more, see Amazon Aurora connection management.
> - For global databases:
>   - Windows clients can connect using instance endpoints or cluster endpoints in the primary AWS Region of the global database only.
>   - Windows clients can't connect using cluster endpoints in secondary AWS Regions.

Make sure that your on-premises Microsoft Active Directory domain name includes a DNS suffix routing that corresponds to the newly created trust relationship. The following screenshot shows an example.



Domains trusted by this domain (outgoing trusts):

Domain Name	Trust Type	Transitive

Domains that trust this domain (incoming trusts):

Domain Name	Trust Type	Transitive
AWSAD.LOCAL	Forest	Yes

If routing is enabled for a particular name suffix, all authentication requests using that suffix are routed to the specified forest.

The specified forest contains multiple name suffixes. To change the routing status of a suffix, select the suffix, and then click Enable or Disable.

Name suffixes in the sanjaytrust.local forest:

Suffix	Routing	Status
*.AWSAD.LOCAL	Enabled	

Step 3: Create an IAM role for use by Amazon Aurora

For Amazon Aurora to call AWS Directory Service for you, you need an AWS Identity and Access Management (IAM) role that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`. This role allows Aurora to make calls to the AWS Directory Service.

When you create a DB cluster using the AWS Management Console, and you have the `iam:CreateRole` permission, the console creates this role automatically. In this case, the role name is `rds-directoryservice-kerberos-access-role`. Otherwise, you must create the IAM role manually. When you create this IAM role, choose `Directory Service`, and attach the AWS managed policy `AmazonRDSDirectoryServiceAccess` to it.

For more information about creating IAM roles for a service, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Optionally, you can create policies with the required permissions instead of using the managed IAM policy `AmazonRDSDirectoryServiceAccess`. In this case, the IAM role must have the following IAM trust policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "directoryservice.rds.amazonaws.com",
          "rds.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

The role must also have the following IAM role policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ds:DescribeDirectories",
        "ds:AuthorizeApplication",
        "ds:UnauthorizeApplication",
        "ds:GetAuthorizedApplicationDetails"
      ],
    "Effect": "Allow",
    "Resource": "*"
    }
  ]
}
```

Step 4: Create and configure users

You can create users with the Active Directory Users and Computers tool. This tool is part of the Active Directory Domain Services and Active Directory Lightweight Directory Services tools. Users represent individual people or entities that have access to your directory.

To create users in an AWS Directory Service directory, you use an on-premises or Amazon EC2 instance based on Microsoft Windows that is joined to your AWS Directory Service directory. You must be logged in to the instance as a user that has privileges to create users. For more information, see Manage users and groups in AWS Managed Microsoft AD in the *AWS Directory Service Administration Guide*.

Step 5: Create or modify an Aurora MySQL DB cluster

Create or modify an Aurora MySQL DB cluster for use with your directory. You can use the console, AWS CLI, or RDS API to associate a DB cluster with a directory. You can do this task in one of the following ways:

- Create a new Aurora MySQL DB cluster using the console, the create-db-cluster CLI command, or the CreateDBCluster RDS API operation.

  For instructions, see Creating an Amazon Aurora DB cluster.
- Modify an existing Aurora MySQL DB cluster using the console, the modify-db-cluster CLI command, or the ModifyDBCluster RDS API operation.

  For instructions, see Modifying an Amazon Aurora DB cluster.
- Restore an Aurora MySQL DB cluster from a DB snapshot using the console, the restore-db-cluster-from-snapshot CLI command, or the RestoreDBClusterFromSnapshot RDS API operation.

  For instructions, see Restoring from a DB cluster snapshot.
- Restore an Aurora MySQL DB cluster to a point-in-time using the console, the restore-db-cluster-to-point-in-time CLI command, or the RestoreDBClusterToPointInTime RDS API operation.

For instructions, see Restoring a DB cluster to a specified time.

Kerberos authentication is only supported for Aurora MySQL DB clusters in a VPC. The DB cluster can be in the same VPC as the directory, or in a different VPC. The DB cluster's VPC must have a VPC security group that allows outbound communication to your directory.

Console

When you use the console to create, modify, or restore a DB cluster, choose **Kerberos authentication** in the **Database authentication** section. Choose **Browse Directory** and then select the directory, or choose **Create a new directory**.

AWS CLI

When you use the AWS CLI or RDS API, associate a DB cluster with a directory. The following parameters are required for the DB cluster to use the domain directory you created:

- For the `--domain` parameter, use the domain identifier ("d-*" identifier) generated when you created the directory.
- For the `--domain-iam-role-name` parameter, use the role you created that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`.

For example, the following CLI command modifies a DB cluster to use a directory.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --domain d-ID \
    --domain-iam-role-name role-name
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --domain d-ID ^
    --domain-iam-role-name role-name
```

⚠ **Important**

If you modify a DB cluster to turn on Kerberos authentication, reboot the reader DB instances after making the change.

Step 6: Create Aurora MySQL users that use Kerberos authentication

The DB cluster is joined to the AWS Managed Microsoft AD domain. Thus, you can create Aurora MySQL users from the Active Directory users in your domain. Database permissions are managed through standard Aurora MySQL permissions that are granted to and revoked from these users.

You can allow an Active Directory user to authenticate with Aurora MySQL. To do this, first use the Amazon RDS primary user credentials to connect to the Aurora MySQL DB cluster as with any other DB cluster. After you're logged in, create an externally authenticated user with Kerberos authentication in Aurora MySQL as shown here:

```
CREATE USER user_name@'host_name' IDENTIFIED WITH 'authentication_kerberos' BY
 'realm_name';
```

- Replace *user_name* with the username. Users (both humans and applications) from your domain can now connect to the DB cluster from a domain-joined client machine using Kerberos authentication.
- Replace *host_name* with the hostname. You can use `%` as a wild card. You can also use specific IP addresses for the hostname.
- Replace *realm_name* with the directory realm name of the domain. The realm name is usually the same as the DNS domain name in uppercase letters, such as `CORP.EXAMPLE.COM`. A realm is a group of systems that use the same Kerberos Key Distribution Center.

The following example creates a database user with the name `Admin` that authenticates against the Active Directory with the realm name `MYSQL.LOCAL`.

```
CREATE USER Admin@'%' IDENTIFIED WITH 'authentication_kerberos' BY 'MYSQL.LOCAL';
```

Modifying an existing Aurora MySQL login

You can also modify an existing Aurora MySQL login to use Kerberos authentication by using the following syntax:

```
ALTER USER user_name IDENTIFIED WITH 'authentication_kerberos' BY 'realm_name';
```

Step 7: Configure a MySQL client

To configure a MySQL client, take the following steps:

1. Create a `krb5.conf` file (or equivalent) to point to the domain.
2. Verify that traffic can flow between the client host and AWS Directory Service. Use a network utility such as Netcat, for the following:
   - Verify traffic over DNS for port 53.
   - Verify traffic over TCP/UDP for port 53 and for Kerberos, which includes ports 88 and 464 for AWS Directory Service.
3. Verify that traffic can flow between the client host and the DB instance over the database port. For example, use `mysql` to connect and access the database.

The following is sample `krb5.conf` content for AWS Managed Microsoft AD.

```
[libdefaults]
 default_realm = EXAMPLE.COM
[realms]
 EXAMPLE.COM = {
  kdc = example.com
  admin_server = example.com
 }
[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
```

The following is sample `krb5.conf` content for an on-premises Microsoft Active Directory.

```
[libdefaults]
 default_realm = EXAMPLE.COM
[realms]
 EXAMPLE.COM = {
  kdc = example.com
  admin_server = example.com
 }
 ONPREM.COM = {
  kdc = onprem.com
```

```
   admin_server = onprem.com
  }
 [domain_realm]
  .example.com = EXAMPLE.COM
  example.com = EXAMPLE.COM
  .onprem.com = ONPREM.COM
  onprem.com = ONPREM.COM
  .rds.amazonaws.com = EXAMPLE.COM
  .amazonaws.com.cn = EXAMPLE.COM
  .amazon.com = EXAMPLE.COM
```

Step 8: (Optional) Configure case-insensitive username comparison

By default, the character case of the MySQL database username must match that of the Active Directory login. However, you can now use case-insensitive username comparison with the `authentication_kerberos` plugin. To do so, you set the `authentication_kerberos_caseins_cmp` DB cluster parameter to `true`.

To use case-insensitive username comparison

1. Create a custom DB cluster parameter group. Follow the procedures in [Creating a DB cluster parameter group](#).
2. Edit the new parameter group to set the value of `authentication_kerberos_caseins_cmp` to `true`. Follow the procedures in [Modifying parameters in a DB cluster parameter group](#).
3. Associate the DB cluster parameter group with your Aurora MySQL DB cluster. Follow the procedures in [Associating a DB cluster parameter group with a DB cluster](#).
4. Reboot the DB cluster.

Connecting to Aurora MySQL with Kerberos authentication

To avoid errors, use a MySQL client with version 8.0.26 or higher on Unix platforms, 8.0.27 or higher on Windows.

Using the Aurora MySQL Kerberos login to connect to the DB cluster

To connect to Aurora MySQL with Kerberos authentication, you log in as a database user that you created using the instructions in [Step 6: Create Aurora MySQL users that use Kerberos authentication](#).

At a command prompt, connect to one of the endpoints associated with your Aurora MySQL DB cluster. When you're prompted for the password, enter the Kerberos password associated with that username.

When you authenticate with Kerberos, a *ticket-granting ticket* (TGT) is generated if one doesn't already exist. The `authentication_kerberos` plugin uses the TGT to get a *service ticket*, which is then presented to the Aurora MySQL database server.

You can use the MySQL client to connect to Aurora MySQL with Kerberos authentication using either Windows or Unix.

Unix

You can connect by using either one of the following methods:

- Obtain the TGT manually. In this case, you don't need to supply the password to the MySQL client.
- Supply the password for the Active Directory login directly to the MySQL client.

The client-side plugin is supported on Unix platforms for MySQL client versions 8.0.26 and higher.

To connect by obtaining the TGT manually

1. At the command line interface, use the following command to obtain the TGT.

   ```
   kinit user_name
   ```

2. Use the following `mysql` command to log in to the DB instance endpoint of your DB cluster.

   ```
   mysql -h DB_instance_endpoint -P 3306 -u user_name -p
   ```

   > **Note**
   >
   > Authentication can fail if the keytab is rotated on the DB instance. In this case, obtain a new TGT by rerunning `kinit`.

To connect directly

1. At the command line interface, use the following mysql command to log in to the DB instance endpoint of your DB cluster.

   ```
   mysql -h DB_instance_endpoint -P 3306 -u user_name -p
   ```

2. Enter the password for the Active Directory user.

Windows

On Windows, authentication is usually done at login time, so you don't need to obtain the TGT manually to connect to the Aurora MySQL DB cluster. The case of the database username must match the character case of the user in the Active Directory. For example, if the user in the Active Directory appears as Admin, the database username must be Admin.

The client-side plugin is supported on Windows for MySQL client versions 8.0.27 and higher.

To connect directly

- At the command line interface, use the following mysql command to log in to the DB instance endpoint of your DB cluster.

  ```
  mysql -h DB_instance_endpoint -P 3306 -u user_name
  ```

Kerberos authentication with Aurora global databases

Kerberos authentication for Aurora MySQL is supported for Aurora global databases. To authenticate users on the secondary DB cluster using the Active Directory of the primary DB cluster, replicate the Active Directory to the secondary AWS Region. You turn on Kerberos authentication on the secondary cluster using the same domain ID as for the primary cluster. AWS Managed Microsoft AD replication is supported only with the Enterprise version of Active Directory. For more information, see Multi-Region replication in the *AWS Directory Service Administration Guide*.

Migrating from RDS for MySQL to Aurora MySQL

After you migrate from RDS for MySQL with Kerberos authentication enabled to Aurora MySQL, modify users created with the `auth_pam` plugin to use the `authentication_kerberos` plugin. For example:

```
ALTER USER user_name IDENTIFIED WITH 'authentication_kerberos' BY 'realm_name';
```

Preventing ticket caching

If a valid TGT doesn't exist when the MySQL client application starts, the application can obtain and cache the TGT. If you want to prevent the TGT from being cached, set a configuration parameter in the `/etc/krb5.conf` file.

> **Note**
>
> This configuration only applies to client hosts running Unix, not Windows.

To prevent TGT caching

- Add an `[appdefaults]` section to `/etc/krb5.conf` as follows:

  ```
  [appdefaults]
    mysql = {
      destroy_tickets = true
    }
  ```

Logging for Kerberos authentication

The `AUTHENTICATION_KERBEROS_CLIENT_LOG` environment variable sets the logging level for Kerberos authentication. You can use the logs for client-side debugging.

The permitted values are 1–5. Log messages are written to the standard error output. The following table describes each logging level.

Logging level	Description
1 or not set	No logging

Logging level	Description
2	Error messages
3	Error and warning messages
4	Error, warning, and information messages
5	Error, warning, information, and debug messages

Managing a DB cluster in a domain

You can use the AWS CLI or the RDS API to manage your DB cluster and its relationship with your managed Active Directory. For example, you can associate an Active Directory for Kerberos authentication and disassociate an Active Directory to turn off Kerberos authentication. You can also move a DB cluster to be externally authenticated by one Active Directory to another.

For example, using the Amazon RDS API, you can do the following:

- To reattempt turning on Kerberos authentication for a failed membership, use the `ModifyDBInstance` API operation and specify the current membership's directory ID.
- To update the IAM role name for membership, use the `ModifyDBInstance` API operation and specify the current membership's directory ID and the new IAM role.
- To turn off Kerberos authentication on a DB cluster, use the `ModifyDBInstance` API operation and specify `none` as the domain parameter.
- To move a DB cluster from one domain to another, use the `ModifyDBInstance` API operation and specify the domain identifier of the new domain as the domain parameter.
- To list membership for each DB cluster, use the `DescribeDBInstances` API operation.

Understanding domain membership

After you create or modify your DB cluster, it becomes a member of the domain. You can view the status of the domain membership for the DB cluster by running the describe-db-clusters CLI command. The status of the DB cluster can be one of the following:

- `kerberos-enabled` – The DB cluster has Kerberos authentication turned on.

- `enabling-kerberos` – AWS is in the process of turning on Kerberos authentication on this DB cluster.
- `pending-enable-kerberos` – Turning on Kerberos authentication is pending on this DB cluster.
- `pending-maintenance-enable-kerberos` – AWS will attempt to turn on Kerberos authentication on the DB cluster during the next scheduled maintenance window.
- `pending-disable-kerberos` – Turning off Kerberos authentication is pending on this DB cluster.
- `pending-maintenance-disable-kerberos` – AWS will attempt to turn off Kerberos authentication on the DB cluster during the next scheduled maintenance window.
- `enable-kerberos-failed` – A configuration problem has prevented AWS from turning on Kerberos authentication on the DB cluster. Check and fix your configuration before reissuing the DB cluster modify command.
- `disabling-kerberos` – AWS is in the process of turning off Kerberos authentication on this DB cluster.

A request to turn on Kerberos authentication can fail because of a network connectivity issue or an incorrect IAM role. For example, suppose that you create a DB cluster or modify an existing DB cluster and the attempt to turn on Kerberos authentication fails. In this case, reissue the modify command or modify the newly created DB cluster to join the domain.

Migrating data to an Amazon Aurora MySQL DB cluster

You have several options for migrating data from your existing database to an Amazon Aurora MySQL DB cluster. Your migration options also depend on the database that you are migrating from and the size of the data that you are migrating.

There are two different types of migration: physical and logical. Physical migration means that physical copies of database files are used to migrate the database. Logical migration means that the migration is accomplished by applying logical database changes, such as inserts, updates, and deletes.

Physical migration has the following advantages:

- Physical migration is faster than logical migration, especially for large databases.
- Database performance does not suffer when a backup is taken for physical migration.
- Physical migration can migrate everything in the source database, including complex database components.

Physical migration has the following limitations:

- The `innodb_page_size` parameter must be set to its default value (16KB).
- The `innodb_data_file_path` parameter must be configured with only one data file that uses the default data file name "`ibdata1:12M:autoextend`". Databases with two data files, or with a data file with a different name, can't be migrated using this method.

  The following are examples of file names that are not allowed: "`innodb_data_file_path=ibdata1:50M; ibdata2:50M:autoextend`" and "`innodb_data_file_path=ibdata01:50M:autoextend`".
- The `innodb_log_files_in_group` parameter must be set to its default value (2).

Logical migration has the following advantages:

- You can migrate subsets of the database, such as specific tables or parts of a table.
- The data can be migrated regardless of the physical storage structure.

Logical migration has the following limitations:

- Logical migration is usually slower than physical migration.
- Complex database components can slow down the logical migration process. In some cases, complex database components can even block logical migration.

The following table describes your options and the type of migration for each option.

Migrating from	Migration type	Solution
An RDS for MySQL DB instance	Physical	You can migrate from an RDS for MySQL DB instance by first creating an Aurora MySQL read replica of a MySQL DB instance. When the replica lag between the MySQL DB instance and the Aurora MySQL read replica is 0, you can direct your client applicati ons to read from the Aurora read replica and then stop replication to make the Aurora MySQL read replica a standalone Aurora MySQL DB cluster for reading and writing. For details, see Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica.
An RDS for MySQL DB snapshot	Physical	You can migrate data directly from an RDS for MySQL DB snapshot to an Amazon Aurora MySQL DB cluster. For details, see Migrating an RDS for MySQL snapshot to Aurora.
A MySQL database external to Amazon RDS	Logical	You can create a dump of your data using the `mysqldump` utility, and then import that data into an existing Amazon Aurora MySQL DB cluster. For details, see Logical migration from MySQL to Amazon Aurora MySQL by using mysqldump. To export metadata for database users during the migration from an external MySQL database, you can also use

Migrating from	Migration type	Solution
		a MySQL Shell command instead of `mysqldump` . For more information, see Instance Dump Utility, Schema Dump Utility, and Table Dump Utility. **Note** The mysqlpump utility is deprecated as of MySQL 8.0.34.
A MySQL database external to Amazon RDS	Physical	You can copy the backup files from your database to an Amazon Simple Storage Service (Amazon S3) bucket, and then restore an Amazon Aurora MySQL DB cluster from those files. This option can be considerably faster than migrating data using `mysqldump` . For details, see Physical migration from MySQL by using Percona XtraBackup and Amazon S3.
A MySQL database external to Amazon RDS	Logical	You can save data from your database as text files and copy those files to an Amazon S3 bucket. You can then load that data into an existing Aurora MySQL DB cluster using the `LOAD DATA FROM S3` MySQL command. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.

Migrating from	Migration type	Solution
A database that isn't MySQL-compatible	Logical	You can use AWS Database Migration Service (AWS DMS) to migrate data from a database that isn't MySQL-compatible. For more information on AWS DMS, see What is AWS database migration service?

Note

If you're migrating a MySQL database external to Amazon RDS, the migration options described in the table are supported only if your database supports the InnoDB or MyISAM tablespaces.

If the MySQL database you're migrating to Aurora MySQL uses `memcached`, remove `memcached` before migrating it.

You can't migrate to Aurora MySQL version 3.05 and higher from some older MySQL 8.0 versions, including 8.0.11, 8.0.13, and 8.0.15. We recommend that you upgrade to MySQL version 8.0.28 before migrating.

Migrating data from an external MySQL database to an Amazon Aurora MySQL DB cluster

If your database supports the InnoDB or MyISAM tablespaces, you have these options for migrating your data to an Amazon Aurora MySQL DB cluster:

- You can create a dump of your data using the `mysqldump` utility, and then import that data into an existing Amazon Aurora MySQL DB cluster. For more information, see Logical migration from MySQL to Amazon Aurora MySQL by using mysqldump.
- You can copy the full and incremental backup files from your database to an Amazon S3 bucket, and then restore to an Amazon Aurora MySQL DB cluster from those files. This option can be considerably faster than migrating data using `mysqldump`. For more information, see Physical migration from MySQL by using Percona XtraBackup and Amazon S3.

Topics

Physical migration from MySQL by using Percona XtraBackup and Amazon S3

You can copy the full and incremental backup files from your source MySQL version 5.7 or 8.0 database to an Amazon S3 bucket. Then you can restore to an Amazon Aurora MySQL DB cluster with the same major DB engine version from those files.

This option can be considerably faster than migrating data using `mysqldump`, because using `mysqldump` replays all of the commands to recreate the schema and data from your source database in your new Aurora MySQL DB cluster. By copying your source MySQL data files, Aurora MySQL can immediately use those files as the data for an Aurora MySQL DB cluster.

You can also minimize downtime by using binary log replication during the migration process. If you use binary log replication, the external MySQL database remains open to transactions while the data is being migrated to the Aurora MySQL DB cluster. After the Aurora MySQL DB cluster has been created, you use binary log replication to synchronize the Aurora MySQL DB cluster with the transactions that happened after the backup. When the Aurora MySQL DB cluster is caught up with the MySQL database, you finish the migration by completely switching to the Aurora MySQL DB cluster for new transactions. For more information, see Synchronizing the Amazon Aurora MySQL DB cluster with the MySQL database using replication.

Contents

Limitations and considerations

The following limitations and considerations apply to restoring to an Amazon Aurora MySQL DB cluster from an Amazon S3 bucket:

- You can migrate your data only to a new DB cluster, not an existing DB cluster.
- You must use Percona XtraBackup to back up your data to S3. For more information, see Installing Percona XtraBackup.
- The Amazon S3 bucket and the Aurora MySQL DB cluster must be in the same AWS Region.
- You can't restore from the following:
  - A DB cluster snapshot export to Amazon S3. You also can't migrate data from a DB cluster snapshot export to your S3 bucket.

  - An encrypted source database, but you can encrypt the data being migrated. You can also leave the data unencrypted during the migration process.
  - A MySQL 5.5 or 5.6 database
- Percona Server for MySQL isn't supported as a source database, because it can contain `compression_dictionary*` tables in the `mysql` schema.
- You can't restore to an Aurora Serverless DB cluster.
- Backward migration isn't supported for either major versions or minor versions. For example, you can't migrate from MySQL version 8.0 to Aurora MySQL version 2 (compatible with MySQL 5.7), and you can't migrate from MySQL version 8.0.32 to Aurora MySQL version 3.03, which is compatible with MySQL community version 8.0.26.
- You can't migrate to Aurora MySQL version 3.05 and higher from some older MySQL 8.0 versions, including 8.0.11, 8.0.13, and 8.0.15. We recommend that you upgrade to MySQL version 8.0.28 before migrating.
- Importing from Amazon S3 isn't supported on the db.t2.micro DB instance class. However, you can restore to a different DB instance class, and change the DB instance class later. For more information about DB instance classes, see Aurora DB instance classes.
- Amazon S3 limits the size of a file uploaded to an S3 bucket to 5 TB. If a backup file exceeds 5 TB, then you must split the backup file into smaller files.
- Amazon RDS limits the number of files uploaded to an S3 bucket to 1 million. If the backup data for your database, including all full and incremental backups, exceeds 1 million files, use a Gzip (.gz), tar (.tar.gz), or Percona xbstream (.xbstream) file to store full and incremental backup files in the S3 bucket. Percona XtraBackup 8.0 only supports Percona xbstream for compression.
- To provide management services for each DB cluster, the `rdsadmin` user is created when the DB cluster is created. As this is a reserved user in RDS, the following limitations apply:
  - Functions, procedures, views, events, and triggers with the `'rdsadmin'@'localhost'` definer aren't imported. For more information, see Stored objects with 'rdsadmin'@'localhost' as the definer and Master user privileges with Amazon Aurora MySQL.
  - When the Aurora MySQL DB cluster is created, a master user is created with the maximum privileges supported. While restoring from backup, any unsupported privileges assigned to users being imported are removed automatically during import.

    To identify users that might be affected by this, see User accounts with unsupported privileges. For more information on supported privileges in Aurora MySQL, see Role-based privilege model.

- For Aurora MySQL version 3, dynamic privileges aren't imported. Aurora-supported dynamic privileges can be imported after migration. For more information, see Dynamic privileges in Aurora MySQL version 3.
- User-created tables in the `mysql` schema aren't migrated.
- The `innodb_data_file_path` parameter must be configured with only one data file that uses the default data file name `ibdata1:12M:autoextend`. Databases with two data files, or with a data file with a different name, can't be migrated using this method.

  The following are examples of file names that aren't allowed: `innodb_data_file_path=ibdata1:50M`, `ibdata2:50M:autoextend`, and `innodb_data_file_path=ibdata01:50M:autoextend`.
- You can't migrate from a source database that has tables defined outside of the default MySQL data directory.
- The maximum supported size for uncompressed backups using this method is currently limited to 64 TiB. For compressed backups, this limit goes lower to account for the uncompression space requirements. In such cases, the maximum supported backup size would be (`64 TiB – compressed backup size`).
- Aurora MySQL doesn't support the importing of MySQL and other external components and plugins.
- Aurora MySQL doesn't restore everything from your database. We recommend that you save the database schema and values for the following items from your source MySQL database, then add them to your restored Aurora MySQL DB cluster after it has been created:
  - User accounts
  - Functions
  - Stored procedures
  - Time zone information. Time zone information is loaded from the local operating system of your Aurora MySQL DB cluster. For more information, see Local time zone for Amazon Aurora DB clusters.

Before you begin

Before you can copy your data to an Amazon S3 bucket and restore to a DB cluster from those files, you must do the following:

- Install Percona XtraBackup on your local server.

- Permit Aurora MySQL to access your Amazon S3 bucket on your behalf.

Installing Percona XtraBackup

Amazon Aurora can restore a DB cluster from files that were created using Percona XtraBackup. You can install Percona XtraBackup from Software Downloads - Percona.

For MySQL 5.7 migration, use Percona XtraBackup 2.4.

For MySQL 8.0 migration, use Percona XtraBackup 8.0. Make sure that the Percona XtraBackup version is compatible with the engine version of your source database.

Required permissions

To migrate your MySQL data to an Amazon Aurora MySQL DB cluster, several permissions are required:

- The user that is requesting that Aurora create a new cluster from an Amazon S3 bucket must have permission to list the buckets for your AWS account. You grant the user this permission using an AWS Identity and Access Management (IAM) policy.
- Aurora requires permission to act on your behalf to access the Amazon S3 bucket where you store the files used to create your Amazon Aurora MySQL DB cluster. You grant Aurora the required permissions using an IAM service role.
- The user making the request must also have permission to list the IAM roles for your AWS account.
- If the user making the request is to create the IAM service role or request that Aurora create the IAM service role (by using the console), then the user must have permission to create an IAM role for your AWS account.
- If you plan to encrypt the data during the migration process, update the IAM policy of the user who will perform the migration to grant RDS access to the AWS KMS keys used for encrypting the backups. For instructions, see Creating an IAM policy to access AWS KMS resources.

For example, the following IAM policy grants a user the minimum required permissions to use the console to list IAM roles, create an IAM role, list the Amazon S3 buckets for your account, and list the KMS keys.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iam:ListRoles",
                "iam:CreateRole",
                "iam:CreatePolicy",
                "iam:AttachRolePolicy",
                "s3:ListBucket",
                "kms:ListKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

Additionally, for a user to associate an IAM role with an Amazon S3 bucket, the IAM user must have the `iam:PassRole` permission for that IAM role. This permission allows an administrator to restrict which IAM roles a user can associate with Amazon S3 buckets.

For example, the following IAM policy allows a user to associate the role named `S3Access` with an Amazon S3 bucket.

```
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"AllowS3AccessRole",
            "Effect":"Allow",
            "Action":"iam:PassRole",
            "Resource":"arn:aws:iam::123456789012:role/S3Access"
        }
    ]
}
```

For more information on IAM user permissions, see Managing access using policies.

Creating the IAM service role

You can have the AWS Management Console create a role for you by choosing the **Create a New Role** option (shown later in this topic). If you select this option and specify a name for the new role, then Aurora creates the IAM service role required for Aurora to access your Amazon S3 bucket with the name that you supply.

As an alternative, you can manually create the role using the following procedure.

To create an IAM role for Aurora to access Amazon S3

1. Complete the steps in Creating an IAM policy to access Amazon S3 resources.
2. Complete the steps in Creating an IAM role to allow Amazon Aurora to access AWS services.
3. Complete the steps in Associating an IAM role with an Amazon Aurora MySQL DB cluster.

Backing up files to be restored as an Amazon Aurora MySQL DB cluster

You can create a full backup of your MySQL database files using Percona XtraBackup and upload the backup files to an Amazon S3 bucket. Alternatively, if you already use Percona XtraBackup to back up your MySQL database files, you can upload your existing full and incremental backup directories and files to an Amazon S3 bucket.

Topics

- Creating a full backup with Percona XtraBackup
- Using incremental backups with Percona XtraBackup
- Backup considerations

Creating a full backup with Percona XtraBackup

To create a full backup of your MySQL database files that can be restored from Amazon S3 to create an Aurora MySQL DB cluster, use the Percona XtraBackup utility (`xtrabackup`) to back up your database.

For example, the following command creates a backup of a MySQL database and stores the files in the `/on-premises/s3-restore/backup` folder.

```
xtrabackup --backup --user=<myuser> --password=<password> --target-dir=</on-premises/
s3-restore/backup>
```

If you want to compress your backup into a single file (which can be split, if needed), you can use the `--stream` option to save your backup in one of the following formats:

- Gzip (.gz)
- tar (.tar)
- Percona xbstream (.xbstream)

The following command creates a backup of your MySQL database split into multiple Gzip files.

```
xtrabackup --backup --user=<myuser> --password=<password> --stream=tar \
   --target-dir=</on-premises/s3-restore/backup> | gzip - | split -d --bytes=500MB \
   - </on-premises/s3-restore/backup/backup>.tar.gz
```

The following command creates a backup of your MySQL database split into multiple tar files.

```
xtrabackup --backup --user=<myuser> --password=<password> --stream=tar \
   --target-dir=</on-premises/s3-restore/backup> | split -d --bytes=500MB \
   - </on-premises/s3-restore/backup/backup>.tar
```

The following command creates a backup of your MySQL database split into multiple xbstream files.

```
xtrabackup --backup --user=<myuser> --password=<password> --stream=xbstream \
   --target-dir=</on-premises/s3-restore/backup> | split -d --bytes=500MB \
   - </on-premises/s3-restore/backup/backup>.xbstream
```

Note

If you see the following error, it might be caused by mixing file formats in your command:

```
ERROR:/bin/tar: This does not look like a tar archive
```

Once you have backed up your MySQL database using the Percona XtraBackup utility, you can copy your backup directories and files to an Amazon S3 bucket.

For information on creating and uploading a file to an Amazon S3 bucket, see Getting started with Amazon Simple Storage Service in the *Amazon S3 Getting Started Guide*.

Using incremental backups with Percona XtraBackup

Amazon Aurora MySQL supports both full and incremental backups created using Percona XtraBackup. If you already use Percona XtraBackup to perform full and incremental backups of your MySQL database files, you don't need to create a full backup and upload the backup files to Amazon S3. Instead, you can save a significant amount of time by copying your existing backup directories and files for your full and incremental backups to an Amazon S3 bucket. For more information, see Create an incremental backup on the Percona website.

When copying your existing full and incremental backup files to an Amazon S3 bucket, you must recursively copy the contents of the base directory. Those contents include the full backup and also all incremental backup directories and files. This copy must preserve the directory structure in the Amazon S3 bucket. Aurora iterates through all files and directories. Aurora uses the `xtrabackup-checkpoints` file included with each incremental backup to identify the base directory and to order incremental backups by log sequence number (LSN) range.

For information on creating and uploading a file to an Amazon S3 bucket, see Getting started with Amazon Simple Storage Service in the *Amazon S3 Getting Started Guide*.

Backup considerations

Aurora doesn't support partial backups created using Percona XtraBackup. You can't use the following options to create a partial backup when you back up the source files for your database: `--tables`, `--tables-exclude`, `--tables-file`, `--databases`, `--databases-exclude`, or `--databases-file`.

For more information about backing up your database with Percona XtraBackup, see Percona XtraBackup - Documentation and Work with binary logs on the Percona website.

Aurora supports incremental backups created using Percona XtraBackup. For more information, see Create an incremental backup on the Percona website.

Aurora consumes your backup files based on the file name. Be sure to name your backup files with the appropriate file extension based on the file format—for example, `.xbstream` for files stored using the Percona xbstream format.

Aurora consumes your backup files in alphabetical order and also in natural number order. Always use the `split` option when you issue the `xtrabackup` command to ensure that your backup files are written and named in the proper order.

Amazon S3 limits the size of a file uploaded to an Amazon S3 bucket to 5 TB. If the backup data for your database exceeds 5 TB, use the `split` command to split the backup files into multiple files that are each less than 5 TB.

Aurora limits the number of source files uploaded to an Amazon S3 bucket to 1 million files. In some cases, backup data for your database, including all full and incremental backups, can come to a large number of files. In these cases, use a tarball (.tar.gz) file to store full and incremental backup files in the Amazon S3 bucket.

When you upload a file to an Amazon S3 bucket, you can use server-side encryption to encrypt the data. You can then restore an Amazon Aurora MySQL DB cluster from those encrypted files. Amazon Aurora MySQL can restore a DB cluster with files encrypted using the following types of server-side encryption:

- Server-side encryption with Amazon S3–managed keys (SSE-S3) – Each object is encrypted with a unique key employing strong multifactor encryption.
- Server-side encryption with AWS KMS–managed keys (SSE-KMS) – Similar to SSE-S3, but you have the option to create and manage encryption keys yourself, and also other differences.

For information about using server-side encryption when uploading files to an Amazon S3 bucket, see Protecting data using server-side encryption in the *Amazon S3 Developer Guide*.

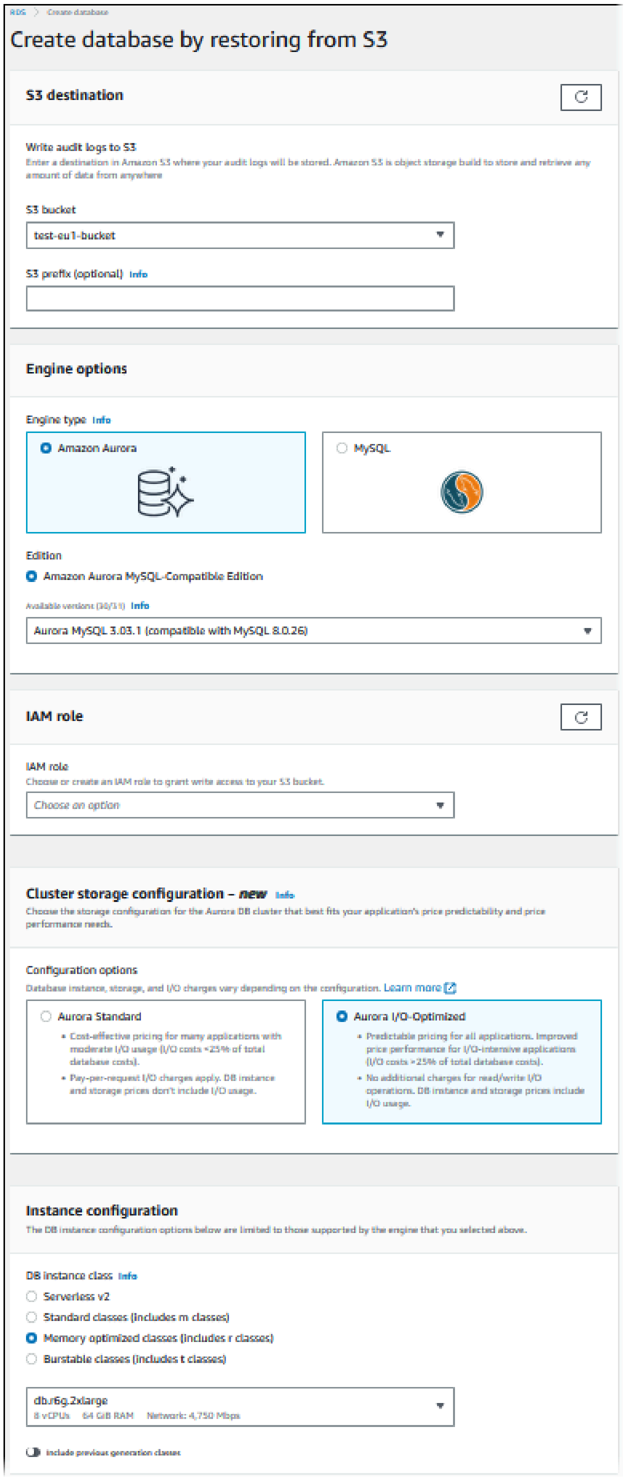
Restoring an Amazon Aurora MySQL DB cluster from an Amazon S3 bucket

You can restore your backup files from your Amazon S3 bucket to create a new Amazon Aurora MySQL DB cluster by using the Amazon RDS console.

To restore an Amazon Aurora MySQL DB cluster from files on an Amazon S3 bucket

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the top right corner of the Amazon RDS console, choose the AWS Region in which to create your DB cluster. Choose the same AWS Region as the Amazon S3 bucket that contains your database backup.
3. In the navigation pane, choose **Databases**, and then choose **Restore from S3**.
4. Choose **Restore from S3**.

   The **Create database by restoring from S3** page appears.



Create database by restoring from S3

S3 destination

Write audit logs to S3

Enter a destination in Amazon S3 where your audit logs will be stored. Amazon S3 is object storage build to store and retrieve any amount of data from anywhere

S3 bucket: test-eu1-bucket

S3 prefix (optional) Info

Engine options

Engine type Info

● Amazon Aurora

○ MySQL

Edition

● Amazon Aurora MySQL-Compatible Edition

Available versions (30/31) Info

Aurora MySQL 3.03.1 (compatible with MySQL 8.0.26)

IAM role

IAM role

Choose or create an IAM role to grant write access to your S3 bucket.

Choose an option

Cluster storage configuration – *new* Info

Choose the storage configuration for the Aurora DB cluster that best fits your application's price predictability and price performance needs.

Configuration options

Database instance, storage, and I/O charges vary depending on the configuration. Learn more

○ Aurora Standard

- Cost-effective pricing for many applications with moderate I/O usage (I/O costs <25% of total database costs).
- Pay-per-request I/O charges apply. DB instance and storage prices don't include I/O usage.

● Aurora I/O-Optimized

- Predictable pricing for all applications. Improved price performance for I/O-intensive applications (I/O costs >25% of total database costs).
- No additional charges for read/write I/O operations. DB instance and storage prices include I/O usage.

Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

○ Serverless v2

○ Standard classes (includes m classes)

● Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

db.r6g.2xlarge
8 vCPUs 64 GiB RAM Network: 4,750 Mbps

Include previous generation classes

5. Under **S3 destination**:

   a. Choose the **S3 bucket** that contains the backup files.

   b. (Optional) For **S3 folder path prefix**, enter a file path prefix for the files stored in your Amazon S3 bucket.

If you don't specify a prefix, then RDS creates your DB instance using all of the files and folders in the root folder of the S3 bucket. If you do specify a prefix, then RDS creates your DB instance using the files and folders in the S3 bucket where the path for the file begins with the specified prefix.

For example, suppose that you store your backup files on S3 in a subfolder named backups, and you have multiple sets of backup files, each in its own directory (gzip_backup1, gzip_backup2, and so on). In this case, you specify a prefix of backups/gzip_backup1 to restore from the files in the gzip_backup1 folder.

6. Under **Engine options**:

   a. For **Engine type**, choose **Amazon Aurora**.

   b. For **Version**, choose the Aurora MySQL engine version for your restored DB instance.

7. For **IAM role**, you can choose an existing IAM role.

8. (Optional) You can also have a new IAM role created for you by choosing **Create a new role**. If so:

   a. Enter the **IAM role name**.

   b. Choose whether to **Allow access to KMS key**:

      - If you didn't encrypt the backup files, choose **No**.
      - If you encrypted the backup files with AES-256 (SSE-S3) when you uploaded them to Amazon S3, choose **No**. In this case, the data is decrypted automatically.
      - If you encrypted the backup files with AWS KMS (SSE-KMS) server-side encryption when you uploaded them to Amazon S3, choose **Yes**. Next, choose the correct KMS key for **AWS KMS key**.

        The AWS Management Console creates an IAM policy that enables Aurora to decrypt the data.

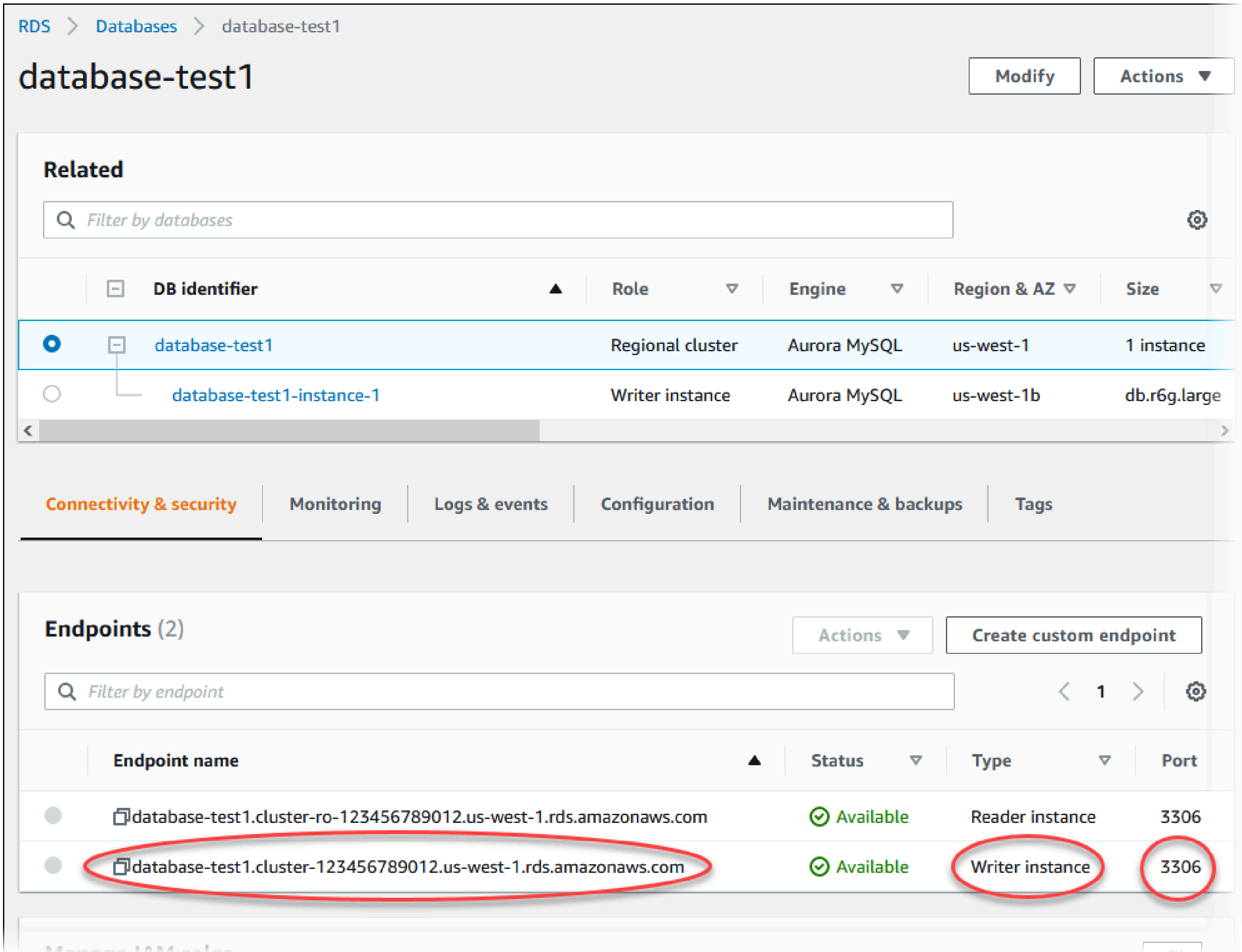
   For more information, see Protecting data using server-side encryption in the *Amazon S3 Developer Guide*.

9. Choose settings for your DB cluster, such as the DB cluster storage configuration, DB instance class, DB cluster identifier, and login credentials. For information about each setting, see Settings for Aurora DB clusters.

10. Customize additional settings for your Aurora MySQL DB cluster as needed.
11. Choose **Create database** to launch your Aurora DB instance.

On the Amazon RDS console, the new DB instance appears in the list of DB instances. The DB instance has a status of **creating** until the DB instance is created and ready for use. When the state changes to **available**, you can connect to the primary instance for your DB cluster. Depending on the DB instance class and store allocated, it can take several minutes for the new instance to be available.

To view the newly created cluster, choose the **Databases** view in the Amazon RDS console and choose the DB cluster. For more information, see Viewing an Amazon Aurora DB cluster.



		DB identifier	Role	Engine	Region & AZ	Size
●	⊟	database-test1	Regional cluster	Aurora MySQL	us-west-1	1 instance
○		database-test1-instance-1	Writer instance	Aurora MySQL	us-west-1b	db.r6g.large

	Endpoint name	Status	Type	Port
○	database-test1.cluster-ro-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Reader instance	3306
○	database-test1.cluster-123456789012.us-west-1.rds.amazonaws.com	✓ Available	Writer instance	3306

Note the port and the writer endpoint of the DB cluster. Use the writer endpoint and port of the DB cluster in your JDBC and ODBC connection strings for any application that performs write or read operations.

Synchronizing the Amazon Aurora MySQL DB cluster with the MySQL database using replication

To achieve little or no downtime during the migration, you can replicate transactions that were committed on your MySQL database to your Aurora MySQL DB cluster. Replication enables the DB cluster to catch up with the transactions on the MySQL database that happened during the migration. When the DB cluster is completely caught up, you can stop the replication and finish the migration to Aurora MySQL.

Topics

Configuring your external MySQL database and your Aurora MySQL DB cluster for encrypted replication

To replicate data securely, you can use encrypted replication.

> **Note**
>
> If you don't need to use encrypted replication, you can skip these steps and move on to the instructions in Synchronizing the Amazon Aurora MySQL DB cluster with the external MySQL database.

The following are prerequisites for using encrypted replication:

- Secure Sockets Layer (SSL) must be enabled on the external MySQL primary database.
- A client key and client certificate must be prepared for the Aurora MySQL DB cluster.

During encrypted replication, the Aurora MySQL DB cluster acts a client to the MySQL database server. The certificates and keys for the Aurora MySQL client are in files in .pem format.

To configure your external MySQL database and your Aurora MySQL DB cluster for encrypted replication

1. Ensure that you are prepared for encrypted replication:

   - If you don't have SSL enabled on the external MySQL primary database and don't have a client key and client certificate prepared, enable SSL on the MySQL database server and generate the required client key and client certificate.
   - If SSL is enabled on the external primary, supply a client key and certificate for the Aurora MySQL DB cluster. If you don't have these, generate a new key and certificate for the Aurora MySQL DB cluster. To sign the client certificate, you must have the certificate authority key that you used to configure SSL on the external MySQL primary database.

   For more information, see Creating SSL certificates and keys using openssl in the MySQL documentation.

   You need the certificate authority certificate, the client key, and the client certificate.

2. Connect to the Aurora MySQL DB cluster as the primary user using SSL.

   For information about connecting to an Aurora MySQL DB cluster with SSL, see Using TLS with Aurora MySQL DB clusters.

3. Run the mysql.rds_import_binlog_ssl_material stored procedure to import the SSL information into the Aurora MySQL DB cluster.

   For the `ssl_material_value` parameter, insert the information from the .pem format files for the Aurora MySQL DB cluster in the correct JSON payload.

   The following example imports SSL information into an Aurora MySQL DB cluster. In .pem format files, the body code typically is longer than the body code shown in the example.

   ```
   call mysql.rds_import_binlog_ssl_material(
   '{"ssl_ca":"-----BEGIN CERTIFICATE-----
   AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
   hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
   lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
   qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
   BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
   -----END CERTIFICATE-----\n","ssl_cert":"-----BEGIN CERTIFICATE-----
   AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
   ```

```
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END CERTIFICATE-----\n","ssl_key":"-----BEGIN RSA PRIVATE KEY-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END RSA PRIVATE KEY-----\n"}');
```

For more information, see mysql.rds_import_binlog_ssl_material and Using TLS with Aurora MySQL DB clusters.

Note

After running the procedure, the secrets are stored in files. To erase the files later, you can run the mysql.rds_remove_binlog_ssl_material stored procedure.

Synchronizing the Amazon Aurora MySQL DB cluster with the external MySQL database

You can synchronize your Amazon Aurora MySQL DB cluster with the MySQL database using replication.

To synchronize your Aurora MySQL DB cluster with the MySQL database using replication

1. Ensure that the /etc/my.cnf file for the external MySQL database has the relevant entries.

   If encrypted replication is not required, ensure that the external MySQL database is started with binary logs (binlogs) enabled and SSL disabled. The following are the relevant entries in the /etc/my.cnf file for unencrypted data.

   ```
   log-bin=mysql-bin
   server-id=2133421
   innodb_flush_log_at_trx_commit=1
   sync_binlog=1
   ```

If encrypted replication is required, ensure that the external MySQL database is started with SSL and binlogs enabled. The entries in the /etc/my.cnf file include the .pem file locations for the MySQL database server.

```
log-bin=mysql-bin
server-id=2133421
innodb_flush_log_at_trx_commit=1
sync_binlog=1

# Setup SSL.
ssl-ca=/home/sslcerts/ca.pem
ssl-cert=/home/sslcerts/server-cert.pem
ssl-key=/home/sslcerts/server-key.pem
```

You can verify that SSL is enabled with the following command.

```
mysql> show variables like 'have_ssl';
```

Your output should be similar the following.

```
+~-~-~-~-~-~-~-~-~-~-~-~-~-~--+~-~-~-~-~-~--+
| Variable_name | Value |
+~-~-~-~-~-~-~-~-~-~-~-~-~-~--+~-~-~-~-~-~--+
| have_ssl      | YES   |
+~-~-~-~-~-~-~-~-~-~-~-~-~-~--+~-~-~-~-~-~--+
1 row in set (0.00 sec)
```

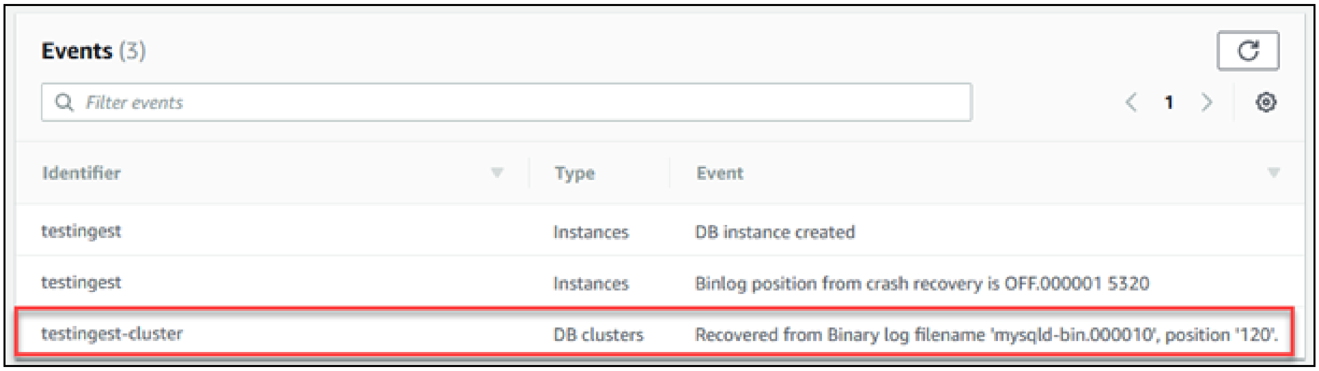
2. Determine the starting binary log position for replication. You specify the position to start replication in a later step.

   Using the AWS Management Console

   a. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   b. In the navigation pane, choose **Events**.

   c. In the **Events** list, note the position in the **Recovered from Binary log filename** event.



Events (3)

Filter events

Identifier	Type	Event
testingest	Instances	DB instance created
testingest	Instances	Binlog position from crash recovery is OFF.000001 5320
testingest-cluster	DB clusters	Recovered from Binary log filename 'mysqld-bin.000010', position '120'.

Using the AWS CLI

You can also get the binlog file name and position by using the describe-events AWS CLI command. The following shows an example `describe-events` command.

```
PROMPT> aws rds describe-events
```

In the output, identify the event that shows the binlog position.

3. While connected to the external MySQL database, create a user to be used for replication. This account is used solely for replication and must be restricted to your domain to improve security. The following is an example.

```
mysql> CREATE USER '<user_name>'@'<domain_name>' IDENTIFIED BY '<password>';
```

The user requires the `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges. Grant these privileges to the user.

```
GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO
 '<user_name>'@'<domain_name>';
```

If you need to use encrypted replication, require SSL connections for the replication user. For example, you can use the following statement to require SSL connections on the user account *<user_name>*.

```
GRANT USAGE ON *.* TO '<user_name>'@'<domain_name>' REQUIRE SSL;
```

Note

If `REQUIRE SSL` is not included, the replication connection might silently fall back to an unencrypted connection.

4. In the Amazon RDS console, add the IP address of the server that hosts the external MySQL database to the VPC security group for the Aurora MySQL DB cluster. For more information on modifying a VPC security group, see Security groups for your VPC in the *Amazon Virtual Private Cloud User Guide*.

   You might also need to configure your local network to permit connections from the IP address of your Aurora MySQL DB cluster, so that it can communicate with your external MySQL database. To find the IP address of the Aurora MySQL DB cluster, use the `host` command.

   ```
   host <db_cluster_endpoint>
   ```

   The host name is the DNS name from the Aurora MySQL DB cluster endpoint.

5. Enable binary log replication by running the mysql.rds_reset_external_master (Aurora MySQL version 2) or mysql.rds_reset_external_source (Aurora MySQL version 3) stored procedure. This stored procedure has the following syntax.

   ```

   CALL mysql.rds_set_external_master (
     host_name
     , host_port
     , replication_user_name
     , replication_user_password
     , mysql_binary_log_file_name
     , mysql_binary_log_file_location
     , ssl_encryption
   );

   CALL mysql.rds_set_external_source (
     host_name
     , host_port
     , replication_user_name
     , replication_user_password
     , mysql_binary_log_file_name
   ```

```
  , mysql_binary_log_file_location
  , ssl_encryption
);
```

For information about the parameters, see mysql.rds_reset_external_master (Aurora MySQL version 2) and mysql.rds_reset_external_source (Aurora MySQL version 3).

For `mysql_binary_log_file_name` and `mysql_binary_log_file_location`, use the position in the **Recovered from Binary log filename** event you noted earlier.

If the data in the Aurora MySQL DB cluster is not encrypted, the `ssl_encryption` parameter must be set to `0`. If the data is encrypted, the `ssl_encryption` parameter must be set to 1.

The following example runs the procedure for an Aurora MySQL DB cluster that has encrypted data.

```
CALL mysql.rds_set_external_master(
  'Externaldb.some.com',
  3306,
  'repl_user'@'mydomain.com',
  'password',
  'mysql-bin.000010',
  120,
  1);

CALL mysql.rds_set_external_source(
  'Externaldb.some.com',
  3306,
  'repl_user'@'mydomain.com',
  'password',
  'mysql-bin.000010',
  120,
  1);
```

This stored procedure sets the parameters that the Aurora MySQL DB cluster uses for connecting to the external MySQL database and reading its binary log. If the data is encrypted, it also downloads the SSL certificate authority certificate, client certificate, and client key to the local disk.

6. Start binary log replication by running the mysql.rds_start_replication stored procedure.

```
CALL mysql.rds_start_replication;
```

7. Monitor how far the Aurora MySQL DB cluster is behind the MySQL replication primary database. To do so, connect to the Aurora MySQL DB cluster and run the following command.

   ```
   Aurora MySQL version 2:
   SHOW SLAVE STATUS;

   Aurora MySQL version 3:
   SHOW REPLICA STATUS;
   ```

   In the command output, the `Seconds Behind Master` field shows how far the Aurora MySQL DB cluster is behind the MySQL primary. When this value is `0` (zero), the Aurora MySQL DB cluster has caught up to the primary, and you can move on to the next step to stop replication.

8. Connect to the MySQL replication primary database and stop replication. To do so, run the mysql.rds_stop_replication stored procedure.

   ```
   CALL mysql.rds_stop_replication;
   ```

Reducing the time for physical migration to Amazon Aurora MySQL

You can make the following database modifications to speed up the process of migrating a database to Amazon Aurora MySQL.

> ⚠ **Important**
>
> Make sure to perform these updates on a copy of a production database, rather than on a production database. You can then back up the copy and restore it to your Aurora MySQL DB cluster to avoid any service interruptions on your production database.

Unsupported table types

Aurora MySQL supports only the InnoDB engine for database tables. If you have MyISAM tables in your database, then those tables must be converted before migrating to Aurora MySQL. The conversion process requires additional space for the MyISAM to InnoDB conversion during the migration procedure.

To reduce your chances of running out of space or to speed up the migration process, convert all of your MyISAM tables to InnoDB tables before migrating them. The size of the resulting InnoDB table is equivalent to the size required by Aurora MySQL for that table. To convert a MyISAM table to InnoDB, run the following command:

```
ALTER TABLE schema.table_name engine=innodb, algorithm=copy;
```

Aurora MySQL doesn't support compressed tables or pages, that is, tables created with `ROW_FORMAT=COMPRESSED` or `COMPRESSION = {"zlib"|"lz4"}`.

To reduce your chances of running out of space or to speed up the migration process, expand your compressed tables by setting `ROW_FORMAT` to `DEFAULT`, `COMPACT`, `DYNAMIC`, or `REDUNDANT`. For compressed pages, set `COMPRESSION="none"`.

For more information, see InnoDB row formats and InnoDB table and page compressionin the MySQL documentation.

You can use the following SQL script on your existing MySQL DB instance to list the tables in your database that are MyISAM tables or compressed tables.

```
-- This script examines a MySQL database for conditions that block
-- migrating the database into Aurora MySQL.
-- It must be run from an account that has read permission for the
-- INFORMATION_SCHEMA database.

-- Verify that this is a supported version of MySQL.

select msg as `==> Checking current version of MySQL.`
from
  (
  select
    'This script should be run on MySQL version 5.6 or higher. ' +
    'Earlier versions are not supported.' as msg,
    cast(substring_index(version(), '.', 1) as unsigned) * 100 +
      cast(substring_index(substring_index(version(), '.', 2), '.', -1)
      as unsigned)
    as major_minor
  ) as T
where major_minor <> 506;


-- List MyISAM and compressed tables. Include the table size.

select concat(TABLE_SCHEMA, '.', TABLE_NAME) as `==> MyISAM or Compressed Tables`,
round(((data_length + index_length) / 1024 / 1024), 2) "Approx size (MB)"
from INFORMATION_SCHEMA.TABLES
where
  ENGINE <> 'InnoDB'
  and
  (
    -- User tables
    TABLE_SCHEMA not in ('mysql', 'performance_schema',
                         'information_schema')
    or
    -- Non-standard system tables
    (
      TABLE_SCHEMA = 'mysql' and TABLE_NAME not in
        (
          'columns_priv', 'db', 'event', 'func', 'general_log',
          'help_category', 'help_keyword', 'help_relation',
          'help_topic', 'host', 'ndb_binlog_index', 'plugin',
          'proc', 'procs_priv', 'proxies_priv', 'servers', 'slow_log',
          'tables_priv', 'time_zone', 'time_zone_leap_second',
          'time_zone_name', 'time_zone_transition',
          'time_zone_transition_type', 'user'
        )
    )
  )
  or
  (
    -- Compressed tables
       ROW_FORMAT = 'Compressed'
  );
```

User accounts with unsupported privileges

User accounts with privileges that aren't supported by Aurora MySQL are imported without the unsupported privileges. For the list of supported privileges, see Role-based privilege model.

You can run the following SQL query on your source database to list the user accounts that have unsupported privileges.

```
SELECT
    user,
    host
FROM
    mysql.user
WHERE
    Shutdown_priv = 'y'
    OR File_priv = 'y'
    OR Super_priv = 'y'
    OR Create_tablespace_priv = 'y';
```

Dynamic privileges in Aurora MySQL version 3

Dynamic privileges aren't imported. Aurora MySQL version 3 supports the following dynamic privileges.

```
'APPLICATION_PASSWORD_ADMIN',
'CONNECTION_ADMIN',
'REPLICATION_APPLIER',
'ROLE_ADMIN',
'SESSION_VARIABLES_ADMIN',
'SET_USER_ID',
'XA_RECOVER_ADMIN'
```

The following example script grants the supported dynamic privileges to the user accounts in the Aurora MySQL DB cluster.

```
-- This script finds the user accounts that have Aurora MySQL supported dynamic
 privileges
-- and grants them to corresponding user accounts in the Aurora MySQL DB cluster.

/home/ec2-user/opt/mysql/8.0.26/bin/mysql -uusername -pxxxxx -P8026 -h127.0.0.1 -BNe
 "SELECT
  CONCAT('GRANT ', GRANTS, ' ON *.* TO ', GRANTEE ,';') AS grant_statement
  FROM (select GRANTEE, group_concat(privilege_type) AS GRANTS FROM
 information_schema.user_privileges
      WHERE privilege_type IN (
        'APPLICATION_PASSWORD_ADMIN',
        'CONNECTION_ADMIN',
        'REPLICATION_APPLIER',
        'ROLE_ADMIN',
        'SESSION_VARIABLES_ADMIN',
        'SET_USER_ID',
        'XA_RECOVER_ADMIN')
      AND GRANTEE NOT IN (\"'mysql.session'@'localhost'\",
\"'mysql.infoschema'@'localhost'\",\"'mysql.sys'@'localhost'\") GROUP BY GRANTEE)
      AS PRIVGRANTS; " | /home/ec2-user/opt/mysql/8.0.26/bin/mysql -u master_username -
p master_password -h DB_cluster_endpoint
```

Stored objects with 'rdsadmin'@'localhost' as the definer

Functions, procedures, views, events, and triggers with `'rdsadmin'@'localhost'` as the definer aren't imported.

You can use the following SQL script on your source MySQL database to list the stored objects that have the unsupported definer.

```
-- This SQL query lists routines with `rdsadmin`@`localhost` as the definer.

SELECT
    ROUTINE_SCHEMA,
    ROUTINE_NAME
FROM
    information_schema.routines
WHERE
    definer = 'rdsadmin@localhost';

-- This SQL query lists triggers with `rdsadmin`@`localhost` as the definer.

SELECT
    TRIGGER_SCHEMA,
    TRIGGER_NAME,
    DEFINER
FROM
    information_schema.triggers
WHERE
    DEFINER = 'rdsadmin@localhost';

-- This SQL query lists events with `rdsadmin`@`localhost` as the definer.

SELECT
    EVENT_SCHEMA,
    EVENT_NAME
FROM
    information_schema.events
WHERE
    DEFINER = 'rdsadmin@localhost';

-- This SQL query lists views with `rdsadmin`@`localhost` as the definer.
SELECT
    TABLE_SCHEMA,
    TABLE_NAME
FROM
    information_schema.views
WHERE
    DEFINER = 'rdsadmin@localhost';
```

Logical migration from MySQL to Amazon Aurora MySQL by using mysqldump

Because Amazon Aurora MySQL is a MySQL-compatible database, you can use the `mysqldump` utility to copy data from your MySQL or MariaDB database to an existing Aurora MySQL DB cluster.

For a discussion of how to do so with MySQL databases that are very large, see Importing data to a MySQL or MariaDB DB instance with reduced downtime. For MySQL databases that have smaller amounts of data, see Importing data from a MySQL or MariaDB DB to a MySQL or MariaDB DB instance.

Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster

You can migrate (copy) data to an Amazon Aurora MySQL DB cluster from an RDS for MySQL DB instance.

Topics

> **Note**
>
> Because Amazon Aurora MySQL is compatible with MySQL, you can migrate data from your MySQL database by setting up replication between your MySQL database and an Amazon Aurora MySQL DB cluster. For more information, see Replication with Amazon Aurora.

Migrating an RDS for MySQL snapshot to Aurora

You can migrate a DB snapshot of an RDS for MySQL DB instance to create an Aurora MySQL DB cluster. The new Aurora MySQL DB cluster is populated with the data from the original RDS for MySQL DB instance. The DB snapshot must have been made from an Amazon RDS DB instance running a MySQL version that's compatible with Aurora MySQL.

You can migrate either a manual or automated DB snapshot. After the DB cluster is created, you can then create optional Aurora Replicas.

> **Note**
>
> You can also migrate an RDS for MySQL DB instance to an Aurora MySQL DB cluster by creating an Aurora read replica of your source RDS for MySQL DB instance. For more information, see Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica.
> You can't migrate to Aurora MySQL version 3.05 and higher from some older MySQL 8.0 versions, including 8.0.11, 8.0.13, and 8.0.15. We recommend that you upgrade to MySQL version 8.0.28 before migrating.

The general steps you must take are as follows:

1. Determine the amount of space to provision for your Aurora MySQL DB cluster. For more information, see How much space do I need?
2. Use the console to create the snapshot in the AWS Region where the Amazon RDS MySQL instance is located. For information about creating a DB snapshot, see Creating a DB snapshot.
3. If the DB snapshot is not in the same AWS Region as your DB cluster, use the Amazon RDS console to copy the DB snapshot to that AWS Region. For information about copying a DB snapshot, see Copying a DB snapshot.
4. Use the console to migrate the DB snapshot and create an Amazon Aurora MySQL DB cluster with the same databases as the original MySQL DB instance.

> ⚠ **Warning**
>
> Amazon RDS limits each AWS account to one snapshot copy into each AWS Region at a time.

How much space do I need?

When you migrate a snapshot of a MySQL DB instance into an Aurora MySQL DB cluster, Aurora uses an Amazon Elastic Block Store (Amazon EBS) volume to format the data from the snapshot before migrating it. In some cases, additional space is needed to format the data for migration.

Tables that are not MyISAM tables and are not compressed can be up to 16 TB in size. If you have MyISAM tables, then Aurora must use additional space in the volume to convert the tables to be compatible with Aurora MySQL. If you have compressed tables, then Aurora must use additional space in the volume to expand these tables before storing them on the Aurora cluster volume. Because of this additional space requirement, you should ensure that none of the MyISAM and compressed tables being migrated from your MySQL DB instance exceeds 8 TB in size.

Reducing the amount of space required to migrate data into Amazon Aurora MySQL

You might want to modify your database schema prior to migrating it into Amazon Aurora. Such modification can be helpful in the following cases:

- You want to speed up the migration process.
- You are unsure of how much space you need to provision.

- You have attempted to migrate your data and the migration has failed due to a lack of provisioned space.

You can make the following changes to improve the process of migrating a database into Amazon Aurora.

> **Important**
>
> Be sure to perform these updates on a new DB instance restored from a snapshot of a production database, rather than on a production instance. You can then migrate the data from the snapshot of your new DB instance into your Aurora DB cluster to avoid any service interruptions on your production database.

Table type	Limitation or guideline
MyISAM tables	Aurora MySQL supports InnoDB tables only. If you have MyISAM tables in your database, then those tables must be converted before being migrated into Aurora MySQL. The conversion process requires additional space for the MyISAM to InnoDB conversion during the migration procedure. To reduce your chances of running out of space or to speed up the migration process, convert all of your MyISAM tables to InnoDB tables before migrating them. The size of the resulting InnoDB table is equivalent to the size required by Aurora MySQL for that table. To convert a MyISAM table to InnoDB, run the following command: `alter table <schema>.<table_name> engine=innodb, algorithm=copy;`
Compressed tables	Aurora MySQL doesn't support compressed tables (that is, tables created with `ROW_FORMAT=COMPRESSED`). To reduce your chances of running out of space or to speed up the migration process, expand your compressed tables by setting `ROW_FORMAT` to `DEFAULT`, `COMPACT`, `DYNAMIC`, or `REDUNDANT`

Table type	Limitation or guideline
	. For more information, see InnoDB row formats in the MySQL documentation.

You can use the following SQL script on your existing MySQL DB instance to list the tables in your database that are MyISAM tables or compressed tables.

```
-- This script examines a MySQL database for conditions that block
-- migrating the database into Amazon Aurora.
-- It needs to be run from an account that has read permission for the
-- INFORMATION_SCHEMA database.

-- Verify that this is a supported version of MySQL.

select msg as `==> Checking current version of MySQL.`
from
  (
  select
    'This script should be run on MySQL version 5.6 or higher. ' +
    'Earlier versions are not supported.' as msg,
    cast(substring_index(version(), '.', 1) as unsigned) * 100 +
      cast(substring_index(substring_index(version(), '.', 2), '.', -1)
      as unsigned)
    as major_minor
  ) as T
where major_minor <> 506;


-- List MyISAM and compressed tables. Include the table size.

select concat(TABLE_SCHEMA, '.', TABLE_NAME) as `==> MyISAM or Compressed Tables`,
round(((data_length + index_length) / 1024 / 1024), 2) "Approx size (MB)"
from INFORMATION_SCHEMA.TABLES
where
  ENGINE <> 'InnoDB'
  and
  (
    -- User tables
    TABLE_SCHEMA not in ('mysql', 'performance_schema',
                         'information_schema')
    or
    -- Non-standard system tables
    (
      TABLE_SCHEMA = 'mysql' and TABLE_NAME not in
        (
          'columns_priv', 'db', 'event', 'func', 'general_log',
          'help_category', 'help_keyword', 'help_relation',
          'help_topic', 'host', 'ndb_binlog_index', 'plugin',
          'proc', 'procs_priv', 'proxies_priv', 'servers', 'slow_log',
          'tables_priv', 'time_zone', 'time_zone_leap_second',
          'time_zone_name', 'time_zone_transition',
          'time_zone_transition_type', 'user'
        )
    )
  )
  or
  (
    -- Compressed tables
       ROW_FORMAT = 'Compressed'
  );
```

The script produces output similar to the output in the following example. The example shows two tables that must be converted from MyISAM to InnoDB. The output also includes the approximate size of each table in megabytes (MB).

```
+---------------------------------+------------------+
| ==> MyISAM or Compressed Tables | Approx size (MB) |
+---------------------------------+------------------+
| test.name_table                 |          2102.25 |
| test.my_table                   |            65.25 |
+---------------------------------+------------------+
2 rows in set (0.01 sec)
```

Migrating an RDS for MySQL DB snapshot to an Aurora MySQL DB cluster

You can migrate a DB snapshot of an RDS for MySQL DB instance to create an Aurora MySQL DB cluster using the AWS Management Console or the AWS CLI. The new Aurora MySQL DB cluster is populated with the data from the original RDS for MySQL DB instance. For information about creating a DB snapshot, see Creating a DB snapshot.

If the DB snapshot is not in the AWS Region where you want to locate your data, copy the DB snapshot to that AWS Region. For information about copying a DB snapshot, see Copying a DB snapshot.

Console

When you migrate the DB snapshot by using the AWS Management Console, the console takes the actions necessary to create both the DB cluster and the primary instance.

You can also choose for your new Aurora MySQL DB cluster to be encrypted at rest using an AWS KMS key.

To migrate a MySQL DB snapshot by using the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Either start the migration from the MySQL DB instance or from the snapshot:

   To start the migration from the DB instance:

   1. In the navigation pane, choose **Databases**, and then select the MySQL DB instance.
   2. For **Actions**, choose **Migrate latest snapshot**.

   To start the migration from the snapshot:

   1. Choose **Snapshots**.
   2. On the **Snapshots** page, choose the snapshot that you want to migrate into an Aurora MySQL DB cluster.
   3. Choose **Snapshot Actions**, and then choose **Migrate Snapshot**.

   The **Migrate Database** page appears.
3. Set the following values on the **Migrate Database** page:

   - **Migrate to DB Engine**: Select `aurora`.
   - **DB Engine Version**: Select the DB engine version for the Aurora MySQL DB cluster.
   - **DB Instance Class**: Select a DB instance class that has the required storage and capacity for your database, for example `db.r3.large`. Aurora cluster volumes automatically grow as the amount of data in your database increases. An Aurora cluster volume can grow to a maximum size of 128 tebibytes (TiB). So you only need to select a DB instance class that meets your current storage requirements. For more information, see Overview of Amazon Aurora storage.

- **DB Instance Identifier**: Type a name for the DB cluster that is unique for your account in the AWS Region you selected. This identifier is used in the endpoint addresses for the instances in your DB cluster. You might choose to add some intelligence to the name, such as including the AWS Region and DB engine you selected, for example **`aurora-cluster1`**.

  The DB instance identifier has the following constraints:

  - It must contain from 1 to 63 alphanumeric characters or hyphens.
  - Its first character must be a letter.
  - It cannot end with a hyphen or contain two consecutive hyphens.
  - It must be unique for all DB instances per AWS account, per AWS Region.
- **Virtual Private Cloud (VPC)**: If you have an existing VPC, then you can use that VPC with your Aurora MySQL DB cluster by selecting your VPC identifier, for example `vpc-a464d1c1`. For information on creating a VPC, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

  Otherwise, you can choose to have Aurora create a VPC for you by selecting **Create a new VPC**.
- **DB subnet group**: If you have an existing subnet group, then you can use that subnet group with your Aurora MySQL DB cluster by selecting your subnet group identifier, for example `gs-subnet-group1`.

  Otherwise, you can choose to have Aurora create a subnet group for you by selecting **Create a new subnet group**.
- **Public accessibility**: Select **No** to specify that instances in your DB cluster can only be accessed by resources inside of your VPC. Select **Yes** to specify that instances in your DB cluster can be accessed by resources on the public network. The default is **Yes**.

  > **Note**
  >
  > Your production DB cluster might not need to be in a public subnet, because only your application servers require access to your DB cluster. If your DB cluster doesn't need to be in a public subnet, set **Publicly Accessible** to **No**.
- **Availability Zone**: Select the Availability Zone to host the primary instance for your Aurora MySQL DB cluster. To have Aurora select an Availability Zone for you, select **No Preference**.

- **Database Port**: Type the default port to be used when connecting to instances in the Aurora MySQL DB cluster. The default is `3306`.

  > **Note**
  >
  > You might be behind a corporate firewall that doesn't allow access to default ports such as the MySQL default port, 3306. In this case, provide a port value that your corporate firewall allows. Remember that port value later when you connect to the Aurora MySQL DB cluster.

- **Encryption**: Choose **Enable Encryption** for your new Aurora MySQL DB cluster to be encrypted at rest. If you choose **Enable Encryption**, you must choose a KMS key as the **AWS KMS key** value.

  If your DB snapshot isn't encrypted, specify an encryption key to have your DB cluster encrypted at rest.

  If your DB snapshot is encrypted, specify an encryption key to have your DB cluster encrypted at rest using the specified encryption key. You can specify the encryption key used by the DB snapshot or a different key. You can't create an unencrypted DB cluster from an encrypted DB snapshot.

- **Auto Minor Version Upgrade**: This setting doesn't apply to Aurora MySQL DB clusters.

  For more information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.

4. Choose **Migrate** to migrate your DB snapshot.
5. Choose **Instances**, and then choose the arrow icon to show the DB cluster details and monitor the progress of the migration. On the details page, you can find the cluster endpoint used to connect to the primary instance of the DB cluster. For more information on connecting to an Aurora MySQL DB cluster, see Connecting to an Amazon Aurora DB cluster.

AWS CLI

You can create an Aurora DB cluster from a DB snapshot of an RDS for MySQL DB instance by using the `restore-db-cluster-from-snapshot` command with the following parameters:

- `--db-cluster-identifier` – The name of the DB cluster to create.

- `--engine aurora-mysql` – For a MySQL 5.7–compatible or 8.0–compatible DB cluster.
- `--kms-key-id` – The AWS KMS key to optionally encrypt the DB cluster with, depending on whether your DB snapshot is encrypted.
  - If your DB snapshot isn't encrypted, specify an encryption key to have your DB cluster encrypted at rest. Otherwise, your DB cluster isn't encrypted.
  - If your DB snapshot is encrypted, specify an encryption key to have your DB cluster encrypted at rest using the specified encryption key. Otherwise, your DB cluster is encrypted at rest using the encryption key for the DB snapshot.

  > **Note**
  >
  > You can't create an unencrypted DB cluster from an encrypted DB snapshot.
- `--snapshot-identifier` – The Amazon Resource Name (ARN) of the DB snapshot to migrate. For more information about Amazon RDS ARNs, see [Amazon Relational Database Service (Amazon RDS)](#).

When you migrate the DB snapshot by using the `RestoreDBClusterFromSnapshot` command, the command creates both the DB cluster and the primary instance.

In this example, you create a MySQL 5.7–compatible DB cluster named *`mydbcluster`* from a DB snapshot with an ARN set to *`mydbsnapshotARN`*.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier mydbcluster \
    --snapshot-identifier mydbsnapshotARN \
    --engine aurora-mysql
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-cluster-identifier mydbcluster ^
    --snapshot-identifier mydbsnapshotARN ^
    --engine aurora-mysql
```

In this example, you create a MySQL 5.7–compatible DB cluster named *mydbcluster* from a DB snapshot with an ARN set to *mydbsnapshotARN*.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier mydbcluster \
    --snapshot-identifier mydbsnapshotARN \
    --engine aurora-mysql
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-cluster-identifier mydbcluster ^
    --snapshot-identifier mydbsnapshotARN ^
    --engine aurora-mysql
```

Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica

Aurora uses the MySQL DB engines' binary log replication functionality to create a special type of DB cluster called an Aurora read replica for a source RDS for MySQL DB instance. Updates made to the source RDS for MySQL DB instance are asynchronously replicated to the Aurora read replica.

We recommend using this functionality to migrate from a RDS for MySQL DB instance to an Aurora MySQL DB cluster by creating an Aurora read replica of your source RDS for MySQL DB instance. When the replica lag between the RDS for MySQL DB instance and the Aurora read replica is 0, you can direct your client applications to the Aurora read replica and then stop replication to make the Aurora read replica a standalone Aurora MySQL DB cluster. Be prepared for migration to take a while, roughly several hours per tebibyte (TiB) of data.

For a list of regions where Aurora is available, see Amazon Aurora in the *AWS General Reference*.

When you create an Aurora read replica of an RDS for MySQL DB instance, Amazon RDS creates a DB snapshot of your source RDS for MySQL DB instance (private to Amazon RDS, and incurring no charges). Amazon RDS then migrates the data from the DB snapshot to the Aurora read replica. After the data from the DB snapshot has been migrated to the new Aurora MySQL DB cluster, Amazon RDS starts replication between your RDS for MySQL DB instance and the Aurora MySQL DB cluster. If your RDS for MySQL DB instance contains tables that use storage engines other than InnoDB, or that use compressed row format, you can speed up the process of creating an Aurora read replica by altering those tables to use the InnoDB storage engine and dynamic row format before you create your Aurora read replica. For more information about the process of copying a MySQL DB snapshot to an Aurora MySQL DB cluster, see Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster.

You can have only one Aurora read replica for an RDS for MySQL DB instance.

Note

Replication issues can arise due to feature differences between Aurora MySQL and the MySQL database engine version of your RDS for MySQL DB instance that is the replication primary. If you encounter an error, you can find help in the Amazon RDS community forum or by contacting AWS Support.
You can't create an Aurora read replica if your RDS for MySQL DB instance is already the source for a cross-Region read replica.

You can't migrate to Aurora MySQL version 3.05 and higher from some older RDS for MySQL 8.0 versions, including 8.0.11, 8.0.13, and 8.0.15. We recommend that you upgrade to RDS for MySQL version 8.0.28 before migrating.

For more information on MySQL read replicas, see Working with read replicas of MariaDB, MySQL, and PostgreSQL DB instances.

Creating an Aurora read replica

You can create an Aurora read replica for an RDS for MySQL DB instance by using the console, the AWS CLI, or the RDS API.

Console

To create an Aurora read replica from a source RDS for MySQL DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the MySQL DB instance that you want to use as the source for your Aurora read replica.
4. For **Actions**, choose **Create Aurora read replica**.
5. Choose the DB cluster specifications you want to use for the Aurora read replica, as described in the following table.

Option	Description
DB instance class	Choose a DB instance class that defines the processin g and memory requirements for the primary instance in the DB cluster. For more information about DB instance class options, see Aurora DB instance classes.
Multi-AZ deployment	Choose **Create Replica in Different Zone** to create a standby replica of the new DB cluster in another Availability Zone in the target AWS Region for failover support. For more information about multiple Availability Zones, see Regions and Availability Zones.

Option	Description
DB instance identifier	Type a name for the primary instance in your Aurora read replica DB cluster. This identifier is used in the endpoint address for the primary instance of the new DB cluster. The DB instance identifier has the following constrain ts: • It must contain from 1 to 63 alphanumeric characters or hyphens. • Its first character must be a letter. • It cannot end with a hyphen or contain two consecutive hyphens. • It must be unique for all DB instances for each AWS account, for each AWS Region. Because the Aurora read replica DB cluster is created from a snapshot of the source DB instance, the master user name and master password for the Aurora read replica are the same as the master user name and master password for the source DB instance.
Virtual Private Cloud (VPC)	Select the VPC to host the DB cluster. Select **Create new VPC** to have Aurora create a VPC for you. For more information, see DB cluster prerequisites.
DB subnet group	Select the DB subnet group to use for the DB cluster. Select **Create new DB subnet group** to have Aurora create a DB subnet group for you. For more informati on, see DB cluster prerequisites.

Option	Description
Public accessibility	Select Yes to give the DB cluster a public IP address; otherwise, select No. The instances in your DB cluster can be a mix of both public and private DB instances . For more information about hiding instances from public access, see Hiding a DB cluster in a VPC from the internet.
Availability zone	Determine if you want to specify a particular Availability Zone. For more information about Availability Zones, see Regions and Availability Zones.
VPC security group (firewall)	Select **Create new VPC security group** to have Aurora create a VPC security group for you. Select **Select existing VPC security groups** to specify one or more VPC security groups to secure network access to the DB cluster. For more information, see DB cluster prerequisites.
Database port	Specify the port for applications and utilities to use to access the database. Aurora MySQL DB clusters default to the default MySQL port, 3306. Firewalls at some companies block connections to this port. If your company firewall blocks the default port, choose another port for the new DB cluster.
DB parameter group	Select a DB parameter group for the Aurora MySQL DB cluster. Aurora has a default DB parameter group you can use, or you can create your own DB parameter group. For more information about DB parameter groups, see Working with parameter groups.

Option	Description
DB cluster parameter group	Select a DB cluster parameter group for the Aurora MySQL DB cluster. Aurora has a default DB cluster parameter group you can use, or you can create your own DB cluster parameter group. For more informati on about DB cluster parameter groups, see Working with parameter groups.
Encryption	Choose **Disable encryption** if you don't want your new Aurora DB cluster to be encrypted. Choose **Enable encryption** for your new Aurora DB cluster to be encrypted at rest. If you choose **Enable encryptio n**, you must choose a KMS key as the **AWS KMS key** value. If your MySQL DB instance isn't encrypted, specify an encryption key to have your DB cluster encrypted at rest. If your MySQL DB instance is encrypted, specify an encryption key to have your DB cluster encrypted at rest using the specified encryption key. You can specify the encryption key used by the MySQL DB instance or a different key. You can't create an unencrypted DB cluster from an encrypted MySQL DB instance.
Priority	Choose a failover priority for the DB cluster. If you don't select a value, the default is **tier-1**. This priority determines the order in which Aurora Replicas are promoted when recovering from a primary instance failure. For more information, see Fault tolerance for an Aurora DB cluster.

Option	Description
Backup retention period	Select the length of time, from 1 to 35 days, that Aurora retains backup copies of the database. Backup copies can be used for point-in-time restores (PITR) of your database down to the second.
Enhanced Monitoring	Choose **Enable enhanced monitoring** to enable gathering metrics in real time for the operating system that your DB cluster runs on. For more information, see Monitoring OS metrics with Enhanced Monitoring.
Monitoring Role	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Choose the IAM role that you created to permit Aurora to communicate with Amazon CloudWatch Logs for you, or choose **Default** to have Aurora create a role for you named `rds-monitoring-role` . For more information, see Monitoring OS metrics with Enhanced Monitoring.
Granularity	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Set the interval, in seconds, between when metrics are collected for your DB cluster.
Auto minor version upgrade	This setting doesn't apply to Aurora MySQL DB clusters. For more information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.
Maintenance window	Select **Select window** and specify the weekly time range during which system maintenance can occur. Or, select **No preference** for Aurora to assign a period randomly.

6. Choose **Create read replica**.

AWS CLI

To create an Aurora read replica from a source RDS for MySQL DB instance, use the `create-db-cluster` and `create-db-instance` AWS CLI commands to create a new Aurora MySQL DB cluster. When you call the `create-db-cluster` command, include the `--replication-source-identifier` parameter to identify the Amazon Resource Name (ARN) for the source MySQL DB instance. For more information about Amazon RDS ARNs, see Amazon Relational Database Service (Amazon RDS).

Don't specify the master username, master password, or database name as the Aurora read replica uses the same master username, master password, and database name as the source MySQL DB instance.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster --db-cluster-identifier sample-replica-cluster --engine
 aurora \
    --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2 \
    --replication-source-identifier arn:aws:rds:us-west-2:123456789012:db:primary-
mysql-instance
```

For Windows:

```
aws rds create-db-cluster --db-cluster-identifier sample-replica-cluster --engine
 aurora ^
    --db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2 ^
    --replication-source-identifier arn:aws:rds:us-west-2:123456789012:db:primary-
mysql-instance
```

If you use the console to create an Aurora read replica, then Aurora automatically creates the primary instance for your DB cluster Aurora read replica. If you use the AWS CLI to create an Aurora read replica, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.

You can create a primary instance for your DB cluster by using the `create-db-instance` AWS CLI command with the following parameters.

- `--db-cluster-identifier`

  The name of your DB cluster.

- `--db-instance-class`

The name of the DB instance class to use for your primary instance.

- `--db-instance-identifier`

  The name of your primary instance.

- `--engine aurora`

In this example, you create a primary instance named *myreadreplicainstance* for the DB cluster named *myreadreplicacluster*, using the DB instance class specified in *myinstanceclass*.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
    --db-cluster-identifier myreadreplicacluster \
    --db-instance-class myinstanceclass \
    --db-instance-identifier myreadreplicainstance \
    --engine aurora
```

For Windows:

```
aws rds create-db-instance ^
    --db-cluster-identifier myreadreplicacluster ^
    --db-instance-class myinstanceclass ^
    --db-instance-identifier myreadreplicainstance ^
    --engine aurora
```

RDS API

To create an Aurora read replica from a source RDS for MySQL DB instance, use the `CreateDBCluster` and `CreateDBInstance` Amazon RDS API commands to create a new Aurora DB cluster and primary instance. Do not specify the master username, master password, or database name as the Aurora read replica uses the same master username, master password, and database name as the source RDS for MySQL DB instance.

You can create a new Aurora DB cluster for an Aurora read replica from a source RDS for MySQL DB instance by using the `CreateDBCluster` Amazon RDS API command with the following parameters:

- `DBClusterIdentifier`

  The name of the DB cluster to create.

- `DBSubnetGroupName`

  The name of the DB subnet group to associate with this DB cluster.

- `Engine=aurora`
- `KmsKeyId`

  The AWS KMS key to optionally encrypt the DB cluster with, depending on whether your MySQL DB instance is encrypted.

  - If your MySQL DB instance isn't encrypted, specify an encryption key to have your DB cluster encrypted at rest. Otherwise, your DB cluster is encrypted at rest using the default encryption key for your account.
  - If your MySQL DB instance is encrypted, specify an encryption key to have your DB cluster encrypted at rest using the specified encryption key. Otherwise, your DB cluster is encrypted at rest using the encryption key for the MySQL DB instance.

  > **Note**
  >
  > You can't create an unencrypted DB cluster from an encrypted MySQL DB instance.

- `ReplicationSourceIdentifier`

  The Amazon Resource Name (ARN) for the source MySQL DB instance. For more information about Amazon RDS ARNs, see Amazon Relational Database Service (Amazon RDS).

- `VpcSecurityGroupIds`

  The list of EC2 VPC security groups to associate with this DB cluster.

In this example, you create a DB cluster named *`myreadreplicacluster`* from a source MySQL DB instance with an ARN set to *`mysqlprimaryARN`*, associated with a DB subnet group named *`mysubnetgroup`* and a VPC security group named *`mysecuritygroup`*.

Example

```
https://rds.us-east-1.amazonaws.com/
    ?Action=CreateDBCluster
    &DBClusterIdentifier=myreadreplicacluster
    &DBSubnetGroupName=mysubnetgroup
    &Engine=aurora
    &ReplicationSourceIdentifier=mysqlprimaryARN
    &SignatureMethod=HmacSHA256
    &SignatureVersion=4
    &Version=2014-10-31
    &VpcSecurityGroupIds=mysecuritygroup
    &X-Amz-Algorithm=AWS4-HMAC-SHA256
    &X-Amz-Credential=AKIADQKE4SARGYLE/20150927/us-east-1/rds/aws4_request
    &X-Amz-Date=20150927T164851Z
    &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
    &X-Amz-Signature=6a8f4bd6a98f649c75ea04a6b3929ecc75ac09739588391cd7250f5280e716db
```

If you use the console to create an Aurora read replica, then Aurora automatically creates the primary instance for your DB cluster Aurora read replica. If you use the AWS CLI to create an Aurora read replica, you must explicitly create the primary instance for your DB cluster. The primary instance is the first instance that is created in a DB cluster.

You can create a primary instance for your DB cluster by using the `CreateDBInstance` Amazon RDS API command with the following parameters:

- `DBClusterIdentifier`

  The name of your DB cluster.
- `DBInstanceClass`

  The name of the DB instance class to use for your primary instance.
- `DBInstanceIdentifier`

  The name of your primary instance.
- `Engine=aurora`

In this example, you create a primary instance named *`myreadreplicainstance`* for the DB cluster named *`myreadreplicacluster`*, using the DB instance class specified in *`myinstanceclass`*.

Example

```
https://rds.us-east-1.amazonaws.com/
```

```
    ?Action=CreateDBInstance
    &DBClusterIdentifier=myreadreplicacluster
    &DBInstanceClass=myinstanceclass
    &DBInstanceIdentifier=myreadreplicainstance
    &Engine=aurora
    &SignatureMethod=HmacSHA256
    &SignatureVersion=4
    &Version=2014-09-01
    &X-Amz-Algorithm=AWS4-HMAC-SHA256
    &X-Amz-Credential=AKIADQKE4SARGYLE/20140424/us-east-1/rds/aws4_request
    &X-Amz-Date=20140424T194844Z
    &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
    &X-Amz-Signature=bee4aabc750bf7dad0cd9e22b952bd6089d91e2a16592c2293e532eeaab8bc77
```

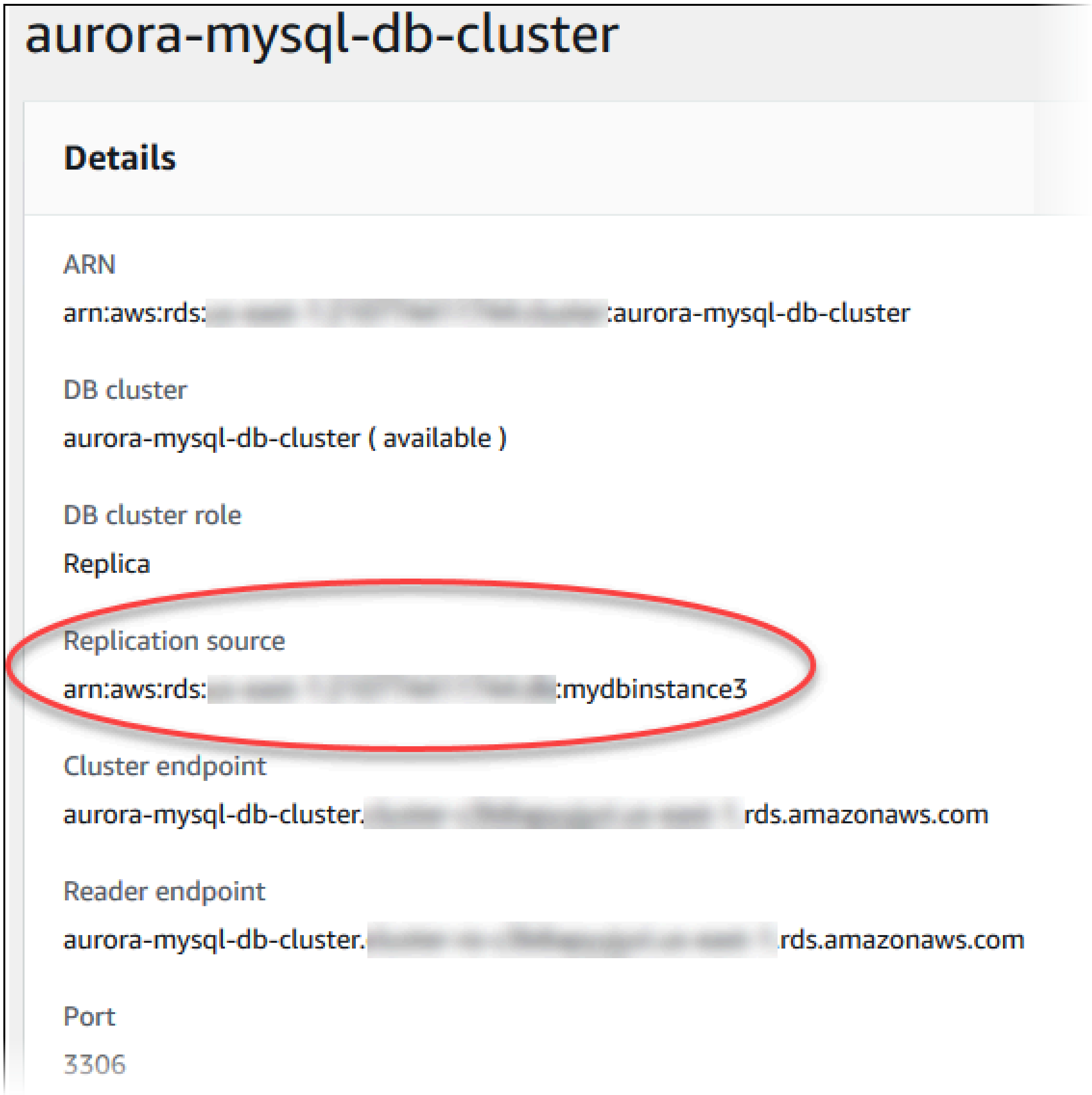
Viewing an Aurora read replica

You can view the MySQL to Aurora MySQL replication relationships for your Aurora MySQL DB clusters by using the AWS Management Console or the AWS CLI.

Console

To view the primary MySQL DB instance for an Aurora read replica

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster for the Aurora read replica to display its details. The primary MySQL DB instance information is in the **Replication source** field.



aurora-mysql-db-cluster

Details

ARN

arn:aws:rds:[redacted]:aurora-mysql-db-cluster

DB cluster

aurora-mysql-db-cluster (available)

DB cluster role

Replica

Replication source

arn:aws:rds:[redacted]:mydbinstance3

Cluster endpoint

aurora-mysql-db-cluster.[redacted]rds.amazonaws.com

Reader endpoint

aurora-mysql-db-cluster.[redacted]rds.amazonaws.com

Port

3306

AWS CLI

To view the MySQL to Aurora MySQL replication relationships for your Aurora MySQL DB clusters by using the AWS CLI, use the `describe-db-clusters` and `describe-db-instances` commands.

To determine which MySQL DB instance is the primary, use the `describe-db-clusters` and specify the cluster identifier of the Aurora read replica for the `--db-cluster-identifier` option. Refer to the `ReplicationSourceIdentifier` element in the output for the ARN of the DB instance that is the replication primary.

To determine which DB cluster is the Aurora read replica, use the [describe-db-instances](#) and specify the instance identifier of the MySQL DB instance for the `--db-instance-identifier` option. Refer to the `ReadReplicaDBClusterIdentifiers` element in the output for the DB cluster identifier of the Aurora read replica.

Example

For Linux, macOS, or Unix:

```
aws rds describe-db-clusters \
    --db-cluster-identifier myreadreplicacluster
```

```
aws rds describe-db-instances \
    --db-instance-identifier mysqlprimary
```

For Windows:

```
aws rds describe-db-clusters ^
    --db-cluster-identifier myreadreplicacluster
```

```
aws rds describe-db-instances ^
    --db-instance-identifier mysqlprimary
```

Promoting an Aurora read replica

After migration completes, you can promote the Aurora read replica to a stand-alone DB cluster using the AWS Management Console or AWS CLI.

Then you can direct your client applications to the endpoint for the Aurora read replica. For more information on the Aurora endpoints, see [Amazon Aurora connection management](#). Promotion should complete fairly quickly, and you can read from and write to the Aurora read replica during promotion. However, you can't delete the primary MySQL DB instance or unlink the DB Instance and the Aurora read replica during this time.

Before you promote your Aurora read replica, stop any transactions from being written to the source MySQL DB instance, and then wait for the replica lag on the Aurora read replica to reach 0. You can view the replica lag for an Aurora read replica by calling the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command on your Aurora read replica. Check the **Seconds behind master** value.

You can start writing to the Aurora read replica after write transactions to the primary have stopped and replica lag is 0. If you write to the Aurora read replica before this and you modify tables that are also being modified on the MySQL primary, you risk breaking replication to Aurora. If this happens, you must delete and recreate your Aurora read replica.

Console

To promote an Aurora read replica to an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster for the Aurora read replica.
4. For **Actions**, choose **Promote**.
5. Choose **Promote read replica**.

After you promote, confirm that the promotion has completed by using the following procedure.

To confirm that the Aurora read replica was promoted

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Events**.
3. On the **Events** page, verify that there is a `Promoted Read Replica cluster to a stand-alone database cluster` event for the cluster that you promoted.

After promotion is complete, the primary MySQL DB instance and the Aurora read replica are unlinked, and you can safely delete the DB instance if you want.

AWS CLI

To promote an Aurora read replica to a stand-alone DB cluster, use the `promote-read-replica-db-cluster` AWS CLI command.

Example

For Linux, macOS, or Unix:

```
aws rds promote-read-replica-db-cluster \
    --db-cluster-identifier myreadreplicacluster
```

For Windows:

```
aws rds promote-read-replica-db-cluster ^
    --db-cluster-identifier myreadreplicacluster
```

Managing Amazon Aurora MySQL

The following sections discuss managing an Amazon Aurora MySQL DB cluster.

Topics

Managing performance and scaling for Amazon Aurora MySQL

Scaling Aurora MySQL DB instances

You can scale Aurora MySQL DB instances in two ways, instance scaling and read scaling. For more information about read scaling, see Read scaling.

You can scale your Aurora MySQL DB cluster by modifying the DB instance class for each DB instance in the DB cluster. Aurora MySQL supports several DB instance classes optimized for Aurora. Don't use db.t2 or db.t3 instance classes for larger Aurora clusters of size greater than 40 TB. For the specifications of the DB instance classes supported by Aurora MySQL, see Aurora DB instance classes.

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see Using T instance classes for development and testing.

Maximum connections to an Aurora MySQL DB instance

The maximum number of connections allowed to an Aurora MySQL DB instance is determined by the `max_connections` parameter in the instance-level parameter group for the DB instance.

The following table lists the resulting default value of `max_connections` for each DB instance class available to Aurora MySQL. You can increase the maximum number of connections to your

Aurora MySQL DB instance by scaling the instance up to a DB instance class with more memory, or by setting a larger value for the `max_connections` parameter in the DB parameter group for your instance, up to 16,000.

> **Tip**
>
> If your applications frequently open and close connections, or keep a large number of long-lived connections open, we recommend that you use Amazon RDS Proxy. RDS Proxy is a fully managed, highly available database proxy that uses connection pooling to share database connections securely and efficiently. To learn more about RDS Proxy, see Using Amazon RDS Proxy for Aurora.

For details about how Aurora Serverless v2 instances handle this parameter, see Maximum connections for Aurora Serverless v2.

Instance class	max_connections default value		
db.t2.small	45		
db.t2.medium	90		
db.t3.small	45		
db.t3.medium	90		
db.t3.large	135		
db.t4g.medium	90		
db.t4g.large	135		
db.r3.large	1000		
db.r3.xlarge	2000		
db.r3.2xlarge	3000		

Instance class	max_connections default value		
db.r3.4xlarge	4000		
db.r3.8xlarge	5000		
db.r4.large	1000		
db.r4.xlarge	2000		
db.r4.2xlarge	3000		
db.r4.4xlarge	4000		
db.r4.8xlarge	5000		
db.r4.16xlarge	6000		
db.r5.large	1000		
db.r5.xlarge	2000		
db.r5.2xlarge	3000		
db.r5.4xlarge	4000		
db.r5.8xlarge	5000		
db.r5.12xlarge	6000		
db.r5.16xlarge	6000		
db.r5.24xlarge	7000		
db.r6g.large	1000		
db.r6g.xlarge	2000		
db.r6g.2xlarge	3000		

Instance class	max_connections default value		
db.r6g.4xlarge	4000		
db.r6g.8xlarge	5000		
db.r6g.12xlarge	6000		
db.r6g.16xlarge	6000		
db.r6i.large	1000		
db.r6i.xlarge	2000		
db.r6i.2xlarge	3000		
db.r6i.4xlarge	4000		
db.r6i.8xlarge	5000		
db.r6i.12xlarge	6000		
db.r6i.16xlarge	6000		
db.r6i.24xlarge	7000		
db.r6i.32xlarge	7000		
db.r7g.large	1000		
db.r7g.xlarge	2000		
db.r7g.2xlarge	3000		
db.r7g.4xlarge	4000		
db.r7g.8xlarge	5000		
db.r7g.12xlarge	6000		

Instance class	max_connections default value
db.r7g.16xlarge	6000
db.x2g.large	2000
db.x2g.xlarge	3000
db.x2g.2xlarge	4000
db.x2g.4xlarge	5000
db.x2g.8xlarge	6000
db.x2g.12xlarge	7000
db.x2g.16xlarge	7000

If you create a new parameter group to customize your own default for the connection limit, you'll see that the default connection limit is derived using a formula based on the `DBInstanceClassMemory` value. As shown in the preceding table, the formula produces connection limits that increase by 1000 as the memory doubles between progressively larger R3, R4, and R5 instances, and by 45 for different memory sizes of T2 and T3 instances.

See Specifying DB parameters for more details on how `DBInstanceClassMemory` is calculated.

Aurora MySQL and RDS for MySQL DB instances have different amounts of memory overhead. Therefore, the `max_connections` value can be different for Aurora MySQL and RDS for MySQL DB instances that use the same instance class. The values in the table only apply to Aurora MySQL DB instances.

Note

The much lower connectivity limits for T2 and T3 instances are because with Aurora, those instance classes are intended only for development and test scenarios, not for production workloads.

The default connection limits are tuned for systems that use the default values for other major memory consumers, such as the buffer pool and query cache. If you change those other settings for your cluster, consider adjusting the connection limit to account for the increase or decrease in available memory on the DB instances.

Temporary storage limits for Aurora MySQL

Aurora MySQL stores tables and indexes in the Aurora storage subsystem. Aurora MySQL uses separate temporary or local storage for nonpersistent temporary files and non-InnoDB temporary tables. Local storage also includes files that are used for such purposes as sorting large datasets during query processing or for index build operations. It doesn't include InnoDB temporary tables.

For more information on temporary tables in Aurora MySQL version 3, see New temporary table behavior in Aurora MySQL version 3. For more information on temporary tables in version 2, see Temporary tablespace behavior in Aurora MySQL version 2.

The data and temporary files on these volumes are lost when starting and stopping the DB instance, and during host replacement.

These local storage volumes are backed by Amazon Elastic Block Store (EBS) and can be extended by using a larger DB instance class. For more information about storage, see Amazon Aurora storage and reliability.

Local storage is also used for importing data from Amazon S3 using `LOAD DATA FROM S3` or `LOAD XML FROM S3`, and for exporting data to S3 using SELECT INTO OUTFILE S3. For more information on importing from and exporting to S3, see the following:

- Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket
- Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket

Aurora MySQL uses separate permanent storage for error logs, general logs, slow query logs, and audit logs for most of the Aurora MySQL DB instance classes (not including burstable-performance instance class types such as db.t2, db.t3, and db.t4g). The data on this volume is retained when starting and stopping the DB instance, and during host replacement.

This permanent storage volume is also backed by Amazon EBS and has a fixed size according to the DB instance class. It can't be extended by using a larger DB instance class.

The following table shows the maximum amount of temporary and permanent storage available for each Aurora MySQL DB instance class. For more information on DB instance class support for Aurora, see Aurora DB instance classes.

DB instance class	Maximum temporary/local storage available (GiB)	Additional maximum storage available for log files (GiB)
db.x2g.16xlarge	1280	500
db.x2g.12xlarge	960	500
db.x2g.8xlarge	640	500
db.x2g.4xlarge	320	500
db.x2g.2xlarge	160	60
db.x2g.xlarge	80	60
db.x2g.large	40	60
db.r7g.16xlarge	1280	500
db.r7g.12xlarge	960	500
db.r7g.8xlarge	640	500
db.r7g.4xlarge	320	500
db.r7g.2xlarge	160	60
db.r7g.xlarge	80	60
db.r7g.large	32	60
db.r6i.32xlarge	2560	500
db.r6i.24xlarge	1920	500
db.r6i.16xlarge	1280	500

DB instance class	Maximum temporary/local storage available (GiB)	Additional maximum storage available for log files (GiB)
db.r6i.12xlarge	960	500
db.r6i.8xlarge	640	500
db.r6i.4xlarge	320	500
db.r6i.2xlarge	160	60
db.r6i.xlarge	80	60
db.r6i.large	32	60
db.r6g.16xlarge	1280	500
db.r6g.12xlarge	960	500
db.r6g.8xlarge	640	500
db.r6g.4xlarge	320	500
db.r6g.2xlarge	160	60
db.r6g.xlarge	80	60
db.r6g.large	32	60
db.r5.24xlarge	1920	500
db.r5.16xlarge	1280	500
db.r5.12xlarge	960	500
db.r5.8xlarge	640	500
db.r5.4xlarge	320	500
db.r5.2xlarge	160	60
db.r5.xlarge	80	60

DB instance class	Maximum temporary/local storage available (GiB)	Additional maximum storage available for log files (GiB)
db.r5.large	32	60
db.r4.16xlarge	1280	500
db.r4.8xlarge	640	500
db.r4.4xlarge	320	500
db.r4.2xlarge	160	60
db.r4.xlarge	80	60
db.r4.large	32	60
db.t4g.large	32	–
db.t4g.medium	32	–
db.t3.large	32	–
db.t3.medium	32	–
db.t3.small	32	–
db.t2.medium	32	–
db.t2.small	32	–

⚠ **Important**

These values represent the theoretical maximum amount of free storage on each DB instance. The actual local storage available to you might be lower. Aurora uses some local storage for its management processes, and the DB instance uses some local storage even before you load any data. You can monitor the temporary storage available for a specific DB instance with the `FreeLocalStorage` CloudWatch metric, described in Amazon CloudWatch metrics for Amazon Aurora. You can check the amount of free storage at the present time. You can also chart the amount of free storage over time. Monitoring the free

storage over time helps you to determine whether the value is increasing or decreasing, or to find the minimum, maximum, or average values.
(This doesn't apply to Aurora Serverless v2.)

Backtracking an Aurora DB cluster

With Amazon Aurora MySQL-Compatible Edition, you can backtrack a DB cluster to a specific time, without restoring data from a backup.

Contents

Overview of backtracking

Backtracking "rewinds" the DB cluster to the time you specify. Backtracking is not a replacement for backing up your DB cluster so that you can restore it to a point in time. However, backtracking provides the following advantages over traditional backup and restore:

- You can easily undo mistakes. If you mistakenly perform a destructive action, such as a DELETE without a WHERE clause, you can backtrack the DB cluster to a time before the destructive action with minimal interruption of service.

- You can backtrack a DB cluster quickly. Restoring a DB cluster to a point in time launches a new DB cluster and restores it from backup data or a DB cluster snapshot, which can take hours. Backtracking a DB cluster doesn't require a new DB cluster and rewinds the DB cluster in minutes.
- You can explore earlier data changes. You can repeatedly backtrack a DB cluster back and forth in time to help determine when a particular data change occurred. For example, you can backtrack a DB cluster three hours and then backtrack forward in time one hour. In this case, the backtrack time is two hours before the original time.

> ⓘ **Note**
>
> For information about restoring a DB cluster to a point in time, see Overview of backing up and restoring an Aurora DB cluster.

Backtrack window

With backtracking, there is a target backtrack window and an actual backtrack window:

- The *target backtrack window* is the amount of time you want to be able to backtrack your DB cluster. When you enable backtracking, you specify a *target backtrack window*. For example, you might specify a target backtrack window of 24 hours if you want to be able to backtrack the DB cluster one day.
- The *actual backtrack window* is the actual amount of time you can backtrack your DB cluster, which can be smaller than the target backtrack window. The actual backtrack window is based on your workload and the storage available for storing information about database changes, called *change records.*

As you make updates to your Aurora DB cluster with backtracking enabled, you generate change records. Aurora retains change records for the target backtrack window, and you pay an hourly rate for storing them. Both the target backtrack window and the workload on your DB cluster determine the number of change records you store. The workload is the number of changes you make to your DB cluster in a given amount of time. If your workload is heavy, you store more change records in your backtrack window than you do if your workload is light.

You can think of your target backtrack window as the goal for the maximum amount of time you want to be able to backtrack your DB cluster. In most cases, you can backtrack the maximum amount of time that you specified. However, in some cases, the DB cluster can't store enough change records to backtrack the maximum amount of time, and your actual backtrack window is smaller than your target. Typically, the actual backtrack window is smaller than the target when you have extremely heavy workload on your DB cluster. When your actual backtrack window is smaller than your target, we send you a notification.

When backtracking is enabled for a DB cluster, and you delete a table stored in the DB cluster, Aurora keeps that table in the backtrack change records. It does this so that you can revert back to a time before you deleted the table. If you don't have enough space in your backtrack window to store the table, the table might be removed from the backtrack change records eventually.

Backtracking time

Aurora always backtracks to a time that is consistent for the DB cluster. Doing so eliminates the possibility of uncommitted transactions when the backtrack is complete. When you specify a time for a backtrack, Aurora automatically chooses the nearest possible consistent time. This approach means that the completed backtrack might not exactly match the time you specify, but you can determine the exact time for a backtrack by using the describe-db-cluster-backtracks AWS CLI command. For more information, see Retrieving existing backtracks.

Backtracking limitations

The following limitations apply to backtracking:

- Backtracking is only available for DB clusters that were created with the Backtrack feature enabled. You can't modify a DB cluster to enable the Backtrack feature. You can enable the Backtrack feature when you create a new DB cluster or restore a snapshot of a DB cluster.
- The limit for a backtrack window is 72 hours.
- Backtracking affects the entire DB cluster. For example, you can't selectively backtrack a single table or a single data update.
- You can't create cross-Region read replicas from a backtrack-enabled cluster, but you can still enable binary log (binlog) replication on the cluster. If you try to backtrack a DB cluster for which binary logging is enabled, an error typically occurs unless you choose to force the backtrack. Any attempts to force a backtrack will break downstream read replicas and interfere with other operations such as blue/green deployments.
- You can't backtrack a database clone to a time before that database clone was created. However, you can use the original database to backtrack to a time before the clone was created. For more information about database cloning, see Cloning a volume for an Amazon Aurora DB cluster.

- Backtracking causes a brief DB instance disruption. You must stop or pause your applications before starting a backtrack operation to ensure that there are no new read or write requests. During the backtrack operation, Aurora pauses the database, closes any open connections, and drops any uncommitted reads and writes. It then waits for the backtrack operation to complete.
- You can't restore a cross-Region snapshot of a backtrack-enabled cluster in an AWS Region that doesn't support backtracking.
- If you perform an in-place upgrade for a backtrack-enabled cluster from Aurora MySQL version 2 to version 3, you can't backtrack to a point in time before the upgrade happened.

Region and version availability

Backtrack is not available for Aurora PostgreSQL.

Following are the supported engines and Region availability for Backtrack with Aurora MySQL.

Region	Aurora MySQL version 3	Aurora MySQL version 2
US East (Ohio)	All versions	All versions
US East (N. Virginia)	All versions	All versions
US West (N. California)	All versions	All versions
US West (Oregon)	All versions	All versions
Africa (Cape Town)	–	–
Asia Pacific (Hong Kong)	–	–
Asia Pacific (Jakarta)	–	–

Region	Aurora MySQL version 3	Aurora MySQL version 2
Asia Pacific (Melbourne)	–	–
Asia Pacific (Mumbai)	All versions	All versions
Asia Pacific (Osaka)	All versions	Version 2.07.3 and higher
Asia Pacific (Seoul)	All versions	All versions
Asia Pacific (Singapore)	All versions	All versions
Asia Pacific (Sydney)	All versions	All versions
Asia Pacific (Tokyo)	All versions	All versions
Canada (Central)	All versions	All versions
Canada West (Calgary)	–	–
China (Beijing)	–	–
China (Ningxia)	–	–
Europe (Frankfurt)	All versions	All versions
Europe (Ireland)	All versions	All versions
Europe (London)	All versions	All versions
Europe (Milan)	–	–

Region	Aurora MySQL version 3	Aurora MySQL version 2
Europe (Paris)	All versions	All versions
Europe (Spain)	–	–
Europe (Stockholm)	–	–
Europe (Zurich)	–	–
Israel (Tel Aviv)	–	–
Middle East (Bahrain)	–	–
Middle East (UAE)	–	–
South America (São Paulo)	–	–
AWS GovCloud (US-East)	–	–
AWS GovCloud (US-West)	–	–

Upgrade considerations for backtrack-enabled clusters

You can upgrade a backtrack-enabled DB cluster from Aurora MySQL version 2 to version 3, because all minor versions of Aurora MySQL version 3 are supported for Backtrack.

Configuring backtracking

To use the Backtrack feature, you must enable backtracking and specify a target backtrack window. Otherwise, backtracking is disabled.

For the target backtrack window, specify the amount of time that you want to be able to rewind your database using Backtrack. Aurora tries to retain enough change records to support that window of time.

Console

You can use the console to configure backtracking when you create a new DB cluster. You can also modify a DB cluster to change the backtrack window for a backtrack-enabled cluster. If you turn off backtracking entirely for a cluster by setting the backtrack window to 0, you can't enable backtrack again for that cluster.

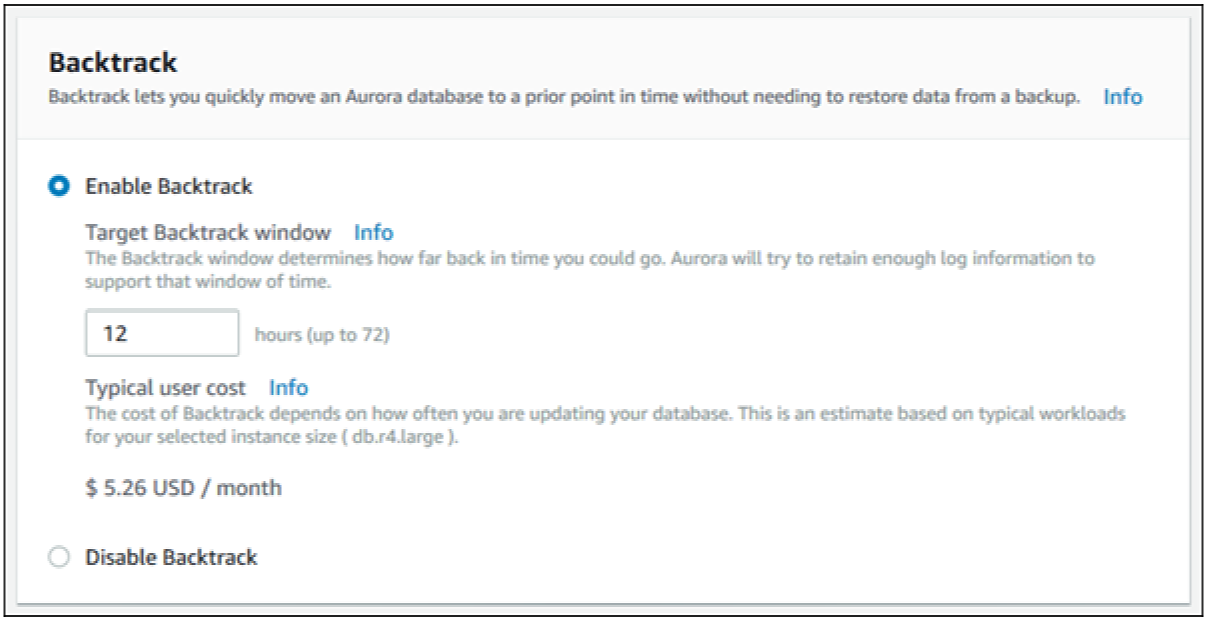
Topics

- Configuring backtracking with the console when creating a DB cluster
- Configuring backtrack with the console when modifying a DB cluster

Configuring backtracking with the console when creating a DB cluster

When you create a new Aurora MySQL DB cluster, backtracking is configured when you choose **Enable Backtrack** and specify a **Target Backtrack window** value that is greater than zero in the **Backtrack** section.

To create a DB cluster, follow the instructions in Creating an Amazon Aurora DB cluster. The following image shows the **Backtrack** section.



Backtrack

Backtrack lets you quickly move an Aurora database to a prior point in time without needing to restore data from a backup. Info

● Enable Backtrack

Target Backtrack window Info

The Backtrack window determines how far back in time you could go. Aurora will try to retain enough log information to support that window of time.

12 hours (up to 72)

Typical user cost Info

The cost of Backtrack depends on how often you are updating your database. This is an estimate based on typical workloads for your selected instance size (db.r4.large).

$ 5.26 USD / month

○ Disable Backtrack

When you create a new DB cluster, Aurora has no data for the DB cluster's workload. So it can't estimate a cost specifically for the new DB cluster. Instead, the console presents a typical user cost for the specified target backtrack window based on a typical workload. The typical cost is meant to provide a general reference for the cost of the Backtrack feature.

> **Important**
>
> Your actual cost might not match the typical cost, because your actual cost is based on your DB cluster's workload.

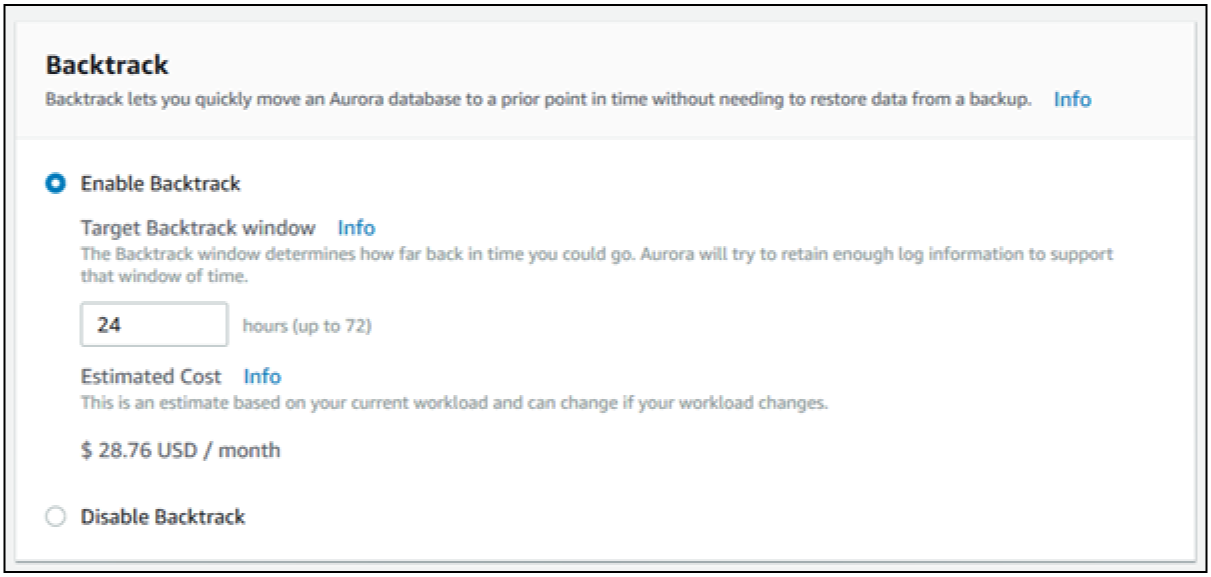
Configuring backtrack with the console when modifying a DB cluster

You can modify backtracking for a DB cluster using the console.

> **Note**
>
> Currently, you can modify backtracking only for a DB cluster that has the Backtrack feature enabled. The **Backtrack** section doesn't appear for a DB cluster that was created with the Backtrack feature disabled or if the Backtrack feature has been disabled for the DB cluster.

To modify backtracking for a DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose the cluster that you want to modify, and choose **Modify**.
4. For **Target Backtrack window**, modify the amount of time that you want to be able to backtrack. The limit is 72 hours.



Backtrack

Backtrack lets you quickly move an Aurora database to a prior point in time without needing to restore data from a backup. Info

● Enable Backtrack

Target Backtrack window Info

The Backtrack window determines how far back in time you could go. Aurora will try to retain enough log information to support that window of time.

24 hours (up to 72)

Estimated Cost Info

This is an estimate based on your current workload and can change if your workload changes.

$ 28.76 USD / month

○ Disable Backtrack

The console shows the estimated cost for the amount of time you specified based on the DB cluster's past workload:

- If backtracking was disabled on the DB cluster, the cost estimate is based on the `VolumeWriteIOPS` metric for the DB cluster in Amazon CloudWatch.
- If backtracking was enabled previously on the DB cluster, the cost estimate is based on the `BacktrackChangeRecordsCreationRate` metric for the DB cluster in Amazon CloudWatch.

5. Choose **Continue**.
6. For **Scheduling of Modifications**, choose one of the following:
   - **Apply during the next scheduled maintenance window** – Wait to apply the **Target Backtrack window** modification until the next maintenance window.
   - **Apply immediately** – Apply the **Target Backtrack window** modification as soon as possible.
7. Choose **Modify cluster**.

AWS CLI

When you create a new Aurora MySQL DB cluster using the create-db-cluster AWS CLI command, backtracking is configured when you specify a `--backtrack-window` value that is greater than zero. The `--backtrack-window` value specifies the target backtrack window. For more information, see Creating an Amazon Aurora DB cluster.

You can also specify the `--backtrack-window` value using the following AWS CLI commands:

- modify-db-cluster
- restore-db-cluster-from-s3
- restore-db-cluster-from-snapshot
- restore-db-cluster-to-point-in-time

The following procedure describes how to modify the target backtrack window for a DB cluster using the AWS CLI.

To modify the target backtrack window for a DB cluster using the AWS CLI

- Call the modify-db-cluster AWS CLI command and supply the following values:

  - `--db-cluster-identifier` – The name of the DB cluster.
  - `--backtrack-window` – The maximum number of seconds that you want to be able to backtrack the DB cluster.

  The following example sets the target backtrack window for `sample-cluster` to one day (86,400 seconds).

  For Linux, macOS, or Unix:

  ```
  aws rds modify-db-cluster \
      --db-cluster-identifier sample-cluster \
      --backtrack-window 86400
  ```

  For Windows:

  ```
  aws rds modify-db-cluster ^
      --db-cluster-identifier sample-cluster ^
      --backtrack-window 86400
  ```

Note

Currently, you can enable backtracking only for a DB cluster that was created with the Backtrack feature enabled.

RDS API

When you create a new Aurora MySQL DB cluster using the CreateDBCluster Amazon RDS API operation, backtracking is configured when you specify a `BacktrackWindow` value that is greater than zero. The `BacktrackWindow` value specifies the target backtrack window for the DB cluster specified in the `DBClusterIdentifier` value. For more information, see Creating an Amazon Aurora DB cluster.

You can also specify the `BacktrackWindow` value using the following API operations:

- ModifyDBCluster
- RestoreDBClusterFromS3
- RestoreDBClusterFromSnapshot
- RestoreDBClusterToPointInTime

Note

Currently, you can enable backtracking only for a DB cluster that was created with the Backtrack feature enabled.

Performing a backtrack

You can backtrack a DB cluster to a specified backtrack time stamp. If the backtrack time stamp isn't earlier than the earliest possible backtrack time, and isn't in the future, the DB cluster is backtracked to that time stamp.

Otherwise, an error typically occurs. Also, if you try to backtrack a DB cluster for which binary logging is enabled, an error typically occurs unless you've chosen to force the backtrack to occur. Forcing a backtrack to occur can interfere with other operations that use binary logging.

⚠ **Important**

Backtracking doesn't generate binlog entries for the changes that it makes. If you have binary logging enabled for the DB cluster, backtracking might not be compatible with your binlog implementation.

ⓘ **Note**

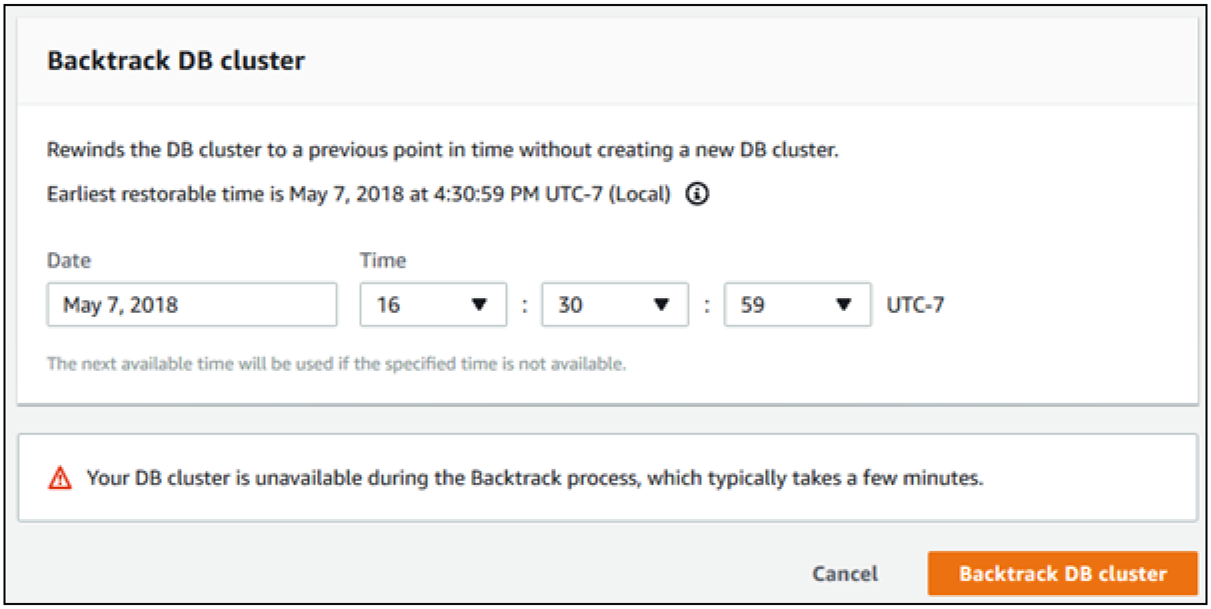
For database clones, you can't backtrack the DB cluster earlier than the date and time when the clone was created. For more information about database cloning, see Cloning a volume for an Amazon Aurora DB cluster.

Console

The following procedure describes how to perform a backtrack operation for a DB cluster using the console.

To perform a backtrack operation using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Instances**.
3. Choose the primary instance for the DB cluster that you want to backtrack.
4. For **Actions**, choose **Backtrack DB cluster**.
5. On the **Backtrack DB cluster** page, enter the backtrack time stamp to backtrack the DB cluster to.



Backtrack DB cluster

Rewinds the DB cluster to a previous point in time without creating a new DB cluster.

Earliest restorable time is May 7, 2018 at 4:30:59 PM UTC-7 (Local)

Date: May 7, 2018

Time: 16 : 30 : 59 UTC-7

The next available time will be used if the specified time is not available.

Your DB cluster is unavailable during the Backtrack process, which typically takes a few minutes.

6. Choose **Backtrack DB cluster**.

AWS CLI

The following procedure describes how to backtrack a DB cluster using the AWS CLI.

To backtrack a DB cluster using the AWS CLI

- Call the backtrack-db-cluster AWS CLI command and supply the following values:
  - `--db-cluster-identifier` – The name of the DB cluster.
  - `--backtrack-to` – The backtrack time stamp to backtrack the DB cluster to, specified in ISO 8601 format.

  The following example backtracks the DB cluster `sample-cluster` to March 19, 2018, at 10 a.m.

  For Linux, macOS, or Unix:

```
aws rds backtrack-db-cluster \
    --db-cluster-identifier sample-cluster \
    --backtrack-to 2018-03-19T10:00:00+00:00
```

  For Windows:

```
aws rds backtrack-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --backtrack-to 2018-03-19T10:00:00+00:00
```

RDS API

To backtrack a DB cluster using the Amazon RDS API, use the BacktrackDBCluster operation. This operation backtracks the DB cluster specified in the `DBClusterIdentifier` value to the specified time.

Monitoring backtracking

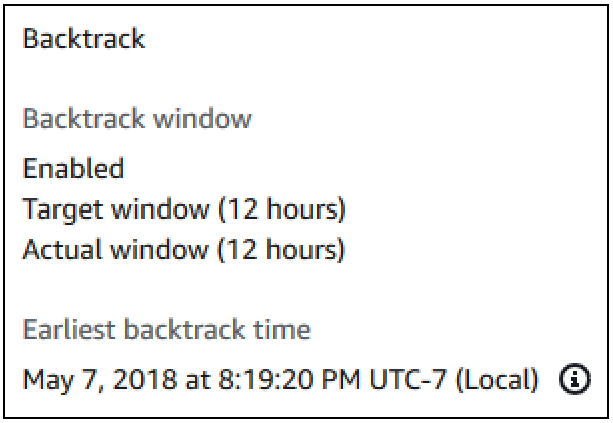
You can view backtracking information and monitor backtracking metrics for a DB cluster.

Console

To view backtracking information and monitor backtracking metrics using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose the DB cluster name to open information about it.

   The backtrack information is in the **Backtrack** section.



Backtrack

Backtrack window

Enabled
Target window (12 hours)
Actual window (12 hours)

Earliest backtrack time

May 7, 2018 at 8:19:20 PM UTC-7 (Local) ⓘ

   When backtracking is enabled, the following information is available:

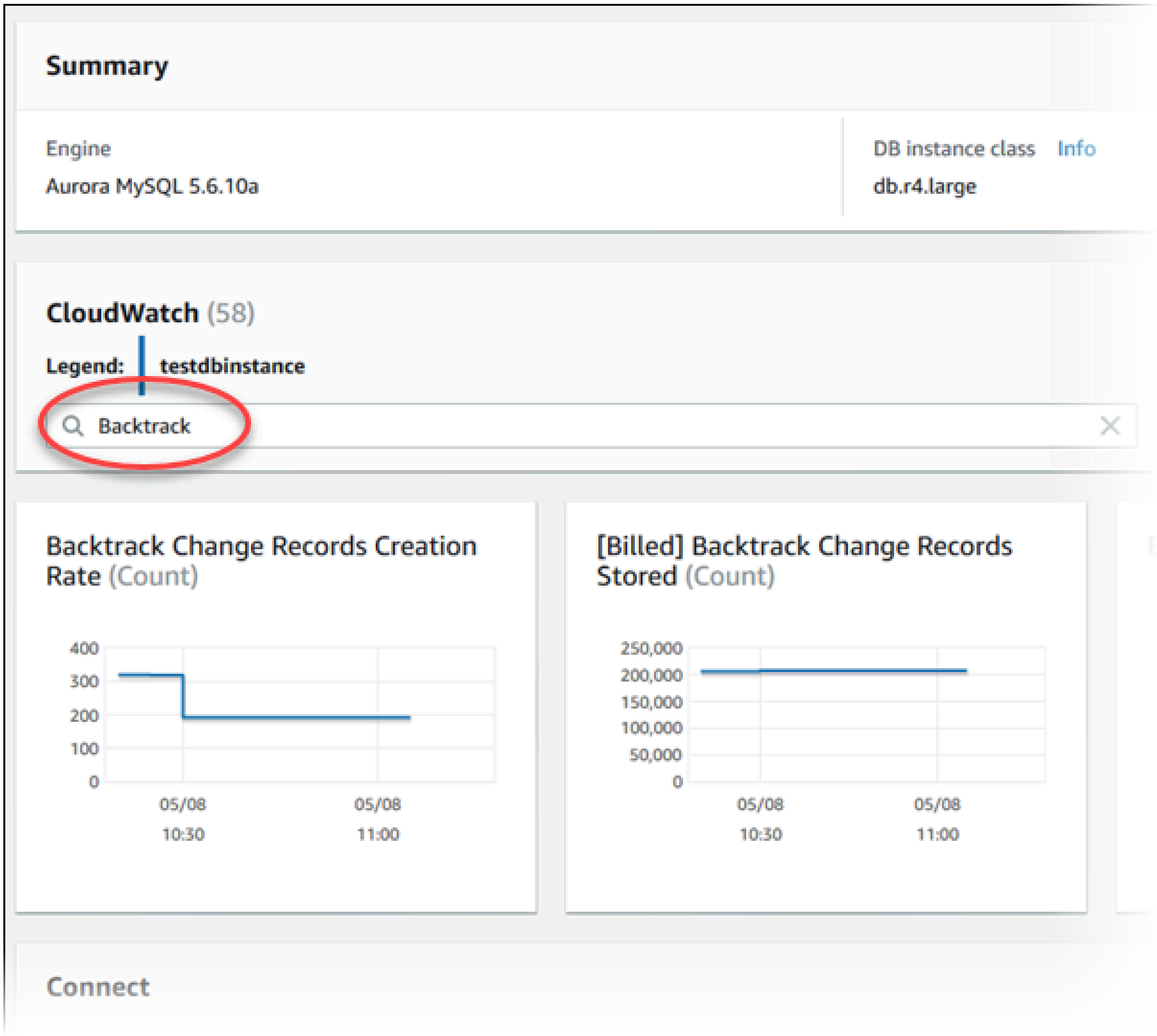
- **Target window –** The current amount of time specified for the target backtrack window. The target is the maximum amount of time that you can backtrack if there is sufficient storage.
- **Actual window –** The actual amount of time you can backtrack, which can be smaller than the target backtrack window. The actual backtrack window is based on your workload and the storage available for retaining backtrack change records.
- **Earliest backtrack time –** The earliest possible backtrack time for the DB cluster. You can't backtrack the DB cluster to a time before the displayed time.

4. Do the following to view backtracking metrics for the DB cluster:

   a. In the navigation pane, choose **Instances**.

   b. Choose the name of the primary instance for the DB cluster to display its details.

   c. In the **CloudWatch** section, type **`Backtrack`** into the **CloudWatch** box to show only the Backtrack metrics.



Summary

Engine: Aurora MySQL 5.6.10a

DB instance class: db.r4.large

CloudWatch (58)

Legend: testdbinstance

Backtrack

The following metrics are displayed:

- **Backtrack Change Records Creation Rate (Count)** – This metric shows the number of backtrack change records created over five minutes for your DB cluster. You can use this metric to estimate the backtrack cost for your target backtrack window.
- **[Billed] Backtrack Change Records Stored (Count)** – This metric shows the actual number of backtrack change records used by your DB cluster.
- **Backtrack Window Actual (Minutes)** – This metric shows whether there is a difference between the target backtrack window and the actual backtrack window. For example, if your target backtrack window is 2 hours (120 minutes), and this metric shows that the actual backtrack window is 100 minutes, then the actual backtrack window is smaller than the target.

- **Backtrack Window Alert (Count)** – This metric shows how often the actual backtrack window is smaller than the target backtrack window for a given period of time.

> **Note**
>
> The following metrics might lag behind the current time:
>
> - **Backtrack Change Records Creation Rate (Count)**
> - **[Billed] Backtrack Change Records Stored (Count)**

AWS CLI

The following procedure describes how to view backtrack information for a DB cluster using the AWS CLI.

To view backtrack information for a DB cluster using the AWS CLI

- Call the describe-db-clusters AWS CLI command and supply the following values:
  - `--db-cluster-identifier` – The name of the DB cluster.

  The following example lists backtrack information for `sample-cluster`.

  For Linux, macOS, or Unix:

  ```
  aws rds describe-db-clusters \
      --db-cluster-identifier sample-cluster
  ```

  For Windows:

  ```
  aws rds describe-db-clusters ^
      --db-cluster-identifier sample-cluster
  ```

RDS API

To view backtrack information for a DB cluster using the Amazon RDS API, use the DescribeDBClusters operation. This operation returns backtrack information for the DB cluster specified in the `DBClusterIdentifier` value.

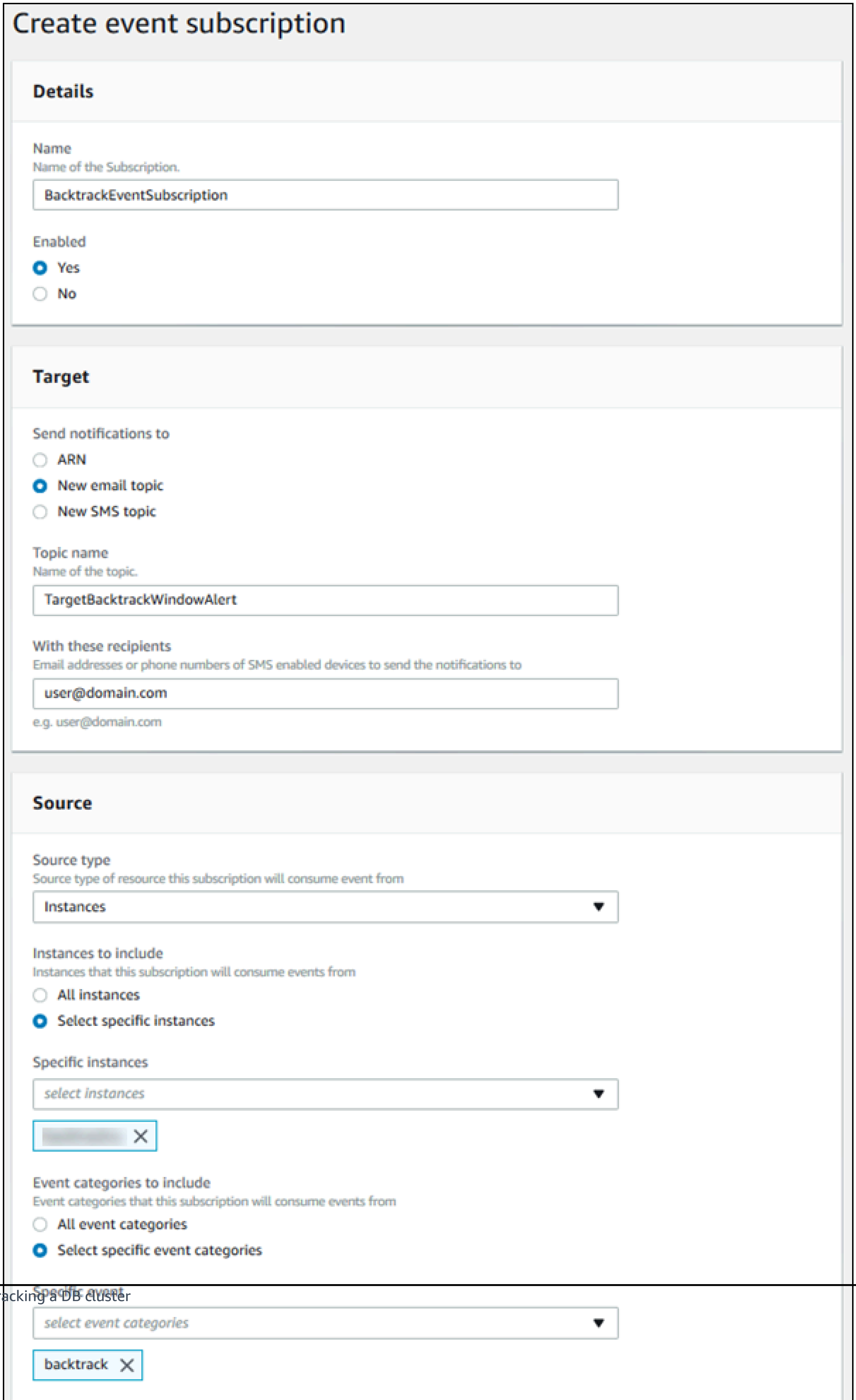
Subscribing to a backtrack event with the console

The following procedure describes how to subscribe to a backtrack event using the console. The event sends you an email or text notification when your actual backtrack window is smaller than your target backtrack window.

To view backtrack information using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Event subscriptions**.
3. Choose **Create event subscription**.
4. In the **Name** box, type a name for the event subscription, and ensure that **Yes** is selected for **Enabled**.
5. In the **Target** section, choose **New email topic**.
6. For **Topic name**, type a name for the topic, and for **With these recipients**, enter the email addresses or phone numbers to receive the notifications.
7. In the **Source** section, choose **Instances** for **Source type**.
8. For **Instances to include**, choose **Select specific instances**, and choose your DB instance.
9. For **Event categories to include**, choose **Select specific event categories**, and choose **backtrack**.

   Your page should look similar to the following page.



Create event subscription

Details

Name
Name of the Subscription.
BacktrackEventSubscription

Enabled
● Yes
○ No

Target

Send notifications to
○ ARN
● New email topic
○ New SMS topic

Topic name
Name of the topic.
TargetBacktrackWindowAlert

With these recipients
Email addresses or phone numbers of SMS enabled devices to send the notifications to
user@domain.com
e.g. user@domain.com

Source

Source type
Source type of resource this subscription will consume event from
Instances

Instances to include
Instances that this subscription will consume events from
○ All instances
● Select specific instances

Specific instances
select instances
[redacted] ✕

Event categories to include
Event categories that this subscription will consume events from
○ All event categories
● Select specific event categories

Specific event
select event categories
backtrack ✕

10. Choose **Create**.

Retrieving existing backtracks

You can retrieve information about existing backtracks for a DB cluster. This information includes the unique identifier of the backtrack, the date and time backtracked to and from, the date and time the backtrack was requested, and the current status of the backtrack.

> **Note**
>
> Currently, you can't retrieve existing backtracks using the console.

AWS CLI

The following procedure describes how to retrieve existing backtracks for a DB cluster using the AWS CLI.

To retrieve existing backtracks using the AWS CLI

- Call the describe-db-cluster-backtracks AWS CLI command and supply the following values:

  - `--db-cluster-identifier` – The name of the DB cluster.

  The following example retrieves existing backtracks for `sample-cluster`.

  For Linux, macOS, or Unix:

  ```
  aws rds describe-db-cluster-backtracks \
      --db-cluster-identifier sample-cluster
  ```

  For Windows:

  ```
  aws rds describe-db-cluster-backtracks ^
      --db-cluster-identifier sample-cluster
  ```

RDS API

To retrieve information about the backtracks for a DB cluster using the Amazon RDS API, use the DescribeDBClusterBacktracks operation. This operation returns information about backtracks for the DB cluster specified in the `DBClusterIdentifier` value.

Disabling backtracking for a DB cluster

You can disable the Backtrack feature for a DB cluster.

Console

You can disable backtracking for a DB cluster using the console. After you turn off backtracking entirely for a cluster, you can't enable it again for that cluster.

To disable the Backtrack feature for a DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose the cluster you want to modify, and choose **Modify**.
4. In the **Backtrack** section, choose **Disable Backtrack**.
5. Choose **Continue**.
6. For **Scheduling of Modifications**, choose one of the following:
   - **Apply during the next scheduled maintenance window** – Wait to apply the modification until the next maintenance window.
   - **Apply immediately** – Apply the modification as soon as possible.
7. Choose **Modify Cluster**.

AWS CLI

You can disable the Backtrack feature for a DB cluster using the AWS CLI by setting the target backtrack window to `0` (zero). After you turn off backtracking entirely for a cluster, you can't enable it again for that cluster.

To modify the target backtrack window for a DB cluster using the AWS CLI

- Call the modify-db-cluster AWS CLI command and supply the following values:

- `--db-cluster-identifier` – The name of the DB cluster.
- `--backtrack-window` – specify `0` to turn off backtracking.

The following example disables the Backtrack feature for the `sample-cluster` by setting `--backtrack-window` to `0`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --backtrack-window 0
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --backtrack-window 0
```

RDS API

To disable the Backtrack feature for a DB cluster using the Amazon RDS API, use the ModifyDBCluster operation. Set the `BacktrackWindow` value to `0` (zero), and specify the DB cluster in the `DBClusterIdentifier` value. After you turn off backtracking entirely for a cluster, you can't enable it again for that cluster.

Testing Amazon Aurora MySQL using fault injection queries

You can test the fault tolerance of your Aurora MySQL DB cluster by using fault injection queries. Fault injection queries are issued as SQL commands to an Amazon Aurora instance. They let you schedule a simulated occurrence of one of the following events:

- A crash of a writer or reader DB instance
- A failure of an Aurora Replica
- A disk failure
- Disk congestion

When a fault injection query specifies a crash, it forces a crash of the Aurora MySQL DB instance. The other fault injection queries result in simulations of failure events, but don't cause the event to occur. When you submit a fault injection query, you also specify an amount of time for the failure event simulation to occur for.

You can submit a fault injection query to one of your Aurora Replica instances by connecting to the endpoint for the Aurora Replica. For more information, see Amazon Aurora connection management.

Running fault injection queries requires all of the master user privileges. For more information, see Master user account privileges.

Testing an instance crash

You can force a crash of an Amazon Aurora instance using the `ALTER SYSTEM CRASH` fault injection query.

For this fault injection query, a failover will not occur. If you want to test a failover, then you can choose the **Failover** instance action for your DB cluster in the RDS console, or use the failover-db-cluster AWS CLI command or the FailoverDBCluster RDS API operation.

Syntax

```
ALTER SYSTEM CRASH [ INSTANCE | DISPATCHER | NODE ];
```

Options

This fault injection query takes one of the following crash types:

- **`INSTANCE`** — A crash of the MySQL-compatible database for the Amazon Aurora instance is simulated.
- **`DISPATCHER`** — A crash of the dispatcher on the writer instance for the Aurora DB cluster is simulated. The *dispatcher* writes updates to the cluster volume for an Amazon Aurora DB cluster.
- **`NODE`** — A crash of both the MySQL-compatible database and the dispatcher for the Amazon Aurora instance is simulated. For this fault injection simulation, the cache is also deleted.

The default crash type is `INSTANCE`.

Testing an Aurora replica failure

You can simulate the failure of an Aurora Replica using the `ALTER SYSTEM SIMULATE READ REPLICA FAILURE` fault injection query.

An Aurora Replica failure blocks all requests from the writer instance to an Aurora Replica or all Aurora Replicas in the DB cluster for a specified time interval. When the time interval completes, the affected Aurora Replicas will be automatically synced up with master instance.

Syntax

```
ALTER SYSTEM SIMULATE percentage_of_failure PERCENT READ REPLICA FAILURE
    [ TO ALL | TO "replica name" ]
    FOR INTERVAL quantity { YEAR | QUARTER | MONTH | WEEK | DAY | HOUR | MINUTE |
 SECOND };
```

Options

This fault injection query takes the following parameters:

- **`percentage_of_failure`** — The percentage of requests to block during the failure event. This value can be a double between 0 and 100. If you specify 0, then no requests are blocked. If you specify 100, then all requests are blocked.
- **Failure type** — The type of failure to simulate. Specify `TO ALL` to simulate failures for all Aurora Replicas in the DB cluster. Specify `TO` and the name of the Aurora Replica to simulate a failure of a single Aurora Replica. The default failure type is `TO ALL`.
- **`quantity`** — The amount of time for which to simulate the Aurora Replica failure. The interval is an amount followed by a time unit. The simulation will occur for that amount of the specified unit. For example, `20 MINUTE` will result in the simulation running for 20 minutes.

> **Note**
>
> Take care when specifying the time interval for your Aurora Replica failure event. If you specify too long of a time interval, and your writer instance writes a large amount of data during the failure event, then your Aurora DB cluster might assume that your Aurora Replica has crashed and replace it.

Testing a disk failure

You can simulate a disk failure for an Aurora DB cluster using the `ALTER SYSTEM SIMULATE DISK FAILURE` fault injection query.

During a disk failure simulation, the Aurora DB cluster randomly marks disk segments as faulting. Requests to those segments will be blocked for the duration of the simulation.

Syntax

```
ALTER SYSTEM SIMULATE percentage_of_failure PERCENT DISK FAILURE
    [ IN DISK index | NODE index ]
    FOR INTERVAL quantity { YEAR | QUARTER | MONTH | WEEK | DAY | HOUR | MINUTE |
 SECOND };
```

Options

This fault injection query takes the following parameters:

- **`percentage_of_failure`** — The percentage of the disk to mark as faulting during the failure event. This value can be a double between 0 and 100. If you specify 0, then none of the disk is marked as faulting. If you specify 100, then the entire disk is marked as faulting.
- **`DISK index`** — A specific logical block of data to simulate the failure event for. If you exceed the range of available logical blocks of data, you will receive an error that tells you the maximum index value that you can specify. For more information, see Displaying volume status for an Aurora MySQL DB cluster.
- **`NODE index`** — A specific storage node to simulate the failure event for. If you exceed the range of available storage nodes, you will receive an error that tells you the maximum index value that you can specify. For more information, see Displaying volume status for an Aurora MySQL DB cluster.
- **`quantity`** — The amount of time for which to simulate the disk failure. The interval is an amount followed by a time unit. The simulation will occur for that amount of the specified unit. For example, `20 MINUTE` will result in the simulation running for 20 minutes.

Testing disk congestion

You can simulate a disk failure for an Aurora DB cluster using the `ALTER SYSTEM SIMULATE DISK CONGESTION` fault injection query.

During a disk congestion simulation, the Aurora DB cluster randomly marks disk segments as congested. Requests to those segments will be delayed between the specified minimum and maximum delay time for the duration of the simulation.

Syntax

```
ALTER SYSTEM SIMULATE percentage_of_failure PERCENT DISK CONGESTION
    BETWEEN minimum AND maximum MILLISECONDS
    [ IN DISK index | NODE index ]
    FOR INTERVAL quantity { YEAR | QUARTER | MONTH | WEEK | DAY | HOUR | MINUTE |
 SECOND };
```

Options

This fault injection query takes the following parameters:

- **`percentage_of_failure`** — The percentage of the disk to mark as congested during the failure event. This value can be a double between 0 and 100. If you specify 0, then none of the disk is marked as congested. If you specify 100, then the entire disk is marked as congested.
- **`DISK index` Or `NODE index`** — A specific disk or node to simulate the failure event for. If you exceed the range of indexes for the disk or node, you will receive an error that tells you the maximum index value that you can specify.
- **`minimum` And `maximum`** — The minimum and maximum amount of congestion delay, in milliseconds. Disk segments marked as congested will be delayed for a random amount of time within the range of the minimum and maximum amount of milliseconds for the duration of the simulation.
- **`quantity`** — The amount of time for which to simulate the disk congestion. The interval is an amount followed by a time unit. The simulation will occur for that amount of the specified time unit. For example, `20 MINUTE` will result in the simulation running for 20 minutes.

Altering tables in Amazon Aurora using Fast DDL

Amazon Aurora includes optimizations to run an `ALTER TABLE` operation in place, nearly instantaneously. The operation completes without requiring the table to be copied and without having a material impact on other DML statements. Because the operation doesn't consume temporary storage for a table copy, it makes DDL statements practical even for large tables on small instance classes.

Aurora MySQL version 3 is compatible with the MySQL 8.0 feature called instant DDL. Aurora MySQL version 2 uses a different implementation called Fast DDL.

Topics

- Instant DDL (Aurora MySQL version 3)
- Fast DDL (Aurora MySQL version 2)

Instant DDL (Aurora MySQL version 3)

The optimization performed by Aurora MySQL version 3 to improve the efficiency of some DDL operations is called instant DDL.

Aurora MySQL version 3 is compatible with the instant DDL from community MySQL 8.0. You perform an instant DDL operation by using the clause `ALGORITHM=INSTANT` with the `ALTER TABLE` statement. For syntax and usage details about instant DDL, see ALTER TABLE and Online DDL Operations in the MySQL documentation.

The following examples demonstrate the instant DDL feature. The `ALTER TABLE` statements add columns and change default column values. The examples include both regular and virtual columns, and both regular and partitioned tables. At each step, you can see the results by issuing `SHOW CREATE TABLE` and `DESCRIBE` statements.

```
mysql> CREATE TABLE t1 (a INT, b INT, KEY(b)) PARTITION BY KEY(b) PARTITIONS 6;
Query OK, 0 rows affected (0.02 sec)

mysql> ALTER TABLE t1 RENAME TO t2, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

mysql> ALTER TABLE t2 ALTER COLUMN b SET DEFAULT 100, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.00 sec)

mysql> ALTER TABLE t2 ALTER COLUMN b DROP DEFAULT, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

mysql> ALTER TABLE t2 ADD COLUMN c ENUM('a', 'b', 'c'), ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

mysql> ALTER TABLE t2 MODIFY COLUMN c ENUM('a', 'b', 'c', 'd', 'e'), ALGORITHM =
 INSTANT;
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> ALTER TABLE t2 ADD COLUMN (d INT GENERATED ALWAYS AS (a + 1) VIRTUAL), ALGORITHM
 = INSTANT;
Query OK, 0 rows affected (0.02 sec)

mysql> ALTER TABLE t2 ALTER COLUMN a SET DEFAULT 20,
    ->   ALTER COLUMN b SET DEFAULT 200, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE t3 (a INT, b INT) PARTITION BY LIST(a)(
    ->   PARTITION mypart1 VALUES IN (1,3,5),
    ->   PARTITION MyPart2 VALUES IN (2,4,6)
    -> );
Query OK, 0 rows affected (0.03 sec)

mysql> ALTER TABLE t3 ALTER COLUMN a SET DEFAULT 20, ALTER COLUMN b SET DEFAULT 200,
 ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE t4 (a INT, b INT) PARTITION BY RANGE(a)
    ->   (PARTITION p0 VALUES LESS THAN(100), PARTITION p1 VALUES LESS THAN(1000),
    ->   PARTITION p2 VALUES LESS THAN MAXVALUE);
Query OK, 0 rows affected (0.05 sec)

mysql> ALTER TABLE t4 ALTER COLUMN a SET DEFAULT 20,
    ->   ALTER COLUMN b SET DEFAULT 200, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)

/* Sub-partitioning example */
mysql> CREATE TABLE ts (id INT, purchased DATE, a INT, b INT)
    ->   PARTITION BY RANGE( YEAR(purchased) )
    ->     SUBPARTITION BY HASH( TO_DAYS(purchased) )
    ->     SUBPARTITIONS 2 (
    ->       PARTITION p0 VALUES LESS THAN (1990),
    ->       PARTITION p1 VALUES LESS THAN (2000),
    ->       PARTITION p2 VALUES LESS THAN MAXVALUE
    ->    );
Query OK, 0 rows affected (0.10 sec)

mysql> ALTER TABLE ts ALTER COLUMN a SET DEFAULT 20,
    ->   ALTER COLUMN b SET DEFAULT 200, ALGORITHM = INSTANT;
Query OK, 0 rows affected (0.01 sec)
```

Fast DDL (Aurora MySQL version 2)

In MySQL, many data definition language (DDL) operations have a significant performance impact.

For example, suppose that you use an `ALTER TABLE` operation to add a column to a table. Depending on the algorithm specified for the operation, this operation can involve the following:

- Creating a full copy of the table
- Creating a temporary table to process concurrent data manipulation language (DML) operations
- Rebuilding all indexes for the table
- Applying table locks while applying concurrent DML changes
- Slowing concurrent DML throughput

The optimization performed by Aurora MySQL version 2 to improve the efficiency of some DDL operations is called Fast DDL.

In Aurora MySQL version 3, Aurora uses the MySQL 8.0 feature called instant DDL. Aurora MySQL version 2 uses a different implementation called Fast DDL.

> ⚠ **Important**
>
> Currently, Aurora lab mode must be enabled to use Fast DDL for Aurora MySQL. We don't recommend using Fast DDL for production DB clusters. For information about enabling Aurora lab mode, see Amazon Aurora MySQL lab mode.

Fast DDL limitations

Currently, Fast DDL has the following limitations:

- Fast DDL only supports adding nullable columns, without default values, to the end of an existing table.
- Fast DDL doesn't work for partitioned tables.
- Fast DDL doesn't work for InnoDB tables that use the REDUNDANT row format.
- Fast DDL doesn't work for tables with full-text search indexes.
- If the maximum possible record size for the DDL operation is too large, Fast DDL is not used. A record size is too large if it is greater than half the page size. The maximum size of a record is computed by adding the maximum sizes of all columns. For variable sized columns, according to InnoDB standards, extern bytes are not included for computation.

Fast DDL syntax

```
ALTER TABLE tbl_name ADD COLUMN col_name column_definition
```

This statement takes the following options:

- **`tbl_name`** — The name of the table to be modified.
- **`col_name`** — The name of the column to be added.
- **`col_definition`** — The definition of the column to be added.

> **Note**
>
> You must specify a nullable column definition without a default value. Otherwise, Fast DDL isn't used.

Fast DDL examples

The following examples demonstrate the speedup from Fast DDL operations. The first SQL example runs `ALTER TABLE` statements on a large table without using Fast DDL. This operation takes substantial time. A CLI example shows how to enable Fast DDL for the cluster. Then another SQL example runs the same `ALTER TABLE` statements on an identical table. With Fast DDL enabled, the operation is very fast.

This example uses the `ORDERS` table from the TPC-H benchmark, containing 150 million rows. This cluster intentionally uses a relatively small instance class, to demonstrate how long `ALTER TABLE` statements can take when you can't use Fast DDL. The example creates a clone of the original table containing identical data. Checking the `aurora_lab_mode` setting confirms that the cluster can't use Fast DDL, because lab mode isn't enabled. Then `ALTER TABLE ADD COLUMN` statements take substantial time to add new columns at the end of the table.

```
mysql> create table orders_regular_ddl like orders;
Query OK, 0 rows affected (0.06 sec)

mysql> insert into orders_regular_ddl select * from orders;
Query OK, 150000000 rows affected (1 hour 1 min 25.46 sec)

mysql> select @@aurora_lab_mode;
+-------------------+
| @@aurora_lab_mode |
+-------------------+
|                 0 |
+-------------------+

mysql> ALTER TABLE orders_regular_ddl ADD COLUMN o_refunded boolean;
Query OK, 0 rows affected (40 min 31.41 sec)

mysql> ALTER TABLE orders_regular_ddl ADD COLUMN o_coverletter varchar(512);
Query OK, 0 rows affected (40 min 44.45 sec)
```

This example does the same preparation of a large table as the previous example. However, you can't simply enable lab mode within an interactive SQL session. That setting must be enabled in a custom parameter group. Doing so requires switching out of the `mysql` session and running some AWS CLI commands or using the AWS Management Console.

```
mysql> create table orders_fast_ddl like orders;
Query OK, 0 rows affected (0.02 sec)

mysql> insert into orders_fast_ddl select * from orders;
Query OK, 150000000 rows affected (58 min 3.25 sec)

mysql> set aurora_lab_mode=1;
ERROR 1238 (HY000): Variable 'aurora_lab_mode' is a read only variable
```

Enabling lab mode for the cluster requires some work with a parameter group. This AWS CLI example uses a cluster parameter group, to ensure that all DB instances in the cluster use the same value for the lab mode setting.

```
$ aws rds create-db-cluster-parameter-group \
  --db-parameter-group-family aurora5.7 \
    --db-cluster-parameter-group-name lab-mode-enabled-57 --description 'TBD'
$ aws rds describe-db-cluster-parameters \
  --db-cluster-parameter-group-name lab-mode-enabled-57 \
    --query '*[*].[ParameterName,ParameterValue]' \
      --output text | grep aurora_lab_mode
aurora_lab_mode 0
$ aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name lab-mode-enabled-57 \
    --parameters ParameterName=aurora_lab_mode,ParameterValue=1,ApplyMethod=pending-
reboot
{
    "DBClusterParameterGroupName": "lab-mode-enabled-57"
}

# Assign the custom parameter group to the cluster that's going to use Fast DDL.
$ aws rds modify-db-cluster --db-cluster-identifier tpch100g \
  --db-cluster-parameter-group-name lab-mode-enabled-57
{
  "DBClusterIdentifier": "tpch100g",
  "DBClusterParameterGroup": "lab-mode-enabled-57",
  "Engine": "aurora-mysql",
  "EngineVersion": "5.7.mysql_aurora.2.10.2",
  "Status": "available"
}

# Reboot the primary instance for the cluster tpch100g:
$ aws rds reboot-db-instance --db-instance-identifier instance-2020-12-22-5208
{
  "DBInstanceIdentifier": "instance-2020-12-22-5208",
  "DBInstanceStatus": "rebooting"
}

$ aws rds describe-db-clusters --db-cluster-identifier tpch100g \
  --query '*[].[DBClusterParameterGroup]' --output text
lab-mode-enabled-57

$ aws rds describe-db-cluster-parameters \
  --db-cluster-parameter-group-name lab-mode-enabled-57 \
    --query '*[*].{ParameterName:ParameterName,ParameterValue:ParameterValue}' \
      --output text | grep aurora_lab_mode
aurora_lab_mode 1
```

The following example shows the remaining steps after the parameter group change takes effect. It tests the `aurora_lab_mode` setting to make sure that the cluster can use Fast DDL. Then it runs `ALTER TABLE` statements to add columns to the end of another large table. This time, the statements finish very quickly.

```
mysql> select @@aurora_lab_mode;
+-------------------+
| @@aurora_lab_mode |
+-------------------+
|                 1 |
+-------------------+
mysql> ALTER TABLE orders_fast_ddl ADD COLUMN o_refunded boolean;
Query OK, 0 rows affected (1.51 sec)

mysql> ALTER TABLE orders_fast_ddl ADD COLUMN o_coverletter varchar(512);
Query OK, 0 rows affected (0.40 sec)
```

Displaying volume status for an Aurora MySQL DB cluster

In Amazon Aurora, a DB cluster volume consists of a collection of logical blocks. Each of these represents 10 gigabytes of allocated storage. These blocks are called *protection groups*.

The data in each protection group is replicated across six physical storage devices, called *storage nodes*. These storage nodes are allocated across three Availability Zones (AZs) in the AWS Region where the DB cluster resides. In turn, each storage node contains one or more logical blocks of data for the DB cluster volume. For more information about protection groups and storage nodes, see Introducing the Aurora storage engine on the AWS Database Blog.

You can simulate the failure of an entire storage node, or a single logical block of data within a storage node. To do so, you use the `ALTER SYSTEM SIMULATE DISK FAILURE` fault injection statement. For the statement, you specify the index value of a specific logical block of data or storage node. However, if you specify an index value greater than the number of logical blocks of data or storage nodes used by the DB cluster volume, the statement returns an error. For more information about fault injection queries, see Testing Amazon Aurora MySQL using fault injection queries.

You can avoid that error by using the `SHOW VOLUME STATUS` statement. The statement returns two server status variables, `Disks` and `Nodes`. These variables represent the total number of logical blocks of data and storage nodes, respectively, for the DB cluster volume.

Syntax

```
SHOW VOLUME STATUS
```

Example

The following example illustrates a typical SHOW VOLUME STATUS result.

```

mysql> SHOW VOLUME STATUS;
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Disks         | 96    |
| Nodes         | 74    |
+---------------+-------+
```

Tuning Aurora MySQL

Wait events and thread states are important tuning tools for Aurora MySQL. If you can find out why sessions are waiting for resources and what they are doing, you are better able to reduce bottlenecks. You can use the information in this section to find possible causes and corrective actions.

Amazon DevOps Guru for RDS can proactively determine whether your Aurora MySQL databases are experiencing problematic conditions that might cause bigger problems later. Amazon DevOps Guru for RDS publishes an explanation and recommendations for corrective actions in a proactive insight. This section contains insights for common problems.

> ⚠ **Important**
>
> The wait events and thread states in this section are specific to Aurora MySQL. Use the information in this section to tune only Amazon Aurora, not Amazon RDS for MySQL. Some wait events in this section have no analogs in the open source versions of these database engines. Other wait events have the same names as events in open source engines, but behave differently. For example, Amazon Aurora storage works different from open source storage, so storage-related wait events indicate different resource conditions.

Topics

Essential concepts for Aurora MySQL tuning

Before you tune your Aurora MySQL database, make sure to learn what wait events and thread states are and why they occur. Also review the basic memory and disk architecture of Aurora MySQL when using the InnoDB storage engine. For a helpful architecture diagram, see the MySQL Reference Manual.

Topics

- Aurora MySQL wait events
- Aurora MySQL thread states
- Aurora MySQL memory
- Aurora MySQL processes

Aurora MySQL wait events

A *wait event* indicates a resource for which a session is waiting. For example, the wait event `io/socket/sql/client_connection` indicates that a thread is in the process of handling a new connection. Typical resources that a session waits for include the following:

- Single-threaded access to a buffer, for example, when a session is attempting to modify a buffer
- A row that is currently locked by another session
- A data file read
- A log file write

For example, to satisfy a query, the session might perform a full table scan. If the data isn't already in memory, the session waits for the disk I/O to complete. When the buffers are read into memory, the session might need to wait because other sessions are accessing the same buffers. The database records the waits by using a predefined wait event. These events are grouped into categories.

A wait event doesn't by itself show a performance problem. For example, if requested data isn't in memory, reading data from disk is necessary. If one session locks a row for an update, another session waits for the row to be unlocked so that it can update it. A commit requires waiting for the write to a log file to complete. Waits are integral to the normal functioning of a database.

Large numbers of wait events typically show a performance problem. In such cases, you can use wait event data to determine where sessions are spending time. For example, if a report that typically runs in minutes now runs for hours, you can identify the wait events that contribute the most to total wait time. If you can determine the causes of the top wait events, you can sometimes make changes that improve performance. For example, if your session is waiting on a row that has been locked by another session, you can end the locking session.

Aurora MySQL thread states

A *general thread state* is a `State` value that is associated with general query processing. For example, the thread state `sending data` indicates that a thread is reading and filtering rows for a query to determine the correct result set.

You can use thread states to tune Aurora MySQL in a similar fashion to how you use wait events. For example, frequent occurrences of `sending data` usually indicate that a query isn't using an index. For more information about thread states, see General Thread States in the *MySQL Reference Manual*.

When you use Performance Insights, one of the following conditions is true:

- Performance Schema is turned on – Aurora MySQL shows wait events rather than the thread state.
- Performance Schema isn't turned on – Aurora MySQL shows the thread state.

We recommend that you configure the Performance Schema for automatic management. The Performance Schema provides additional insights and better tools to investigate potential performance problems. For more information, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

Aurora MySQL memory

In Aurora MySQL, the most important memory areas are the buffer pool and log buffer.

Topics

Buffer pool

The *buffer pool* is the shared memory area where Aurora MySQL caches table and index data. Queries can access frequently used data directly from memory without reading from disk.

The buffer pool is structured as a linked list of pages. A *page* can hold multiple rows. Aurora MySQL uses a least recently used (LRU) algorithm to age pages out of the pool.

For more information, see Buffer Pool in the *MySQL Reference Manual*.

Aurora MySQL processes

Aurora MySQL uses a process model that is very different from Aurora PostgreSQL.

Topics

- MySQL server (mysqld)
- Threads
- Thread pool

MySQL server (mysqld)

The MySQL server is a single operating-system process named mysqld. The MySQL server doesn't spawn additional processes. Thus, an Aurora MySQL database uses mysqld to perform most of its work.

When the MySQL server starts, it listens for network connections from MySQL clients. When a client connects to the database, mysqld opens a thread.

Threads

Connection manager threads associate each client connection with a dedicated thread. This thread manages authentication, runs statements, and returns results to the client. Connection manager creates new threads when necessary.

The *thread cache* is the set of available threads. When a connection ends, MySQL returns the thread to the thread cache if the cache isn't full. The `thread_cache_size` system variable determines the thread cache size.

Thread pool

The *thread pool* consists of a number of thread groups. Each group manages a set of client connections. When a client connects to the database, the thread pool assigns the connections to thread groups in round-robin fashion. The thread pool separates connections and threads. There is no fixed relationship between connections and the threads that run statements received from those connections.

Tuning Aurora MySQL with wait events

The following table summarizes the Aurora MySQL wait events that most commonly indicate performance problems. The following wait events are a subset of the list in Aurora MySQL wait events.

Wait event	Description
cpu	This event occurs when a thread is active in CPU or is waiting for CPU.
io/aurora_redo_log_flush	This event occurs when a session is writing persistent data to Aurora storage.
io/aurora_respond_to_client	This event occurs when a thread is waiting to return a result set to a client.
io/redo_log_flush	This event occurs when a session is writing persistent data to Aurora storage.
io/socket/sql/client_connection	This event occurs when a thread is in the process of handling a new connection.
io/table/sql/handler	This event occurs when work has been delegated to a storage engine.
synch/cond/innodb/row_lock_wait	This event occurs when one session has locked a row for an update, and another session tries to update the same row.
synch/cond/innodb/row_lock_wait_cond	This event occurs when one session has locked a row for an update, and another session tries to update the same row.
synch/cond/sql/MDL_context::COND_wait_status	This event occurs when there are threads waiting on a table metadata lock.

Wait event	Description
synch/mutex/innodb/aurora_lock_thread_slot_futex	This event occurs when one session has locked a row for an update, and another session tries to update the same row.
synch/mutex/innodb/buf_pool_mutex	This event occurs when a thread has acquired a lock on the InnoDB buffer pool to access a page in memory.
synch/mutex/innodb/fil_system_mutex	This event occurs when a session is waiting to access the tablespace memory cache.
synch/mutex/innodb/trx_sys_mutex	This event occurs when there is high database activity with a large number of transactions.
synch/sxlock/innodb/hash_table_locks	This event occurs when pages not found in the buffer pool must be read from a file.

cpu

The `cpu` wait event occurs when a thread is active in CPU or is waiting for CPU.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL versions 2 and 3

Context

For every vCPU, a connection can run work on this CPU. In some situations, the number of active connections that are ready to run is higher than the number of vCPUs. This imbalance results in connections waiting for CPU resources. If the number of active connections stays consistently higher than the number of vCPUs, then your instance experiences CPU contention. The contention causes the `cpu` wait event to occur.

> **Note**
>
> The Performance Insights metric for CPU is `DBLoadCPU`. The value for `DBLoadCPU` can differ from the value for the CloudWatch metric `CPUUtilization`. The latter metric is collected from the HyperVisor for a database instance.

Performance Insights OS metrics provide detailed information about CPU utilization. For example, you can display the following metrics:

- `os.cpuUtilization.nice.avg`
- `os.cpuUtilization.total.avg`
- `os.cpuUtilization.wait.avg`
- `os.cpuUtilization.idle.avg`

Performance Insights reports the CPU usage by the database engine as `os.cpuUtilization.nice.avg`.

Likely causes of increased waits

When this event occurs more than normal, possibly indicating a performance problem, typical causes include the following:

- Analytic queries
- Highly concurrent transactions
- Long-running transactions
- A sudden increase in the number of connections, known as a *login storm*
- An increase in context switching

Actions

If the cpu wait event dominates database activity, it doesn't necessarily indicate a performance problem. Respond to this event only when performance degrades.

Depending on the cause of the increase in CPU utilization, consider the following strategies:

- Increase the CPU capacity of the host. This approach typically gives only temporary relief.
- Identify top queries for potential optimization.
- Redirect some read-only workload to reader nodes, if applicable.

Topics

- Identify the sessions or queries that are causing the problem
- Analyze and optimize the high CPU workload

Identify the sessions or queries that are causing the problem

To find the sessions and queries, look at the **Top SQL** table in Performance Insights for the SQL statements that have the highest CPU load. For more information, see Analyzing metrics with the Performance Insights dashboard.

Typically, one or two SQL statements consume the majority of CPU cycles. Concentrate your efforts on these statements. Suppose that your DB instance has 2 vCPUs with a DB load of 3.1 average active sessions (AAS), all in the CPU state. In this case, your instance is CPU bound. Consider the following strategies:

- Upgrade to a larger instance class with more vCPUs.
- Tune your queries to have lower CPU load.

In this example, the top SQL queries have a DB load of 1.5 AAS, all in the CPU state. Another SQL statement has a load of 0.1 in the CPU state. In this example, if you stopped the lowest-load SQL statement, you don't significantly reduce database load. However, if you optimize the two high-load queries to be twice as efficient, you eliminate the CPU bottleneck. If you reduce the CPU load of 1.5 AAS by 50 percent, the AAS for each statement decreases to 0.75. The total DB load spent on CPU is now 1.6 AAS. This value is below the maximum vCPU line of 2.0.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights. Also see the AWS Support article How can I troubleshoot and resolve high CPU utilization on my Amazon RDS for MySQL instances?.

Analyze and optimize the high CPU workload

After you identify the query or queries increasing CPU usage, you can either optimize them or end the connection. The following example shows how to end a connection.

```
CALL mysql.rds_kill(processID);
```

For more information, see mysql.rds_kill.

If you end a session, the action might trigger a long rollback.

Follow the guidelines for optimizing queries

To optimize queries, consider the following guidelines:

- Run the `EXPLAIN` statement.

  This command shows the individual steps involved in running a query. For more information, see Optimizing Queries with EXPLAIN in the MySQL documentation.
- Run the `SHOW PROFILE` statement.

  Use this statement to review profile details that can indicate resource usage for statements that are run during the current session. For more information, see SHOW PROFILE Statement in the MySQL documentation.
- Run the `ANALYZE TABLE` statement.

  Use this statement to refresh the index statistics for the tables accessed by the high-CPU consuming query. By analyzing the statement, you can help the optimizer choose an appropriate execution plan. For more information, see ANALYZE TABLE Statement in the MySQL documentation.

Follow the guidelines for improving CPU usage

To improve CPU usage in a database instance, follow these guidelines:

- Ensure that all queries are using proper indexes.

- Find out whether you can use Aurora parallel queries. You can use this technique to reduce CPU usage on the head node by pushing down function processing, row filtering, and column projection for the `WHERE` clause.
- Find out whether the number of SQL executions per second meets the expected thresholds.
- Find out whether index maintenance or new index creation takes up CPU cycles needed by your production workload. Schedule maintenance activities outside of peak activity times.
- Find out whether you can use partitioning to help reduce the query data set. For more information, see the blog post How to plan and optimize Amazon Aurora with MySQL compatibility for consolidated workloads.

Check for connection storms

If the `DBLoadCPU` metric is not very high, but the `CPUUtilization` metric is high, the cause of the high CPU utilization lies outside of the database engine. A classic example is a connection storm.

Check whether the following conditions are true:

- There is an increase in both the Performance Insights `CPUUtilization` metric and the Amazon CloudWatch `DatabaseConnections` metric.
- The number of threads in the CPU is greater than the number of vCPUs.

If the preceding conditions are true, consider decreasing the number of database connections. For example, you can use a connection pool such as RDS Proxy. To learn the best practices for effective connection management and scaling, see the whitepaper Amazon Aurora MySQL DBA Handbook for Connection Management.

io/aurora_redo_log_flush

The `io/aurora_redo_log_flush` event occurs when a session is writing persistent data to Amazon Aurora storage.

Topics

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 2

Context

The `io/aurora_redo_log_flush` event is for a write input/output (I/O) operation in Aurora MySQL.

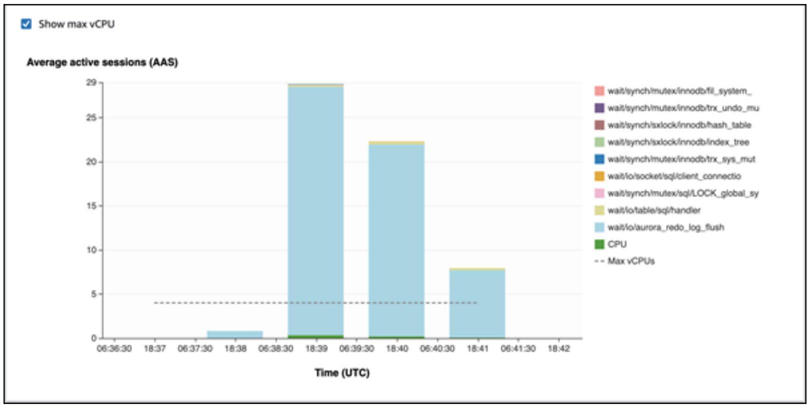
> **Note**
>
> In Aurora MySQL version 3, this wait event is named [io/redo_log_flush](#).

Likely causes of increased waits

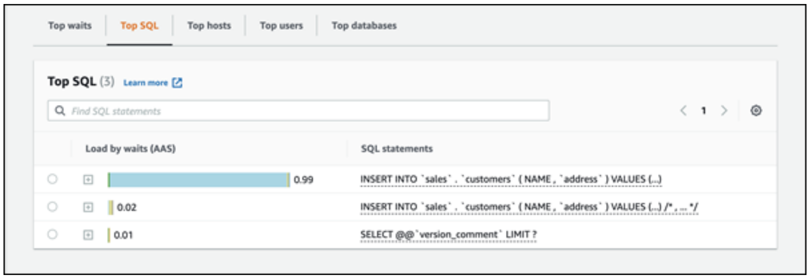
For data persistence, commits require a durable write to stable storage. If the database is doing too many commits, there is a wait event on the write I/O operation, the `io/aurora_redo_log_flush` wait event.

In the following examples, 50,000 records are inserted into an Aurora MySQL DB cluster using the db.r5.xlarge DB instance class:

- In the first example, each session inserts 10,000 records row by row. By default, if a data manipulation language (DML) command isn't within a transaction, Aurora MySQL uses implicit commits. Autocommit is turned on. This means that for each row insertion there is a commit. Performance Insights shows that the connections spend most of their time waiting on the `io/aurora_redo_log_flush` wait event.



This is caused by the simple insert statements used.

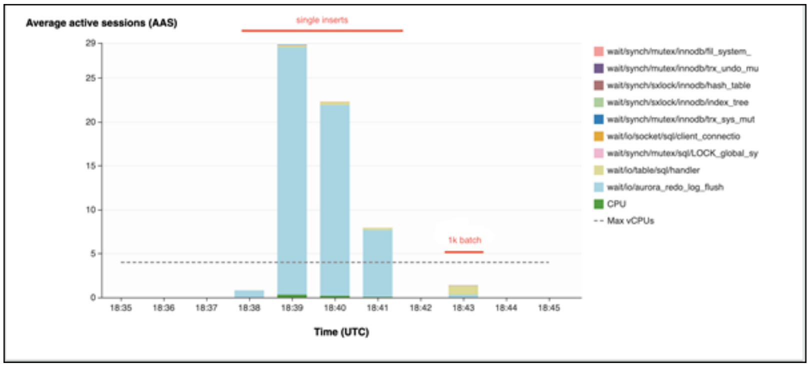


Top SQL (3)

Load by waits (AAS)	SQL statements
0.99	INSERT INTO `sales` . `customers` (NAME , `address`) VALUES (...)
0.02	INSERT INTO `sales` . `customers` (NAME , `address`) VALUES (...) /* , ... */
0.01	SELECT @@ `version_comment` LIMIT ?

The 50,000 records take 3.5 minutes to be inserted.

- In the second example, inserts are made in 1,000 batches, that is each connection performs 10 commits instead of 10,000. Performance Insights shows that the connections don't spend most of their time on the `io/aurora_redo_log_flush` wait event.



The 50,000 records take 4 seconds to be inserted.

Actions

We recommend different actions depending on the causes of your wait event.

Identify the problematic sessions and queries

If your DB instance is experiencing a bottleneck, your first task is to find the sessions and queries that cause it. For a useful AWS Database Blog post, see [Analyze Amazon Aurora MySQL Workloads with Performance Insights](https://aws.amazon.com/blogs/database/analyze-amazon-aurora-mysql-workloads-with-performance-insights/).

To identify sessions and queries causing a bottleneck

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.

3. Choose your DB instance.
4. In **Database load**, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The queries at the top of the list are causing the highest load on the database.

Group your write operations

The following examples trigger the `io/aurora_redo_log_flush` wait event. (Autocommit is turned on.)

```
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
....
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');

UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
....
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
....
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
```

To reduce the time spent waiting on the `io/aurora_redo_log_flush` wait event, group your write operations logically into a single commit to reduce persistent calls to storage.

Turn off autocommit

Turn off autocommit before making large changes that aren't within a transaction, as shown in the following example.

```
SET SESSION AUTOCOMMIT=OFF;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
....
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
-- Other DML statements here
COMMIT;

SET SESSION AUTOCOMMIT=ON;
```

Use transactions

You can use transactions, as shown in the following example.

```
BEGIN
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
....
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
....
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;

-- Other DML statements here
END
```

Use batches

You can make changes in batches, as shown in the following example. However, using batches that are too large can cause performance issues, especially in read replicas or when doing point-in-time recovery (PITR).

```
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES
('xxxx','xxxxx'),('xxxx','xxxxx'),...,('xxxx','xxxxx'),('xxxx','xxxxx');

UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1 BETWEEN xx AND
 xxx;

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1<xx;
```

io/aurora_respond_to_client

The `io/aurora_respond_to_client` event occurs when a thread is waiting to return a result set to a client.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 2

In versions before version 2.07.7, 2.09.3, and 2.10.2, this wait event erroneously includes idle time.

Context

The event `io/aurora_respond_to_client` indicates that a thread is waiting to return a result set to a client.

The query processing is complete, and the results are being returned back to the application client. However, because there isn't enough network bandwidth on the DB cluster, a thread is waiting to return the result set.

Likely causes of increased waits

When the `io/aurora_respond_to_client` event appears more than normal, possibly indicating a performance problem, typical causes include the following:

DB instance class insufficient for the workload

> The DB instance class used by the DB cluster doesn't have the necessary network bandwidth to process the workload efficiently.

Large result sets

There was an increase in size of the result set being returned, because the query returns higher numbers of rows. The larger result set consumes more network bandwidth.

Increased load on the client

There might be CPU pressure, memory pressure, or network saturation on the client. An increase in load on the client delays the reception of data from the Aurora MySQL DB cluster.

Increased network latency

There might be increased network latency between the Aurora MySQL DB cluster and client. Higher network latency increases the time required for the client to receive the data.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Identify the sessions and queries causing the events

You can use Performance Insights to show queries blocked by the `io/aurora_respond_to_client` wait event. Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance isn't optimal, then examine where the database is spending the most time. Look at the wait events that contribute to the highest load, and find out whether you can optimize the database and application to reduce those events.

To find SQL queries that are responsible for high load

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is shown for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the AWS Database Blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Scale the DB instance class

Check for the increase in the value of the Amazon CloudWatch metrics related to network throughput, such as `NetworkReceiveThroughput` and `NetworkTransmitThroughput`. If the DB instance class network bandwidth is being reached, you can scale the DB instance class used by the DB cluster by modifying the DB cluster. A DB instance class with larger network bandwidth returns data to clients more efficiently.

For information about monitoring Amazon CloudWatch metrics, see Viewing metrics in the Amazon RDS console. For information about DB instance classes, see Aurora DB instance classes. For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

Check workload for unexpected results

Check the workload on the DB cluster and make sure that it isn't producing unexpected results. For example, there might be queries that are returning a higher number of rows than expected. In this case, you can use Performance Insights counter metrics such as `Innodb_rows_read`. For more information, see Performance Insights counter metrics.

Distribute workload with reader instances

You can distribute read-only workload with Aurora replicas. You can scale horizontally by adding more Aurora replicas. Doing so can result in an increase in the throttling limits for network bandwidth. For more information, see Amazon Aurora DB clusters.

Use the SQL_BUFFER_RESULT modifier

You can add the `SQL_BUFFER_RESULT` modifier to `SELECT` statements to force the result into a temporary table before they are returned to the client. This modifier can help with performance issues when InnoDB locks aren't being freed because queries are in the `io/aurora_respond_to_client` wait state. For more information, see SELECT Statement in the MySQL documentation.

io/redo_log_flush

The `io/redo_log_flush` event occurs when a session is writing persistent data to Amazon Aurora storage.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 3

Context

The `io/redo_log_flush` event is for a write input/output (I/O) operation in Aurora MySQL.

> **Note**
>
> In Aurora MySQL version 2, this wait event is named io/aurora_redo_log_flush.

Likely causes of increased waits

For data persistence, commits require a durable write to stable storage. If the database is doing too many commits, there is a wait event on the write I/O operation, the `io/redo_log_flush` wait event.

For examples of the behavior of this wait event, see io/aurora_redo_log_flush.

Actions

We recommend different actions depending on the causes of your wait event.

Identify the problematic sessions and queries

If your DB instance is experiencing a bottleneck, your first task is to find the sessions and queries that cause it. For a useful AWS Database Blog post, see [Analyze Amazon Aurora MySQL Workloads with Performance Insights](#).

To identify sessions and queries causing a bottleneck

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose your DB instance.
4. In **Database load**, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The queries at the top of the list are causing the highest load on the database.

Group your write operations

The following examples trigger the `io/redo_log_flush` wait event. (Autocommit is turned on.)

```
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
....
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');

UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;
....
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE id=xx;

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
```

```
....
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
```

To reduce the time spent waiting on the `io/redo_log_flush` wait event, group your write operations logically into a single commit to reduce persistent calls to storage.

Turn off autocommit

Turn off autocommit before making large changes that aren't within a transaction, as shown in the following example.

```
SET SESSION AUTOCOMMIT=OFF;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
....
UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1=xx;
-- Other DML statements here
COMMIT;

SET SESSION AUTOCOMMIT=ON;
```

Use transactions

You can use transactions, as shown in the following example.

```
BEGIN
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');
....
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES ('xxxx','xxxxx');

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;
....
DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1=xx;

-- Other DML statements here
END
```

Use batches

You can make changes in batches, as shown in the following example. However, using batches that are too large can cause performance issues, especially in read replicas or when doing point-in-time recovery (PITR).

```
INSERT INTO `sampleDB`.`sampleTable` (sampleCol2, sampleCol3) VALUES
('xxxx','xxxxx'),('xxxx','xxxxx'),...,('xxxx','xxxxx'),('xxxx','xxxxx');

UPDATE `sampleDB`.`sampleTable` SET sampleCol3='xxxxx' WHERE sampleCol1 BETWEEN xx AND
 xxx;

DELETE FROM `sampleDB`.`sampleTable` WHERE sampleCol1<xx;
```

io/socket/sql/client_connection

The `io/socket/sql/client_connection` event occurs when a thread is in the process of handling a new connection.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL versions 2 and 3

Context

The event `io/socket/sql/client_connection` indicates that mysqld is busy creating threads to handle incoming new client connections. In this scenario, the processing of servicing new client connection requests slows down while connections wait for the thread to be assigned. For more information, see MySQL server (mysqld).

Likely causes of increased waits

When this event appears more than normal, possibly indicating a performance problem, typical causes include the following:

- There is a sudden increase in new user connections from the application to your Amazon RDS instance.
- Your DB instance can't process new connections because the network, CPU, or memory is being throttled.

Actions

If `io/socket/sql/client_connection` dominates database activity, it doesn't necessarily indicate a performance problem. In a database that isn't idle, a wait event is always on top. Act only when performance degrades. We recommend different actions depending on the causes of your wait event.

Topics

Identify the problematic sessions and queries

If your DB instance is experiencing a bottleneck, your first task is to find the sessions and queries that cause it. For a useful blog post, see Analyze Amazon Aurora MySQL Workloads with Performance Insights.

To identify sessions and queries causing a bottleneck

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Performance Insights**.
3. Choose your DB instance.
4. In **Database load**, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The queries at the top of the list are causing the highest load on the database.

Follow best practices for connection management

To manage your connections, consider the following strategies:

- Use connection pooling.

  You can gradually increase the number of connections as required. For more information, see the whitepaper Amazon Aurora MySQL Database Administrator's Handbook.
- Use a reader node to redistribute read-only traffic.

  For more information, see Aurora Replicas and Amazon Aurora connection management.

Scale up your instance if resources are being throttled

Look for examples of throttling in the following resources:

- CPU

  Check your Amazon CloudWatch metrics for high CPU usage.
- Network

  Check for an increase in the value of the CloudWatch metrics `network receive throughput` and `network transmit throughput`. If your instance has reached the network bandwidth limit for your instance class, consider scaling up your RDS instance to a higher instance class type. For more information, see Aurora DB instance classes.
- Freeable memory

  Check for a drop in the CloudWatch metric `FreeableMemory`. Also, consider turning on Enhanced Monitoring. For more information, see Monitoring OS metrics with Enhanced Monitoring.

Check the top hosts and top users

Use Performance Insights to check the top hosts and top users. For more information, see Analyzing metrics with the Performance Insights dashboard.

Query the performance_schema tables

To get an accurate count of the current and total connections, query the `performance_schema` tables. With this technique, you identify the source user or host that is responsible for creating a high number of connections. For example, query the `performance_schema` tables as follows.

```
SELECT * FROM performance_schema.accounts;
SELECT * FROM performance_schema.users;
SELECT * FROM performance_schema.hosts;
```

Check the thread states of your queries

If your performance issue is ongoing, check the thread states of your queries. In the `mysql` client, issue the following command.

```
show processlist;
```

Audit your requests and queries

To check the nature of the requests and queries from user accounts, use AuroraAurora MySQL Advanced Auditing. To learn how to turn on auditing, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

Pool your database connections

Consider using Amazon RDS Proxy for connection management. By using RDS Proxy, you can allow your applications to pool and share database connections to improve their ability to scale. RDS Proxy makes applications more resilient to database failures by automatically connecting to a standby DB instance while preserving application connections. For more information, see Using Amazon RDS Proxy for Aurora.

io/table/sql/handler

The `io/table/sql/handler` event occurs when work has been delegated to a storage engine.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 3: 3.01.0 and 3.01.1
- Aurora MySQL version 2

Context

The event `io/table` indicates a wait for access to a table. This event occurs regardless of whether the data is cached in the buffer pool or accessed on disk. The `io/table/sql/handler` event indicates an increase in workload activity.

A *handler* is a routine specialized in a certain type of data or focused on certain special tasks. For example, an event handler receives and digests events and signals from the operating system or from a user interface. A memory handler performs tasks related to memory. A file input handler is a function that receives file input and performs special tasks on the data, according to context.

Views such as `performance_schema.events_waits_current` often show `io/table/sql/handler` when the actual wait is a nested wait event such as a lock. When the actual wait isn't `io/table/sql/handler`, Performance Insights reports the nested wait event. When Performance Insights reports `io/table/sql/handler`, it represents InnoDB processing of the I/O request and not a hidden nested wait event. For more information, see Performance Schema Atom and Molecule Events in the *MySQL Reference Manual*.

> **Note**
>
> However, in Aurora MySQL versions 3.01.0 and 3.01.1, synch/mutex/innodb/aurora_lock_thread_slot_futex is reported as `io/table/sql/handler`.

The `io/table/sql/handler` event often appears in top wait events with I/O waits such as `io/aurora_redo_log_flush` and `io/file/innodb/innodb_data_file`.

Likely causes of increased waits

In Performance Insights, sudden spikes in the `io/table/sql/handler` event indicate an increase in workload activity. Increased activity means increased I/O.

Performance Insights filters the nesting event IDs and doesn't report a `io/table/sql/handler` wait when the underlying nested event is a lock wait. For example, if the root cause event is synch/mutex/innodb/aurora_lock_thread_slot_futex, Performance Insights displays this wait in top wait events and not `io/table/sql/handler`.

In views such as `performance_schema.events_waits_current`, waits for `io/table/sql/handler` often appear when the actual wait is a nested wait event such as a lock. When the actual wait differs from `io/table/sql/handler`, Performance Insights looks up the nested wait and reports the actual wait instead of `io/table/sql/handler`. When Performance Insights reports `io/table/sql/handler`, the real wait is `io/table/sql/handler` and not a hidden nested wait event. For more information, see Performance Schema Atom and Molecule Events in the *MySQL 5.7 Reference Manual*.

Note

However, in Aurora MySQL versions 3.01.0 and 3.01.1, synch/mutex/innodb/aurora_lock_thread_slot_futex is reported as `io/table/sql/handler`.

Actions

If this wait event dominates database activity, it doesn't necessarily indicate a performance problem. A wait event is always on top when the database is active. You need to act only when performance degrades.

We recommend different actions depending on the other wait events that you see.

Topics

- Identify the sessions and queries causing the events
- Check for a correlation with Performance Insights counter metrics
- Check for other correlated wait events

Identify the sessions and queries causing the events

Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance is isn't optimal, then examine where the database is spending the most time. Look at the wait events that contribute to the highest load, and find out whether you can optimize the database and application to reduce those events.

To find SQL queries that are responsible for high load

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is shown for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Check for a correlation with Performance Insights counter metrics

Check for Performance Insights counter metrics such as `Innodb_rows_changed`. If counter metrics are correlated with `io/table/sql/handler`, follow these steps:

1. In Performance Insights, look for the SQL statements accounting for the `io/table/sql/handler` top wait event. If possible, optimize this statement so that it returns fewer rows.
2. Retrieve the top tables from the `schema_table_statistics` and `x$schema_table_statistics` views. These views show the amount of time spent per table. For more information, see The schema_table_statistics and x$schema_table_statistics Views in the *MySQL Reference Manual*.

   By default, rows are sorted by descending total wait time. Tables with the most contention appear first. The output indicates whether time is spent on reads, writes, fetches, inserts, updates, or deletes. The following example was run on an Aurora MySQL 2.09.1 instance.

```
mysql> select * from sys.schema_table_statistics limit 1\G

*************************** 1. row ***************************
     table_schema: read_only_db
       table_name: sbtest41
    total_latency: 54.11 m
     rows_fetched: 6001557
    fetch_latency: 39.14 m
    rows_inserted: 14833
   insert_latency: 5.78 m
     rows_updated: 30470
   update_latency: 5.39 m
     rows_deleted: 14833
   delete_latency: 3.81 m
 io_read_requests: NULL
          io_read: NULL
  io_read_latency: NULL
io_write_requests: NULL
         io_write: NULL
 io_write_latency: NULL
 io_misc_requests: NULL
  io_misc_latency: NULL
1 row in set (0.11 sec)
```

Check for other correlated wait events

If `synch/sxlock/innodb/btr_search_latch` and `io/table/sql/handler` contribute most to the DB load anomaly together, check whether the `innodb_adaptive_hash_index` variable is turned on. If it is, consider increasing the `innodb_adaptive_hash_index_parts` parameter value.

If the Adaptive Hash Index is turned off, consider turning it on. To learn more about the MySQL Adaptive Hash Index, see the following resources:

- The article Is Adaptive Hash Index in InnoDB right for my workload? on the Percona website
- Adaptive Hash Index in the *MySQL Reference Manual*
- The article Contention in MySQL InnoDB: Useful Info From the Semaphores Section on the Percona website

Note

The Adaptive Hash Index isn't supported on Aurora reader DB instances.

In some cases, performance might be poor on a reader instance when `synch/sxlock/innodb/btr_search_latch` and `io/table/sql/handler` are dominant. If so, consider redirecting the workload temporarily to the writer DB instance and turning on the Adaptive Hash Index.

synch/cond/innodb/row_lock_wait

The `synch/cond/innodb/row_lock_wait` event occurs when one session has locked a row for an update, and another session tries to update the same row. For more information, see InnoDB locking in the *MySQL Reference*.

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 3: 3.02.0, 3.02.1, 3.02.2

Likely causes of increased waits

Multiple data manipulation language (DML) statements are accessing the same row or rows simultaneously.

Actions

We recommend different actions depending on the other wait events that you see.

Topics

- Find and respond to the SQL statements responsible for this wait event
- Find and respond to the blocking session

Find and respond to the SQL statements responsible for this wait event

Use Performance Insights to identify the SQL statements responsible for this wait event. Consider the following strategies:

- If row locks are a persistent problem, consider rewriting the application to use optimistic locking.
- Use multirow statements.
- Spread the workload over different database objects. You can do this through partitioning.
- Check the value of the innodb_lock_wait_timeout parameter. It controls how long transactions wait before generating a timeout error.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Find and respond to the blocking session

Determine whether the blocking session is idle or active. Also, find out whether the session comes from an application or an active user.

To identify the session holding the lock, you can run SHOW ENGINE INNODB STATUS. The following example shows sample output.

```
mysql> SHOW ENGINE INNODB STATUS;

---TRANSACTION 1688153, ACTIVE 82 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 2 row lock(s)
MySQL thread id 4244, OS thread handle 70369524330224, query id 4020834 172.31.14.179
 reinvent executing
select id1 from test.t1 where id1=1 for update
------- TRX HAS BEEN WAITING 24 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 11 page no 4 n bits 72 index GEN_CLUST_INDEX of table test.t1 trx
 id 1688153 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
```

Or you can use the following query to extract details on current locks.

```
mysql> SELECT p1.id waiting_thread,
    p1.user waiting_user,
    p1.host waiting_host,
    it1.trx_query waiting_query,
    ilw.requesting_engine_transaction_id waiting_transaction,
    ilw.blocking_engine_lock_id blocking_lock,
    il.lock_mode blocking_mode,
    il.lock_type blocking_type,
    ilw.blocking_engine_transaction_id blocking_transaction,
    CASE it.trx_state
        WHEN 'LOCK WAIT'
        THEN it.trx_state
        ELSE p.state end blocker_state,
    concat(il.object_schema,'.', il.object_name) as locked_table,
    it.trx_mysql_thread_id blocker_thread,
    p.user blocker_user,
    p.host blocker_host
FROM performance_schema.data_lock_waits ilw
JOIN performance_schema.data_locks il
ON ilw.blocking_engine_lock_id = il.engine_lock_id
AND ilw.blocking_engine_transaction_id = il.engine_transaction_id
JOIN information_schema.innodb_trx it
ON ilw.blocking_engine_transaction_id = it.trx_id join information_schema.processlist p
ON it.trx_mysql_thread_id = p.id join information_schema.innodb_trx it1
ON ilw.requesting_engine_transaction_id = it1.trx_id join
 information_schema.processlist p1
ON it1.trx_mysql_thread_id = p1.id\G

*************************** 1. row ***************************
waiting_thread: 4244
waiting_user: reinvent
waiting_host: 123.456.789.012:18158
waiting_query: select id1 from test.t1 where id1=1 for update
waiting_transaction: 1688153
blocking_lock: 70369562074216:11:4:2:70369549808672
blocking_mode: X
blocking_type: RECORD
blocking_transaction: 1688142
blocker_state: User sleep
locked_table: test.t1
blocker_thread: 4243
blocker_user: reinvent
blocker_host: 123.456.789.012:18156
1 row in set (0.00 sec)
```

When you identify the session, your options include the following:

- Contact the application owner or the user.
- If the blocking session is idle, consider ending the blocking session. This action might trigger a long rollback. To learn how to end a session, see Ending a session or query.

For more information about identifying blocking transactions, see Using InnoDB Transaction and Locking Information in the *MySQL Reference Manual*.

synch/cond/innodb/row_lock_wait_cond

The `synch/cond/innodb/row_lock_wait_cond` event occurs when one session has locked a row for an update, and another session tries to update the same row. For more information, see InnoDB locking in the *MySQL Reference*.

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 2

Likely causes of increased waits

Multiple data manipulation language (DML) statements are accessing the same row or rows simultaneously.

Actions

We recommend different actions depending on the other wait events that you see.

Topics

- Find and respond to the SQL statements responsible for this wait event
- Find and respond to the blocking session

Find and respond to the SQL statements responsible for this wait event

Use Performance Insights to identify the SQL statements responsible for this wait event. Consider the following strategies:

- If row locks are a persistent problem, consider rewriting the application to use optimistic locking.
- Use multirow statements.
- Spread the workload over different database objects. You can do this through partitioning.
- Check the value of the `innodb_lock_wait_timeout` parameter. It controls how long transactions wait before generating a timeout error.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Find and respond to the blocking session

Determine whether the blocking session is idle or active. Also, find out whether the session comes from an application or an active user.

To identify the session holding the lock, you can run `SHOW ENGINE INNODB STATUS`. The following example shows sample output.

```
mysql> SHOW ENGINE INNODB STATUS;

---TRANSACTION 2771110, ACTIVE 112 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 1136, 1 row lock(s)
MySQL thread id 24, OS thread handle 70369573642160, query id 13271336 172.31.14.179
 reinvent Sending data
select id1 from test.t1 where id1=1 for update
------- TRX HAS BEEN WAITING 43 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 11 page no 3 n bits 0 index GEN_CLUST_INDEX of table test.t1 trx
 id 2771110 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 5; compact format; info bits 0
```

Or you can use the following query to extract details on current locks.

```
mysql> SELECT p1.id waiting_thread,
              p1.user waiting_user,
              p1.host waiting_host,
              it1.trx_query waiting_query,
              ilw.requesting_trx_id waiting_transaction,
              ilw.blocking_lock_id blocking_lock,
              il.lock_mode blocking_mode,
              il.lock_type blocking_type,
              ilw.blocking_trx_id blocking_transaction,
              CASE it.trx_state
                WHEN 'LOCK WAIT'
                THEN it.trx_state
                ELSE p.state
              END blocker_state,
              il.lock_table locked_table,
              it.trx_mysql_thread_id blocker_thread,
              p.user blocker_user,
              p.host blocker_host
       FROM information_schema.innodb_lock_waits ilw
       JOIN information_schema.innodb_locks il
         ON ilw.blocking_lock_id = il.lock_id
        AND ilw.blocking_trx_id = il.lock_trx_id
       JOIN information_schema.innodb_trx it
         ON ilw.blocking_trx_id = it.trx_id
       JOIN information_schema.processlist p
         ON it.trx_mysql_thread_id = p.id
       JOIN information_schema.innodb_trx it1
         ON ilw.requesting_trx_id = it1.trx_id
       JOIN information_schema.processlist p1
         ON it1.trx_mysql_thread_id = p1.id\G

*************************** 1. row ***************************
      waiting_thread: 3561959471
        waiting_user: reinvent
        waiting_host: 123.456.789.012:20485
       waiting_query: select id1 from test.t1 where id1=1 for update
 waiting_transaction: 312337314
       blocking_lock: 312337287:261:3:2
       blocking_mode: X
       blocking_type: RECORD
blocking_transaction: 312337287
       blocker_state: User sleep
        locked_table: `test`.`t1`
      blocker_thread: 3561223876
        blocker_user: reinvent
        blocker_host: 123.456.789.012:17746
1 row in set (0.04 sec)
```

When you identify the session, your options include the following:

- Contact the application owner or the user.
- If the blocking session is idle, consider ending the blocking session. This action might trigger a long rollback. To learn how to end a session, see Ending a session or query.

For more information about identifying blocking transactions, see Using InnoDB Transaction and Locking Information in the *MySQL Reference Manual*.

synch/cond/sql/MDL_context::COND_wait_status

The `synch/cond/sql/MDL_context::COND_wait_status` event occurs when there are threads waiting on a table metadata lock.

Topics

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL versions 2 and 3

Context

The event `synch/cond/sql/MDL_context::COND_wait_status` indicates that there are threads waiting on a table metadata lock. In some cases, one session holds a metadata lock on a table and another session tries to get the same lock on the same table. In such a case, the second session waits on the `synch/cond/sql/MDL_context::COND_wait_status` wait event.

MySQL uses metadata locking to manage concurrent access to database objects and to ensure data consistency. Metadata locking applies to tables, schemas, scheduled events, tablespaces, and user locks acquired with the `get_lock` function, and stored programs. Stored programs include procedures, functions, and triggers. For more information, see Metadata locking in the MySQL documentation.

The MySQL process list shows this session in the state `waiting for metadata lock`. In Performance Insights, if `Performance_schema` is turned on, the event `synch/cond/sql/MDL_context::COND_wait_status` appears.

The default timeout for a query waiting on a metadata lock is based on the value of the `lock_wait_timeout` parameter, which defaults to 31,536,000 seconds (365 days).

For more details on different InnoDB locks and the types of locks that can cause conflicts, see InnoDB Locking in the MySQL documentation.

Likely causes of increased waits

When the `synch/cond/sql/MDL_context::COND_wait_status` event appears more than normal, possibly indicating a performance problem, typical causes include the following:

Long-running transactions

One or more transactions are modifying a large amount of data and holding locks on tables for a very long time.

Idle transactions

One or more transactions remain open for a long time, without being committed or rolled back.

DDL statements on large tables

One or more data definition statements (DDL) statements, such as `ALTER TABLE` commands, were run on very large tables.

Explicit table locks

There are explicit locks on tables that aren't being released in a timely manner. For example, an application might run `LOCK TABLE` statements improperly.

Actions

We recommend different actions depending on the causes of your wait event and on the version of the Aurora MySQL DB cluster.

Topics

- Identify the sessions and queries causing the events
- Check for past events
- Run queries on Aurora MySQL version 2
- Respond to the blocking session

Identify the sessions and queries causing the events

You can use Performance Insights to show queries blocked by the `synch/cond/sql/MDL_context::COND_wait_status` wait event. However, to identify the blocking session, query metadata tables from `performance_schema` and `information_schema` on the DB cluster.

Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance isn't optimal, then examine where the database is spending the most time. Look at the wait events that contribute to the highest load, and find out whether you can optimize the database and application to reduce those events.

To find SQL queries that are responsible for high load

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard for that DB instance appears.
4. In the **Database load** chart, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the AWS Database Blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Check for past events

You can gain insight into this wait event to check for past occurrences of it. To do so, complete the following actions:

- Check the data manipulation language (DML) and DDL throughput and latency to see if there were any changes in workload.

  You can use Performance Insights to find queries waiting on this event at the time of the issue. Also, you can view the digest of the queries run near the time of issue.
- If audit logs or general logs are turned on for the DB cluster, you can check for all queries run on the objects (schema.table) involved in the waiting transaction. You can also check for the queries that completed running before the transaction.

The information available to troubleshoot past events is limited. Performing these checks doesn't show which object is waiting for information. However, you can identify tables with heavy load at the time of the event and the set of frequently operated rows causing conflict at the time of issue.

You can then use this information to reproduce the issue in a test environment and provide insights about its cause.

Run queries on Aurora MySQL version 2

In Aurora MySQL version 2, you can identify the blocked session directly by querying `performance_schema` tables or `sys` schema views. An example can illustrate how to query tables to identify blocking queries and sessions.

In the following process list output, the connection ID 89 is waiting on a metadata lock, and it's running a `TRUNCATE TABLE` command. In a query on the `performance_schema` tables or `sys` schema views, the output shows that the blocking session is 76.

```
MySQL [(none)]> select @@version, @@aurora_version;
+-----------+------------------+
| @@version | @@aurora_version |
+-----------+------------------+
| 5.7.12    | 2.09.0           |
+-----------+------------------+
1 row in set (0.01 sec)

MySQL [(none)]> show processlist;
+----+-----------------+--------------------+-----------+---------+------
+---------------------------------+-------------------------------+
| Id | User            | Host               | db        | Command | Time | State
                    | Info                          |
+----+-----------------+--------------------+-----------+---------+------
+---------------------------------+-------------------------------+
|  2 | rdsadmin        | localhost          | NULL      | Sleep   |    0 | NULL
                    | NULL                          |
|  4 | rdsadmin        | localhost          | NULL      | Sleep   |    2 | NULL
                    | NULL                          |
|  5 | rdsadmin        | localhost          | NULL      | Sleep   |    1 | NULL
                    | NULL                          |
| 20 | rdsadmin        | localhost          | NULL      | Sleep   |    0 | NULL
                    | NULL                          |
| 21 | rdsadmin        | localhost          | NULL      | Sleep   |  261 | NULL
                    | NULL                          |
| 66 | auroramysql5712 | 172.31.21.51:52154 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                          |
| 67 | auroramysql5712 | 172.31.21.51:52158 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                          |
| 68 | auroramysql5712 | 172.31.21.51:52150 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 69 | auroramysql5712 | 172.31.21.51:52162 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 70 | auroramysql5712 | 172.31.21.51:52160 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 71 | auroramysql5712 | 172.31.21.51:52152 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 72 | auroramysql5712 | 172.31.21.51:52156 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 73 | auroramysql5712 | 172.31.21.51:52164 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 74 | auroramysql5712 | 172.31.21.51:52166 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 75 | auroramysql5712 | 172.31.21.51:52168 | sbtest123 | Sleep   |    0 | NULL
                    | NULL                            |
| 76 | auroramysql5712 | 172.31.21.51:52170 | NULL      | Query   |    0 | starting
                    | show processlist                |
| 88 | auroramysql5712 | 172.31.21.51:52194 | NULL      | Query   |   22 | User sleep
                    | select sleep(10000)             |
| 89 | auroramysql5712 | 172.31.21.51:52196 | NULL      | Query   |    5 | Waiting for
 table metadata lock | truncate table sbtest.sbtest1 |
+----+-----------------+--------------------+-----------+---------+------
+---------------------------------+-------------------------------+
18 rows in set (0.00 sec)
```

Next, a query on the `performance_schema` tables or `sys` schema views shows that the blocking session is 76.

```
MySQL [(none)]> select * from sys.schema_table_lock_waits;

+---------------+-------------+-------------------+-------------
+------------------------------+-------------------+-----------------------
+-------------------------------+--------------------+-----------------------------
+-----------------------------+--------------------+--------------
+------------------------------+--------------------+------------------------
+-------------------------+-------------------------------+
| object_schema | object_name | waiting_thread_id | waiting_pid | waiting_account
           | waiting_lock_type | waiting_lock_duration | waiting_query
   | waiting_query_secs | waiting_query_rows_affected | waiting_query_rows_examined |
 blocking_thread_id | blocking_pid | blocking_account             | blocking_lock_type
 | blocking_lock_duration | sql_kill_blocking_query | sql_kill_blocking_connection |
+---------------+-------------+-------------------+-------------
+-----------------------------+-------------------+----------------------
+------------------------------+--------------------+-----------------------------
+----------------------------+-------------------+-------------
+-----------------------------+--------------------+------------------------
+-------------------------+-----------------------------+
| sbtest        | sbtest1     |               121 |          89 |
 auroramysql5712@192.0.2.0    | EXCLUSIVE         | TRANSACTION           | truncate
 table sbtest.sbtest1 |                 10 |                           0 |
               0 |                 108 |          76 | auroramysql5712@192.0.2.0    |
 SHARED_READ        | TRANSACTION            | KILL QUERY 76           | KILL 76
               |
+---------------+-------------+-------------------+-------------
+-----------------------------+-------------------+----------------------
+------------------------------+--------------------+-----------------------------
+----------------------------+-------------------+-------------
+-----------------------------+--------------------+------------------------
+-------------------------+-----------------------------+
1 row in set (0.00 sec)
```

Respond to the blocking session

When you identify the session, your options include the following:

- Contact the application owner or the user.
- If the blocking session is idle, consider ending the blocking session. This action might trigger a long rollback. To learn how to end a session, see Ending a session or query.

For more information about identifying blocking transactions, see Using InnoDB Transaction and Locking Information in the MySQL documentation.

synch/mutex/innodb/aurora_lock_thread_slot_futex

The `synch/mutex/innodb/aurora_lock_thread_slot_futex` event occurs when one session has locked a row for an update, and another session tries to update the same row. For more information, see InnoDB locking in the *MySQL Reference*.

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 2

> **Note**
>
> In Aurora MySQL versions 3.01.0 and 3.01.1, this wait event is reported as io/table/sql/handler.

Likely causes of increased waits

Multiple data manipulation language (DML) statements are accessing the same row or rows simultaneously.

Actions

We recommend different actions depending on the other wait events that you see.

Topics

- Find and respond to the SQL statements responsible for this wait event
- Find and respond to the blocking session

Find and respond to the SQL statements responsible for this wait event

Use Performance Insights to identify the SQL statements responsible for this wait event. Consider the following strategies:

- If row locks are a persistent problem, consider rewriting the application to use optimistic locking.
- Use multirow statements.
- Spread the workload over different database objects. You can do this through partitioning.
- Check the value of the `innodb_lock_wait_timeout` parameter. It controls how long transactions wait before generating a timeout error.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Find and respond to the blocking session

Determine whether the blocking session is idle or active. Also, find out whether the session comes from an application or an active user.

To identify the session holding the lock, you can run `SHOW ENGINE INNODB STATUS`. The following example shows sample output.

```
mysql> SHOW ENGINE INNODB STATUS;

---------------------TRANSACTION 302631452, ACTIVE 2 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 376, 1 row lock(s)
MySQL thread id 80109, OS thread handle 0x2ae915060700, query id 938819 10.0.4.12
 reinvent updating
UPDATE sbtest1 SET k=k+1 WHERE id=503
------- TRX HAS BEEN WAITING 2 SEC FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 148 page no 11 n bits 30 index `PRIMARY` of table
 `sysbench2`.`sbtest1` trx id 302631452 lock_mode X locks rec but not gap waiting
Record lock, heap no 30 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
```

Or you can use the following query to extract details on current locks.

```
mysql> SELECT p1.id waiting_thread,
              p1.user waiting_user,
              p1.host waiting_host,
              it1.trx_query waiting_query,
              ilw.requesting_trx_id waiting_transaction,
              ilw.blocking_lock_id blocking_lock,
              il.lock_mode blocking_mode,
              il.lock_type blocking_type,
              ilw.blocking_trx_id blocking_transaction,
              CASE it.trx_state
                WHEN 'LOCK WAIT'
                THEN it.trx_state
                ELSE p.state
              END blocker_state,
              il.lock_table locked_table,
              it.trx_mysql_thread_id blocker_thread,
              p.user blocker_user,
              p.host blocker_host
       FROM information_schema.innodb_lock_waits ilw
       JOIN information_schema.innodb_locks il
```

```
         ON ilw.blocking_lock_id = il.lock_id
        AND ilw.blocking_trx_id = il.lock_trx_id
       JOIN information_schema.innodb_trx it
         ON ilw.blocking_trx_id = it.trx_id
       JOIN information_schema.processlist p
         ON it.trx_mysql_thread_id = p.id
       JOIN information_schema.innodb_trx it1
         ON ilw.requesting_trx_id = it1.trx_id
       JOIN information_schema.processlist p1
         ON it1.trx_mysql_thread_id = p1.id\G

*************************** 1. row ***************************
      waiting_thread: 3561959471
        waiting_user: reinvent
        waiting_host: 123.456.789.012:20485
       waiting_query: select id1 from test.t1 where id1=1 for update
 waiting_transaction: 312337314
       blocking_lock: 312337287:261:3:2
       blocking_mode: X
       blocking_type: RECORD
blocking_transaction: 312337287
       blocker_state: User sleep
        locked_table: `test`.`t1`
      blocker_thread: 3561223876
        blocker_user: reinvent
        blocker_host: 123.456.789.012:17746
1 row in set (0.04 sec)
```

When you identify the session, your options include the following:

- Contact the application owner or the user.
- If the blocking session is idle, consider ending the blocking session. This action might trigger a long rollback. To learn how to end a session, see Ending a session or query.

For more information about identifying blocking transactions, see Using InnoDB Transaction and Locking Information in the *MySQL Reference Manual*.

synch/mutex/innodb/buf_pool_mutex

The `synch/mutex/innodb/buf_pool_mutex` event occurs when a thread has acquired a lock on the InnoDB buffer pool to access a page in memory.

Topics

Relevant engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL version 2

Context

The `buf_pool` mutex is a single mutex that protects the control data structures of the buffer pool.

For more information, see Monitoring InnoDB Mutex Waits Using Performance Schema in the MySQL documentation.

Likely causes of increased waits

This is a workload-specific wait event. Common causes for `synch/mutex/innodb/buf_pool_mutex` to appear among the top wait events include the following:

- The buffer pool size isn't large enough to hold the working set of data.
- The workload is more specific to certain pages from a specific table in the database, leading to contention in the buffer pool.

Actions

We recommend different actions depending on the causes of your wait event.

Identify the sessions and queries causing the events

Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance isn't optimal, then examine where the database is spending the most time. Look at the wait events that contribute to the highest load, and find out whether you can optimize the database and application to reduce those events.

To view the Top SQL chart in the AWS Management Console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is shown for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. Underneath the **Database load** chart, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Use Performance Insights

This event is related to workload. You can use Performance Insights to do the following:

- Identify when wait events start, and whether there's any change in the workload around that time from the application logs or related sources.
- Identify the SQL statements responsible for this wait event. Examine the execution plan of the queries to make sure that these queries are optimized and using appropriate indexes.

  If the top queries responsible for the wait event are related to the same database object or table, then consider partitioning that object or table.

Create Aurora Replicas

You can create Aurora Replicas to serve read-only traffic. You can also use Aurora Auto Scaling to handle surges in read traffic. Make sure to run scheduled read-only tasks and logical backups on Aurora Replicas.

For more information, see Using Amazon Aurora Auto Scaling with Aurora Replicas.

Examine the buffer pool size

Check whether the buffer pool size is sufficient for the workload by looking at the metric `innodb_buffer_pool_wait_free`. If the value of this metric is high and increasing continuously, that indicates that the size of the buffer pool isn't sufficient to handle the workload. If `innodb_buffer_pool_size` has been set properly, the value of `innodb_buffer_pool_wait_free` should be small. For more information, see Innodb_buffer_pool_wait_free in the MySQL documentation.

Increase the buffer pool size if the DB instance has enough memory for session buffers and operating-system tasks. If it doesn't, change the DB instance to a larger DB instance class to get additional memory that can be allocated to the buffer pool.

> **Note**
>
> Aurora MySQL automatically adjusts the value of `innodb_buffer_pool_instances` based on the configured `innodb_buffer_pool_size`.

Monitor the global status history

By monitoring the change rates of status variables, you can detect locking or memory issues on your DB instance. Turn on Global Status History (GoSH) if it isn't already turned on. For more information on GoSH, see Managing the global status history.

You can also create custom Amazon CloudWatch metrics to monitor status variables. For more information, see Publishing custom metrics.

synch/mutex/innodb/fil_system_mutex

The `synch/mutex/innodb/fil_system_mutex` event occurs when a session is waiting to access the tablespace memory cache.

Topics

Supported engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL versions 2 and 3

Context

InnoDB uses tablespaces to manage the storage area for tables and log files. The *tablespace memory cache* is a global memory structure that maintains information about tablespaces. MySQL uses `synch/mutex/innodb/fil_system_mutex` waits to control concurrent access to the tablespace memory cache.

The event `synch/mutex/innodb/fil_system_mutex` indicates that there is currently more than one operation that needs to retrieve and manipulate information in the tablespace memory cache for the same tablespace.

Likely causes of increased waits

When the `synch/mutex/innodb/fil_system_mutex` event appears more than normal, possibly indicating a performance problem, this typically occurs when all of the following conditions are present:

- An increase in concurrent data manipulation language (DML) operations that update or delete data in the same table.
- The tablespace for this table is very large and has a lot of data pages.
- The fill factor for these data pages is low.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Identify the sessions and queries causing the events
- Reorganize large tables during off-peak hours

Identify the sessions and queries causing the events

Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance isn't optimal, examine where the database is spending the most time. Look at the wait events that contribute to the highest load, and find out whether you can optimize the database and application to reduce those events.

To find SQL queries that are responsible for high load

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard appears for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Another way to find out which queries are causing high numbers of synch/mutex/innodb/fil_system_mutex waits is to check performance_schema, as in the following example.

```
mysql> select * from performance_schema.events_waits_current where EVENT_NAME='wait/
synch/mutex/innodb/fil_system_mutex'\G
*************************** 1. row ***************************
            THREAD_ID: 19
             EVENT_ID: 195057
         END_EVENT_ID: 195057
           EVENT_NAME: wait/synch/mutex/innodb/fil_system_mutex
               SOURCE: fil0fil.cc:6700
          TIMER_START: 1010146190118400
            TIMER_END: 1010146196524000
           TIMER_WAIT: 6405600
                SPINS: NULL
        OBJECT_SCHEMA: NULL
          OBJECT_NAME: NULL
           INDEX_NAME: NULL
          OBJECT_TYPE: NULL
OBJECT_INSTANCE_BEGIN: 47285552262176
     NESTING_EVENT_ID: NULL
   NESTING_EVENT_TYPE: NULL
            OPERATION: lock
      NUMBER_OF_BYTES: NULL
                FLAGS: NULL
```

```
*************************** 2. row ***************************
            THREAD_ID: 23
             EVENT_ID: 5480
         END_EVENT_ID: 5480
           EVENT_NAME: wait/synch/mutex/innodb/fil_system_mutex
               SOURCE: fil0fil.cc:5906
          TIMER_START: 995269979908800
            TIMER_END: 995269980159200
           TIMER_WAIT: 250400
                SPINS: NULL
        OBJECT_SCHEMA: NULL
          OBJECT_NAME: NULL
           INDEX_NAME: NULL
          OBJECT_TYPE: NULL
OBJECT_INSTANCE_BEGIN: 47285552262176
     NESTING_EVENT_ID: NULL
   NESTING_EVENT_TYPE: NULL
            OPERATION: lock
      NUMBER_OF_BYTES: NULL
                FLAGS: NULL
*************************** 3. row ***************************
            THREAD_ID: 55
             EVENT_ID: 23233794
         END_EVENT_ID: NULL
           EVENT_NAME: wait/synch/mutex/innodb/fil_system_mutex
               SOURCE: fil0fil.cc:449
          TIMER_START: 1010492125341600
            TIMER_END: 1010494304900000
           TIMER_WAIT: 2179558400
                SPINS: NULL
        OBJECT_SCHEMA: NULL
          OBJECT_NAME: NULL
           INDEX_NAME: NULL
          OBJECT_TYPE: NULL
OBJECT_INSTANCE_BEGIN: 47285552262176
     NESTING_EVENT_ID: 23233786
   NESTING_EVENT_TYPE: WAIT
            OPERATION: lock
      NUMBER_OF_BYTES: NULL
                FLAGS: NULL
```

Reorganize large tables during off-peak hours

Reorganize large tables that you identify as the source of high numbers of `synch/mutex/innodb/fil_system_mutex` wait events during a maintenance window outside of production hours. Doing so ensures that the internal tablespaces map cleanup doesn't occur when quick access to the table is critical. For information about reorganizing tables, see OPTIMIZE TABLE Statement in the *MySQL Reference*.

synch/mutex/innodb/trx_sys_mutex

The `synch/mutex/innodb/trx_sys_mutex` event occurs when there is high database activity with a large number of transactions.

Topics

Relevant engine versions

This wait event information is supported for the following engine versions:

- Aurora MySQL versions 2 and 3

Context

Internally, the InnoDB database engine uses the repeatable read isolation level with snapshots to provide read consistency. This gives you a point-in-time view of the database at the time the snapshot was created.

In InnoDB, all changes are applied to the database as soon as they arrive, regardless of whether they're committed. This approach means that without multiversion concurrency control (MVCC), all users connected to the database see all of the changes and the latest rows. Therefore, InnoDB requires a way to track the changes to understand what to roll back when necessary.

To do this, InnoDB uses a transaction system (`trx_sys`) to track snapshots. The transaction system does the following:

- Tracks the transaction ID for each row in the undo logs.
- Uses an internal InnoDB structure called `ReadView` that helps to identify which transaction IDs are visible for a snapshot.

Likely causes of increased waits

Any database operation that requires the consistent and controlled handling (creating, reading, updating, and deleting) of transactions IDs generates a call from `trx_sys` to the mutex.

These calls happen inside three functions:

- `trx_sys_mutex_enter` – Creates the mutex.
- `trx_sys_mutex_exit` – Releases the mutex.
- `trx_sys_mutex_own` – Tests whether the mutex is owned.

The InnoDB Performance Schema instrumentation tracks all `trx_sys` mutex calls. Tracking includes, but isn't limited to, management of `trx_sys` on database startup or shutdown, rollback operations, undo cleanups, row read access, and buffer pool loads. High database activity with a large number of transactions results in `synch/mutex/innodb/trx_sys_mutex` appearing among the top wait events.

For more information, see [Monitoring InnoDB Mutex Waits Using Performance Schema](#) in the MySQL documentation.

Actions

We recommend different actions depending on the causes of your wait event.

Identify the sessions and queries causing the events

Typically, databases with moderate to significant load have wait events. The wait events might be acceptable if performance is optimal. If performance isn't optimal, then examine where the database is spending the most time. Look at the wait events that contribute to the highest load. Find out whether you can optimize the database and application to reduce those events.

To view the Top SQL chart in the AWS Management Console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is shown for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. Under the **Database load** chart, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Examine other wait events

Examine the other wait events associated with the `synch/mutex/innodb/trx_sys_mutex` wait event. Doing this can provide more information about the nature of the workload. A large number of transactions might reduce throughput, but the workload might also make this necessary.

For more information on how to optimize transactions, see Optimizing InnoDB Transaction Management in the MySQL documentation.

synch/sxlock/innodb/hash_table_locks

The `synch/sxlock/innodb/hash_table_locks` event occurs when pages not found in the buffer pool must be read from storage.

Topics

Supported engine versions

This wait event information is supported for the following versions:

- Aurora MySQL versions 2 and 3

Context

The event `synch/sxlock/innodb/hash_table_locks` indicates that a workload is frequently accessing data that isn't stored in the buffer pool. This wait event is associated with new page additions and old data evictions from the buffer pool. The data stored in the buffer pool aged and new data must be cached, so the aged pages are evicted to allow caching of the new pages. MySQL uses a least recently used (LRU) algorithm to evict pages from the buffer pool. The workload is trying to access data that hasn't been loaded into the buffer pool or data that has been evicted from the buffer pool.

This wait event occurs when the workload must access the data in files on disk or when blocks are freed from or added to the buffer pool's LRU list. These operations wait to obtain a shared excluded lock (SX-lock). This SX-lock is used for the synchronization over the *hash table*, which is a table in memory designed to improve buffer pool access performance.

For more information, see Buffer Pool in the MySQL documentation.

Likely causes of increased waits

When the `synch/sxlock/innodb/hash_table_locks` wait event appears more than normal, possibly indicating a performance problem, typical causes include the following:

An undersized buffer pool

The size of the buffer pool is too small to keep all of the frequently accessed pages in memory.

Heavy workload

The workload is causing frequent evictions and data pages reloads in the buffer cache.

Errors reading the pages

There are errors reading pages in the buffer pool, which might indicate data corruption.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Increase the size of the buffer pool
- Improve data access patterns

- Reduce or avoid full-table scans
- Check the error logs for page corruption

Increase the size of the buffer pool

Make sure that the buffer pool is appropriately sized for the workload. To do so, you can check the buffer pool cache hit rate. Typically, if the value drops below 95 percent, consider increasing the buffer pool size. A larger buffer pool can keep frequently accessed pages in memory longer. To increase the size of the buffer pool, modify the value of the `innodb_buffer_pool_size` parameter. The default value of this parameter is based on the DB instance class size. For more information, see Best practices for Amazon Aurora MySQL database configuration.

Improve data access patterns

Check the queries affected by this wait and their execution plans. Consider improving data access patterns. For example, if you are using mysqli_result::fetch_array, you can try increasing the array fetch size.

You can use Performance Insights to show queries and sessions that might be causing the `synch/sxlock/innodb/hash_table_locks` wait event.

To find SQL queries that are responsible for high load

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is shown for that DB instance.
4. In the **Database load** chart, choose **Slice by wait**.
5. At the bottom of the page, choose **Top SQL**.

   The chart lists the SQL queries that are responsible for the load. Those at the top of the list are most responsible. To resolve a bottleneck, focus on these statements.

For a useful overview of troubleshooting using Performance Insights, see the AWS Database Blog post Analyze Amazon Aurora MySQL Workloads with Performance Insights.

Reduce or avoid full-table scans

Monitor your workload to see if it's running full-table scans, and, if it is, reduce or avoid them. For example, you can monitor status variables such as `Handler_read_rnd_next`. For more information, see Server Status Variables in the MySQL documentation.

Check the error logs for page corruption

You can check the mysql-error.log for corruption-related messages that were detected near the time of the issue. Messages that you can work with to resolve the issue are in the error log. You might need to recreate objects that were reported as corrupted.

Tuning Aurora MySQL with thread states

The following table summarizes the most common general thread states for Aurora MySQL.

General thread state	Description
???	This thread state indicates that a thread is processing a SELECT statement that requires the use of an internal temporary table to sort the data.
???	This thread state indicates that a thread is reading and filtering rows for a query to determine the correct result set.

creating sort index

The `creating sort index` thread state indicates that a thread is processing a `SELECT` statement that requires the use of an internal temporary table to sort the data.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This thread state information is supported for the following versions:

- Aurora MySQL version 2 up to 2.09.2

Context

The `creating sort index` state appears when a query with an `ORDER BY` or `GROUP BY` clause can't use an existing index to perform the operation. In this case, MySQL needs to perform a more expensive `filesort` operation. This operation is typically performed in memory if the result set isn't too large. Otherwise, it involves creating a file on disk.

Likely causes of increased waits

The appearance of `creating sort index` doesn't by itself indicate a problem. If performance is poor, and you see frequent instances of `creating sort index`, the most likely cause is slow queries with `ORDER BY` or `GROUP BY` operators.

Actions

The general guideline is to find queries with `ORDER BY` or `GROUP BY` clauses that are associated with the increases in the `creating sort index` state. Then see whether adding an index or increasing the sort buffer size solves the problem.

Topics

- Turn on the Performance Schema if it isn't turned on
- Identify the problem queries
- Examine the explain plans for filesort usage
- Increase the sort buffer size

Turn on the Performance Schema if it isn't turned on

Performance Insights reports thread states only if Performance Schema instruments aren't turned on. When Performance Schema instruments are turned on, Performance Insights reports wait events instead. Performance Schema instruments provide additional insights and better tools when you investigate potential performance problems. Therefore, we recommend that you turn on the Performance Schema. For more information, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

Identify the problem queries

To identify current queries that are causing increases in the `creating sort index` state, run `show processlist` and see if any of the queries have `ORDER BY` or `GROUP BY`. Optionally, run `explain for connection N`, where `N` is the process list ID of the query with `filesort`.

To identify past queries that are causing these increases, turn on the slow query log and find the queries with `ORDER BY`. Run `EXPLAIN` on the slow queries and look for "using filesort." For more information, see Examine the explain plans for filesort usage.

Examine the explain plans for filesort usage

Identify the statements with `ORDER BY` or `GROUP BY` clauses that result in the `creating sort index` state.

The following example shows how to run `explain` on a query. The `Extra` column shows that this query uses `filesort`.

```
mysql> explain select * from mytable order by c1 limit 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: mytable
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2064548
     filtered: 100.00
        Extra: Using filesort
1 row in set, 1 warning (0.01 sec)
```

The following example shows the result of running `EXPLAIN` on the same query after an index is created on column `c1`.

```
mysql> alter table mytable add index (c1);
```

```
mysql> explain select * from mytable order by c1 limit 10\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: mytable
   partitions: NULL
         type: index
possible_keys: NULL
          key: c1
      key_len: 1023
          ref: NULL
         rows: 10
```

```
     filtered: 100.00
        Extra: Using index
1 row in set, 1 warning (0.01 sec)
```

For information on using indexes for sort order optimization, see ORDER BY Optimization in the MySQL documentation.

Increase the sort buffer size

To see whether a specific query required a `filesort` process that created a file on disk, check the `sort_merge_passes` variable value after running the query. The following shows an example.

```
mysql> show session status like 'sort_merge_passes';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Sort_merge_passes | 0     |
+-------------------+-------+
1 row in set (0.01 sec)

--- run query
mysql> select * from mytable order by u limit 10;
--- run status again:

mysql> show session status like 'sort_merge_passes';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Sort_merge_passes | 0     |
+-------------------+-------+
1 row in set (0.01 sec)
```

If the value of `sort_merge_passes` is high, consider increasing the sort buffer size. Apply the increase at the session level, because increasing it globally can significantly increase the amount of RAM MySQL uses. The following example shows how to change the sort buffer size before running a query.

```
mysql> set session sort_buffer_size=10*1024*1024;
Query OK, 0 rows affected (0.00 sec)
-- run query
```

sending data

The `sending data` thread state indicates that a thread is reading and filtering rows for a query to determine the correct result set. The name is misleading because it implies the state is transferring data, not collecting and preparing data to be sent later.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This thread state information is supported for the following versions:

- Aurora MySQL version 2 up to 2.09.2

Context

Many thread states are short-lasting. Operations occurring during `sending data` tend to perform large numbers of disk or cache reads. Therefore, `sending data` is often the longest-running state over the lifetime of a given query. This state appears when Aurora MySQL is doing the following:

- Reading and processing rows for a `SELECT` statement
- Performing a large number of reads from either disk or memory
- Completing a full read of all data from a specific query
- Reading data from a table, an index, or the work of a stored procedure
- Sorting, grouping, or ordering data

After the `sending data` state finishes preparing the data, the thread state `writing to net` indicates the return of data to the client. Typically, `writing to net` is captured only when the result set is very large or severe network latency is slowing the transfer.

Likely causes of increased waits

The appearance of `sending data` doesn't by itself indicate a problem. If performance is poor, and you see frequent instances of `sending data`, the most likely causes are as follows.

Topics

- Inefficient query
- Suboptimal server configuration

Inefficient query

In most cases, what's responsible for this state is a query that isn't using an appropriate index to find the result set of a specific query. For example, consider a query reading a 10 million record table for all orders placed in California, where the state column isn't indexed or is poorly indexed. In the latter case, the index might exist, but the optimizer ignores it because of low cardinality.

Suboptimal server configuration

If several queries appear in the `sending data` state, the database server might be configured poorly. Specifically, the server might have the following issues:

- The database server doesn't have enough computing capacity: disk I/O, disk type and speed, CPU, or number of CPUs.
- The server is starved for allocated resources, such as the InnoDB buffer pool for InnoDB tables or the key buffer for MyIsam tables.
- Per-thread memory settings such as `sort_buffer`, `read_buffer`, and `join_buffer` consume more RAM than required, starving the physical server for memory resources.

Actions

The general guideline is to find queries that return large numbers of rows by checking the Performance Schema. If logging queries that don't use indexes is turned on, you can also examine the results from the slow logs.

Topics

- Turn on the Performance Schema if it isn't turned on
- Examine memory settings

- Examine the explain plans for index usage
- Check the volume of data returned
- Check for concurrency issues
- Check the structure of your queries

Turn on the Performance Schema if it isn't turned on

Performance Insights reports thread states only if Performance Schema instruments aren't turned on. When Performance Schema instruments are turned on, Performance Insights reports wait events instead. Performance Schema instruments provide additional insights and better tools when you investigate potential performance problems. Therefore, we recommend that you turn on the Performance Schema. For more information, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

Examine memory settings

Examine the memory settings for the primary buffer pools. Make sure that these pools are appropriately sized for the workload. If your database uses multiple buffer pool instances, make sure that they aren't divided into many small buffer pools. Threads can only use one buffer pool at a time.

Make sure that the following memory settings used for each thread are properly sized:

- read_buffer
- read_rnd_buffer
- sort_buffer
- join_buffer
- binlog_cache

Unless you have a specific reason to modify the settings, use the default values.

Examine the explain plans for index usage

For queries in the `sending data` thread state, examine the plan to determine whether appropriate indexes are used. If a query isn't using a useful index, consider adding hints like `USE INDEX` or `FORCE INDEX`. Hints can greatly increase or decrease the time it takes to run a query, so use care before adding them.

Check the volume of data returned

Check the tables that are being queried and the amount of data that they contain. Can any of this data be archived? In many cases, the cause of poor query execution times isn't the result of the query plan, but the volume of data to be processed. Many developers are very efficient in adding data to a database but seldom consider dataset life cycle in the design and development phases.

Look for queries that perform well in low-volume databases but perform poorly in your current system. Sometimes developers who design specific queries might not realize that these queries are returning 350,000 rows. The developers might have developed the queries in a lower-volume environment with smaller datasets than production environments have.

Check for concurrency issues

Check whether multiple queries of the same type are running at the same time. Some forms of queries run efficiently when they run alone. However, if similar forms of query run together, or in high volume, they can cause concurrency issues. Often, these issues are caused when the database uses temp tables to render results. A restrictive transaction isolation level can also cause concurrency issues.

If tables are read and written to concurrently, the database might be using locks. To help identify periods of poor performance, examine the use of databases through large-scale batch processes. To see recent locks and rollbacks, examine the output of the `SHOW ENGINE INNODB STATUS` command.

Check the structure of your queries

Check whether captured queries from these states use subqueries. This type of query often leads to poor performance because the database compiles the results internally and then substitutes them back into the query to render data. This process is an extra step for the database. In many cases, this step can cause poor performance in a highly concurrent loading condition.

Also check whether your queries use large numbers of `ORDER BY` and `GROUP BY` clauses. In such operations, often the database must first form the entire dataset in memory. Then it must order or group it in a specific manner before returning it to the client.

Tuning Aurora MySQL with Amazon DevOps Guru proactive insights

DevOps Guru proactive insights detect known problematic conditions on your Aurora MySQL DB clusters before they occur. DevOps Guru can do the following:

- Prevent many common database issues by cross-checking your database configuration against common recommended settings.
- Alert you to critical issues in your fleet that, if left unchecked, can lead to larger problems later.
- Alert you to newly discovered problems.

Every proactive insight contains an analysis of the cause of the problem and recommendations for corrective actions.

Topics

- The InnoDB history list length increased significantly
- Database is creating temporary tables on disk

The InnoDB history list length increased significantly

Starting on *date*, your history list for row changes increased significantly, up to *length* on *db-instance*. This increase affects query and database shutdown performance.

Topics

- Supported engine versions
- Context
- Likely causes for this issue
- Actions
- Relevant metrics

Supported engine versions

This insight information is supported for all versions of Aurora MySQL.

Context

The InnoDB transaction system maintains multiversion concurrency control (MVCC). When a row is modified, the pre-modification version of the data being modified is stored as an undo record in an undo log. Every undo record has a reference to its previous redo record, forming a linked list.

The *InnoDB history list* is a global list of the undo logs for committed transactions. MySQL uses the history list to purge records and log pages when transactions no longer require the history.

The *history list length* is the total number of undo logs that contain modifications in the history list. Each log contains one or more modifications. If the InnoDB history list length grows too large, indicating a large number of old row versions, queries and database shutdowns become slower.

Likely causes for this issue

Typical causes of a long history list include the following:

- Long-running transactions, either read or write
- A heavy write load

Actions

We recommend different actions depending on the causes of your insight.

Topics

- Don't begin any operation involving a database shutdown until the InnoDB history list decreases
- Identify and end long-running transactions
- Identify the top hosts and top users by using Performance Insights.

Don't begin any operation involving a database shutdown until the InnoDB history list decreases

Because a long InnoDB history list slows database shutdowns, reduce the list size before initiating operations involving a database shutdown. These operations include major version database upgrades.

Identify and end long-running transactions

You can find long-running transactions by querying `information_schema.innodb_trx`.

> **Note**
>
> Make sure also to look for long-running transactions on read replicas.

To identify and end long-running transactions

1. In your SQL client, run the following query:

```
SELECT a.trx_id,
      a.trx_state,
      a.trx_started,
      TIMESTAMPDIFF(SECOND,a.trx_started, now()) as "Seconds Transaction Has Been
 Open",
      a.trx_rows_modified,
      b.USER,
      b.host,
      b.db,
      b.command,
      b.time,
      b.state
FROM  information_schema.innodb_trx a,
      information_schema.processlist b
WHERE a.trx_mysql_thread_id=b.id
  AND TIMESTAMPDIFF(SECOND,a.trx_started, now()) > 10
ORDER BY trx_started
```

2. End each long-running transaction with a `COMMIT` or `ROLLBACK` command.

Identify the top hosts and top users by using Performance Insights.

Optimize transactions so that large numbers of modified rows are immediately committed.

Relevant metrics

The following metrics are related to this insight:

- trx_rseg_history_len

  For more information, see InnoDB INFORMATION_SCHEMA Metrics Table in the *MySQL 5.7 Reference Manual*.

Database is creating temporary tables on disk

Your recent on-disk temporary table usage increased significantly, up to *`percentage`*. The database is creating around *`number`* temporary tables per second. This might impact performance and increase disk operations on *`db-instance`*.

Topics

- Supported engine versions
- Context
- Likely causes for this issue
- Actions
- Relevant metrics

Supported engine versions

This insight information is supported for all versions of Aurora MySQL.

Context

Sometimes it's necessary for the MySQL server to create an internal temporary table while processing a query. Aurora MySQL can hold an internal temporary table in memory, where it can be processed by the TempTable or MEMORY storage engine, or stored on disk by InnoDB. For more information, see Internal Temporary Table Use in MySQL in the *MySQL Reference Manual*.

Likely causes for this issue

An increase in on-disk temporary tables indicates the use of complex queries. If the configured memory is insufficient to store temporary tables in memory, Aurora MySQL creates the tables on disk. This can impact performance and increase disk operations.

Actions

We recommend different actions depending on the causes of your insight.

- For Aurora MySQL version 3, we recommend that you use the TempTable storage engine.
- Optimize your queries to return less data by selecting only necessary columns.

  If you turn on the Performance Schema with all `statement` instruments enabled and timed, you can query `SYS.statements_with_temp_tables` to retrieve the list of queries that use temporary tables. For more information, see Prerequisites for Using the sys Schema in the MySQL documentation.
- Consider indexing columns that are involved in sorting and grouping operations.
- Rewrite your queries to avoid `BLOB` and `TEXT` columns. These columns always use disk.
- Tune the following database parameters: `tmp_table_size` and `max_heap_table_size`.

The default values for these parameters is 16 MiB. When using the MEMORY storage engine for in-memory temporary tables, their maximum size is defined by the `tmp_table_size` or `max_heap_table_size` value, whichever is smaller. When this maximum size is reached, MySQL automatically converts the in-memory internal temporary table to an InnoDB on-disk internal temporary table. For more information, see Use the TempTable storage engine on Amazon RDS for MySQL and Amazon Aurora MySQL.

Note

When explicitly creating MEMORY tables with CREATE TABLE, only the `max_heap_table_size` variable determines how large a table can grow. There is also no conversion to an on-disk format.

Relevant metrics

The following Performance Insights metrics are related to this insight:

- Created_tmp_disk_tables
- Created_tmp_tables

For more information, see Created_tmp_disk_tables in the MySQL documentation.

Working with parallel query for Amazon Aurora MySQL

This topic describes the parallel query performance optimization for Amazon Aurora MySQL-Compatible Edition. This feature uses a special processing path for certain data-intensive queries, taking advantage of the Aurora shared storage architecture. Parallel query works best with Aurora MySQL DB clusters that have tables with millions of rows and analytic queries that take minutes or hours to complete.

Contents

Overview of parallel query for Aurora MySQL

Aurora MySQL parallel query is an optimization that parallelizes some of the I/O and computation involved in processing data-intensive queries. The work that is parallelized includes retrieving rows from storage, extracting column values, and determining which rows match the conditions in the `WHERE` clause and join clauses. This data-intensive work is delegated (in database optimization terms, *pushed down*) to multiple nodes in the Aurora distributed storage layer. Without parallel query, each query brings all the scanned data to a single node within the Aurora MySQL cluster (the head node) and performs all the query processing there.

> **Tip**
>
> The PostgreSQL database engine also has a feature called "parallel query." That feature is unrelated to Aurora parallel query.

When the parallel query feature is turned on, the Aurora MySQL engine automatically determines when queries can benefit, without requiring SQL changes such as hints or table attributes. In the following sections, you can find an explanation of when parallel query is applied to a query. You can also find how to make sure that parallel query is applied where it provides the most benefit.

> **Note**
>
> The parallel query optimization provides the most benefit for long-running queries that take minutes or hours to complete. Aurora MySQL generally doesn't perform parallel query optimization for inexpensive queries. It also generally doesn't perform parallel query optimization if another optimization technique makes more sense, such as query caching, buffer pool caching, or index lookups. If you find that parallel query isn't being used when you expect it, see Verifying which statements use parallel query.

Topics

- Benefits
- Architecture
- Prerequisites
- Limitations
- I/O costs with parallel query

Benefits

With parallel query, you can run data-intensive analytic queries on Aurora MySQL tables. In many cases, you can get an order-of-magnitude performance improvement over the traditional division of labor for query processing.

Benefits of parallel query include the following:

- Improved I/O performance, due to parallelizing physical read requests across multiple storage nodes.
- Reduced network traffic. Aurora doesn't transmit entire data pages from storage nodes to the head node and then filter out unnecessary rows and columns afterward. Instead, Aurora transmits compact tuples containing only the column values needed for the result set.
- Reduced CPU usage on the head node, due to pushing down function processing, row filtering, and column projection for the `WHERE` clause.
- Reduced memory pressure on the buffer pool. The pages processed by the parallel query aren't added to the buffer pool. This approach reduces the chance of a data-intensive scan evicting frequently used data from the buffer pool.
- Potentially reduced data duplication in your extract, transform, load (ETL) pipeline, by making it practical to perform long-running analytic queries on existing data.

Architecture

The parallel query feature uses the major architectural principles of Aurora MySQL: decoupling the database engine from the storage subsystem, and reducing network traffic by streamlining communication protocols. Aurora MySQL uses these techniques to speed up write-intensive operations such as redo log processing. Parallel query applies the same principles to read operations.

> **Note**
>
> The architecture of Aurora MySQL parallel query differs from that of similarly named features in other database systems. Aurora MySQL parallel query doesn't involve symmetric multiprocessing (SMP) and so doesn't depend on the CPU capacity of the database server. The parallel processing happens in the storage layer, independent of the Aurora MySQL server that serves as the query coordinator.

By default, without parallel query, the processing for an Aurora query involves transmitting raw data to a single node within the Aurora cluster (the *head node*). Aurora then performs all further processing for that query in a single thread on that single node. With parallel query, much of this I/O-intensive and CPU-intensive work is delegated to nodes in the storage layer. Only the compact rows of the result set are transmitted back to the head node, with rows already filtered, and column values already extracted and transformed. The performance benefit comes from the reduction in network traffic, reduction in CPU usage on the head node, and parallelizing the I/O across the storage nodes. The amount of parallel I/O, filtering, and projection is independent of the number of DB instances in the Aurora cluster that runs the query.

Prerequisites

To use all features of parallel query requires an Aurora MySQL DB cluster that's running version 2.09 or higher. If you already have a cluster that you want to use with parallel query, you can upgrade it to a compatible version and turn on parallel query afterward. In that case, make sure to follow the upgrade procedure in Upgrade considerations for parallel query because the configuration setting names and default values are different in these newer versions.

The DB instances in your cluster must use the `db.r*` instance classes.

Make sure that hash join optimization is turned on for your cluster. To learn how, see Turning on hash join for parallel query clusters.

To customize parameters such as `aurora_parallel_query` and `aurora_disable_hash_join`, you must have a custom parameter group that you use with your cluster. You can specify these parameters individually for each DB instance by using a DB parameter group. However, we recommend that you specify them in a DB cluster parameter group. That way, all DB instances in your cluster inherit the same settings for these parameters.

Limitations

The following limitations apply to the parallel query feature:

- Parallel query isn't supported with the Aurora I/O-Optimized DB cluster storage configuration.
- You can't use parallel query with the db.t2 or db.t3 instance classes. This limitation applies even if you request parallel query using the `aurora_pq_force` session variable.
- Parallel query doesn't apply to tables using the `COMPRESSED` or `REDUNDANT` row formats. Use the `COMPACT` or `DYNAMIC` row formats for tables you plan to use with parallel query.
- Aurora uses a cost-based algorithm to determine whether to use the parallel query mechanism for each SQL statement. Using certain SQL constructs in a statement can prevent parallel query or make parallel query unlikely for that statement. For information about compatibility of SQL constructs with parallel query, see How parallel query works with SQL constructs.
- Each Aurora DB instance can run only a certain number of parallel query sessions at one time. If a query has multiple parts that use parallel query, such as subqueries, joins, or `UNION` operators,

those phases run in sequence. The statement only counts as a single parallel query session at any one time. You can monitor the number of active sessions using the parallel query status variables. You can check the limit on concurrent sessions for a given DB instance by querying the status variable `Aurora_pq_max_concurrent_requests`.

- Parallel query is available in all AWS Regions that Aurora supports. For most AWS Regions, the minimum required Aurora MySQL version to use parallel query is 2.09.
- Parallel query is designed to improve the performance of data-intensive queries. It isn't designed for lightweight queries.
- We recommend that you use reader nodes for SELECT statements, especially data-intensive ones.

I/O costs with parallel query

If your Aurora MySQL cluster uses parallel query, you might see an increase in `VolumeReadIOPS` values. Parallel queries don't use the buffer pool. Thus, although the queries are fast, this optimized processing can result in an increase in read operations and associated charges.

Parallel query I/O costs for your query are metered at the storage layer, and will be the same or larger with parallel query turned on. Your benefit is the improvement in query performance. There are two reasons for potentially higher I/O costs with parallel query:

- Even if some of the data in a table is in the buffer pool, parallel query requires all data to be scanned at the storage layer, incurring I/O costs.
- Running a parallel query doesn't warm up the buffer pool. As a result, consecutive runs of the same parallel query incur the full I/O cost.

Planning for a parallel query cluster

Planning for a DB cluster that has parallel query turned on requires making some choices. These include performing setup steps (either creating or restoring a full Aurora MySQL cluster) and deciding how broadly to turn on parallel query across your DB cluster.

Consider the following as part of planning:

- If you use Aurora MySQL that's compatible with MySQL 5.7, you must choose Aurora MySQL 2.09 or higher. In this case, you always create a provisioned cluster. Then you turn on parallel query using the `aurora_parallel_query` parameter.

If you have an existing Aurora MySQL cluster that's running version 2.09 or higher, you don't have to create a new cluster to use parallel query. You can associate your cluster, or specific DB instances in the cluster, with a parameter group that has the `aurora_parallel_query` parameter turned on. By doing so, you can reduce the time and effort to set up the relevant data to use with parallel query.

- Plan for any large tables that you need to reorganize so that you can use parallel query when accessing them. You might need to create new versions of some large tables where parallel query is useful. For example, you might need to remove full-text search indexes. For details, see Creating schema objects to take advantage of parallel query.

Checking Aurora MySQL version compatibility for parallel query

To check which Aurora MySQL versions are compatible with parallel query clusters, use the `describe-db-engine-versions` AWS CLI command and check the value of the `SupportsParallelQuery` field. The following code example shows how to check which combinations are available for parallel query clusters in a specified AWS Region. Make sure to specify the full `--query` parameter string on a single line.

```
aws rds describe-db-engine-versions --region us-east-1 --engine aurora-mysql \
--query '*[]|[?SupportsParallelQuery == `true`].[EngineVersion]' --output text
```

The preceding commands produce output similar to the following. The output might vary depending on which Aurora MySQL versions are available in the specified AWS Region.

```
5.7.mysql_aurora.2.11.1
8.0.mysql_aurora.3.01.0
8.0.mysql_aurora.3.01.1
8.0.mysql_aurora.3.02.0
8.0.mysql_aurora.3.02.1
8.0.mysql_aurora.3.02.2
8.0.mysql_aurora.3.03.0
```

After you start using parallel query with a cluster, you can monitor performance and remove obstacles to parallel query usage. For those instructions, see Performance tuning for parallel query.

Creating a DB cluster that works with parallel query

To create an Aurora MySQL cluster with parallel query, add new instances to it, or perform other administrative operations, you use the same AWS Management Console and AWS CLI techniques that you do with other Aurora MySQL clusters. You can create a new cluster to work with parallel query. You can also create a DB cluster to work with parallel query by restoring from a snapshot of a MySQL-compatible Aurora DB cluster. If you aren't familiar with the process for creating a new Aurora MySQL cluster, you can find background information and prerequisites in Creating an Amazon Aurora DB cluster.

When you choose an Aurora MySQL engine version, we recommend that you choose the latest one available. Currently, Aurora MySQL versions 2.09 and higher support parallel query. You have more flexibility to turn parallel query on and off, or use parallel query with existing clusters, if you use Aurora MySQL 2.09 and higher.

Whether you create a new cluster or restore from a snapshot, you use the same techniques to add new DB instances that you do with other Aurora MySQL clusters.

Creating a parallel query cluster using the console

You can create a new parallel query cluster with the console as described following.

To create a parallel query cluster with the AWS Management Console

1. Follow the general AWS Management Console procedure in Creating an Amazon Aurora DB cluster.
2. On the **Select engine** screen, choose Aurora MySQL.

   For **Engine version**, choose Aurora MySQL 2.09 or higher. With these versions, you have the fewest limitations on parallel query usage. Those versions also have the most flexibility to turn parallel query on or off at any time.

   If it isn't practical to use a recent Aurora MySQL version for this cluster, choose **Show versions that support the parallel query feature**. Doing so filters the **Version** menu to show only the specific Aurora MySQL versions that are compatible with parallel query.
3. For **Additional configuration**, choose a parameter group that you created for **DB cluster parameter group**. Using such a custom parameter group is required for Aurora MySQL 2.09 and higher. In your DB cluster parameter group, specify the parameter settings `aurora_parallel_query=ON` and `aurora_disable_hash_join=OFF`. Doing so turns

on parallel query for the cluster, and turns on the hash join optimization that works in combination with parallel query.

To verify that a new cluster can use parallel query

1. Create a cluster using the preceding technique.
2. (For Aurora MySQL version 2 or 3) Check that the `aurora_parallel_query` configuration setting is true.

```
mysql> select @@aurora_parallel_query;
+-------------------------+
| @@aurora_parallel_query |
+-------------------------+
|                       1 |
+-------------------------+
```

3. (For Aurora MySQL version 2) Check that the `aurora_disable_hash_join` setting is false.

```
mysql> select @@aurora_disable_hash_join;
+----------------------------+
| @@aurora_disable_hash_join |
+----------------------------+
|                          0 |
+----------------------------+
```

4. With some large tables and data-intensive queries, check the query plans to confirm that some of your queries are using the parallel query optimization. To do so, follow the procedure in Verifying which statements use parallel query.

Creating a parallel query cluster using the CLI

You can create a new parallel query cluster with the CLI as described following.

To create a parallel query cluster with the AWS CLI

1. (Optional) Check which Aurora MySQL versions are compatible with parallel query clusters. To do so, use the `describe-db-engine-versions` command and check the value of the `SupportsParallelQuery` field. For an example, see Checking Aurora MySQL version compatibility for parallel query.

2. (Optional) Create a custom DB cluster parameter group with the settings `aurora_parallel_query=ON` and `aurora_disable_hash_join=OFF`. Use commands such as the following.

   ```
   aws rds create-db-cluster-parameter-group --db-parameter-group-family aurora-
   mysql5.7 --db-cluster-parameter-group-name pq-enabled-57-compatible
   aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name pq-
   enabled-57-compatible \
     --parameters
    ParameterName=aurora_parallel_query,ParameterValue=ON,ApplyMethod=pending-reboot
   aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name pq-
   enabled-57-compatible \
     --parameters
    ParameterName=aurora_disable_hash_join,ParameterValue=OFF,ApplyMethod=pending-
   reboot
   ```

   If you perform this step, specify the option `--db-cluster-parameter-group-name` *`my_cluster_parameter_group`* in the subsequent `create-db-cluster` statement. Substitute the name of your own parameter group. If you omit this step, you create the parameter group and associate it with the cluster later, as described in Turning parallel query on and off.

3. Follow the general AWS CLI procedure in Creating an Amazon Aurora DB cluster.

4. Specify the following set of options:

   - For the `--engine` option, use `aurora-mysql`. These values produce parallel query clusters that are compatible with MySQL 5.7 or 8.0.
   - For the `--db-cluster-parameter-group-name` option, specify the name of a DB cluster parameter group that you created and specified the parameter value `aurora_parallel_query=ON`. If you omit this option, you can create the cluster with a default parameter group and later modify it to use such a custom parameter group.
   - For the `--engine-version` option, use an Aurora MySQL version that's compatible with parallel query. Use the procedure from Planning for a parallel query cluster to get a list of versions if necessary. Use at least version 2.09.0. These versions and all higher ones contain substantial enhancements to parallel query.

     The following code example shows how. Substitute your own value for each of the environment variables such as *`$CLUSTER_ID`*. This example also specifies the `--manage-master-user-password` option to generate the master user password and manage it in

Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

```
aws rds create-db-cluster --db-cluster-identifier $CLUSTER_ID \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --master-username $MASTER_USER_ID --manage-master-user-password \
  --db-cluster-parameter-group-name $CUSTOM_CLUSTER_PARAM_GROUP

aws rds create-db-instance --db-instance-identifier ${INSTANCE_ID}-1 \
  --engine same_value_as_in_create_cluster_command \
  --db-cluster-identifier $CLUSTER_ID --db-instance-class $INSTANCE_CLASS
```

5. Verify that a cluster you created or restored has the parallel query feature available.

   Check that the `aurora_parallel_query` configuration setting exists. If this setting has the value 1, parallel query is ready for you to use. If this setting has the value 0, set it to 1 before you can use parallel query. Either way, the cluster is capable of performing parallel queries.

```
mysql> select @@aurora_parallel_query;
+------------------------+
| @@aurora_parallel_query|
+------------------------+
|                      1 |
+------------------------+
```

To restore a snapshot to a parallel query cluster with the AWS CLI

1. Check which Aurora MySQL versions are compatible with parallel query clusters. To do so, use the `describe-db-engine-versions` command and check the value of the `SupportsParallelQuery` field. For an example, see Checking Aurora MySQL version compatibility for parallel query. Decide which version to use for the restored cluster. Choose Aurora MySQL 2.09.0 or higher for a MySQL 5.7-compatible cluster.
2. Locate an Aurora MySQL-compatible cluster snapshot.
3. Follow the general AWS CLI procedure in Restoring from a DB cluster snapshot.

```
aws rds restore-db-cluster-from-snapshot \
  --db-cluster-identifier mynewdbcluster \
  --snapshot-identifier mydbclustersnapshot \
    --engine aurora-mysql
```

4. Verify that a cluster you created or restored has the parallel query feature available. Use the same verification procedure as in Creating a parallel query cluster using the CLI.

Turning parallel query on and off

When parallel query is turned on, Aurora MySQL determines whether to use it at runtime for each query. In the case of joins, unions, subqueries, and so on, Aurora MySQL determines whether to use parallel query at runtime for each query block. For details, see Verifying which statements use parallel query and How parallel query works with SQL constructs.

You can turn on and turn off parallel query dynamically at both the global and session level for a DB instance by using the **aurora_parallel_query** option. You can change the `aurora_parallel_query` setting in your DB cluster group to turn parallel query on or off by default.

```
mysql> select @@aurora_parallel_query;
+------------------------+
| @@aurora_parallel_query|
+------------------------+
|                      1 |
+------------------------+
```

To toggle the `aurora_parallel_query` parameter at the session level, use the standard methods to change a client configuration setting. For example, you can do so through the `mysql` command line or within a JDBC or ODBC application. The command on the standard MySQL client is `set session aurora_parallel_query = {'ON'/'OFF'}`. You can also add the session-level parameter to the JDBC configuration or within your application code to turn on or turn off parallel query dynamically.

You can permanently change the setting for the `aurora_parallel_query` parameter, either for a specific DB instance or for your whole cluster. If you specify the parameter value in a DB parameter group, that value only applies to specific DB instance in your cluster. If you specify the parameter value in a DB cluster parameter group, all DB instances in the cluster inherit the same setting. To toggle the `aurora_parallel_query` parameter, use the techniques for working with parameter groups, as described in Working with parameter groups. Follow these steps:

1. Create a custom cluster parameter group (recommended) or a custom DB parameter group.

2. In this parameter group, update `parallel_query` to the value that you want.
3. Depending on whether you created a DB cluster parameter group or a DB parameter group, attach the parameter group to your Aurora cluster or to the specific DB instances where you plan to use the parallel query feature.

> **Tip**
>
> Because `aurora_parallel_query` is a dynamic parameter, it doesn't require a cluster restart after changing this setting. However, any connections that were using parallel query before toggling the option will continue to do so until the connection is closed, or the instance is rebooted.

You can modify the parallel query parameter by using the ModifyDBClusterParameterGroup or ModifyDBParameterGroup API operation or the AWS Management Console.

Turning on hash join for parallel query clusters

Parallel query is typically used for the kinds of resource-intensive queries that benefit from the hash join optimization. Thus, it's helpful to make sure that hash joins are turned on for clusters where you plan to use parallel query. For information about how to use hash joins effectively, see Optimizing large Aurora MySQL join queries with hash joins.

Turning on and turning off parallel query using the console

You can turn on or turn off parallel query at the DB instance level or the DB cluster level by working with parameter groups.

To turn on or turn off parallel query for a DB cluster with the AWS Management Console

1. Create a custom parameter group, as described in Working with parameter groups.
2. Update **aurora_parallel_query** to **1** (turned on) or **0** (turned off). For clusters where the parallel query feature is available, **aurora_parallel_query** is turned off by default.
3. If you use a custom cluster parameter group, attach it to the Aurora DB cluster where you plan to use the parallel query feature. If you use a custom DB parameter group, attach it to one or more DB instances in the cluster. We recommend using a cluster parameter group. Doing so makes sure that all DB instances in the cluster have the same settings for parallel query and associated features such as hash join.

Turning on and turning off parallel query using the CLI

You can modify the parallel query parameter by using the `modify-db-cluster-parameter-group` or `modify-db-parameter-group` command. Choose the appropriate command depending on whether you specify the value of `aurora_parallel_query` through a DB cluster parameter group or a DB parameter group.

To turn on or turn off parallel query for a DB cluster with the CLI

- Modify the parallel query parameter by using the `modify-db-cluster-parameter-group` command. Use a command such as the following. Substitute the appropriate name for your own custom parameter group. Substitute either `ON` or `OFF` for the `ParameterValue` portion of the `--parameters` option.

```
$ aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-
name cluster_param_group_name \
  --parameters
 ParameterName=aurora_parallel_query,ParameterValue=ON,ApplyMethod=pending-reboot
{
    "DBClusterParameterGroupName": "cluster_param_group_name"
}

aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-
name cluster_param_group_name \
  --parameters ParameterName=aurora_pq,ParameterValue=ON,ApplyMethod=pending-reboot
```

You can also turn on or turn off parallel query at the session level, for example through the `mysql` command line or within a JDBC or ODBC application. To do so, use the standard methods to change a client configuration setting. For example, the command on the standard MySQL client is `set session aurora_parallel_query = {'ON'/'OFF'}` for Aurora MySQL.

You can also add the session-level parameter to the JDBC configuration or within your application code to turn on or turn off parallel query dynamically.

Overriding the parallel query optimizer

You can use the `aurora_pq_force` session variable to override the parallel query optimizer and request parallel query for every query. We recommend that you do this only for testing purposes The following example shows how to use `aurora_pq_force` in a session.

```
set SESSION aurora_parallel_query = ON;
set SESSION aurora_pq_force = ON;
```

To turn off the override, do the following:

```
set SESSION aurora_pq_force = OFF;
```

Upgrade considerations for parallel query

Depending on the original and destination versions when you upgrade a parallel query cluster, you might find enhancements in the types of queries that parallel query can optimize. You might also find that you don't need to specify a special engine mode parameter for parallel query. The following sections explain the considerations when you upgrade a cluster that has parallel query turned on.

Upgrading parallel query clusters to Aurora MySQL version 3

Several SQL statements, clauses, and data types have new or improved parallel query support starting in Aurora MySQL version 3. When you upgrade from a release that's earlier than version 3, check whether additional queries can benefit from parallel query optimizations. For information about these parallel query enhancements, see Column data types, Partitioned tables, and Aggregate functions, GROUP BY clauses, and HAVING clauses.

If you're upgrading a parallel query cluster from Aurora MySQL 2.08 or lower, also learn about changes in how to turn on parallel query. To do so, read Upgrading to Aurora MySQL 2.09 and higher.

In Aurora MySQL version 3, hash join optimization is turned on by default. The `aurora_disable_hash_join` configuration option from earlier versions isn't used.

Upgrading to Aurora MySQL 2.09 and higher

In Aurora MySQL version 2.09 and higher, parallel query works for provisioned clusters and doesn't require the `parallelquery` engine mode parameter. Thus, you don't need to create a new cluster or restore from an existing snapshot to use parallel query with these versions. You can use the upgrade procedures described in Upgrading the minor version or patch level of an Aurora MySQL DB cluster to upgrade your cluster to such a version. You can upgrade an older cluster regardless of whether it was a parallel query cluster or a provisioned cluster. To reduce the number of choices in the **Engine version** menu, you can choose **Show versions that support the parallel query feature** to filter the entries in that menu. Then choose Aurora MySQL 2.09 or higher.

After you upgrade an earlier parallel query cluster to Aurora MySQL 2.09 or higher, you turn on parallel query in the upgraded cluster. Parallel query is turned off by default in these versions, and the procedure for enabling it is different. The hash join optimization is also turned off by default and must be turned on separately. Thus, make sure that you turn on these settings again after the upgrade. For instructions on doing so, see Turning parallel query on and off and Turning on hash join for parallel query clusters.

In particular, you turn on parallel query by using the configuration parameters `aurora_parallel_query=ON` and `aurora_disable_hash_join=OFF` instead of `aurora_pq_supported` and `aurora_pq`. The `aurora_pq_supported` and `aurora_pq` parameters are deprecated in the newer Aurora MySQL versions.

In the upgraded cluster, the `EngineMode` attribute has the value `provisioned` instead of `parallelquery`. To check whether parallel query is available for a specified engine version, now you check the value of the `SupportsParallelQuery` field in the output of the `describe-db-engine-versions` AWS CLI command. In earlier Aurora MySQL versions, you checked for the presence of `parallelquery` in the `SupportedEngineModes` list.

After you upgrade to Aurora MySQL version 2.09 or higher, you can take advantage of the following features. These features aren't available to parallel query clusters running older Aurora MySQL versions.

- Performance Insights. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.
- Backtracking. For more information, see Backtracking an Aurora DB cluster.
- Stopping and starting the cluster. For more information, see Stopping and starting an Amazon Aurora DB cluster.

Performance tuning for parallel query

To manage the performance of a workload with parallel query, make sure that parallel query is used for the queries where this optimization helps the most.

To do so, you can do the following:

- Make sure that your biggest tables are compatible with parallel query. You might change table properties or recreate some tables so that queries for those tables can take advantage of the parallel query optimization. To learn how, see Creating schema objects to take advantage of parallel query.
- Monitor which queries use parallel query. To learn how, see Monitoring parallel query.
- Verify that parallel query is being used for the most data-intensive and long-running queries, and with the right level of concurrency for your workload. To learn how, see Verifying which statements use parallel query.
- Fine-tune your SQL code to turn on parallel query to apply to the queries that you expect. To learn how, see How parallel query works with SQL constructs.

Creating schema objects to take advantage of parallel query

Before you create or modify tables that you plan to use for parallel query, make sure to familiarize yourself with the requirements described in Prerequisites and Limitations.

Because parallel query requires tables to use the `ROW_FORMAT=Compact` or `ROW_FORMAT=Dynamic` setting, check your Aurora configuration settings for any changes to the `INNODB_FILE_FORMAT` configuration option. Issue the `SHOW TABLE STATUS` statement to confirm the row format for all the tables in a database.

Before changing your schema to turn on parallel query to work with more tables, make sure to test. Your tests should confirm if parallel query results in a net increase in performance for those tables. Also, make sure that the schema requirements for parallel query are otherwise compatible with your goals.

For example, before switching from `ROW_FORMAT=Compressed` to `ROW_FORMAT=Compact` or `ROW_FORMAT=Dynamic`, test the performance of workloads for the original and new tables. Also, consider other potential effects such as increased data volume.

Verifying which statements use parallel query

In typical operation, you don't need to perform any special actions to take advantage of parallel query. After a query meets the essential requirements for parallel query, the query optimizer automatically decides whether to use parallel query for each specific query.

If you run experiments in a development or test environment, you might find that parallel query isn't used because your tables are too small in number of rows or overall data volume. The data for the table might also be entirely in the buffer pool, especially for tables that you created recently to perform experiments.

As you monitor or tune cluster performance, make sure to decide whether parallel query is being used in the appropriate contexts. You might adjust the database schema, settings, SQL queries, or even the cluster topology and application connection settings to take advantage of this feature.

To check if a query is using parallel query, check the query plan (also known as the "explain plan") by running the EXPLAIN statement. For examples of how SQL statements, clauses, and expressions affect EXPLAIN output for parallel query, see How parallel query works with SQL constructs.

The following example demonstrates the difference between a traditional query plan and a parallel query plan. This explain plan is from Query 3 from the TPC-H benchmark. Many of the sample queries throughout this section use the tables from the TPC-H dataset. You can get the table definitions, queries, and the dbgen program that generates sample data from the TPC-h website.

```
EXPLAIN SELECT l_orderkey,
  sum(l_extendedprice * (1 - l_discount)) AS revenue,
  o_orderdate,
  o_shippriority
FROM customer,
  orders,
  lineitem
WHERE c_mktsegment = 'AUTOMOBILE'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < date '1995-03-13'
AND l_shipdate > date '1995-03-13'
GROUP BY l_orderkey,
  o_orderdate,
  o_shippriority
ORDER BY revenue DESC,
  o_orderdate LIMIT 10;
```

By default, the query might have a plan like the following. If you don't see hash join used in the query plan, make sure that optimization is turned on first.

```
+----+-------------+----------+------------+------+---------------+------+---------
+------+----------+----------+----------------------------------------------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len |
 ref  | rows      | filtered | Extra                                              |
+----+-------------+----------+------------+------+---------------+------+---------
+------+----------+----------+----------------------------------------------------+
|  1 | SIMPLE      | customer | NULL       | ALL  | NULL          | NULL | NULL    |
 NULL |  1480234 |    10.00 | Using where; Using temporary; Using filesort       |
|  1 | SIMPLE      | orders   | NULL       | ALL  | NULL          | NULL | NULL    |
 NULL | 14875240 |     3.33 | Using where; Using join buffer (Block Nested Loop) |
|  1 | SIMPLE      | lineitem | NULL       | ALL  | NULL          | NULL | NULL    |
 NULL | 59270573 |     3.33 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+----------+------------+------+---------------+------+---------
+------+----------+----------+----------------------------------------------------+
```

For Aurora MySQL version 3, you turn on hash join at the session level by issuing the following statement.

```
SET optimizer_switch='block_nested_loop=on';
```

For Aurora MySQL version 2.09 and higher, you set the `aurora_disable_hash_join` DB parameter or DB cluster parameter to `0` (off). Turning off `aurora_disable_hash_join` sets the value of `optimizer_switch` to `hash_join=on`.

After you turn on hash join, try running the `EXPLAIN` statement again. For information about how to use hash joins effectively, see Optimizing large Aurora MySQL join queries with hash joins.

With hash join turned on but parallel query turned off, the query might have a plan like the following, which uses hash join but not parallel query.

```
+----+-------------+----------+...+-----------
+-------------------------------------------------------------------+
| id | select_type | table    |...| rows      | Extra
                         |
+----+-------------+----------+...+-----------
+-------------------------------------------------------------------+
|  1 | SIMPLE      | customer |...|   5798330 | Using where; Using index; Using
 temporary; Using filesort       |
|  1 | SIMPLE      | orders   |...| 154545408 | Using where; Using join buffer (Hash
 Join Outer table orders)   |
|  1 | SIMPLE      | lineitem |...| 606119300 | Using where; Using join buffer (Hash
 Join Outer table lineitem) |
+----+-------------+----------+...+-----------
+-------------------------------------------------------------------+
```

After parallel query is turned on, two steps in this query plan can use the parallel query optimization, as shown under the `Extra` column in the `EXPLAIN` output. The I/O-intensive and CPU-intensive processing for those steps is pushed down to the storage layer.

```
+----+...
+-----------------------------------------------------------------------------------------------
+
| id |...| Extra
                                                   |
+----+...
+-----------------------------------------------------------------------------------------------
+
|  1 |...| Using where; Using index; Using temporary; Using filesort
                                                   |
|  1 |...| Using where; Using join buffer (Hash Join Outer table orders); Using
 parallel query (4 columns, 1 filters, 1 exprs; 0 extra)   |
|  1 |...| Using where; Using join buffer (Hash Join Outer table lineitem); Using
 parallel query (4 columns, 1 filters, 1 exprs; 0 extra) |
+----+...
+-----------------------------------------------------------------------------------------------
+
```

For information about how to interpret `EXPLAIN` output for a parallel query and the parts of SQL statements that parallel query can apply to, see How parallel query works with SQL constructs.

The following example output shows the results of running the preceding query on a db.r4.2xlarge instance with a cold buffer pool. The query runs substantially faster when using parallel query.

Note

Because timings depend on many environmental factors, your results might be different. Always conduct your own performance tests to confirm the findings with your own environment, workload, and so on.

```
-- Without parallel query
+------------+-------------+-------------+----------------+
| l_orderkey | revenue     | o_orderdate | o_shippriority |
+------------+-------------+-------------+----------------+
|   92511430 | 514726.4896 | 1995-03-06  |              0 |
.
```

```
.
|   28840519 | 454748.2485 | 1995-03-08  |              0 |
+------------+-------------+------------+----------------+
10 rows in set (24 min 49.99 sec)
```

```
-- With parallel query
+------------+-------------+------------+----------------+
| l_orderkey | revenue     | o_orderdate | o_shippriority |
+------------+-------------+------------+----------------+
|   92511430 | 514726.4896 | 1995-03-06  |              0 |
.
.
|   28840519 | 454748.2485 | 1995-03-08  |              0 |
+------------+-------------+------------+----------------+
10 rows in set (1 min 49.91 sec)
```

Many of the sample queries throughout this section use the tables from this TPC-H dataset, particularly the PART table, which has 20 million rows and the following definition.

```
+---------------+---------------+------+-----+---------+-------+
| Field         | Type          | Null | Key | Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| p_partkey     | int(11)       | NO   | PRI | NULL    |       |
| p_name        | varchar(55)   | NO   |     | NULL    |       |
| p_mfgr        | char(25)      | NO   |     | NULL    |       |
| p_brand       | char(10)      | NO   |     | NULL    |       |
| p_type        | varchar(25)   | NO   |     | NULL    |       |
| p_size        | int(11)       | NO   |     | NULL    |       |
| p_container   | char(10)      | NO   |     | NULL    |       |
| p_retailprice | decimal(15,2) | NO   |     | NULL    |       |
| p_comment     | varchar(23)   | NO   |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
```

Experiment with your workload to get a sense of whether individual SQL statements can take advantage of parallel query. Then use the following monitoring techniques to help verify how often parallel query is used in real workloads over time. For real workloads, extra factors such as concurrency limits apply.

Monitoring parallel query

If your Aurora MySQL cluster uses parallel query, you might see an increase in VolumeReadIOPS values. Parallel queries don't use the buffer pool. Thus, although the queries are fast, this optimized processing can result in an increase in read operations and associated charges.

In addition to the Amazon CloudWatch metrics described in Viewing metrics in the Amazon RDS console, Aurora provides other global status variables. You can use these global status variables to help monitor parallel query execution. They can give you insights into why the optimizer might use or not use parallel query in a given situation. To access these variables, you can use the SHOW GLOBAL STATUS command. You can also find these variables listed following.

A parallel query session isn't necessarily a one-to-one mapping with the queries performed by the database. For example, suppose that your query plan has two steps that use parallel query. In that case, the query involves two parallel sessions and the counters for requests attempted and requests successful are incremented by two.

When you experiment with parallel query by issuing EXPLAIN statements, expect to see increases in the counters designated as "not chosen" even though the queries aren't actually running. When you work with parallel query in production, you can check if the "not chosen" counters are increasing faster than you expect. At this point, you can adjust so that parallel query runs for the queries that you expect. To do so, you can change your cluster settings, query mix, DB instances where parallel query is turned on, and so on.

These counters are tracked at the DB instance level. When you connect to a different endpoint, you might see different metrics because each DB instance runs its own set of parallel queries. You might also see different metrics when the reader endpoint connects to a different DB instance for each session.

Name	Description
Aurora_pq_bytes_returned	The number of bytes for the tuple data structures transmitted to the head node during parallel queries. Divide by 16,384 to compare against Aurora_pq_pages_pushed_down .
Aurora_pq_max_concurrent_requests	The maximum number of parallel query sessions that can run concurrently on this

Name	Description
	Aurora DB instance. This is a fixed number that depends on the AWS DB instance class.
`Aurora_pq_pages_pushed_down`	The number of data pages (each with a fixed size of 16 KiB) where parallel query avoided a network transmission to the head node.
`Aurora_pq_request_attempted`	The number of parallel query sessions requested. This value might represent more than one session per query, depending on SQL constructs such as subqueries and joins.
`Aurora_pq_request_executed`	The number of parallel query sessions run successfully.
`Aurora_pq_request_failed`	The number of parallel query sessions that returned an error to the client. In some cases, a request for a parallel query might fail, for example due to a problem in the storage layer. In these cases, the query part that failed is retried using the nonparallel query mechanism . If the retried query also fails, an error is returned to the client and this counter is incremented.
`Aurora_pq_request_in_progress`	The number of parallel query sessions currently in progress. This number applies to the particular Aurora DB instance that you are connected to, not the entire Aurora DB cluster. To see if a DB instance is close to its concurrency limit, compare this value to `Aurora_pq_max_concurrent_requests` .

Name	Description
`Aurora_pq_request_not_chosen`	The number of times parallel query wasn't chosen to satisfy a query. This value is the sum of several other more granular counters. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_below_min_rows`	The number of times parallel query wasn't chosen due to the number of rows in the table. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_column_bit`	The number of parallel query requests that use the nonparallel query processing path because of an unsupported data type in the list of projected columns.
`Aurora_pq_request_not_chosen_column_geometry`	The number of parallel query requests that use the nonparallel query processing path because the table has columns with the `GEOMETRY` data type. For information about Aurora MySQL versions that remove this limitation, see Upgrading parallel query clusters to Aurora MySQL version 3.
`Aurora_pq_request_not_chosen_column_lob`	The number of parallel query requests that use the nonparallel query processing path because the table has columns with a `LOB` data type, or `VARCHAR` columns that are stored externally due to the declared length. For information about Aurora MySQL versions that remove this limitation, see Upgrading parallel query clusters to Aurora MySQL version 3.

Name	Description
`Aurora_pq_request_not_chosen_column_virtual`	The number of parallel query requests that use the nonparallel query processing path because the table contains a virtual column.
`Aurora_pq_request_not_chosen_custom_charset`	The number of parallel query requests that use the nonparallel query processing path because the table has columns with a custom character set.
`Aurora_pq_request_not_chosen_fast_ddl`	The number of parallel query requests that use the nonparallel query processing path because the table is currently being altered by a fast DDL `ALTER` statement.
`Aurora_pq_request_not_chosen_few_pages_outside_buffer_pool`	The number of times parallel query wasn't chosen, even though less than 95 percent of the table data was in the buffer pool, because there wasn't enough unbuffered table data to make parallel query worthwhile.
`Aurora_pq_request_not_chosen_full_text_index`	The number of parallel query requests that use the nonparallel query processing path because the table has full-text indexes.
`Aurora_pq_request_not_chosen_high_buffer_pool_pct`	The number of times parallel query wasn't chosen because a high percentage of the table data (currently, greater than 95 percent) was already in the buffer pool. In these cases, the optimizer determines that reading the data from the buffer pool is more efficient . An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.

Name	Description
`Aurora_pq_request_not_chosen_index_hint`	The number of parallel query requests that use the nonparallel query processing path because the query includes an index hint.
`Aurora_pq_request_not_chosen_innodb_table_format`	The number of parallel query requests that use the nonparallel query processing path because the table uses an unsupported InnoDB row format. Aurora parallel query only applies to the `COMPACT`, `REDUNDANT` , and `DYNAMIC` row formats.
`Aurora_pq_request_not_chosen_long_trx`	The number of parallel query requests that used the nonparallel query processing path, due to the query being started inside a long-running transaction. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_no_where_clause`	The number of parallel query requests that use the nonparallel query processing path because the query doesn't include any `WHERE` clause.
`Aurora_pq_request_not_chosen_range_scan`	The number of parallel query requests that use the nonparallel query processing path because the query uses a range scan on an index.
`Aurora_pq_request_not_chosen_row_length_too_long`	The number of parallel query requests that use the nonparallel query processing path because the total combined length of all the columns is too long.

Name	Description
`Aurora_pq_request_not_chosen_small_table`	The number of times parallel query wasn't chosen due to the overall size of the table, as determined by number of rows and average row length. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_temporary_table`	The number of parallel query requests that use the nonparallel query processing path because the query refers to temporary tables that use the unsupported `MyISAM` or `memory` table types.
`Aurora_pq_request_not_chosen_tx_isolation`	The number of parallel query requests that use the nonparallel query processing path because query uses an unsupported transaction isolation level. On reader DB instances, parallel query only applies to the `REPEATABLE READ` and `READ COMMITTED` isolation levels.
`Aurora_pq_request_not_chosen_update_delete_stmts`	The number of parallel query requests that use the nonparallel query processing path because the query is part of an `UPDATE` or `DELETE` statement.
`Aurora_pq_request_not_chosen_unsupported_access`	The number of parallel query requests that use the nonparallel query processing path because the `WHERE` clause doesn't meet the criteria for parallel query. This result can occur if the query doesn't require a data-intensive scan, or if the query is a `DELETE` or `UPDATE` statement.

Name	Description
`Aurora_pq_request_not_chosen_unsupported_storage_type`	The number of parallel query requests that use the nonparallel query processing path because the Aurora MySQL DB cluster isn't using a supported Aurora cluster storage configuration. This parameter is available in Aurora MySQL version 3.04 and higher. For more information, see Limitations.
`Aurora_pq_request_throttled`	The number of times parallel query wasn't chosen due to the maximum number of concurrent parallel queries already running on a particular Aurora DB instance.

How parallel query works with SQL constructs

In the following section, you can find more detail about why particular SQL statements use or don't use parallel query. This section also details how Aurora MySQL features interact with parallel query. This information can help you diagnose performance issues for a cluster that uses parallel query or understand how parallel query applies for your particular workload.

The decision to use parallel query relies on many factors that occur at the moment that the statement runs. Thus, parallel query might be used for certain queries always, never, or only under certain conditions.

> **Tip**
>
> When you view these examples in HTML, you can use the **Copy** widget in the upper-right corner of each code listing to copy the SQL code to try yourself. Using the **Copy** widget avoids copying the extra characters around the `mysql>` prompt and `->` continuation lines.

Topics

- Column data types
- Partitioned tables
- Aggregate functions, GROUP BY clauses, and HAVING clauses
- Function calls in WHERE clause
- LIMIT clause
- Comparison operators
- Joins
- Subqueries
- UNION
- Views
- Data manipulation language (DML) statements
- Transactions and locking
- B-tree indexes
- Full-text search (FTS) indexes
- Virtual columns
- Built-in caching mechanisms
- Optimizer hints
- MyISAM temporary tables

EXPLAIN statement

As shown in examples throughout this section, the `EXPLAIN` statement indicates whether each stage of a query is currently eligible for parallel query. It also indicates which aspects of a query can be pushed down to the storage layer. The most important items in the query plan are the following:

- A value other than `NULL` for the `key` column suggests that the query can be performed efficiently using index lookups, and parallel query is unlikely.
- A small value for the `rows` column (a value not in the millions) suggests that the query isn't accessing enough data to make parallel query worthwhile. This means that parallel query is unlikely.
- The `Extra` column shows you if parallel query is expected to be used. This output looks like the following example.

```
Using parallel query (A columns, B filters, C exprs; D extra)
```

The `columns` number represents how many columns are referred to in the query block.

The `filters` number represents the number of `WHERE` predicates representing a simple comparison of a column value to a constant. The comparison can be for equality, inequality, or a range. Aurora can parallelize these kinds of predicates most effectively.

The `exprs` number represents the number of expressions such as function calls, operators, or other expressions that can also be parallelized, though not as effectively as a filter condition.

The `extra` number represents how many expressions can't be pushed down and are performed by the head node.

For example, consider the following `EXPLAIN` output.

```
mysql> explain select p_name, p_mfgr from part
    -> where p_brand is not null
    -> and upper(p_type) is not null
    -> and round(p_retailprice) is not null;
+----+-------------+-------+...+----------
+----------------------------------------------------------------------------+
| id | select_type | table |...| rows     | Extra
                                |
+----+-------------+-------+...+----------
+----------------------------------------------------------------------------+
|  1 | SIMPLE      | part  |...| 20427936 | Using where; Using parallel query (5
 columns, 1 filters, 2 exprs; 0 extra) |
+----+-------------+-------+...+----------
+----------------------------------------------------------------------------+
```

The information from the `Extra` column shows that five columns are extracted from each row to evaluate the query conditions and construct the result set. One `WHERE` predicate involves a filter, that is, a column that is directly tested in the `WHERE` clause. Two `WHERE` clauses require evaluating more complicated expressions, in this case involving function calls. The `0 extra` field confirms that all the operations in the `WHERE` clause are pushed down to the storage layer as part of parallel query processing.

In cases where parallel query isn't chosen, you can typically deduce the reason from the other columns of the `EXPLAIN` output. For example, the `rows` value might be too small, or the `possible_keys` column might indicate that the query can use an index lookup instead of a data-intensive scan. The following example shows a query where the optimizer can estimate that the query will scan only a small number of rows. It does so based on the characteristics of the primary key. In this case, parallel query isn't required.

```
mysql> explain select count(*) from part where p_partkey between 1 and 100;
+----+-------------+-------+-------+---------------+---------+---------+------+------
+--------------------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows |
 Extra                    |
+----+-------------+-------+-------+---------------+---------+---------+------+------
+--------------------------+
|  1 | SIMPLE      | part  | range | PRIMARY       | PRIMARY | 4       | NULL |   99 |
 Using where; Using index |
+----+-------------+-------+-------+---------------+---------+---------+------+------
+--------------------------+
```

The output showing whether parallel query will be used takes into account all available factors at the moment that the `EXPLAIN` statement is run. The optimizer might make a different choice when the query is actually run, if the situation changed in the meantime. For example, `EXPLAIN` might report that a statement will use parallel query. But when the query is actually run later, it might not use parallel query based on the conditions then. Such conditions can include several other parallel queries running concurrently. They can also include rows being deleted from the table, a new index being created, too much time passing within an open transaction, and so on.

WHERE clause

For a query to use the parallel query optimization, it *must* include a `WHERE` clause.

The parallel query optimization speeds up many kinds of expressions used in the `WHERE` clause:

- Simple comparisons of a column value to a constant, known as *filters*. These comparisons benefit the most from being pushed down to the storage layer. The number of filter expressions in a query is reported in the `EXPLAIN` output.
- Other kinds of expressions in the `WHERE` clause are also pushed down to the storage layer where possible. The number of such expressions in a query is reported in the `EXPLAIN` output. These expressions can be function calls, `LIKE` operators, `CASE` expressions, and so on.

- Certain functions and operators aren't currently pushed down by parallel query. The number of such expressions in a query is reported as the `extra` counter in the `EXPLAIN` output. The rest of the query can still use parallel query.
- While expressions in the select list aren't pushed down, queries containing such functions can still benefit from reduced network traffic for the intermediate results of parallel queries. For example, queries that call aggregation functions in the select list can benefit from parallel query, even though the aggregation functions aren't pushed down.

For example, the following query does a full-table scan and processes all the values for the `P_BRAND` column. However, it doesn't use parallel query because the query doesn't include any `WHERE` clause.

```
mysql> explain select count(*), p_brand from part group by p_brand;
+----+-------------+-------+------+---------------+------+---------+------+----------
+---------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows     |
 Extra                           |
+----+-------------+-------+------+---------------+------+---------+------+----------
+---------------------------------+
|  1 | SIMPLE      | part  | ALL  | NULL          | NULL | NULL    | NULL | 20427936 |
 Using temporary; Using filesort |
+----+-------------+-------+------+---------------+------+---------+------+----------
+---------------------------------+
```

In contrast, the following query includes `WHERE` predicates that filter the results, so parallel query can be applied:

```
mysql> explain select count(*), p_brand from part where p_name is not null
    ->   and p_mfgr in ('Manufacturer#1', 'Manufacturer#3') and p_retailprice > 1000
    -> group by p_brand;
+----+...+----------
+----------------------------------------------------------------------------------------------
+
| id |...| rows     | Extra
                                            |
+----+...+----------
+----------------------------------------------------------------------------------------------
+
|  1 |...| 20427936 | Using where; Using temporary; Using filesort; Using parallel
 query (5 columns, 1 filters, 2 exprs; 0 extra) |
+----+...+----------
+----------------------------------------------------------------------------------------------
+
```

If the optimizer estimates that the number of returned rows for a query block is small, parallel query isn't used for that query block. The following example shows a case where a greater-than operator on the primary key column applies to millions of rows, which causes parallel query to be used. The converse less-than test is estimated to apply to only a few rows and doesn't use parallel query.

```
mysql> explain select count(*) from part where p_partkey > 10;
+----+...+----------
+----------------------------------------------------------------------------+
| id |...| rows     | Extra
          |
+----+...+----------
+----------------------------------------------------------------------------+
|  1 |...| 20427936 | Using where; Using parallel query (1 columns, 1 filters, 0 exprs;
 0 extra) |
+----+...+----------
+----------------------------------------------------------------------------+

mysql> explain select count(*) from part where p_partkey < 10;
+----+...+------+--------------------------+
| id |...| rows | Extra                    |
+----+...+------+--------------------------+
|  1 |...|    9 | Using where; Using index |
+----+...+------+--------------------------+
```

Data definition language (DDL)

In Aurora MySQL version 2, parallel query is only available for tables for which no fast data definition language (DDL) operations are pending. In Aurora MySQL version 3, you can use parallel query on a table at the same time as an instant DDL operation.

Instant DDL in Aurora MySQL version 3 replaces the fast DDL feature in Aurora MySQL version 2. For information about instant DDL, see Instant DDL (Aurora MySQL version 3).

Column data types

In Aurora MySQL version 3, parallel query can work with tables containing columns with data types TEXT, BLOB, JSON, and GEOMETRY. It can also work with VARCHAR and CHAR columns with a maximum declared length longer than 768 bytes. If your query refers to any columns containing such large object types, the additional work to retrieve them does add some overhead to query processing. In that case, check if the query can omit the references to those columns. If not, run benchmarks to confirm if such queries are faster with parallel query turned on or turned off.

In Aurora MySQL version 2, parallel query has these limitations for large object types:

- `TEXT`, `BLOB`, `JSON`, and `GEOMETRY` data types aren't supported with parallel query. A query that refers to any columns of these types can't use parallel query.
- Variable-length columns (`VARCHAR` and `CHAR` data types) are compatible with parallel query up to a maximum declared length of 768 bytes. A query that refers to any columns of the types declared with a longer maximum length can't use parallel query. For columns that use multibyte character sets, the byte limit takes into account the maximum number of bytes in the character set. For example, for the character set `utf8mb4` (which has a maximum character length of 4 bytes), a `VARCHAR(192)` column is compatible with parallel query but a `VARCHAR(193)` column isn't.

Partitioned tables

You can use partitioned tables with parallel query in Aurora MySQL version 3. Because partitioned tables are represented internally as multiple smaller tables, a query that uses parallel query on a nonpartitioned table might not use parallel query on an identical partitioned table. Aurora MySQL considers whether each partition is large enough to qualify for the parallel query optimization, instead of evaluating the size of the entire table. Check whether the `Aurora_pq_request_not_chosen_small_table` status variable is incremented if a query on a partitioned table doesn't use parallel query when you expect it to.

For example, consider one table partitioned with PARTITION BY HASH (*column*) PARTITIONS 2 and another table partitioned with PARTITION BY HASH (*column*) PARTITIONS 10. In the table with two partititions, the partitions are five times as large as the table with ten partitions. Thus, parallel query is more likely to be used for queries against the table with fewer partitions. In the following example, the table `PART_BIG_PARTITIONS` has two partitions and `PART_SMALL_PARTITIONS` has ten partitions. With identical data, parallel query is more likely to be used for the table with fewer big partitions.

```
mysql> explain select count(*), p_brand from part_big_partitions where p_name is not
 null
```

```
    ->   and p_mfgr in ('Manufacturer#1', 'Manufacturer#3') and p_retailprice > 1000
 group by p_brand;
+----+-------------+---------------------+------------
+-----------------------------------------------------------------------------------------------------
+
| id | select_type | table               | partitions | Extra
                                                                                    |
+----+-------------+---------------------+------------
+-----------------------------------------------------------------------------------------------------
+
|  1 | SIMPLE      | part_big_partitions | p0,p1      | Using where; Using temporary;
 Using parallel query (4 columns, 1 filters, 1 exprs; 0 extra; 1 group-bys, 1 aggrs) |
+----+-------------+---------------------+------------
+-----------------------------------------------------------------------------------------------------
+

mysql> explain select count(*), p_brand from part_small_partitions where p_name is not
 null
    ->   and p_mfgr in ('Manufacturer#1', 'Manufacturer#3') and p_retailprice > 1000
 group by p_brand;
+----+-------------+-----------------------+-------------------------------
+-----------------------------+
| id | select_type | table                 | partitions                    | Extra
                   |
+----+-------------+-----------------------+-------------------------------
+-----------------------------+
|  1 | SIMPLE      | part_small_partitions | p0,p1,p2,p3,p4,p5,p6,p7,p8,p9 | Using
 where; Using temporary |
+----+-------------+-----------------------+-------------------------------
+-----------------------------+
```

Aggregate functions, GROUP BY clauses, and HAVING clauses

Queries involving aggregate functions are often good candidates for parallel query, because they involve scanning large numbers of rows within large tables.

In Aurora MySQL 3, parallel query can optimize aggregate function calls in the select list and the `HAVING` clause.

Before Aurora MySQL 3, aggregate function calls in the select list or the `HAVING` clause aren't pushed down to the storage layer. However, parallel query can still improve the performance of such queries with aggregate functions. It does so by first extracting column values from the raw data pages in parallel at the storage layer. It then transmits those values back to the head node in a compact tuple format instead of as entire data pages. As always, the query requires at least one WHERE predicate for parallel query to be activated.

The following simple examples illustrate the kinds of aggregate queries that can benefit from parallel query. They do so by returning intermediate results in compact form to the head node, filtering nonmatching rows from the intermediate results, or both.

```
mysql> explain select sql_no_cache count(distinct p_brand) from part where p_mfgr =
 'Manufacturer#5';
+----+...+----------------------------------------------------------------------------+
| id |...| Extra                                                                      |
+----+...+----------------------------------------------------------------------------+
|  1 |...| Using where; Using parallel query (2 columns, 1 filters, 0 exprs; 0 extra) |
+----+...+----------------------------------------------------------------------------+

mysql> explain select sql_no_cache p_mfgr from part where p_retailprice > 1000 group by
 p_mfgr having count(*) > 100;
+----+...
+------------------------------------------------------------------------------------------------------------
+
| id |...| Extra
                                |
+----+...
+------------------------------------------------------------------------------------------------------------
+
|  1 |...| Using where; Using temporary; Using filesort; Using parallel query (3
 columns, 0 filters, 1 exprs; 0 extra) |
+----+...
+------------------------------------------------------------------------------------------------------------
+
```

Function calls in WHERE clause

Aurora can apply the parallel query optimization to calls to most built-in functions in the WHERE clause. Parallelizing these function calls offloads some CPU work from the head node. Evaluating the predicate functions in parallel during the earliest query stage helps Aurora minimize the amount of data transmitted and processed during later stages.

Currently, the parallelization doesn't apply to function calls in the select list. Those functions are evaluated by the head node, even if identical function calls appear in the WHERE clause. The original values from relevant columns are included in the tuples transmitted from the storage nodes back to the head node. The head node performs any transformations such as UPPER, CONCATENATE, and so on to produce the final values for the result set.

In the following example, parallel query parallelizes the call to LOWER because it appears in the WHERE clause. Parallel query doesn't affect the calls to SUBSTR and UPPER because they appear in the select list.

```
mysql> explain select sql_no_cache distinct substr(upper(p_name),1,5) from part
    -> where lower(p_name) like '%cornflower%' or lower(p_name) like '%goldenrod%';
+----+...
+---------------------------------------------------------------------------------------------
+
| id |...| Extra
                |
+----+...
+---------------------------------------------------------------------------------------------
+
|  1 |...| Using where; Using temporary; Using parallel query (2 columns, 0 filters, 1
 exprs; 0 extra) |
+----+...
+---------------------------------------------------------------------------------------------
+
```

The same considerations apply to other expressions, such as CASE expressions or LIKE operators. For example, the following example shows that parallel query evaluates the CASE expression and LIKE operators in the WHERE clause.

```
mysql> explain select p_mfgr, p_retailprice from part
    -> where p_retailprice > case p_mfgr
    ->   when 'Manufacturer#1' then 1000
    ->   when 'Manufacturer#2' then 1200
    ->   else 950
    -> end
    -> and p_name like '%vanilla%'
    -> group by p_retailprice;
+----+...
+----------------------------------------------------------------------------------------------
+
| id |...| Extra
                                |
+----+...
+----------------------------------------------------------------------------------------------
+
|  1 |...| Using where; Using temporary; Using filesort; Using parallel query (4
 columns, 0 filters, 2 exprs; 0 extra) |
+----+...
+----------------------------------------------------------------------------------------------
+
```

LIMIT clause

Currently, parallel query isn't used for any query block that includes a `LIMIT` clause. Parallel query might still be used for earlier query phases with `GROUP` by, `ORDER BY`, or joins.

Comparison operators

The optimizer estimates how many rows to scan to evaluate comparison operators, and determines whether to use parallel query based on that estimate.

The first example following shows that an equality comparison against the primary key column can be performed efficiently without parallel query. The second example following shows that a similar comparison against an unindexed column requires scanning millions of rows and therefore can benefit from parallel query.

```
mysql> explain select * from part where p_partkey = 10;
+----+...+------+-------+
| id |...| rows | Extra |
+----+...+------+-------+
|  1 |...|    1 | NULL  |
+----+...+------+-------+

mysql> explain select * from part where p_type = 'LARGE BRUSHED BRASS';
+----+...+----------
+-----------------------------------------------------------------------------+
| id |...| rows     | Extra
          |
+----+...+----------
+-----------------------------------------------------------------------------+
|  1 |...| 20427936 | Using where; Using parallel query (9 columns, 1 filters, 0 exprs;
 0 extra) |
+----+...+----------
+-----------------------------------------------------------------------------+
```

The same considerations apply for not-equals tests and for range comparisons such as less than, greater than or equal to, or BETWEEN. The optimizer estimates the number of rows to scan, and determines whether parallel query is worthwhile based on the overall volume of I/O.

Joins

Join queries with large tables typically involve data-intensive operations that benefit from the parallel query optimization. The comparisons of column values between multiple tables (that is, the join predicates themselves) currently aren't parallelized. However, parallel query can push down some of the internal processing for other join phases, such as constructing the Bloom filter during a hash join. Parallel query can apply to join queries even without a WHERE clause. Therefore, a join query is an exception to the rule that a WHERE clause is required to use parallel query.

Each phase of join processing is evaluated to check if it is eligible for parallel query. If more than one phase can use parallel query, these phases are performed in sequence. Thus, each join query counts as a single parallel query session in terms of concurrency limits.

For example, when a join query includes WHERE predicates to filter the rows from one of the joined tables, that filtering option can use parallel query. As another example, suppose that a join query uses the hash join mechanism, for example to join a big table with a small table. In this case, the table scan to produce the Bloom filter data structure might be able to use parallel query.

Note

Parallel query is typically used for the kinds of resource-intensive queries that benefit from the hash join optimization. The method for turning on the hash join optimization depends on the Aurora MySQL version. For details for each version, see Turning on hash join for parallel query clusters. For information about how to use hash joins effectively, see Optimizing large Aurora MySQL join queries with hash joins.

```
mysql> explain select count(*) from orders join customer where o_custkey = c_custkey;
+-----+...+----------+-------+---------------+-------------+...+-----------
+----------------------------------------------------------------------------------------------
+
| id |...| table    | type  | possible_keys | key         |...| rows      | Extra

              |
+----+...+----------+-------+--------------+-------------+...+-----------
+------------------------------------------------------------------------------------------------------
+
|  1 |...| customer | index | PRIMARY      | c_nationkey |...|  15051972 | Using index

              |
|  1 |...| orders   | ALL   | o_custkey    | NULL        |...| 154545408 | Using join
 buffer (Hash Join Outer table orders); Using parallel query (1 columns, 0 filters, 1
 exprs; 0 extra) |
+----+...+----------+-------+--------------+-------------+...+-----------
+------------------------------------------------------------------------------------------------------
+
```

For a join query that uses the nested loop mechanism, the outermost nested loop block might use parallel query. The use of parallel query depends on the same factors as usual, such as the presence of additional filter conditions in the WHERE clause.

```
mysql> -- Nested loop join with extra filter conditions can use parallel query.
mysql> explain select count(*) from part, partsupp where p_partkey != ps_partkey and
 p_name is not null and ps_availqty > 0;
+----+-------------+----------+...+----------
+-----------------------------------------------------------------------------+
| id | select_type | table    |...| rows     | Extra
                                   |
+----+-------------+----------+...+----------
+-----------------------------------------------------------------------------+
|  1 | SIMPLE      | part     |...| 20427936 | Using where; Using parallel query (2
 columns, 1 filters, 0 exprs; 0 extra) |
|  1 | SIMPLE      | partsupp |...| 78164450 | Using where; Using join buffer (Block
 Nested Loop)                          |
+----+-------------+----------+...+----------
+-----------------------------------------------------------------------------+
```

Subqueries

The outer query block and inner subquery block might each use parallel query, or not. Whether they do is based on the usual characteristics of the table, WHERE clause, and so on, for each block. For example, the following query uses parallel query for the subquery block but not the outer block.

```
mysql> explain select count(*) from part where
```

```
   --> p_partkey < (select max(p_partkey) from part where p_name like '%vanilla%');
+----+-------------+...+----------
+--------------------------------------------------------------------------------+
| id | select_type |...| rows     | Extra
                         |
+----+-------------+...+----------
+--------------------------------------------------------------------------------+
|  1 | PRIMARY     |...|     NULL | Impossible WHERE noticed after reading const tables
                         |
|  2 | SUBQUERY    |...| 20427936 | Using where; Using parallel query (2 columns, 0
 filters, 1 exprs; 0 extra) |
+----+-------------+...+----------
+--------------------------------------------------------------------------------+
```

Currently, correlated subqueries can't use the parallel query optimization.

UNION

Each query block in a UNION query can use parallel query, or not, based on the usual characteristics of the table, WHERE clause, and so on, for each part of the UNION.

```
mysql> explain select p_partkey from part where p_name like '%choco_ate%'
    -> union select p_partkey from part where p_name like '%vanil_a%';
+----+----------------+...+----------
+--------------------------------------------------------------------------------+
| id | select_type    |...| rows     | Extra
                            |
+----+----------------+...+----------
+--------------------------------------------------------------------------------+
|  1 | PRIMARY        |...| 20427936 | Using where; Using parallel query (2 columns, 0
 filters, 1 exprs; 0 extra) |
|  2 | UNION          |...| 20427936 | Using where; Using parallel query (2 columns, 0
 filters, 1 exprs; 0 extra) |
| NULL | UNION RESULT | <union1,2> |...|     NULL | Using temporary
                        |
+----+--------------+...+----------
+--------------------------------------------------------------------------------+
```

> **Note**
>
> Each UNION clause within the query is run sequentially. Even if the query includes multiple stages that all use parallel query, it only runs a single parallel query at any one time.

Therefore, even a complex multistage query only counts as 1 toward the limit of concurrent parallel queries.

Views

The optimizer rewrites any query using a view as a longer query using the underlying tables. Thus, parallel query works the same whether table references are views or real tables. All the same considerations about whether to use parallel query for a query, and which parts are pushed down, apply to the final rewritten query.

For example, the following query plan shows a view definition that usually doesn't use parallel query. When the view is queried with additional `WHERE` clauses, Aurora MySQL uses parallel query.

```
mysql> create view part_view as select * from part;
mysql> explain select count(*) from part_view where p_partkey is not null;
+----+...+----------
+-----------------------------------------------------------------------------+
| id |...| rows     | Extra
          |
+----+...+----------
+-----------------------------------------------------------------------------+
|  1 |...| 20427936 | Using where; Using parallel query (1 columns, 0 filters, 0 exprs;
 1 extra) |
+----+...+----------
+-----------------------------------------------------------------------------+
```

Data manipulation language (DML) statements

The `INSERT` statement can use parallel query for the `SELECT` phase of processing, if the `SELECT` part meets the other conditions for parallel query.

```
mysql> create table part_subset like part;
mysql> explain insert into part_subset select * from part where p_mfgr =
 'Manufacturer#1';
+----+...+----------
+-----------------------------------------------------------------------------+
| id |...| rows     | Extra
          |
+----+...+----------
+-----------------------------------------------------------------------------+
|  1 |...| 20427936 | Using where; Using parallel query (9 columns, 1 filters, 0 exprs;
 0 extra) |
+----+...+----------
+----------------------------------------------------------------------------+
```

> **Note**
>
> Typically, after an INSERT statement, the data for the newly inserted rows is in the buffer pool. Therefore, a table might not be eligible for parallel query immediately after inserting a large number of rows. Later, after the data is evicted from the buffer pool during normal operation, queries against the table might begin using parallel query again.

The CREATE TABLE AS SELECT statement doesn't use parallel query, even if the SELECT portion of the statement would otherwise be eligible for parallel query. The DDL aspect of this statement makes it incompatible with parallel query processing. In contrast, in the INSERT ... SELECT statement, the SELECT portion can use parallel query.

Parallel query is never used for DELETE or UPDATE statements, regardless of the size of the table and predicates in the WHERE clause.

```
mysql> explain delete from part where p_name is not null;
+----+-------------+...+----------+-------------+
| id | select_type |...| rows     | Extra       |
+----+-------------+...+----------+-------------+
|  1 | SIMPLE      |...| 20427936 | Using where |
+----+-------------+...+----------+-------------+
```

Transactions and locking

You can use all the isolation levels on the Aurora primary instance.

On Aurora reader DB instances, parallel query applies to statements performed under the REPEATABLE READ isolation level. Aurora MySQL version 2.09 or higher can also use the READ COMMITTED isolation level on reader DB instances. REPEATABLE READ is the default isolation level for Aurora reader DB instances. To use READ COMMITTED isolation level on reader DB instances requires setting the aurora_read_replica_read_committed configuration option at the session level. The READ COMMITTED isolation level for reader instances complies with SQL standard behavior. However, the isolation is less strict on reader instances than when queries use READ COMMITTED isolation level on the writer instance.

For more information about Aurora isolation levels, especially the differences in READ COMMITTED between writer and reader instances, see Aurora MySQL isolation levels.

After a big transaction is finished, the table statistics might be stale. Such stale statistics might require an ANALYZE TABLE statement before Aurora can accurately estimate the number of rows. A large-scale DML statement might also bring a substantial portion of the table data into the buffer pool. Having this data in the buffer pool can lead to parallel query being chosen less frequently for that table until the data is evicted from the pool.

When your session is inside a long-running transaction (by default, 10 minutes), further queries inside that session don't use parallel query. A timeout can also occur during a single long-running query. This type of timeout might happen if the query runs for longer than the maximum interval (currently 10 minutes) before the parallel query processing starts.

You can reduce the chance of starting long-running transactions accidentally by setting autocommit=1 in mysql sessions where you perform ad hoc (one-time) queries. Even a SELECT statement against a table begins a transaction by creating a read view. A *read view* is a consistent set of data for subsequent queries that remains until the transaction is committed. Be aware of this restriction also when using JDBC or ODBC applications with Aurora, because such applications might run with the autocommit setting turned off.

The following example shows how, with the autocommit setting turned off, running a query against a table creates a read view that implicitly begins a transaction. Queries that are run shortly afterward can still use parallel query. However, after a pause of several minutes, queries are no longer eligible for parallel query. Ending the transaction with COMMIT or ROLLBACK restores parallel query eligibility.

```
mysql> set autocommit=0;

mysql> explain select sql_no_cache count(*) from part where p_retailprice > 10.0;
+----+...+---------
+---------------------------------------------------------------------------+
| id |...| rows    | Extra
         |
+----+...+---------
+---------------------------------------------------------------------------+
|  1 |...| 2976129 | Using where; Using parallel query (1 columns, 1 filters, 0 exprs;
 0 extra) |
+----+...+---------
+-----------------------------------------------------------------------------+

mysql> select sleep(720); explain select sql_no_cache count(*) from part where
 p_retailprice > 10.0;
+------------+
| sleep(720) |
+------------+
|          0 |
+------------+
1 row in set (12 min 0.00 sec)

+----+...+---------+-------------+
| id |...| rows    | Extra       |
+----+...+---------+-------------+
|  1 |...| 2976129 | Using where |
+----+...+---------+-------------+

mysql> commit;

mysql> explain select sql_no_cache count(*) from part where p_retailprice > 10.0;
+----+...+---------
+-----------------------------------------------------------------------------+
| id |...| rows    | Extra
         |
+----+...+---------
+-----------------------------------------------------------------------------+
|  1 |...| 2976129 | Using where; Using parallel query (1 columns, 1 filters, 0 exprs;
 0 extra) |
+----+...+---------
+-----------------------------------------------------------------------------+
```

To see how many times queries weren't eligible for parallel query because they were inside long-running transactions, check the status variable `Aurora_pq_request_not_chosen_long_trx`.

```
mysql> show global status like '%pq%trx%';
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| Aurora_pq_request_not_chosen_long_trx | 4     |
+-------------------------------+-------+
```

Any SELECT statement that acquires locks, such as the SELECT FOR UPDATE or SELECT LOCK IN SHARE MODE syntax, can't use parallel query.

Parallel query can work for a table that is locked by a LOCK TABLES statement.

```
mysql> explain select o_orderpriority, o_shippriority from orders where o_clerk =
 'Clerk#000095055';
+----+...+-----------
+-------------------------------------------------------------------------------+
| id |...| rows      | Extra
           |
+----+...+-----------
+-------------------------------------------------------------------------------+
|  1 |...| 154545408 | Using where; Using parallel query (3 columns, 1 filters, 0
 exprs; 0 extra) |
+----+...+-----------
+-------------------------------------------------------------------------------+

mysql> explain select o_orderpriority, o_shippriority from orders where o_clerk =
 'Clerk#000095055' for update;
+----+...+-----------+-------------+
| id |...| rows      | Extra       |
+----+...+-----------+-------------+
|  1 |...| 154545408 | Using where |
+----+...+-----------+-------------+
```

B-tree indexes

The statistics gathered by the ANALYZE TABLE statement help the optimizer to decide when to use parallel query or index lookups, based on the characteristics of the data for each column. Keep statistics current by running ANALYZE TABLE after DML operations that make substantial changes to the data within a table.

If index lookups can perform a query efficiently without a data-intensive scan, Aurora might use index lookups. Doing so avoids the overhead of parallel query processing. There are also concurrency limits on the number of parallel queries that can run simultaneously on any Aurora DB cluster. Make sure to use best practices for indexing your tables, so that your most frequent and most highly concurrent queries use index lookups.

Full-text search (FTS) indexes

Currently, parallel query isn't used for tables that contain a full-text search index, regardless of whether the query refers to such indexed columns or uses the `MATCH` operator.

Virtual columns

Currently, parallel query isn't used for tables that contain a virtual column, regardless of whether the query refers to any virtual columns.

Built-in caching mechanisms

Aurora includes built-in caching mechanisms, namely the buffer pool and the query cache. The Aurora optimizer chooses between these caching mechanisms and parallel query depending on which one is most effective for a particular query.

When a parallel query filters rows and transforms and extracts column values, data is transmitted back to the head node as tuples rather than as data pages. Therefore, running a parallel query doesn't add any pages to the buffer pool, or evict pages that are already in the buffer pool.

Aurora checks the number of pages of table data that are present in the buffer pool, and what proportion of the table data that number represents. Aurora uses that information to determine whether it is more efficient to use parallel query (and bypass the data in the buffer pool). Alternatively, Aurora might use the nonparallel query processing path, which uses data cached in the buffer pool. Which pages are cached and how data-intensive queries affect caching and eviction depends on configuration settings related to the buffer pool. Therefore, it can be hard to predict whether any particular query uses parallel query, because the choice depends on the ever-changing data within the buffer pool.

Also, Aurora imposes concurrency limits on parallel queries. Because not every query uses parallel query, tables that are accessed by multiple queries simultaneously typically have a substantial portion of their data in the buffer pool. Therefore, Aurora often doesn't choose these tables for parallel queries.

When you run a sequence of nonparallel queries on the same table, the first query might be slow due to the data not being in the buffer pool. Then the second and subsequent queries are much faster because the buffer pool is now "warmed up". Parallel queries typically show consistent performance from the very first query against the table. When conducting performance tests, benchmark the nonparallel queries with both a cold and a warm buffer pool. In some cases, the results with a warm buffer pool can compare well to parallel query times. In these cases, consider factors such as the frequency of queries against that table. Also consider whether it is worthwhile to keep the data for that table in the buffer pool.

The query cache avoids rerunning a query when an identical query is submitted and the underlying table data hasn't changed. Queries optimized by parallel query feature can go into the query cache, effectively making them instantaneous when run again.

> **Note**
>
> When conducting performance comparisons, the query cache can produce artificially low timing numbers. Therefore, in benchmark-like situations, you can use the `sql_no_cache` hint. This hint prevents the result from being served from the query cache, even if the same query had been run previously. The hint comes immediately after the `SELECT` statement in a query. Many parallel query examples in this topic include this hint, to make query times comparable between versions of the query for which parallel query is turned on and turned off.
> Make sure that you remove this hint from your source when you move to production use of parallel query.

Optimizer hints

Another way to control the optimizer is by using optimizer hints, which can be specified within individual statements. For example, you can turn on an optimization for one table in a statement, and then turn off the optimization for a different table. For more information about these hints, see Optimizer Hints in the *MySQL Reference Manual*.

You can use SQL hints with Aurora MySQL queries to fine-tune performance. You can also use hints to prevent execution plans for important queries from changing because of unpredictable conditions.

We have extended the SQL hints feature to help you control optimizer choices for your query plans. These hints apply to queries that use parallel query optimization. For more information, see Aurora MySQL hints.

MyISAM temporary tables

The parallel query optimization only applies to InnoDB tables. Because Aurora MySQL uses MyISAM behind the scenes for temporary tables, internal query phases involving temporary tables never use parallel query. These query phases are indicated by `Using temporary` in the `EXPLAIN` output.

Using Advanced Auditing with an Amazon Aurora MySQL DB cluster

You can use the high-performance Advanced Auditing feature in Amazon Aurora MySQL to audit database activity. To do so, you enable the collection of audit logs by setting several DB cluster parameters. When Advanced Auditing is enabled, you can use it to log any combination of supported events.

You can view or download the audit logs to review the audit information for one DB instance at a time. To do so, you can use the procedures in Monitoring Amazon Aurora log files.

> **Tip**
>
> For an Aurora DB cluster containing multiple DB instances, you might find it more convenient to examine the audit logs for all instances in the cluster. To do so, you can use CloudWatch Logs. You can turn on a setting at the cluster level to publish the Aurora MySQL audit log data to a log group in CloudWatch. Then you can view, filter, and search the audit logs through the CloudWatch interface. For more information, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.

Enabling Advanced Auditing

Use the parameters described in this section to enable and configure Advanced Auditing for your DB cluster.

Use the `server_audit_logging` parameter to enable or disable Advanced Auditing.

Use the `server_audit_events` parameter to specify what events to log.

Use the `server_audit_incl_users` and `server_audit_excl_users` parameters to specify who gets audited. By default, all users are audited. For details about how these parameters work when one or both are left empty, or the same user names are specified in both, see server_audit_incl_users and server_audit_excl_users.

Configure Advanced Auditing by setting these parameters in the parameter group used by your DB cluster. You can use the procedure shown in Modifying parameters in a DB parameter group to modify DB cluster parameters using the AWS Management Console. You can use the modify-db-cluster-parameter-group AWS CLI command or the ModifyDBClusterParameterGroup Amazon RDS API operation to modify DB cluster parameters programmatically.

Modifying these parameters doesn't require a DB cluster restart when the parameter group is already associated with your cluster. When you associate the parameter group with the cluster for the first time, a cluster restart is required.

Topics

- server_audit_logging
- server_audit_events
- server_audit_incl_users
- server_audit_excl_users

server_audit_logging

Enables or disables Advanced Auditing. This parameter defaults to OFF; set it to ON to enable Advanced Auditing.

No audit data appears in the logs unless you also define one or more types of events to audit using the `server_audit_events` parameter.

To confirm that audit data is logged for a DB instance, check that some log files for that instance have names of the form `audit/audit.log.`*`other_identifying_information`*. To see the names of the log files, follow the procedure in Viewing and listing database log files.

server_audit_events

Contains the comma-delimited list of events to log. Events must be specified in all caps, and there should be no white space between the list elements, for example: `CONNECT,QUERY_DDL`. This parameter defaults to an empty string.

You can log any combination of the following events:

- CONNECT – Logs both successful and failed connections and also disconnections. This event includes user information.
- QUERY – Logs all queries in plain text, including queries that fail due to syntax or permission errors.

Tip

With this event type turned on, the audit data includes information about the continuous monitoring and health-checking information that Aurora does automatically. If you are only interested in particular kinds of operations, you can use the more specific kinds of events. You can also use the CloudWatch interface to search in the logs for events related to specific databases, tables, or users.

- QUERY_DCL – Similar to the QUERY event, but returns only data control language (DCL) queries (GRANT, REVOKE, and so on).
- QUERY_DDL – Similar to the QUERY event, but returns only data definition language (DDL) queries (CREATE, ALTER, and so on).
- QUERY_DML – Similar to the QUERY event, but returns only data manipulation language (DML) queries (INSERT, UPDATE, and so on, and also SELECT).
- TABLE – Logs the tables that were affected by query execution.

Note

There's no filter in Aurora that excludes certain queries from audit logs. To exclude `SELECT` queries, you must exclude all DML statements.
If a certain user is reporting these internal `SELECT` queries in the audit logs, then you can exclude that user by setting the server_audit_excl_users DB cluster parameter. However, if that user is also used in other activities and can't be omitted, then there is no other option for excluding `SELECT` queries.

server_audit_incl_users

Contains the comma-delimited list of user names for users whose activity is logged. There should be no white space between the list elements, for example: `user_3,user_4`. This parameter defaults to an empty string. The maximum length is 1024 characters. Specified user names must match corresponding values in the `User` column of the `mysql.user` table. For more information about user names, see Account User Names and Passwords in the MySQL documentation.

If `server_audit_incl_users` and `server_audit_excl_users` are both empty (the default), all users are audited.

If you add users to `server_audit_incl_users` and leave `server_audit_excl_users` empty, then only those users are audited.

If you add users to `server_audit_excl_users` and leave `server_audit_incl_users` empty, then all users are audited, except for those listed in `server_audit_excl_users`.

If you add the same users to both `server_audit_excl_users` and `server_audit_incl_users`, then those users are audited. When the same user is listed in both settings, `server_audit_incl_users` is given higher priority.

Connect and disconnect events aren't affected by this variable; they are always logged if specified. A user is logged even if that user is also specified in the `server_audit_excl_users` parameter, because `server_audit_incl_users` has higher priority.

server_audit_excl_users

Contains the comma-delimited list of user names for users whose activity isn't logged. There should be no white space between the list elements, for example: `rdsadmin,user_1,user_2`. This parameter defaults to an empty string. The maximum length is 1024 characters. Specified user names must match corresponding values in the `User` column of the `mysql.user` table. For more information about user names, see Account User Names and Passwords in the MySQL documentation.

If `server_audit_incl_users` and `server_audit_excl_users` are both empty (the default), all users are audited.

If you add users to `server_audit_excl_users` and leave `server_audit_incl_users` empty, then only those users that you list in `server_audit_excl_users` are not audited, and all other users are.

If you add the same users to both `server_audit_excl_users` and `server_audit_incl_users`, then those users are audited. When the same user is listed in both settings, `server_audit_incl_users` is given higher priority.

Connect and disconnect events aren't affected by this variable; they are always logged if specified. A user is logged if that user is also specified in the `server_audit_incl_users` parameter, because that setting has higher priority than `server_audit_excl_users`.

Viewing audit logs

You can view and download the audit logs by using the console. On the **Databases** page, choose the DB instance to show its details, then scroll to the **Logs** section. The audit logs produced by the Advanced Auditing feature have names of the form `audit/audit.log.`*`other_identifying_information`*.

To download a log file, choose that file in the **Logs** section and then choose **Download**.

You can also get a list of the log files by using the describe-db-log-files AWS CLI command. You can download the contents of a log file by using the download-db-log-file-portion AWS CLI command. For more information, see Viewing and listing database log files and Downloading a database log file.

Audit log details

Log files are represented as comma-separated variable (CSV) files in UTF-8 format. Queries are also wrapped in single quotes (').

The audit log is stored separately on the local storage of each instance. Each Aurora instance distributes writes across four log files at a time. The maximum size of the logs is 100 MB in aggregate. When this non-configurable limit is reached, Aurora rotates the files and generates four new files.

Tip

Log file entries are not in sequential order. To order the entries, use the timestamp value. To see the latest events, you might have to review all log files. For more flexibility in sorting and searching the log data, turn on the setting to upload the audit logs to CloudWatch and view them using the CloudWatch interface.
To view audit data with more types of fields and with output in JSON format, you can also use the Database Activity Streams feature. For more information, see Monitoring Amazon Aurora with Database Activity Streams.

The audit log files include the following comma-delimited information in rows, in the specified order:

Field	Description
timestamp	The Unix time stamp for the logged event with microsecond precision.
serverhost	The name of the instance that the event is logged for.
username	The connected user name of the user.
host	The host that the user connected from.
connectionid	The connection ID number for the logged operation.
queryid	The query ID number, which can be used for finding the relational table events and related queries. For `TABLE` events, multiple lines are added.
operation	The recorded action type. Possible values are: `CONNECT`, `QUERY`, `READ`, `WRITE`, `CREATE`, `ALTER`, `RENAME`, and `DROP`.
database	The active database, as set by the `USE` command.
object	For `QUERY` events, this value indicates the query that the database performed. For `TABLE` events, it indicates the table name.
retcode	The return code of the logged operation.

Replication with Amazon Aurora MySQL

The Aurora MySQL replication features are key to the high availability and performance of your cluster. Aurora makes it easy to create or resize clusters with up to 15 Aurora Replicas.

All the replicas work from the same underlying data. If some database instances go offline, others remain available to continue processing queries or to take over as the writer if needed. Aurora automatically spreads your read-only connections across multiple database instances, helping an Aurora cluster to support query-intensive workloads.

In the following topics, you can find information about how Aurora MySQL replication works and how to fine-tune replication settings for best availability and performance.

Topics

Using Aurora Replicas

Aurora Replicas are independent endpoints in an Aurora DB cluster, best used for scaling read operations and increasing availability. Up to 15 Aurora Replicas can be distributed across the Availability Zones that a DB cluster spans within an AWS Region. Although the DB cluster volume is made up of multiple copies of the data for the DB cluster, the data in the cluster volume is represented as a single, logical volume to the primary instance and to Aurora Replicas in the DB cluster. For more information about Aurora Replicas, see Aurora Replicas.

Aurora Replicas work well for read scaling because they are fully dedicated to read operations on your cluster volume. Write operations are managed by the primary instance. Because the cluster volume is shared among all instances in your Aurora MySQL DB cluster, no additional work is required to replicate a copy of the data for each Aurora Replica. In contrast, MySQL read replicas must replay, on a single thread, all write operations from the source DB instance to their local data store. This limitation can affect the ability of MySQL read replicas to support large volumes of read traffic.

With Aurora MySQL, when an Aurora Replica is deleted, its instance endpoint is removed immediately, and the Aurora Replica is removed from the reader endpoint. If there are statements running on the Aurora Replica that is being deleted, there is a three minute grace period. Existing statements can finish gracefully during the grace period. When the grace period ends, the Aurora Replica is shut down and deleted.

⚠ **Important**

Aurora Replicas for Aurora MySQL always use the `REPEATABLE READ` default transaction isolation level for operations on InnoDB tables. You can use the `SET TRANSACTION ISOLATION LEVEL` command to change the transaction level only for the primary instance of an Aurora MySQL DB cluster. This restriction avoids user-level locks on Aurora Replicas, and allows Aurora Replicas to scale to support thousands of active user connections while still keeping replica lag to a minimum.

ⓘ **Note**

DDL statements that run on the primary instance might interrupt database connections on the associated Aurora Replicas. If an Aurora Replica connection is actively using a database object, such as a table, and that object is modified on the primary instance using a DDL statement, the Aurora Replica connection is interrupted.

ⓘ **Note**

The China (Ningxia) Region does not support cross-Region read replicas.

Replication options for Amazon Aurora MySQL

You can set up replication between any of the following options:

- Two Aurora MySQL DB clusters in different AWS Regions, by creating a cross-Region read replica of an Aurora MySQL DB cluster.

  For more information, see Replicating Amazon Aurora MySQL DB clusters across AWS Regions.
- Two Aurora MySQL DB clusters in the same AWS Region, by using MySQL binary log (binlog) replication.

  For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).
- An RDS for MySQL DB instance as the source and an Aurora MySQL DB cluster, by creating an Aurora read replica of an RDS for MySQL DB instance.

  You can use this approach to bring existing and ongoing data changes into Aurora MySQL during migration to Aurora. For more information, see Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica.

  You can also use this approach to increase the scalability of read queries for your data. You do so by querying the data using one or more DB instances within a read-only Aurora MySQL cluster. For more information, see Using Amazon Aurora to scale reads for your MySQL database.
- An Aurora MySQL DB cluster in one AWS Region and up to five Aurora read-only Aurora MySQL DB clusters in different Regions, by creating an Aurora global database.

  You can use an Aurora global database to support applications with a world-wide footprint. The primary Aurora MySQL DB cluster has a Writer instance and up to 15 Aurora Replicas. The read-only secondary Aurora MySQL DB clusters can each be made up of as many as 16 Aurora Replicas. For more information, see Using Amazon Aurora global databases.

Note

Rebooting the primary instance of an Amazon Aurora DB cluster also automatically reboots the Aurora Replicas for that DB cluster, to re-establish an entry point that guarantees read/write consistency across the DB cluster.

Performance considerations for Amazon Aurora MySQL replication

The following features help you to fine-tune the performance of Aurora MySQL replication.

The replica log compression feature automatically reduces network bandwidth for replication messages. Because each message is transmitted to all Aurora Replicas, the benefits are greater for larger clusters. This feature involves some CPU overhead on the writer node to perform the compression. It's always enabled in Aurora MySQL version 2 and version 3.

The binlog filtering feature automatically reduces network bandwidth for replication messages. Because the Aurora Replicas don't use the binlog information that is included in the replication messages, that data is omitted from the messages sent to those nodes.

In Aurora MySQL version 2, you can control this feature by changing the `aurora_enable_repl_bin_log_filtering` parameter. This parameter is on by default. Because this optimization is intended to be transparent, you might turn off this setting only during diagnosis or troubleshooting for issues related to replication. For example, you can do so to match the behavior of an older Aurora MySQL cluster where this feature was not available.

Binlog filtering is always enabled in Aurora MySQL version 3.

Zero-downtime restart (ZDR) for Amazon Aurora MySQL

The zero-downtime restart (ZDR) feature can preserve some or all of the active connections to DB instances during certain kinds of restarts. ZDR applies to restarts that Aurora performs automatically to resolve error conditions, for example when a replica begins to lag too far behind the source.

> ⚠ **Important**
>
> The ZDR mechanism operates on a best-effort basis. The Aurora MySQL versions, instance classes, error conditions, compatible SQL operations, and other factors that determine where ZDR applies are subject to change at any time.

ZDR for Aurora MySQL 2.x requires version 2.10 and higher. ZDR is available in all minor versions of Aurora MySQL 3.x. In Aurora MySQL version 2 and 3, the ZDR mechanism is turned on by default and Aurora doesn't use the `aurora_enable_zdr` parameter.

Aurora reports on the **Events** page activities related to zero-downtime restart. Aurora records an event when it attempts a restart using the ZDR mechanism. This event states why Aurora performs the restart. Then Aurora records another event when the restart finishes. This final event reports how long the process took, and how many connections were preserved or dropped during the restart. You can consult the database error log to see more details about what happened during the restart.

Although connections remain intact following a successful ZDR operation, some variables and features are reinitialized. The following kinds of information aren't preserved through a restart caused by zero-downtime restart:

- Global variables. Aurora restores session variables, but it doesn't restore global variables after the restart.
- Status variables. In particular, the uptime value reported by the engine status is reset.
- `LAST_INSERT_ID`.
- In-memory `auto_increment` state for tables. The in-memory auto-increment state is reinitialized. For more information about auto-increment values, see MySQL Reference Manual.
- Diagnostic information from `INFORMATION_SCHEMA` and `PERFORMANCE_SCHEMA` tables. This diagnostic information also appears in the output of commands such as `SHOW PROFILE` and `SHOW PROFILES`.

The following table shows the versions, instance roles, and other circumstances that determine whether Aurora can use the ZDR mechanism when restarting DB instances in your cluster.

Aurora MySQL version	ZDR applies to the writer?	ZDR applies to readers?	ZDR always enabled?	Notes
2.x, lower than 2.10.0	No	No	N/A	ZDR isn't available for these versions.
2.10.0–2.11.0	Yes	Yes	Yes	Aurora rolls back any transactions that are in progress on active connections. Your applicati on must retry the transactions.

Aurora MySQL version	ZDR applies to the writer?	ZDR applies to readers?	ZDR always enabled?	Notes
				Aurora cancels any connections that use TLS/SSL, temporary tables, table locks, or user locks.
2.11.1 and higher	Yes	Yes	Yes	Aurora rolls back any transactions that are in progress on active connections. Your application must retry the transactions. Aurora cancels any connections that use temporary tables, table locks, or user locks.
3.01–3.03	Yes	Yes	Yes	Aurora rolls back any transactions that are in progress on active connections. Your application must retry the transactions. Aurora cancels any connections that use TLS/SSL, temporary tables, table locks, or user locks.
3.04 and higher	Yes	Yes	Yes	Aurora rolls back any transactions that are in progress on active connections. Your application must retry the transactions. Aurora cancels any connections that use temporary tables, table locks, or user locks.

Configuring replication filters with Aurora MySQL

You can use replication filters to specify which databases and tables are replicated with a read replica. Replication filters can include databases and tables in replication or exclude them from replication.

The following are some use cases for replication filters:

- To reduce the size of a read replica. With replication filtering, you can exclude the databases and tables that aren't needed on the read replica.
- To exclude databases and tables from read replicas for security reasons.
- To replicate different databases and tables for specific use cases at different read replicas. For example, you might use specific read replicas for analytics or sharding.
- For a DB cluster that has read replicas in different AWS Regions, to replicate different databases or tables in different AWS Regions.
- To specify which databases and tables are replicated with an Aurora MySQL DB cluster that is configured as a replica in an inbound replication topology. For more information about this configuration, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

Topics

Setting replication filtering parameters for Aurora MySQL

To configure replication filters, set the following parameters:

- `binlog-do-db` – Replicate changes to the specified binary logs. When you set this parameter for a binlog source cluster, only the binary logs specified in the parameter are replicated.
- `binlog-ignore-db` – Don't replicate changes to the specified binary logs. When the `binlog-do-db` parameter is set for a binlog source cluster, this parameter isn't evaluated.
- `replicate-do-db` – Replicate changes to the specified databases. When you set this parameter for a binlog replica cluster, only the databases specified in the parameter are replicated.
- `replicate-ignore-db` – Don't replicate changes to the specified databases. When the `replicate-do-db` parameter is set for a binlog replica cluster, this parameter isn't evaluated.
- `replicate-do-table` – Replicate changes to the specified tables. When you set this parameter for a read replica, only the tables specified in the parameter are replicated. Also, when the `replicate-do-db` or `replicate-ignore-db` parameter is set, make sure to include the database that includes the specified tables in replication with the binlog replica cluster.

- `replicate-ignore-table` – Don't replicate changes to the specified tables. When the `replicate-do-table` parameter is set for a binlog replica cluster, this parameter isn't evaluated.
- `replicate-wild-do-table` – Replicate tables based on the specified database and table name patterns. The `%` and `_` wildcard characters are supported. When the `replicate-do-db` or `replicate-ignore-db` parameter is set, make sure to include the database that includes the specified tables in replication with the binlog replica cluster.
- `replicate-wild-ignore-table` – Don't replicate tables based on the specified database and table name patterns. The `%` and `_` wildcard characters are supported. When the `replicate-do-table` or `replicate-wild-do-table` parameter is set for a binlog replica cluster, this parameter isn't evaluated.

The parameters are evaluated in the order that they are listed. For more information about how these parameters work, see the MySQL documentation:

- For general information, see Replica Server Options and Variables.
- For information about how database replication filtering parameters are evaluated, see Evaluation of Database-Level Replication and Binary Logging Options.
- For information about how table replication filtering parameters are evaluated, see Evaluation of Table-Level Replication Options.

By default, each of these parameters has an empty value. On each binlog cluster, you can use these parameters to set, change, and delete replication filters. When you set one of these parameters, separate each filter from others with a comma.

You can use the `%` and `_` wildcard characters in the `replicate-wild-do-table` and `replicate-wild-ignore-table` parameters. The `%` wildcard matches any number of characters, and the `_` wildcard matches only one character.

The binary logging format of the source DB instance is important for replication because it determines the record of data changes. The setting of the `binlog_format` parameter determines whether the replication is row-based or statement-based. For more information, see Configuring Aurora MySQL binary logging.

Note

All data definition language (DDL) statements are replicated as statements, regardless of the binlog_format setting on the source DB instance.

Replication filtering limitations for Aurora MySQL

The following limitations apply to replication filtering for Aurora MySQL:

- Replication filters are supported only for Aurora MySQL version 3.
- Each replication filtering parameter has a 2,000-character limit.
- Commas aren't supported in replication filters.
- Replication filtering doesn't support XA transactions.

  For more information, see Restrictions on XA Transactions in the MySQL documentation.

Replication filtering examples for Aurora MySQL

To configure replication filtering for a read replica, modify the replication filtering parameters in the DB cluster parameter group associated with the read replica.

Note

You can't modify a default DB cluster parameter group. If the read replica is using a default parameter group, create a new parameter group and associate it with the read replica. For more information on DB cluster parameter groups, see Working with parameter groups.

You can set parameters in a DB cluster parameter group using the AWS Management Console, AWS CLI, or RDS API. For information about setting parameters, see Modifying parameters in a DB parameter group. When you set parameters in a DB cluster parameter group, all of the DB clusters associated with the parameter group use the parameter settings. If you set the replication filtering parameters in a DB cluster parameter group, make sure that the parameter group is associated only with read replica clusters. Leave the replication filtering parameters empty for source DB instances.

The following examples set the parameters using the AWS CLI. These examples set ApplyMethod to immediate so that the parameter changes occur immediately after the CLI command completes. If you want a pending change to be applied after the read replica is rebooted, set ApplyMethod to pending-reboot.

The following examples set replication filters:

- Including databases in replication
- Including tables in replication
- Including tables in replication with wildcard characters
- Excluding databases from replication
- Excluding tables from replication
- Excluding tables from replication using wildcard characters

Example Including databases in replication

The following example includes the mydb1 and mydb2 databases in replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-do-
db,ParameterValue='mydb1,mydb2',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-do-
db,ParameterValue='mydb1,mydb2',ApplyMethod=immediate"
```

Example Including tables in replication

The following example includes the table1 and table2 tables in database mydb1 in replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-do-
table,ParameterValue='mydb1.table1,mydb1.table2',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-do-
table,ParameterValue='mydb1.table1,mydb1.table2',ApplyMethod=immediate"
```

Example Including tables in replication using wildcard characters

The following example includes tables with names that begin with `order` and `return` in database `mydb` in replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-wild-do-table,ParameterValue='mydb.order
%,mydb.return%',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-wild-do-table,ParameterValue='mydb.order
%,mydb.return%',ApplyMethod=immediate"
```

Example Excluding databases from replication

The following example excludes the `mydb5` and `mydb6` databases from replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-ignore-
db,ParameterValue='mydb5,mydb6',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-ignore-
db,ParameterValue='mydb5,mydb6,ApplyMethod=immediate"
```

Example Excluding tables from replication

The following example excludes tables `table1` in database `mydb5` and `table2` in database `mydb6` from replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-ignore-
table,ParameterValue='mydb5.table1,mydb6.table2',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-ignore-
table,ParameterValue='mydb5.table1,mydb6.table2',ApplyMethod=immediate"
```

Example Excluding tables from replication using wildcard characters

The following example excludes tables with names that begin with `order` and `return` in database `mydb7` from replication.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=replicate-wild-ignore-table,ParameterValue='mydb7.order
%,mydb7.return%',ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
  --db-cluster-parameter-group-name myparametergroup ^
  --parameters "ParameterName=replicate-wild-ignore-table,ParameterValue='mydb7.order
%,mydb7.return%',ApplyMethod=immediate"
```

Viewing the replication filters for a read replica

You can view the replication filters for a read replica in the following ways:

- Check the settings of the replication filtering parameters in the parameter group associated with the read replica.

  For instructions, see Viewing parameter values for a DB parameter group.
- In a MySQL client, connect to the read replica and run the `SHOW REPLICA STATUS` statement.

  In the output, the following fields show the replication filters for the read replica:
  - `Binlog_Do_DB`
  - `Binlog_Ignore_DB`
  - `Replicate_Do_DB`
  - `Replicate_Ignore_DB`
  - `Replicate_Do_Table`
  - `Replicate_Ignore_Table`
  - `Replicate_Wild_Do_Table`
  - `Replicate_Wild_Ignore_Table`

  For more information about these fields, see Checking Replication Status in the MySQL documentation.

Monitoring Amazon Aurora MySQL replication

Read scaling and high availability depend on minimal lag time. You can monitor how far an Aurora Replica is lagging behind the primary instance of your Aurora MySQL DB cluster by monitoring the

Amazon CloudWatch `AuroraReplicaLag` metric. The `AuroraReplicaLag` metric is recorded in each Aurora Replica.

The primary DB instance also records the `AuroraReplicaLagMaximum` and `AuroraReplicaLagMinimum` Amazon CloudWatch metrics. The `AuroraReplicaLagMaximum` metric records the maximum amount of lag between the primary DB instance and each Aurora Replica in the DB cluster. The `AuroraReplicaLagMinimum` metric records the minimum amount of lag between the primary DB instance and each Aurora Replica in the DB cluster.

If you need the most current value for Aurora Replica lag, you can check the `AuroraReplicaLag` metric in Amazon CloudWatch. The Aurora Replica lag is also recorded on each Aurora Replica of your Aurora MySQL DB cluster in the `information_schema.replica_host_status` table. For more information on this table, see information_schema.replica_host_status.

For more information on monitoring RDS instances and CloudWatch metrics, see Monitoring metrics in an Amazon Aurora cluster.

Using local write forwarding in an Amazon Aurora MySQL DB cluster

Local (in-cluster) write forwarding allows your applications to issue read/write transactions directly on an Aurora Replica. These transactions are then forwarded to the writer DB instance to be committed. You can use local write forwarding when your applications require *read-after-write consistency*, which is the ability to read the latest write in a transaction.

Read replicas receive updates asynchronously from the writer. Without write forwarding, you have to transact any reads that require read-after-write consistency on the writer DB instance. Or you have to develop complex custom application logic to take advantage of multiple read replicas for scalability. Your applications must fully split all read and write traffic, maintaining two sets of database connections to send the traffic to the correct endpoint. This development overhead complicates application design when the queries are part of a single logical session, or transaction, within the application. Moreover, because replication lag can differ among read replicas, it's difficult to achieve global read consistency across all instances in the database.

Write forwarding avoids the need to split those transactions or send them exclusively to the writer, which simplifies application development. This new capability makes it easy to achieve read scale for workloads that need to read the latest write in a transaction and aren't sensitive to write latency.

Local write forwarding is different from global write forwarding, which forwards writes from a secondary DB cluster to the primary DB cluster in an Aurora global database. You can use local write forwarding in a DB cluster that is part of an Aurora global database. For more information, see Using write forwarding in an Amazon Aurora global database.

Local write forwarding requires Aurora MySQL version 3.04 or higher.

Topics

- Amazon CloudWatch metrics and Aurora MySQL status variables for write forwarding
- Identifying forwarded transactions and queries

Enabling local write forwarding

By default, local write forwarding isn't enabled for Aurora MySQL DB clusters. You enable local write forwarding at the cluster level, not at the instance level.

> **Important**
>
> You can also enable local write forwarding for cross-Region read replicas that use binary logging, but write operations aren't forwarded to the source AWS Region. They're forwarded to the writer DB instance of the binlog read replica cluster.
> Use this method only if you have a use case for writing to the binlog read replica in the secondary AWS Region. Otherwise, you might end up with a "split-brain" scenario where replicated datasets are inconsistent with each other.
> We recommend that you use global write forwarding with global databases, rather than local write forwarding on cross-Region read replicas, unless absolutely necessary. For more information, see Using write forwarding in an Amazon Aurora global database.

Console

Using the AWS Management Console, select the **Turn on local write forwarding** check box under **Read replica write forwarding** when you create or modify a DB cluster.

AWS CLI

To enable write forwarding with the AWS CLI, use the `--enable-local-write-forwarding` option. This option works when you create a new DB cluster using the `create-db-cluster` command. It also works when you modify an existing DB cluster using the `modify-db-cluster` command. You can disable write forwarding by using the `--no-enable-local-write-forwarding` option with these same CLI commands.

The following example creates an Aurora MySQL DB cluster with write forwarding enabled.

```
aws rds create-db-cluster \
  --db-cluster-identifier write-forwarding-test-cluster \
  --enable-local-write-forwarding \
  --engine aurora-mysql \
  --engine-version 8.0.mysql_aurora.3.04.0 \
  --master-username myuser \
  --master-user-password mypassword \
  --backup-retention 1
```

You then create writer and reader DB instances so that you can use write forwarding. For more information, see Creating an Amazon Aurora DB cluster.

RDS API

To enable write forwarding using the Amazon RDS API, set the EnableLocalWriteForwarding parameter to true. This parameter works when you create a new DB cluster using the CreateDBCluster operation. It also works when you modify an existing DB cluster using the ModifyDBCluster operation. You can disable write forwarding by setting the EnableLocalWriteForwarding parameter to false.

Enabling write forwarding for database sessions

The aurora_replica_read_consistency parameter is a DB parameter and DB cluster parameter that enables write forwarding. You can specify EVENTUAL, SESSION, or GLOBAL for the read consistency level. To learn more about consistency levels, see Read consistency for write forwarding.

The following rules apply to this parameter:

- The default value is '' (null).
- Write forwarding is available only if you set aurora_replica_read_consistency to EVENTUAL, SESSION, or GLOBAL. This parameter is relevant only in reader instances of DB clusters that have write forwarding enabled.
- You can't set this parameter (when empty) or unset it (when already set) inside a multistatement transaction. You can change it from one valid value to another valid value during such a transaction, but we don't recommend this action.

Checking if a DB cluster has write forwarding enabled

To determine that you can use write forwarding in a DB cluster, confirm that the cluster has the attribute LocalWriteForwardingStatus set to enabled.

In the AWS Management Console, on the **Configuration** tab of the details page for the cluster, you see the status **Enabled** for **Local read replica write forwarding**.

To see the status of the write forwarding setting for all of your clusters, run the following AWS CLI command.

Example

```
aws rds describe-db-clusters \
--query '*[].
{DBClusterIdentifier:DBClusterIdentifier,LocalWriteForwardingStatus:LocalWriteForwardingStatus}

[
    {
        "LocalWriteForwardingStatus": "enabled",
        "DBClusterIdentifier": "write-forwarding-test-cluster-1"
    },
    {
        "LocalWriteForwardingStatus": "disabled",
        "DBClusterIdentifier": "write-forwarding-test-cluster-2"
    },
    {
        "LocalWriteForwardingStatus": "requested",
        "DBClusterIdentifier": "test-global-cluster-2"
    },
    {
        "LocalWriteForwardingStatus": "null",
        "DBClusterIdentifier": "aurora-mysql-v2-cluster"
    }
]
```

A DB cluster can have the following values for `LocalWriteForwardingStatus`:

- `disabled` – Write forwarding is disabled.
- `disabling` – Write forwarding is in the process of being disabled.
- `enabled` – Write forwarding is enabled.
- `enabling` – Write forwarding is in the process of being enabled.
- `null` – Write forwarding isn't available for this DB cluster.
- `requested` – Write forwarding has been requested, but is not yet active.

Application and SQL compatibility with write forwarding

You can use the following kinds of SQL statements with write forwarding:

- Data manipulation language (DML) statements, such as INSERT, DELETE, and UPDATE. There are some restrictions on the properties of these statements that you can use with write forwarding, as described following.
- SELECT ... LOCK IN SHARE MODE and SELECT FOR UPDATE statements.
- PREPARE and EXECUTE statements.

Certain statements aren't allowed or can produce stale results when you use them in a DB cluster with write forwarding. Thus, the EnableLocalWriteForwarding setting is disabled by default for DB clusters. Before enabling it, check to make sure that your application code isn't affected by any of these restrictions.

The following restrictions apply to the SQL statements you use with write forwarding. In some cases, you can use the statements on DB clusters with write forwarding enabled. This approach works if write forwarding isn't enabled within the session by the aurora_replica_read_consistency configuration parameter. If you try to use a statement when it's not allowed because of write forwarding, then you will see an error message similar to the following:

```
ERROR 1235 (42000): This version of MySQL doesn't yet support 'operation with write
 forwarding'.
```

Data definition language (DDL)

Connect to the writer DB instance to run DDL statements. You can't run them from reader DB instances.

Updating a permanent table using data from a temporary table

You can use temporary tables on DB clusters with write forwarding enabled. However, you can't use a DML statement to modify a permanent table if the statement refers to a temporary table. For example, you can't use an INSERT ... SELECT statement that takes the data from a temporary table.

XA transactions

You can't use the following statements on a DB cluster when write forwarding is enabled within the session. You can use these statements on DB clusters that don't have write forwarding enabled, or within sessions where the `aurora_replica_read_consistency` setting is empty. Before enabling write forwarding within a session, check if your code uses these statements.

```
XA {START|BEGIN} xid [JOIN|RESUME]
XA END xid [SUSPEND [FOR MIGRATE]]
XA PREPARE xid
XA COMMIT xid [ONE PHASE]
XA ROLLBACK xid
XA RECOVER [CONVERT XID]
```

LOAD statements for permanent tables

You can't use the following statements on a DB cluster with write forwarding enabled.

```
LOAD DATA INFILE 'data.txt' INTO TABLE t1;
LOAD XML LOCAL INFILE 'test.xml' INTO TABLE t1;
```

Plugin statements

You can't use the following statements on a DB cluster with write forwarding enabled.

```
INSTALL PLUGIN example SONAME 'ha_example.so';
UNINSTALL PLUGIN example;
```

SAVEPOINT statements

You can't use the following statements on a DB cluster when write forwarding is enabled within the session. You can use these statements on DB clusters that don't have write forwarding enabled, or within sessions where the `aurora_replica_read_consistency` setting is blank. Check if your code uses these statements before enabling write forwarding within a session.

```
SAVEPOINT t1_save;
ROLLBACK TO SAVEPOINT t1_save;
RELEASE SAVEPOINT t1_save;
```

Isolation levels for write forwarding

In sessions that use write forwarding, you can only use the `REPEATABLE READ` isolation level. Although you can also use the `READ COMMITTED` isolation level with Aurora Replicas, that isolation level doesn't work with write forwarding. For information about the `REPEATABLE READ` and `READ COMMITTED` isolation levels, see Aurora MySQL isolation levels.

Read consistency for write forwarding

You can control the degree of read consistency on a DB cluster. The read consistency level determines how long the DB cluster waits before each read operation to ensure that some or all changes are replicated from the writer. You can adjust the read consistency level to make sure that all forwarded write operations from your session are visible in the DB cluster before any subsequent queries. You can also use this setting to make sure that queries on the DB cluster always see the most current updates from the writer. This setting also applies to queries submitted by other sessions or other clusters. To specify this type of behavior for your application, choose a value for the `aurora_replica_read_consistency` DB parameter or DB cluster parameter.

> ⚠ **Important**
>
> Always set the `aurora_replica_read_consistency` DB parameter or DB cluster parameter when you want to forward writes. If you don't, then Aurora doesn't forward writes. This parameter has an empty value by default, so choose a specific value when you use this parameter. The `aurora_replica_read_consistency` parameter only affects DB clusters or instances that have write forwarding enabled.

As you increase the consistency level, your application spends more time waiting for changes to be propagated between DB instances. You can choose the balance between fast response time and making sure that changes made in other DB instances are fully available before your queries run.

You can specify the following values for the `aurora_replica_read_consistency` parameter:

- `EVENTUAL` – Results of write operations in the same session aren't visible until the write operation is performed on the writer DB instance. The query doesn't wait for the updated results to be available. Thus it might retrieve the older data or the updated data, depending on the timing of the statements and the amount of replication lag. This is the same consistency as for Aurora MySQL DB clusters that don't use write forwarding.

- SESSION – All queries that use write forwarding see the results of all changes made in that session. The changes are visible regardless of whether the transaction is committed. If necessary, the query waits for the results of forwarded write operations to be replicated.
- GLOBAL – A session sees all committed changes across all sessions and instances in the DB cluster. Each query might wait for a period that varies depending on the amount of session lag. The query proceeds when the DB cluster is up-to-date with all committed data from the writer, as of the time that the query began.

For information about the configuration parameters involved in write forwarding, see Configuration parameters for write forwarding.

> **Note**
>
> You can also use aurora_replica_read_consistency as a session variable, for example:
>
> ```
> mysql> set aurora_replica_read_consistency = 'session';
> ```

Examples of using write forwarding

The following examples show the effects of the aurora_replica_read_consistency parameter on running INSERT statements followed by SELECT statements. The results can differ, depending on the value of aurora_replica_read_consistency and the timing of the statements.

To achieve higher consistency, you might wait briefly before issuing the SELECT statement. Or Aurora can automatically wait until the results finish replicating before proceeding with SELECT.

For information on setting DB parameters, see Working with parameter groups.

Example with aurora_replica_read_consistency set to EVENTUAL

Running an INSERT statement, immediately followed by a SELECT statement, returns a value for COUNT(*) with the number of rows before the new row is inserted. Running the SELECT again a short time later returns the updated row count. The SELECT statements don't wait.

```
mysql> select count(*) from t1;
+----------+
```

```
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)

mysql> insert into t1 values (6); select count(*) from t1;
+----------+
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.00 sec)
```

Example with `aurora_replica_read_consistency` set to `SESSION`

A `SELECT` statement immediately after an `INSERT` waits until the changes from the `INSERT` statement are visible. Subsequent `SELECT` statements don't wait.

```
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.01 sec)

mysql> insert into t1 values (6); select count(*) from t1; select count(*) from t1;
Query OK, 1 row affected (0.08 sec)
+----------+
| count(*) |
+----------+
|        7 |
+----------+
1 row in set (0.37 sec)
```

```
+----------+
| count(*) |
+----------+
|        7 |
+----------+
1 row in set (0.00 sec)
```

With the read consistency setting still set to SESSION, introducing a brief wait after performing an INSERT statement makes the updated row count available by the time the next SELECT statement runs.

```
mysql> insert into t1 values (6); select sleep(2); select count(*) from t1;
Query OK, 1 row affected (0.07 sec)
+----------+
| sleep(2) |
+----------+
|        0 |
+----------+
1 row in set (2.01 sec)
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.00 sec)
```

Example with `aurora_replica_read_consistency` set to `GLOBAL`

Each SELECT statement waits for all data changes, as of the start time of the statement, to be visible before performing the query. The wait time for each SELECT statement varies, depending on the amount of replication lag.

```
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.75 sec)

mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.37 sec)

mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.66 sec)
```

Running multipart statements with write forwarding

A DML statement might consist of multiple parts, such as an `INSERT ... SELECT` statement or a `DELETE ... WHERE` statement. In this case, the entire statement is forwarded to the writer DB instance and run there.

Transactions with write forwarding

If the transaction access mode is set to read only, write forwarding isn't used. You can specify the access mode for the transaction by using the `SET TRANSACTION` statement or the `START TRANSACTION` statement. You can also specify the transaction access mode by changing the value of the transaction_read_only session variable. You can change this session value only while you're connected to a DB cluster that has write forwarding enabled.

If a long-running transaction doesn't issue any statement for a substantial period of time, it might exceed the idle timeout period. This period has a default of one minute. You can set the `aurora_fwd_writer_idle_timeout` parameter to increase it up to one day. A transaction that exceeds the idle timeout is canceled by the writer instance. The next subsequent statement you submit receives a timeout error. Then Aurora rolls back the transaction.

This type of error can occur in other cases when write forwarding becomes unavailable. For example, Aurora cancels any transactions that use write forwarding if you restart the DB cluster or if you disable write forwarding.

When a writer instance in a cluster using local write forwarding is restarted, any active, forwarded transactions and queries on reader instances using local write forwarding are automatically closed. After the writer instance is available again, you can retry these transactions.

Configuration parameters for write forwarding

The Aurora DB parameter groups include settings for the write forwarding feature. Details about these parameters are summarized in the following table, with usage notes after the table.

Parameter	Scope	Type	Default value	Valid values
`aurora_fwd_writer_idle_timeout`	Cluster	Unsigned integer	60	1–86,400
`aurora_fwd_writer_max_connections_pct`	Cluster	Unsigned long integer	10	0–90
`aurora_replica_read_consistency`	Cluster or instance	Enum	'' (null)	`EVENTUAL`, `SESSION`, `GLOBAL`

To control incoming write requests, use these settings:

- `aurora_fwd_writer_idle_timeout` – The number of seconds the writer DB instance waits for activity on a connection that's forwarded from a reader instance before closing it. If the session remains idle beyond this period, Aurora cancels the session.
- `aurora_fwd_writer_max_connections_pct` – The upper limit on database connections that can be used on a writer DB instance to handle queries forwarded from reader instances. It's expressed as a percentage of the `max_connections` setting for the writer. For example, if `max_connections` is 800 and `aurora_fwd_master_max_connections_pct` or `aurora_fwd_writer_max_connections_pct` is 10, then the writer allows a maximum of 80 simultaneous forwarded sessions. These connections come from the same connection pool managed by the `max_connections` setting.

  This setting applies only on the writer when it has write forwarding enabled. If you decrease the value, existing connections aren't affected. Aurora takes the new value of the setting into account when attempting to create a new connection from a DB cluster. The default value is 10, representing 10% of the `max_connections` value.

Note

Because `aurora_fwd_writer_idle_timeout` and `aurora_fwd_writer_max_connections_pct` are DB cluster parameters, all DB instances in each cluster have the same values for these parameters.

For more information about `aurora_replica_read_consistency`, see Read consistency for write forwarding.

For more information on DB parameter groups, see Working with parameter groups.

Amazon CloudWatch metrics and Aurora MySQL status variables for write forwarding

The following Amazon CloudWatch metrics and Aurora MySQL status variables apply when you use write forwarding on one or more DB clusters. These metrics and status variables are all measured on the writer DB instance.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
`ForwardingWriterDMLLatency`	–	Milliseconds	Average time to process each forwarded DML statement on the writer DB instance. It doesn't include the time for the DB cluster to forward the write request, or the time to replicate changes back to the writer.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
ForwardingWriterDMLThroughput	–	Count per second	Number of forwarded DML statements processed each second by this writer DB instance.
ForwardingWriterOpenSessions	Aurora_fwd_writer_open_sessions	Count	Number of forwarded sessions on the writer DB instance.
–	Aurora_fwd_writer_dml_stmt_count	Count	Total number of DML statements forwarded to this writer DB instance.
–	Aurora_fwd_writer_dml_stmt_duration	Microseconds	Total duration of DML statements forwarded to this writer DB instance.
–	Aurora_fwd_writer_select_stmt_count	Count	Total number of SELECT statements forwarded to this writer DB instance.
–	Aurora_fwd_writer_select_stmt_duration	Microseconds	Total duration of SELECT statements forwarded to this writer DB instance.

The following CloudWatch metrics and Aurora MySQL status variables are measured on each reader DB instance in a DB cluster with write forwarding enabled.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
`ForwardingReplicaDMLLatency`	–	Milliseconds	Average response time of forwarded DMLs on replica.
`ForwardingReplicaDMLThroughput`	–	Count per second	Number of forwarded DML statements processed each second.
`ForwardingReplicaOpenSessions`	`Aurora_fwd_replica_open_sessions`	Count	Number of sessions that are using write forwarding on a reader DB instance.
`ForwardingReplicaReadWaitLatency`	–	Milliseconds	Average wait time that a `SELECT` statement on a reader DB instance waits to catch up to the writer. The degree to which the reader DB instance waits before processing a query depends on the `aurora_replica_read_consistency` setting.
`ForwardingReplicaReadWaitThroughput`	–	Count per second	Total number of `SELECT` statements processed each second in all sessions

CloudWatch metric	Aurora MySQL status variable	Unit	Description
			that are forwarding writes.
`ForwardingReplicaSelectLatency`	–	Milliseconds	Forwarded SELECT latency, averaged over all forwarded SELECT statements within the monitoring period.
`ForwardingReplicaSelectThroughput`	–	Count per second	Forwarded SELECT throughput per second average within the monitoring period.
–	`Aurora_fwd_replica_dml_stmt_count`	Count	Total number of DML statements forwarded from this reader DB instance.
–	`Aurora_fwd_replica_dml_stmt_duration`	Microseconds	Total duration of all DML statements forwarded from this reader DB instance.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
–	Aurora_fwd_replica_errors_session_limit	Count	Number of sessions rejected by the primary cluster due to one of the following error conditions: • writer full • Too many forwarded statements in progress.
–	Aurora_fwd_replica_read_wait_count	Count	Total number of read-after-write waits on this reader DB instance.
–	Aurora_fwd_replica_read_wait_duration	Microseconds	Total duration of waits due to the read consistency setting on this reader DB instance.
–	Aurora_fwd_replica_select_stmt_count	Count	Total number of SELECT statements forwarded from this reader DB instance.
–	Aurora_fwd_replica_select_stmt_duration	Microseconds	Total duration of SELECT statements forwarded from this reader DB instance.

Identifying forwarded transactions and queries

You can use the information_schema.aurora_forwarding_processlist table to identify forwarded transactions and queries. For more information on this table, see information_schema.aurora_forwarding_processlist.

The following example shows all forwarded connections on a writer DB instance.

```
mysql> select * from information_schema.AURORA_FORWARDING_PROCESSLIST where
 IS_FORWARDED=1 order by REPLICA_SESSION_ID;

+-----+----------+--------------------+----------+---------+------+--------------
+-----------------------------------------+--------------+--------------------
+--------------------------------+---------------------+----------------+
| ID  | USER     | HOST               | DB       | COMMAND | TIME | STATE        |
 INFO                                        | IS_FORWARDED | REPLICA_SESSION_ID |
 REPLICA_INSTANCE_IDENTIFIER     | REPLICA_CLUSTER_NAME | REPLICA_REGION |
+-----+----------+--------------------+----------+---------+------+--------------
+-----------------------------------------+--------------+--------------------
+--------------------------------+-------------------------------------+
| 648 | myuser   | IP_address:port1   | sysbench | Query   |    0 | async commit |
 UPDATE sbtest58 SET k=k+1 WHERE id=4802579 |            1 |                637 | my-
db-cluster-instance-2        | my-db-cluster        | us-west-2      |
| 650 | myuser   | IP_address:port2   | sysbench | Query   |    0 | async commit |
 UPDATE sbtest54 SET k=k+1 WHERE id=2503953 |            1 |                639 | my-
db-cluster-instance-2        | my-db-cluster        | us-west-2      |
+-----+----------+--------------------+----------+---------+------+--------------
+-----------------------------------------+--------------+--------------------
+--------------------------------+---------------------+----------------+
```

On the forwarding reader DB instance, you can see the threads associated with these writer DB connections by running SHOW PROCESSLIST. The REPLICA_SESSION_ID values on the writer, 637 and 639, are the same as the Id values on the reader.

```
mysql> select @@aurora_server_id;

+---------------------------------+
| @@aurora_server_id              |
+---------------------------------+
| my-db-cluster-instance-2        |
+---------------------------------+
1 row in set (0.00 sec)
```

```
mysql> show processlist;

+-----+----------+--------------------+----------+---------+------+--------------
+---------------------------------------------+
| Id  | User     | Host               | db       | Command | Time | State         | Info
                                        |
+-----+----------+--------------------+----------+---------+------+--------------
+---------------------------------------------+
| 637 | myuser   | IP_address:port1   | sysbench | Query   |    0 | async commit |
 UPDATE sbtest12 SET k=k+1 WHERE id=4802579  |
| 639 | myuser   | IP_address:port2   | sysbench | Query   |    0 | async commit |
 UPDATE sbtest61 SET k=k+1 WHERE id=2503953  |
+-----+----------+--------------------+----------+---------+------+--------------
+---------------------------------------------+
12 rows in set (0.00 sec)
```

Replicating Amazon Aurora MySQL DB clusters across AWS Regions

You can create an Amazon Aurora MySQL DB cluster as a read replica in a different AWS Region than the source DB cluster. Taking this approach can improve your disaster recovery capabilities, let you scale read operations into an AWS Region that is closer to your users, and make it easier to migrate from one AWS Region to another.

You can create read replicas of both encrypted and unencrypted DB clusters. The read replica must be encrypted if the source DB cluster is encrypted.

For each source DB cluster, you can have up to five cross-Region DB clusters that are read replicas.

> **Note**
>
> As an alternative to cross-Region read replicas, you can scale read operations with minimal lag time by using an Aurora global database. An Aurora global database has a primary Aurora DB cluster in one AWS Region and up to five secondary read-only DB clusters in different Regions. Each secondary DB cluster can include up to 16 (rather than 15) Aurora Replicas. Replication from the primary DB cluster to all secondaries is handled by the Aurora storage layer rather than by the database engine, so lag time for replicating changes is minimal—typically, less than 1 second. Keeping the database engine out of the replication process means that the database engine is dedicated to processing workloads. It also means that you don't need to configure or manage the Aurora MySQL binlog (binary logging) replication. To learn more, see Using Amazon Aurora global databases.

When you create an Aurora MySQL DB cluster read replica in another AWS Region, you should be aware of the following:

- Both your source DB cluster and your cross-Region read replica DB cluster can have up to 15 Aurora Replicas, along with the primary instance for the DB cluster. By using this functionality, you can scale read operations for both your source AWS Region and your replication target AWS Region.
- In a cross-Region scenario, there is more lag time between the source DB cluster and the read replica due to the longer network channels between AWS Regions.
- Data transferred for cross-Region replication incurs Amazon RDS data transfer charges. The following cross-Region replication actions generate charges for the data transferred out of the source AWS Region:

  - When you create the read replica, Amazon RDS takes a snapshot of the source cluster and transfers the snapshot to the AWS Region that holds the read replica.
  - For each data modification made in the source databases, Amazon RDS transfers data from the source region to the AWS Region that holds the read replica.

  For more information about Amazon RDS data transfer pricing, see Amazon Aurora pricing.

- You can run multiple concurrent create or delete actions for read replicas that reference the same source DB cluster. However, you must stay within the limit of five read replicas for each source DB cluster.
- For replication to operate effectively, each read replica should have the same amount of compute and storage resources as the source DB cluster. If you scale the source DB cluster, you should also scale the read replicas.

Topics

Before you begin

Before you can create an Aurora MySQL DB cluster that is a cross-Region read replica, you must turn on binary logging on your source Aurora MySQL DB cluster. Cross-region replication for Aurora MySQL uses MySQL binary replication to replay changes on the cross-Region read replica DB cluster.

To turn on binary logging on an Aurora MySQL DB cluster, update the `binlog_format` parameter for your source DB cluster. The `binlog_format` parameter is a cluster-level parameter that is in the default cluster parameter group. If your DB cluster uses the default DB cluster parameter group, create a new DB cluster parameter group to modify `binlog_format` settings. We recommend that you set the `binlog_format` to `MIXED`. However, you can also set `binlog_format` to `ROW` or `STATEMENT` if you need a specific binlog format. Reboot your Aurora DB cluster for the change to take effect.

For more information about using binary logging with Aurora MySQL, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication). For more information about modifying Aurora MySQL configuration parameters, see Amazon Aurora DB cluster and DB instance parameters and Working with parameter groups.

Creating an Amazon Aurora MySQL DB cluster that is a cross-Region read replica

You can create an Aurora DB cluster that is a cross-Region read replica by using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the Amazon RDS API. You can create cross-Region read replicas from both encrypted and unencrypted DB clusters.

When you create a cross-Region read replica for Aurora MySQL by using the AWS Management Console, Amazon RDS creates a DB cluster in the target AWS Region, and then automatically creates a DB instance that is the primary instance for that DB cluster.

When you create a cross-Region read replica using the AWS CLI or RDS API, you first create the DB cluster in the target AWS Region and wait for it to become active. Once it is active, you then create a DB instance that is the primary instance for that DB cluster.

Replication begins when the primary instance of the read replica DB cluster becomes available.

Use the following procedures to create a cross-Region read replica from an Aurora MySQL DB cluster. These procedures work for creating read replicas from either encrypted or unencrypted DB clusters.

Console

To create an Aurora MySQL DB cluster that is a cross-Region read replica with the AWS Management Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the top-right corner of the AWS Management Console, select the AWS Region that hosts your source DB cluster.
3. In the navigation pane, choose **Databases**.
4. Choose the DB cluster for which you want to create a cross-Region read replica.
5. For **Actions**, choose **Create cross-Region read replica**.
6. On the **Create cross region read replica** page, choose the option settings for your cross-Region read replica DB cluster, as described in the following table.

Option	Description
Destination region	Choose the AWS Region to host the new cross-Region read replica DB cluster.
Destination DB subnet group	Choose the DB subnet group to use for the cross-Reg ion read replica DB cluster.
Publicly accessible	Choose **Yes** to give the cross-Region read replica DB cluster a public IP address; otherwise, select **No**.
Encryption	Select **Enable Encryption** to turn on encryption at rest for this DB cluster. For more information, see Encrypting Amazon Aurora resources.
AWS KMS key	Only available if **Encryption** is set to **Enable Encryption**. Select the AWS KMS key to use for encrypting this DB cluster. For more information, see Encrypting Amazon Aurora resources.
DB instance class	Choose a DB instance class that defines the processin g and memory requirements for the primary instance in the DB cluster. For more information about DB instance class options, see Aurora DB instance classes.
Multi-AZ deployment	Choose **Yes** to create a read replica of the new DB cluster in another Availability Zone in the target AWS Region for failover support. For more information about multiple Availability Zones, see Regions and Availability Zones.
Read replica source	Choose the source DB cluster to create a cross-Region read replica for.

Option	Description
DB instance identifier	Type a name for the primary instance in your cross-Region read replica DB cluster. This identifier is used in the endpoint address for the primary instance of the new DB cluster. The DB instance identifier has the following constraints: • It must contain from 1 to 63 alphanumeric characters or hyphens. • Its first character must be a letter. • It cannot end with a hyphen or contain two consecutive hyphens. • It must be unique for all DB instances for each AWS account, for each AWS Region. Because the cross-Region read replica DB cluster is created from a snapshot of the source DB cluster, the master user name and master password for the read replica are the same as the master user name and master password for the source DB cluster.

Option	Description
DB cluster identifier	Type a name for your cross-Region read replica DB cluster that is unique for your account in the target AWS Region for your replica. This identifier is used in the cluster endpoint address for your DB cluster. For information on the cluster endpoint, see Amazon Aurora connection management. The DB cluster identifier has the following constrain ts: • It must contain from 1 to 63 alphanumeric characters or hyphens. • Its first character must be a letter. • It cannot end with a hyphen or contain two consecutive hyphens. • It must be unique for all DB clusters for each AWS account, for each AWS Region.
Priority	Choose a failover priority for the primary instance of the new DB cluster. This priority determines the order in which Aurora Replicas are promoted when recovering from a primary instance failure. If you don't select a value, the default is **tier-1**. For more information, see Fault tolerance for an Aurora DB cluster.
Database port	Specify the port for applications and utilities to use to access the database. Aurora DB clusters default to the default MySQL port, 3306. Firewalls at some companies block connections to this port. If your company firewall blocks the default port, choose another port for the new DB cluster.

Option	Description
Enhanced monitoring	Choose **Enable enhanced monitoring** to turn on gathering metrics in real time for the operating system that your DB cluster runs on. For more information, see Monitoring OS metrics with Enhanced Monitoring.
Monitoring Role	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Choose the IAM role that you created to permit Amazon RDS to communicate with Amazon CloudWatch Logs for you, or choose **Default** to have RDS create a role for you named `rds-monitoring-role` . For more information, see Monitoring OS metrics with Enhanced Monitoring.
Granularity	Only available if **Enhanced Monitoring** is set to **Enable enhanced monitoring**. Set the interval, in seconds, between when metrics are collected for your DB cluster.
Auto minor version upgrade	This setting doesn't apply to Aurora MySQL DB clusters. For more information about engine updates for Aurora MySQL, see Database engine updates for Amazon Aurora MySQL.

7. Choose **Create** to create your cross-Region read replica for Aurora.

AWS CLI

To create an Aurora MySQL DB cluster that is a cross-Region read replica with the CLI

1. Call the AWS CLI create-db-cluster command in the AWS Region where you want to create the read replica DB cluster. Include the `--replication-source-identifier` option and specify the Amazon Resource Name (ARN) of the source DB cluster to create a read replica for.

For cross-Region replication where the DB cluster identified by `--replication-source-identifier` is encrypted, specify the `--kms-key-id` option and the `--storage-encrypted` option.

> **Note**
>
> You can set up cross-Region replication from an unencrypted DB cluster to an encrypted read replica by specifying `--storage-encrypted` and providing a value for `--kms-key-id`.

You can't specify the `--master-username` and `--master-user-password` parameters. Those values are taken from the source DB cluster.

The following code example creates a read replica in the us-east-1 Region from an unencrypted DB cluster snapshot in the us-west-2 Region. The command is called in the us-east-1 Region. This example specifies the `--manage-master-user-password` option to generate the master user password and manage it in Secrets Manager. For more information, see [Password management with Amazon Aurora and AWS Secrets Manager](). Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
  --db-cluster-identifier sample-replica-cluster \
  --engine aurora \
  --replication-source-identifier arn:aws:rds:us-
west-2:123456789012:cluster:sample-master-cluster
```

For Windows:

```
aws rds create-db-cluster ^
  --db-cluster-identifier sample-replica-cluster ^
  --engine aurora ^
  --replication-source-identifier arn:aws:rds:us-
west-2:123456789012:cluster:sample-master-cluster
```

The following code example creates a read replica in the us-east-1 Region from an encrypted DB cluster snapshot in the us-west-2 Region. The command is called in the us-east-1 Region.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
  --db-cluster-identifier sample-replica-cluster \
  --engine aurora \
  --replication-source-identifier arn:aws:rds:us-
west-2:123456789012:cluster:sample-master-cluster \
  --kms-key-id my-us-east-1-key \
  --storage-encrypted
```

For Windows:

```
aws rds create-db-cluster ^
  --db-cluster-identifier sample-replica-cluster ^
  --engine aurora ^
  --replication-source-identifier arn:aws:rds:us-
west-2:123456789012:cluster:sample-master-cluster ^
  --kms-key-id my-us-east-1-key ^
  --storage-encrypted
```

The `--source-region` option is required for cross-Region replication between the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions, where the DB cluster identified by `--replication-source-identifier` is encrypted. For `--source-region`, specify the AWS Region of the source DB cluster.

If `--source-region` isn't specified, specify a `--pre-signed-url` value. A *presigned URL* is a URL that contains a Signature Version 4 signed request for the `create-db-cluster` command that is called in the source AWS Region. To learn more about the `pre-signed-url` option, see create-db-cluster in the *AWS CLI Command Reference*.

2. Check that the DB cluster has become available to use by using the AWS CLI describe-db-clusters command, as shown in the following example.

```
aws rds describe-db-clusters --db-cluster-identifier sample-replica-cluster
```

When the **describe-db-clusters** results show a status of `available`, create the primary instance for the DB cluster so that replication can begin. To do so, use the AWS CLI create-db-instance command as shown in the following example.

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
  --db-cluster-identifier sample-replica-cluster \
  --db-instance-class db.r3.large \
  --db-instance-identifier sample-replica-instance \
  --engine aurora
```

For Windows:

```
aws rds create-db-instance ^
  --db-cluster-identifier sample-replica-cluster ^
  --db-instance-class db.r3.large ^
  --db-instance-identifier sample-replica-instance ^
  --engine aurora
```

When the DB instance is created and available, replication begins. You can determine if the DB instance is available by calling the AWS CLI describe-db-instances command.

RDS API

To create an Aurora MySQL DB cluster that is a cross-Region read replica with the API

1. Call the RDS API CreateDBCluster operation in the AWS Region where you want to create the read replica DB cluster. Include the `ReplicationSourceIdentifier` parameter and specify the Amazon Resource Name (ARN) of the source DB cluster to create a read replica for.

   For cross-Region replication where the DB cluster identified by `ReplicationSourceIdentifier` is encrypted, specify the `KmsKeyId` parameter and set the `StorageEncrypted` parameter to `true`.

> **Note**
>
> You can set up cross-Region replication from an unencrypted DB cluster to an encrypted read replica by specifying `StorageEncrypted` as **`true`** and providing a value for `KmsKeyId`. In this case, you don't need to specify `PreSignedUrl`.

You don't need to include the `MasterUsername` and `MasterUserPassword` parameters, because those values are taken from the source DB cluster.

The following code example creates a read replica in the us-east-1 Region from an unencrypted DB cluster snapshot in the us-west-2 Region. The action is called in the us-east-1 Region.

```
https://rds.us-east-1.amazonaws.com/
  ?Action=CreateDBCluster
  &ReplicationSourceIdentifier=arn:aws:rds:us-west-2:123456789012:cluster:sample-
master-cluster
  &DBClusterIdentifier=sample-replica-cluster
  &Engine=aurora
  &SignatureMethod=HmacSHA256
  &SignatureVersion=4
  &Version=2014-10-31
  &X-Amz-Algorithm=AWS4-HMAC-SHA256
  &X-Amz-Credential=AKIADQKE4SARGYLE/20161117/us-east-1/rds/aws4_request
  &X-Amz-Date=20160201T001547Z
  &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
  &X-Amz-Signature=a04c831a0b54b5e4cd236a90dcb9f5fab7185eb3b72b5ebe9a70a4e95790c8b7
```

The following code example creates a read replica in the us-east-1 Region from an encrypted DB cluster snapshot in the us-west-2 Region. The action is called in the us-east-1 Region.

```
https://rds.us-east-1.amazonaws.com/
  ?Action=CreateDBCluster
  &KmsKeyId=my-us-east-1-key
  &StorageEncrypted=true
  &PreSignedUrl=https%253A%252F%252Frds.us-west-2.amazonaws.com%252F
         %253FAction%253DCreateDBCluster
         %2526DestinationRegion%253Dus-east-1
         %2526KmsKeyId%253Dmy-us-east-1-key
         %2526ReplicationSourceIdentifier%253Darn%25253Aaws%25253Ards%25253Aus-
west-2%25253A123456789012%25253Acluster%25253Asample-master-cluster
         %2526SignatureMethod%253DHmacSHA256
         %2526SignatureVersion%253D4
         %2526Version%253D2014-10-31
         %2526X-Amz-Algorithm%253DAWS4-HMAC-SHA256
         %2526X-Amz-Credential%253DAKIADQKE4SARGYLE%252F20161117%252Fus-
west-2%252Frds%252Faws4_request
         %2526X-Amz-Date%253D20161117T215409Z
         %2526X-Amz-Expires%253D3600
         %2526X-Amz-SignedHeaders%253Dcontent-type%253Bhost%253Buser-agent%253Bx-
amz-content-sha256%253Bx-amz-date
         %2526X-Amz-Signature
%253D255a0f17b4e717d3b67fad163c3ec26573b882c03a65523522cf890a67fca613
  &ReplicationSourceIdentifier=arn:aws:rds:us-west-2:123456789012:cluster:sample-
master-cluster
  &DBClusterIdentifier=sample-replica-cluster
  &Engine=aurora
  &SignatureMethod=HmacSHA256
  &SignatureVersion=4
  &Version=2014-10-31
  &X-Amz-Algorithm=AWS4-HMAC-SHA256
  &X-Amz-Credential=AKIADQKE4SARGYLE/20161117/us-east-1/rds/aws4_request
  &X-Amz-Date=20160201T001547Z
  &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
  &X-Amz-Signature=a04c831a0b54b5e4cd236a90dcb9f5fab7185eb3b72b5ebe9a70a4e95790c8b7
```

For cross-Region replication between the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions, where the DB cluster identified by `ReplicationSourceIdentifier` is encrypted, also specify the `PreSignedUrl` parameter. The presigned URL must be a valid request for the `CreateDBCluster` API operation that can be performed in the source AWS Region that contains the encrypted DB cluster to be replicated. The KMS key identifier is used to encrypt the read replica, and must be a KMS key valid for the destination AWS Region. To automatically rather than manually generate a presigned URL, use the AWS CLI create-db-cluster command with the `--source-region` option instead.

2. Check that the DB cluster has become available to use by using the RDS API DescribeDBClusters operation, as shown in the following example.

```
https://rds.us-east-1.amazonaws.com/
  ?Action=DescribeDBClusters
  &DBClusterIdentifier=sample-replica-cluster
  &SignatureMethod=HmacSHA256
  &SignatureVersion=4
  &Version=2014-10-31
  &X-Amz-Algorithm=AWS4-HMAC-SHA256
  &X-Amz-Credential=AKIADQKE4SARGYLE/20161117/us-east-1/rds/aws4_request
  &X-Amz-Date=20160201T002223Z
  &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
  &X-Amz-Signature=84c2e4f8fba7c577ac5d820711e34c6e45ffcd35be8a6b7c50f329a74f35f426
```

When `DescribeDBClusters` results show a status of `available`, create the primary instance for the DB cluster so that replication can begin. To do so, use the RDS API CreateDBInstance action as shown in the following example.

```
https://rds.us-east-1.amazonaws.com/
  ?Action=CreateDBInstance
  &DBClusterIdentifier=sample-replica-cluster
  &DBInstanceClass=db.r3.large
  &DBInstanceIdentifier=sample-replica-instance
  &Engine=aurora
  &SignatureMethod=HmacSHA256
  &SignatureVersion=4
  &Version=2014-10-31
  &X-Amz-Algorithm=AWS4-HMAC-SHA256
  &X-Amz-Credential=AKIADQKE4SARGYLE/20161117/us-east-1/rds/aws4_request
  &X-Amz-Date=20160201T003808Z
  &X-Amz-SignedHeaders=content-type;host;user-agent;x-amz-content-sha256;x-amz-date
  &X-Amz-Signature=125fe575959f5bbcebd53f2365f907179757a08b5d7a16a378dfa59387f58cdb
```

When the DB instance is created and available, replication begins. You can determine if the DB instance is available by calling the AWS CLI DescribeDBInstances command.

Viewing Amazon Aurora MySQL cross-Region replicas

You can view the cross-Region replication relationships for your Amazon Aurora MySQL DB clusters by calling the describe-db-clusters AWS CLI command or the DescribeDBClusters RDS API operation. In the response, refer to the `ReadReplicaIdentifiers` field for the DB cluster identifiers of any cross-Region read replica DB clusters. Refer to the `ReplicationSourceIdentifier` element for the ARN of the source DB cluster that is the replication source.

Promoting a read replica to be a DB cluster

You can promote an Aurora MySQL read replica to a standalone DB cluster. When you promote an Aurora MySQL read replica, its DB instances are rebooted before they become available.

Typically, you promote an Aurora MySQL read replica to a standalone DB cluster as a data recovery scheme if the source DB cluster fails.

To do this, first create a read replica and then monitor the source DB cluster for failures. In the event of a failure, do the following:

1. Promote the read replica.
2. Direct database traffic to the promoted DB cluster.
3. Create a replacement read replica with the promoted DB cluster as its source.

When you promote a read replica, the read replica becomes a standalone Aurora DB cluster. The promotion process can take several minutes or longer to complete, depending on the size of the read replica. After you promote the read replica to a new DB cluster, it's just like any other DB cluster. For example, you can create read replicas from it and perform point-in-time restore operations. You can also create Aurora Replicas for the DB cluster.

Because the promoted DB cluster is no longer a read replica, you can't use it as a replication target.

The following steps show the general process for promoting a read replica to a DB cluster:

1. Stop any transactions from being written to the read replica source DB cluster, and then wait for all updates to be made to the read replica. Database updates occur on the read replica after they have occurred on the source DB cluster, and this replication lag can vary significantly. Use the `ReplicaLag` metric to determine when all updates have been made to the read replica. The `ReplicaLag` metric records the amount of time a read replica DB instance lags behind the source DB instance. When the `ReplicaLag` metric reaches `0`, the read replica has caught up to the source DB instance.
2. Promote the read replica by using the **Promote** option on the Amazon RDS console, the AWS CLI command [promote-read-replica-db-cluster](), or the [PromoteReadReplicaDBCluster]() Amazon RDS API operation.

   You choose an Aurora MySQL DB instance to promote the read replica. After the read replica is promoted, the Aurora MySQL DB cluster is promoted to a standalone DB cluster. The DB instance with the highest failover priority is promoted to the primary DB instance for the DB cluster. The other DB instances become Aurora Replicas.

> **Note**
>
> The promotion process takes a few minutes to complete. When you promote a read replica, replication is stopped and the DB instances are rebooted. When the reboot is complete, the read replica is available as a new DB cluster.

Console

To promote an Aurora MySQL read replica to a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. On the console, choose **Instances**.

   The **Instance** pane appears.
3. In the **Instances** pane, choose the read replica that you want to promote.

   The read replicas appear as Aurora MySQL DB instances.
4. For **Actions**, choose **Promote read replica**.
5. On the acknowledgment page, choose **Promote read replica**.

AWS CLI

To promote a read replica to a DB cluster, use the AWS CLI promote-read-replica-db-cluster command.

Example

For Linux, macOS, or Unix:

```
aws rds promote-read-replica-db-cluster \
    --db-cluster-identifier mydbcluster
```

For Windows:

```
aws rds promote-read-replica-db-cluster ^
    --db-cluster-identifier mydbcluster
```

RDS API

To promote a read replica to a DB cluster, call PromoteReadReplicaDBCluster.

Troubleshooting Amazon Aurora MySQL cross Region replicas

Following you can find a list of common error messages that you might encounter when creating an Amazon Aurora cross-Region read replica, and how to resolve the specified errors.

Source cluster [DB cluster ARN] doesn't have binlogs enabled

To resolve this issue, turn on binary logging on the source DB cluster. For more information, see Before you begin.

Source cluster [DB cluster ARN] doesn't have cluster parameter group in sync on writer

You receive this error if you have updated the `binlog_format` DB cluster parameter, but have not rebooted the primary instance for the DB cluster. Reboot the primary instance (that is, the writer) for the DB cluster and try again.

Source cluster [DB cluster ARN] already has a read replica in this region

You can have up to five cross-Region DB clusters that are read replicas for each source DB cluster in any AWS Region. If you already have the maximum number of read replicas for a DB cluster in a particular AWS Region, you must delete an existing one before you can create a new cross-Region DB cluster in that Region.

DB cluster [DB cluster ARN] requires a database engine upgrade for cross-Region replication support

To resolve this issue, upgrade the database engine version for all of the instances in the source DB cluster to the most recent database engine version, and then try creating a cross-Region read replica DB again.

Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication)

Because Amazon Aurora MySQL is compatible with MySQL, you can set up replication between a MySQL database and an Amazon Aurora MySQL DB cluster. This type of replication uses the MySQL binary log replication, also referred to as *binlog replication*. If you use binary log replication with Aurora, we recommend that your MySQL database run MySQL version 5.5 or later. You can set up replication where your Aurora MySQL DB cluster is the replication source or the replica. You can replicate with an Amazon RDS MySQL DB instance, a MySQL database external to Amazon RDS, or another Aurora MySQL DB cluster.

> **Note**
>
> You can't use binlog replication to or from certain types of Aurora DB clusters. In particular, binlog replication isn't available for Aurora Serverless v1 clusters. If the `SHOW MASTER STATUS` and `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) statement returns no output, check that the cluster you're using supports binlog replication.
>
> In Aurora MySQL version 3, binary log replication doesn't replicate to the `mysql` system database. Passwords and accounts aren't replicated by binlog replication in Aurora MySQL version 3. Therefore, Data Control Language (DCL) statements such as `CREATE USER`, `GRANT`, and `REVOKE` aren't replicated.

You can also replicate with an RDS for MySQL DB instance or Aurora MySQL DB cluster in another AWS Region. When you're performing replication across AWS Regions, make sure that your DB clusters and DB instances are publicly accessible. If the Aurora MySQL DB clusters are in private subnets in your VPC, use VPC peering between the AWS Regions. For more information, see A DB cluster in a VPC accessed by an EC2 instance in a different VPC.

If you want to configure replication between an Aurora MySQL DB cluster and an Aurora MySQL DB cluster in another AWS Region, you can create an Aurora MySQL DB cluster as a read replica in a different AWS Region from the source DB cluster. For more information, see Replicating Amazon Aurora MySQL DB clusters across AWS Regions.

With Aurora MySQL version 2 and 3, you can replicate between Aurora MySQL and an external source or target that uses global transaction identifiers (GTIDs) for replication. Ensure that the GTID-related parameters in the Aurora MySQL DB cluster have settings that are compatible with the GTID status of the external database. To learn how to do this, see Using GTID-based replication. In Aurora MySQL version 3.01 and higher, you can choose how to assign GTIDs to transactions that are replicated from a source that doesn't use GTIDs. For information about the stored procedure that controls that setting, see mysql.rds_assign_gtids_to_anonymous_transactions (Aurora MySQL version 3).

⚠ **Warning**

When you replicate between Aurora MySQL and MySQL, make sure that you use only InnoDB tables. If you have MyISAM tables that you want to replicate, you can convert them to InnoDB before setting up replication with the following command.

```
alter table <schema>.<table_name> engine=innodb, algorithm=copy;
```

Setting up replication with MySQL or another Aurora DB cluster

Setting up MySQL replication with Aurora MySQL involves the following steps, which are discussed in detail:

1. Turn on binary logging on the replication source

2. Retain binary logs on the replication source until no longer needed

3. Create a snapshot or dump of your replication source

4. Load the snapshot or dump into your replica target

5. Create a replication user on your replication source

6. Turn on replication on your replica target

7. Monitor your replica

1. Turn on binary logging on the replication source

Find instructions on how to turn on binary logging on the replication source for your database engine following.

Database engine	Instructions
Aurora MySQL	**To turn on binary logging on an Aurora MySQL DB cluster** Set the `binlog_format` DB cluster parameter to `ROW`, `STATEMENT` , or `MIXED`. `MIXED` is recommended unless you have a need for a specific binlog format. (The default value is `OFF`.)

Database engine	Instructions
	To change the `binlog_format` parameter, create a custom DB cluster parameter group and associate that custom parameter group with your DB cluster. You can't change parameters in the default DB cluster parameter group. If you're changing the `binlog_format` parameter from `OFF` to another value, reboot your Aurora DB cluster for the change to take effect. For more information, see Amazon Aurora DB cluster and DB instance parameters and Working with parameter groups.
RDS for MySQL	**To turn on binary logging on an Amazon RDS DB instance** You can't turn on binary logging directly for an Amazon RDS DB instance, but you can turn it on by doing one of the following: • Turn on automated backups for the DB instance. You can turn on automated backups when you create a DB instance, or you can turn on backups by modifying an existing DB instance. For more information, see Creating a DB instance in the *Amazon RDS User Guide*. • Create a read replica for the DB instance. For more information, see Working with read replicas in the *Amazon RDS User Guide*.

Database engine	Instructions
MySQL (external)	**To set up encrypted replication** To replicate data securely with Aurora MySQL version 2, you can use encrypted replication. **Note** If you don't need to use encrypted replication, you can skip these steps. The following are prerequisites for using encrypted replication: • Secure Sockets Layer (SSL) must be enabled on the external MySQL source database. • A client key and client certificate must be prepared for the Aurora MySQL DB cluster. During encrypted replication, the Aurora MySQL DB cluster acts a client to the MySQL database server. The certificates and keys for the Aurora MySQL client are in files in .pem format. 1. Ensure that you are prepared for encrypted replication: • If you don't have SSL enabled on the external MySQL source database and don't have a client key and client certificate prepared, turn on SSL on the MySQL database server and generate the required client key and client certificate. • If SSL is enabled on the external source, supply a client key and certificate for the Aurora MySQL DB cluster. If you don't have these, generate a new key and certificate for the Aurora MySQL DB cluster. To sign the client certificate, you must have the certificate authority key that you used to configure SSL on the external MySQL source database.

Database engine	Instructions

For more information, see [Creating SSL certificates and keys using openssl](#) in the MySQL documentation.

You need the certificate authority certificate, the client key, and the client certificate.

2. Connect to the Aurora MySQL DB cluster as the master user using SSL.

   For information about connecting to an Aurora MySQL DB cluster with SSL, see [Using TLS with Aurora MySQL DB clusters](#).

3. Run the `mysql.rds_import_binlog_ssl_material` stored procedure to import the SSL information into the Aurora MySQL DB cluster.

   For the `ssl_material_value` parameter, insert the information from the .pem format files for the Aurora MySQL DB cluster in the correct JSON payload.

   The following example imports SSL information into an Aurora MySQL DB cluster. In .pem format files, the body code typically is longer than the body code shown in the example.

```
call mysql.rds_import_binlog_ssl_material(
'{"ssl_ca":"-----BEGIN CERTIFICATE-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcj
qP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96
xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/
i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz22
1CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END CERTIFICATE-----\n","ssl_cert":"-----BEGIN CERTIFICA
TE-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcj
qP3maAhDFcvBS7O6V
```

Database engine	Instructions

```
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96
xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/
i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz22
1CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END CERTIFICATE-----\n","ssl_key":"-----BEGIN RSA PRIVATE
 KEY-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pc
jqP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSF
JuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJ
tjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMu
cxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END RSA PRIVATE KEY-----\n"}');
```

For more information, see mysql.rds_import_binlog_ssl_material and Using TLS with Aurora MySQL DB clusters.

> **Note**
>
> After running the procedure, the secrets are stored in files. To erase the files later, you can run the mysql.rds_remove_binlog_ssl_material stored procedure.

To turn on binary logging on an external MySQL database

1. From a command shell, stop the mysql service.

```
sudo service mysqld stop
```

2. Edit the `my.cnf` file (this file is usually under `/etc`).

Database engine	Instructions

	`sudo vi /etc/my.cnf`
	
	Add the `log_bin` and `server_id` options to the `[mysqld]` section. The `log_bin` option provides a file name identifier for binary log files. The `server_id` option provides a unique identifier for the server in source-replica relationships.
	
	If encrypted replication isn't required, ensure that the external MySQL database is started with binlogs enabled and SSL is turned off.
	
	The following are the relevant entries in the `/etc/my.cnf` file for unencrypted data.
	
	`log-bin=mysql-bin`
	`server-id=2133421`
	`innodb_flush_log_at_trx_commit=1`
	`sync_binlog=1`
	
	If encrypted replication is required, ensure that the external MySQL database is started with SSL and binlogs enabled.
	
	The entries in the `/etc/my.cnf` file include the .pem file locations for the MySQL database server.
	
	`log-bin=mysql-bin`
	`server-id=2133421`
	`innodb_flush_log_at_trx_commit=1`
	`sync_binlog=1`
	
	`# Setup SSL.`
	`ssl-ca=/home/sslcerts/ca.pem`
	`ssl-cert=/home/sslcerts/server-cert.pem`
	`ssl-key=/home/sslcerts/server-key.pem`
	
	Additionally, the `sql_mode` option for your MySQL DB instance must be set to 0, or must not be included in your my.cnf file.

Database engine	Instructions

While connected to the external MySQL database, record the external MySQL database's binary log position.

```
mysql> SHOW MASTER STATUS;
```

Your output should be similar to the following:

```
+------------------+----------+--------------+------------------
+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
 Executed_Gtid_Set |
+------------------+----------+--------------+------------------
+-------------------+
| mysql-bin.000031 |      107 |              |                  |
              |
+------------------+----------+--------------+------------------
+-------------------+
1 row in set (0.00 sec)
```

For more information, see Setting the replication source configuration in the MySQL documentation.

3. Start the mysql service.

```
sudo service mysqld start
```

2. Retain binary logs on the replication source until no longer needed

When you use MySQL binary log replication, Amazon RDS doesn't manage the replication process. As a result, you need to ensure that the binlog files on your replication source are retained until after the changes have been applied to the replica. This maintenance helps you to restore your source database in the event of a failure.

Use the following instructions to retain binary logs for your database engine.

Database engine	Instructions
Aurora MySQL	**To retain binary logs on an Aurora MySQL DB cluster** You don't have access to the binlog files for an Aurora MySQL DB cluster. As a result, you must choose a time frame to retain the binlog files on your replication source long enough to ensure that the changes have been applied to your replica before the binlog file is deleted by Amazon RDS. You can retain binlog files on an Aurora MySQL DB cluster for up to 90 days. If you're setting up replication with a MySQL database or RDS for MySQL DB instance as the replica, and the database that you are creating a replica for is very large, choose a large time frame to retain binlog files until the initial copy of the database to the replica is complete and the replica lag has reached 0. To set the binary log retention time frame, use the [mysql.rds_set_configuration](#) procedure and specify a configuration parameter of `'binlog retention hours'` along with the number of hours to retain binlog files on the DB cluster. The maximum value for Aurora MySQL version 2.11.0 and higher and version 3 is 2160 (90 days). The following example sets the retention period for binlog files to 6 days: `CALL mysql.rds_set_configuration('binlog retention hours', 144);` After replication has been started, you can verify that changes have been applied to your replica by running the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command on your replica and checking the `Seconds behind master` field. If the `Seconds behind master` field is 0, then there is no replica lag. When there is no replica lag, reduce the length of time that binlog files are retained by setting the `binlog retention hours` configuration parameter to a smaller time frame. If this setting isn't specified, the default for Aurora MySQL is 24 (1 day). If you specify a value for `'binlog retention hours'` that is higher than the maximum value, then Aurora MySQL uses the maximum.

Database engine	Instructions
RDS for MySQL	**To retain binary logs on an Amazon RDS DB instance** You can retain binary log files on an Amazon RDS DB instance by setting the binlog retention hours just as with an Aurora MySQL DB cluster, described in the previous row. You can also retain binlog files on an Amazon RDS DB instance by creating a read replica for the DB instance. This read replica is temporary and solely for the purpose of retaining binlog files. After the read replica has been created, call the mysql.rds_stop_replication procedure on the read replica. While replication is stopped, Amazon RDS doesn't delete any of the binlog files on the replication source. After you have set up replication with your permanent replica, you can delete the read replica when the replica lag (`Seconds behind master` field) between your replication source and your permanent replica reaches 0.
MySQL (external)	**To retain binary logs on an external MySQL database** Because binlog files on an external MySQL database are not managed by Amazon RDS, they are retained until you delete them. After replication has been started, you can verify that changes have been applied to your replica by running the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command on your replica and checking the `Seconds behind master` field. If the `Seconds behind master` field is 0, then there is no replica lag. When there is no replica lag, you can delete old binlog files.

3. Create a snapshot or dump of your replication source

You use a snapshot or dump of your replication source to load a baseline copy of your data onto your replica and then start replicating from that point on.

Use the following instructions to create a snapshot or dump of the replication source for your database engine.

Database engine	Instructions

Aurora MySQL	**To create a snapshot of an Aurora MySQL DB cluster**
	1. Create a DB cluster snapshot of your Amazon Aurora DB cluster. For more information, see Creating a DB cluster snapshot.
	2. Create a new Aurora DB cluster by restoring from the DB cluster snapshot that you just created. Be sure to retain the same DB parameter group for your restored DB cluster as your original DB cluster. Doing this ensures that the copy of your DB cluster has binary logging enabled. For more information, see Restoring from a DB cluster snapshot.
	3. In the console, choose **Databases** and choose the primary instance (writer) for your restored Aurora DB cluster to show its details. Scroll to **Recent Events**. An event message shows that includes the binlog file name and position. The event message is in the following format.
	`Binlog position from crash recovery is` *`binlog-file-name  binlog-position`*
	Save the binlog file name and position values for when you start replication.
	You can also get the binlog file name and position by calling the describe-events command from the AWS CLI. The following shows an example `describe-events` command with example output.
	`PROMPT> aws rds describe-events`
	`{`
	`    "Events": [`
	`        {`
	`            "EventCategories": [],`
	`            "SourceType": "db-instance",`
	`            "SourceArn": "arn:aws:rds:us-west-2:123456789012:db:sample-restored-instance",`
	`            "Date": "2016-10-28T19:43:46.862Z",`
	`            "Message": "Binlog position from crash recovery is mysql-bin-changelog.000003 4278",`

Database engine	Instructions

```
            "SourceIdentifier": "sample-restored-instance"
        }
    ]
}
```

You can also get the binlog file name and position by checking the MySQL error log for the last MySQL binlog file position.

4. If your replica target is an external MySQL database or an RDS for MySQL DB instance, then you can't load the data from an Amazon Aurora DB cluster snapshot. Instead, create a dump of your Aurora DB cluster by connecting to your DB cluster using a MySQL client and issuing the `mysqldump` command. Be sure to run the `mysqldump` command against the copy of your Aurora DB cluster that you created. The following is an example.

```
PROMPT> mysqldump --databases <database_name> --single-transaction
--order-by-primary -r backup.sql -u <local_user> -p
```

5. When you have finished creating the dump of your data from the newly created Aurora DB cluster, delete that DB cluster as it is no longer needed.

Database engine	Instructions
RDS for MySQL	**To create a snapshot of an Amazon RDS DB instance** Create a read replica of your Amazon RDS DB instance. For more information, see Creating a read replica in the *Amazon Relational Database Service User Guide*. 1. Connect to your read replica and stop replication by running the mysql.rds _stop_replication procedure. 2. While the read replica is **Stopped**, Connect to the read replica and run the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command. Retrieve the current binary log file name from the `Relay_Master_Log_File` field and the log file position from the `Exec_Master_Log_Pos` field. Save these values for when you start replicati on. 3. While the read replica remains **Stopped**, create a DB snapshot of the read replica. For more information, see Creating a DB snapshot in the *Amazon Relational Database Service User Guide*. 4. Delete the read replica.

Database engine	Instructions

MySQL (external)	**To create a dump of an external MySQL database**
	1. Before you create a dump, you need to ensure that the binlog location for the dump is current with the data in your source instance. To do this, you must first stop any write operations to the instance with the following command:
	`mysql> FLUSH TABLES WITH READ LOCK;`
	2. Create a dump of your MySQL database using the `mysqldump` command as shown following:
	`PROMPT> sudo mysqldump --databases <database_name> --master-data=2`
	` --single-transaction \`
	`--order-by-primary -r backup.sql -u <local_user> -p`
	3. After you have created the dump, unlock the tables in your MySQL database with the following command:
	`mysql> UNLOCK TABLES;`

4. Load the snapshot or dump into your replica target

If you plan to load data from a dump of a MySQL database that is external to Amazon RDS, then you might want to create an EC2 instance to copy the dump files to, and then load the data into your DB cluster or DB instance from that EC2 instance. Using this approach, you can compress the dump file(s) before copying them to the EC2 instance in order to reduce the network costs associated with copying data to Amazon RDS. You can also encrypt the dump file or files to secure the data as it is being transferred across the network.

Use the following instructions to load the snapshot or dump of your replication source into your replica target for your database engine.

Database engine	Instructions

Aurora MySQL	**To load a snapshot or dump into an Aurora MySQL DB cluster**
	• If the snapshot of your replication source is a DB cluster snapshot, then you can restore from the DB cluster snapshot to create a new Aurora MySQL DB cluster as your replica target. For more information, see Restoring from a DB cluster snapshot.
	• If the snapshot of your replication source is a DB snapshot, then you can migrate the data from your DB snapshot into a new Aurora MySQL DB cluster. For more information, see Migrating data to an Amazon Aurora MySQL DB cluster.
	• If the data from your replication source is the output from the `mysqldump` command, then follow these steps:
	1. Copy the output of the `mysqldump` command from your replication source to a location that can also connect to your Aurora MySQL DB cluster.
	2. Connect to your Aurora MySQL DB cluster using the `mysql` command. The following is an example.
	`PROMPT> mysql -h <host_name> -port=3306 -u <db_master_user> -p`
	3. At the `mysql` prompt, run the `source` command and pass it the name of your database dump file to load the data into the Aurora MySQL DB cluster, for example:
	`mysql> source backup.sql;`

RDS for MySQL	**To load a dump into an Amazon RDS DB instance** 1. Copy the output of the `mysqldump` command from your replication source to a location that can also connect to your MySQL DB instance. 2. Connect to your MySQL DB instance using the `mysql` command. The following is an example. `PROMPT> mysql -h <host_name> -port=3306 -u <db_master_user> -p`

Database engine	Instructions
	3. At the `mysql` prompt, run the `source` command and pass it the name of your database dump file to load the data into the MySQL DB instance, for example: `mysql> source backup.sql;`

MySQL (external)	**To load a dump into an external MySQL database**
	You can't load a DB snapshot or a DB cluster snapshot into an external MySQL database. Instead, you must use the output from the `mysqldump` command.
	1. Copy the output of the `mysqldump` command from your replication source to a location that can also connect to your MySQL database.
	2. Connect to your MySQL database using the `mysql` command. The following is an example.
	`PROMPT> mysql -h <host_name> -port=3306 -u <db_master_user> -p`
	3. At the `mysql` prompt, run the `source` command and pass it the name of your database dump file to load the data into your MySQL database. The following is an example.
	`mysql> source backup.sql;`

5. Create a replication user on your replication source

Create a user ID on the source that is used solely for replication. The following example is for RDS for MySQL or external MySQL source databases.

```
mysql> CREATE USER 'repl_user'@'domain_name' IDENTIFIED BY 'password';
```

For Aurora MySQL source databases, the `skip_name_resolve` DB cluster parameter is set to `1` (`ON`) and can't be modified, so you must use an IP address for the host instead of a domain name. For more information, see skip_name_resolve in the MySQL documentation.

```
mysql> CREATE USER 'repl_user'@'IP_address' IDENTIFIED BY 'password';
```

The user requires the `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges. Grant these privileges to the user.

If you need to use encrypted replication, require SSL connections for the replication user. For example, you can use one of the following statements to require SSL connections on the user account `repl_user`.

```
GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'IP_address';
```

```
GRANT USAGE ON *.* TO 'repl_user'@'IP_address' REQUIRE SSL;
```

> **Note**
>
> If `REQUIRE SSL` isn't included, the replication connection might silently fall back to an unencrypted connection.

6. Turn on replication on your replica target

Before you turn on replication, we recommend that you take a manual snapshot of the Aurora MySQL DB cluster or RDS for MySQL DB instance replica target. If a problem arises and you need to re-establish replication with the DB cluster or DB instance replica target, you can restore the DB cluster or DB instance from this snapshot instead of having to import the data into your replica target again.

Use the following instructions to turn on replication for your database engine.

Database engine	Instructions
Aurora MySQL	**To turn on replication from an Aurora MySQL DB cluster** 1. Find the starting place for replication. You need the binlog file name and binlog position. If your DB cluster replica target was created from the following:

Database engine	Instructions

	- DB cluster snapshot – Retrieve the binlog file name and position from the recent events for your restored DB cluster, as shown in 3. Create a snapshot or dump of your replication source.
	- DB snapshot – You retrieved the binlog file name and position from the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command when you created the snapshot of your replication source.
	2. Connect to the DB cluster and call the following procedures to start replication with your replication source using the binary log file name and location from the previous step:
	- mysql.rds_set_external_source (Aurora MySQL version 3)
	- mysql.rds_set_external_master (Aurora MySQL version 2)
	- mysql.rds_start_replication (all versions)
	The following example is for Aurora MySQL version 3.
	`CALL mysql.rds_set_external_source ('mydbinstance.123456789012`
	`.us-east-1.rds.amazonaws.com', 3306,`
	`    'repl_user', 'password', 'mysql-bin-changelog.000031', 107,`
	` 0);`
	`CALL mysql.rds_start_replication;`
	To use SSL encryption, set the final value to 1 instead of `0`.

<table>
<tr><th>Database engine</th><th>Instructions</th></tr>
<tr><td>RDS for MySQL</td><td>

To turn on replication from an Amazon RDS DB instance

1. If your DB instance replica target was created from a DB snapshot, then you need the binlog file and binlog position that are the starting place for replication. You retrieved these values from the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command when you created the snapshot of your replication source.
2. Connect to the DB instance and call the mysql.rds_set_external_master (Aurora MySQL version 2) or mysql.rds_set_external_source (Aurora MySQL version 3) and mysql.rds_start_replication procedures to start replication with your replication source. Use the binary log file name and location from the previous step. The following is an example.

```
CALL mysql.rds_set_external_master ('mydbcluster.cluster-12345
6789012.us-east-1.rds.amazonaws.com', 3306,
    'repl_user', 'password', 'mysql-bin-changelog.000031', 107,
 0);
CALL mysql.rds_start_replication;
```

To use SSL encryption, set the final value to 1 instead of `0`.

</td></tr>
</table>

<table>
<tr><th>Database engine</th><th>Instructions</th></tr>
<tr><td>MySQL (external)</td><td>

To turn on replication from an external MySQL database

1. Retrieve the binlog file and binlog position that are the starting place for replication. You retrieved these values from the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command when you created the snapshot of your replication source. If your external MySQL replica target was populated from the output of the `mysqldump` command with the `--master-data=2` option, then the binlog file and binlog position are included in the output. The following is an example.

```
--
-- Position to start replication or point-in-time recovery from
--

-- CHANGE MASTER TO MASTER_LOG_FILE='mysql-bin-changelog.000031',
 MASTER_LOG_POS=107;
```

2. Connect to the external MySQL replica target, and issue `CHANGE MASTER TO` and `START SLAVE` (Aurora MySQL version 2) or `START REPLICA` (Aurora MySQL version 3) to start replication with your replication source using the binary log file name and location from the previous step, for example:

```
CHANGE MASTER TO
  MASTER_HOST = 'mydbcluster.cluster-123456789012.us-east-1.rds.amazonaws.com',
  MASTER_PORT = 3306,
  MASTER_USER = 'repl_user',
  MASTER_PASSWORD = 'password',
  MASTER_LOG_FILE = 'mysql-bin-changelog.000031',
  MASTER_LOG_POS = 107;
-- And one of these statements depending on your engine version:
START SLAVE; -- Aurora MySQL version 2
START REPLICA; -- Aurora MySQL version 3
```

</td></tr>
</table>

If replication fails, it can result in a large increase in unintentional I/O on the replica, which can degrade performance. If replication fails or is no longer needed, you can run the mysql.rds_reset_external_master (Aurora MySQL version 2) or mysql.rds_reset_external_source (Aurora MySQL version 3) stored procedure to remove the replication configuration.

Setting a location to stop replication to a read replica

In Aurora MySQL version 3.04 and higher, you can start replication and then stop it at a specified binary log file location using the mysql.rds_start_replication_until (Aurora MySQL version 3) stored procedure.

To start replication to a read replica and stop replication at a specific location

1. Using a MySQL client, connect to the replica Aurora MySQL DB cluster as the master user.
2. Run the mysql.rds_start_replication_until (Aurora MySQL version 3) stored procedure.

   The following example initiates replication and replicates changes until it reaches location `120` in the `mysql-bin-changelog.000777` binary log file. In a disaster recovery scenario, assume that location `120` is just before the disaster.

   ```
   call mysql.rds_start_replication_until(
     'mysql-bin-changelog.000777',
     120);
   ```

Replication stops automatically when the stop point is reached. The following RDS event is generated: `Replication has been stopped since the replica reached the stop point specified by the rds_start_replication_until stored procedure.`

If you use GTID-based replication, use the mysql.rds_start_replication_until_gtid (Aurora MySQL version 3) stored procedure instead of the mysql.rds_start_replication_until (Aurora MySQL version 3) stored procedure. For more information about GTID-based replication, see Using GTID-based replication.

7. Monitor your replica

When you set up MySQL replication with an Aurora MySQL DB cluster, you must monitor failover events for the Aurora MySQL DB cluster when it is the replica target. If a failover occurs, then the DB cluster that is your replica target might be recreated on a new host with a different network address. For information on how to monitor failover events, see Working with Amazon RDS event notification.

You can also monitor how far the replica target is behind the replication source by connecting to the replica target and running the `SHOW SLAVE STATUS` (Aurora MySQL version 2) or `SHOW REPLICA STATUS` (Aurora MySQL version 3) command. In the command output, the `Seconds Behind Master` field tells you how far the replica target is behind the source.

Synchronizing passwords between replication source and target

When you change user accounts and passwords on the replication source using SQL statements, those changes are replicated to the replication target automatically.

If you use the AWS Management Console, the AWS CLI, or the RDS API to change the master password on the replication source, those changes are not automatically replicated to the replication target. If you want to synchronize the master user and master password between the source and target systems, you must make the same change on the replication target yourself.

Stopping replication between Aurora and MySQL or between Aurora and another Aurora DB cluster

To stop binary log replication with a MySQL DB instance, external MySQL database, or another Aurora DB cluster, follow these steps, discussed in detail following in this topic.

1. Stop binary log replication on the replica target

2. Turn off binary logging on the replication source

1. Stop binary log replication on the replica target

Use the following instructions to stop binary log replication for your database engine.

Database engine	Instructions
Aurora MySQL	**To stop binary log replication on an Aurora MySQL DB cluster replica target** Connect to the Aurora DB cluster that is the replica target, and call the mysql.rds_stop_replication procedure.
RDS for MySQL	**To stop binary log replication on an Amazon RDS DB instance** Connect to the RDS DB instance that is the replica target and call the mysql.rds_stop_replication procedure.

Database engine	Instructions
MySQL (external)	**To stop binary log replication on an external MySQL database** Connect to the MySQL database and run the `STOP SLAVE` (version 5.7) or `STOP REPLICA` (version 8.0) command.

2. Turn off binary logging on the replication source

Use the instructions in the following table to turn off binary logging on the replication source for your database engine.

Database engine	Instructions
Aurora MySQL	**To turn off binary logging on an Amazon Aurora DB cluster** 1. Connect to the Aurora DB cluster that is the replication source. 2. Use the mysql.rds_set_configuration procedure and specify the configuration parameter `binlog retention hours`, with the value `NULL`, as shown in the following example. `CALL mysql.rds_set_configuration('binlog retention hours', NULL);` **Note** You can't use the value `0` for `binlog retention hours`. 3. Set the `binlog_format` parameter to `OFF` on the replication source. The `binlog_format` parameter is in the custom DB cluster parameter group associated with your DB cluster. After you've changed the `binlog_format` parameter value, reboot your DB cluster for the change to take effect.

Database engine	Instructions
	For more information, see Amazon Aurora DB cluster and DB instance parameters and Modifying parameters in a DB parameter group.
RDS for MySQL	**To turn off binary logging on an Amazon RDS DB instance** You can't turn off binary logging directly for an Amazon RDS DB instance, but you can turn it off by doing the following: 1. Turn off automated backups for the DB instance. You can turn off automated backups by modifying an existing DB instance and setting the **Backup Retention Period** to 0. For more information, see Modifying an Amazon RDS DB instance and Working with backups in the *Amazon Relational Database Service User Guide*. 2. Delete all read replicas for the DB instance. For more information, see Working with read replicas of MariaDB, MySQL, and PostgreSQL DB instances in the *Amazon Relational Database Service User Guide*.

MySQL (external)	**To turn off binary logging on an external MySQL database**
	Connect to the MySQL database and call the `STOP REPLICATION` command.
	1. From a command shell, stop the `mysqld` service,
	`sudo service mysqld stop`
	2. Edit the `my.cnf` file (this file is usually under `/etc`).
	`sudo vi /etc/my.cnf`
	Delete the `log_bin` and `server_id` options from the `[mysqld]` section.
	For more information, see Setting the replication source configuration in the MySQL documentation.
	3. Start the mysql service.
	`sudo service mysqld start`

Using Amazon Aurora to scale reads for your MySQL database

You can use Amazon Aurora with your MySQL DB instance to take advantage of the read scaling capabilities of Amazon Aurora and expand the read workload for your MySQL DB instance. To use Aurora to scale reads for your MySQL DB instance, create an Amazon Aurora MySQL DB cluster and make it a read replica of your MySQL DB instance. This applies to an RDS for MySQL DB instance, or a MySQL database running external to Amazon RDS.

For information on creating an Amazon Aurora DB cluster, see Creating an Amazon Aurora DB cluster.

When you set up replication between your MySQL DB instance and your Amazon Aurora DB cluster, be sure to follow these guidelines:

- Use the Amazon Aurora DB cluster endpoint address when you reference your Amazon Aurora MySQL DB cluster. If a failover occurs, then the Aurora Replica that is promoted to the primary instance for the Aurora MySQL DB cluster continues to use the DB cluster endpoint address.
- Maintain the binlogs on your writer instance until you have verified that they have been applied to the Aurora Replica. This maintenance ensures that you can restore your writer instance in the event of a failure.

⚠ **Important**

When using self-managed replication, you're responsible for monitoring and resolving any replication issues that may occur. For more information, see Diagnosing and resolving lag between read replicas.

ⓘ **Note**

The permissions required to start replication on an Aurora MySQL DB cluster are restricted and not available to your Amazon RDS master user. Therefore, you must use the mysql.rds_set_external_master (Aurora MySQL version 2) or mysql.rds_set_external_source (Aurora MySQL version 3) and mysql.rds_start_replication procedures to set up replication between your Aurora MySQL DB cluster and your MySQL DB instance.

Start replication between an external source instance and an Aurora MySQL DB cluster

1. Make the source MySQL DB instance read-only:

```
mysql> FLUSH TABLES WITH READ LOCK;
mysql> SET GLOBAL read_only = ON;
```

2. Run the `SHOW MASTER STATUS` command on the source MySQL DB instance to determine the binlog location. You receive output similar to the following example:

```
File                        Position
------------------------------------
 mysql-bin-changelog.000031      107
------------------------------------
```

3. Copy the database from the external MySQL DB instance to the Amazon Aurora MySQL DB cluster using `mysqldump`. For very large databases, you might want to use the procedure in Importing data to a MySQL or MariaDB DB instance with reduced downtime in the *Amazon Relational Database Service User Guide*.

   For Linux, macOS, or Unix:

```
mysqldump \
    --databases <database_name> \
    --single-transaction \
    --compress \
    --order-by-primary \
    -u local_user \
    -p local_password | mysql \
        --host aurora_cluster_endpoint_address \
        --port 3306 \
        -u RDS_user_name \
        -p RDS_password
```

   For Windows:

```
mysqldump ^
    --databases <database_name> ^
    --single-transaction ^
    --compress ^
    --order-by-primary ^
    -u local_user ^
    -p local_password | mysql ^
        --host aurora_cluster_endpoint_address ^
        --port 3306 ^
        -u RDS_user_name ^
        -p RDS_password
```

Note

Make sure that there is not a space between the -p option and the entered password.

Use the --host, --user (-u), --port and -p options in the mysql command to specify the hostname, user name, port, and password to connect to your Aurora DB cluster. The host name is the DNS name from the Amazon Aurora DB cluster endpoint, for example, mydbcluster.cluster-123456789012.us-east-1.rds.amazonaws.com. You can find the endpoint value in the cluster details in the Amazon RDS Management Console.

4. Make the source MySQL DB instance writeable again:

```
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

For more information on making backups for use with replication, see Backing up a source or replica by making it read only in the MySQL documentation.

5. In the Amazon RDS Management Console, add the IP address of the server that hosts the source MySQL database to the VPC security group for the Amazon Aurora DB cluster. For more information on modifying a VPC security group, see Security groups for your VPC in the *Amazon Virtual Private Cloud User Guide*.

You might also need to configure your local network to permit connections from the IP address of your Amazon Aurora DB cluster, so that it can communicate with your source MySQL instance. To find the IP address of the Amazon Aurora DB cluster, use the host command.

```
host aurora_endpoint_address
```

The host name is the DNS name from the Amazon Aurora DB cluster endpoint.

6. Using the client of your choice, connect to the external MySQL instance and create a MySQL user to be used for replication. This account is used solely for replication and must be restricted to your domain to improve security. The following is an example.

```
CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED BY 'password';
```

7. For the external MySQL instance, grant `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges to your replication user. For example, to grant the `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges on all databases for the '`repl_user`' user for your domain, issue the following command.

```
GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com'
    IDENTIFIED BY 'password';
```

8. Take a manual snapshot of the Aurora MySQL DB cluster to be the read replica before setting up replication. If you need to reestablish replication with the DB cluster as a read replica, you can restore the Aurora MySQL DB cluster from this snapshot instead of having to import the data from your MySQL DB instance into a new Aurora MySQL DB cluster.

9. Make the Amazon Aurora DB cluster the replica. Connect to the Amazon Aurora DB cluster as the master user and identify the source MySQL database as the replication master by using the mysql.rds_set_external_master (Aurora MySQL version 2) or mysql.rds_set_external_source (Aurora MySQL version 3) and mysql.rds_start_replication procedures.

   Use the master log file name and master log position that you determined in Step 2. The following is an example.

```
For Aurora MySQL version 2:
CALL mysql.rds_set_external_master ('mymasterserver.mydomain.com', 3306,
    'repl_user', 'password', 'mysql-bin-changelog.000031', 107, 0);

For Aurora MySQL version 3:
CALL mysql.rds_set_external_source ('mymasterserver.mydomain.com', 3306,
    'repl_user', 'password', 'mysql-bin-changelog.000031', 107, 0);
```

10.On the Amazon Aurora DB cluster, call the mysql.rds_start_replication procedure to start replication.

```
CALL mysql.rds_start_replication;
```

After you have established replication between your source MySQL DB instance and your Amazon Aurora DB cluster, you can add Aurora Replicas to your Amazon Aurora DB cluster. You can then connect to the Aurora Replicas to read scale your data. For information on creating an Aurora Replica, see Adding Aurora Replicas to a DB cluster.

Optimizing binary log replication

Following, you can learn how to optimize binary log replication performance and troubleshoot related issues in Aurora MySQL.

Tip

This discussion presumes that you are familiar with the MySQL binary log replication mechanism and how it works. For background information, see Replication Implementation in the MySQL documentation.

Multithreaded binary log replication

With multithreaded binary log replication, a SQL thread reads events from the relay log and queues them up for SQL worker threads to apply. The SQL worker threads are managed by a coordinator thread. The binary log events are applied in parallel when possible.

Multithreaded binary log replication is supported in Aurora MySQL version 3, and in Aurora MySQL version 2.12.1 and higher.

When an Aurora MySQL DB instance is configured to use binary log replication, by default the replica instance uses single-threaded replication for Aurora MySQL versions lower than 3.04. To enable multithreaded replication, you update the `replica_parallel_workers` parameter to a value greater than zero in your custom parameter group.

For Aurora MySQL version 3.04 and higher, replication is multithreaded by default, with `replica_parallel_workers` set to 4. You can modify this parameter in your custom parameter group.

The following configuration options help you to fine-tune multithreaded replication. For usage information, see Replication and Binary Logging Options and Variables in the *MySQL Reference Manual*.

Optimal configuration depends on several factors. For example, performance for binary log replication is influenced by your database workload characteristics and the DB instance class the replica is running on. Thus, we recommend that you thoroughly test all changes to these configuration parameters before applying new parameter settings to a production instance:

- `binlog_group_commit_sync_delay`
- `binlog_group_commit_sync_no_delay_count`
- `binlog_transaction_dependency_history_size`
- `binlog_transaction_dependency_tracking`
- `replica_preserve_commit_order`
- `replica_parallel_type`
- `replica_parallel_workers`

In Aurora MySQL version 3.06 and higher, you can improve performance for binary log replicas when replicating transactions for large tables with more than one secondary index. This feature introduces a thread pool to apply secondary index changes in parallel on a binlog replica. The feature is controlled by the `aurora_binlog_replication_sec_index_parallel_workers` DB cluster parameter, which controls the total number of parallel threads available to apply the secondary index changes. The parameter is set to `0` (disabled) by default. Enabling this feature doesn't require an instance restart. To enable this feature, stop ongoing replication, set the desired number of parallel worker threads, and then start replication again.

You can also use this parameter as a global variable, where *n* is the number of parallel worker threads:

```
SET global aurora_binlog_replication_sec_index_parallel_workers=n;
```

Optimizing binlog replication (Aurora MySQL 2.10 and higher)

In Aurora MySQL 2.10 and higher, Aurora automatically applies an optimization known as the binlog I/O cache to binary log replication. By caching the most recently committed binlog events, this optimization is designed to improve binlog dump thread performance while limiting the impact to foreground transactions on the binlog source instance.

> **Note**
>
> This memory used for this feature is independent of the MySQL `binlog_cache` setting.

This feature doesn't apply to Aurora DB instances that use the db.t2 and db.t3 instance classes.

You don't need to adjust any configuration parameters to turn on this optimization. In particular, if you adjust the configuration parameter aurora_binlog_replication_max_yield_seconds to a nonzero value in earlier Aurora MySQL versions, set it back to zero for Aurora MySQL 2.10 and higher.

The status variables aurora_binlog_io_cache_reads and aurora_binlog_io_cache_read_requests are available in Aurora MySQL 2.10 and higher. These status variables help you to monitor how often the data is read from the binlog I/O cache.

- aurora_binlog_io_cache_read_requests shows the number of binlog I/O read requests from the cache.
- aurora_binlog_io_cache_reads shows the number of binlog I/O reads that retrieve information from the cache.

The following SQL query computes the percentage of binlog read requests that take advantage of the cached information. In this case, the closer the ratio is to 100, the better it is.

```
mysql> SELECT
  (SELECT VARIABLE_VALUE FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    WHERE VARIABLE_NAME='aurora_binlog_io_cache_reads')
  / (SELECT VARIABLE_VALUE FROM INFORMATION_SCHEMA.GLOBAL_STATUS
    WHERE VARIABLE_NAME='aurora_binlog_io_cache_read_requests')
  * 100
  as binlog_io_cache_hit_ratio;
+---------------------------+
| binlog_io_cache_hit_ratio |
+---------------------------+
|         99.99847949080622 |
+---------------------------+
```

The binlog I/O cache feature also includes new metrics related to the binlog dump threads. *Dump threads* are the threads that are created when new binlog replicas are connected to the binlog source instance.

The dump thread metrics are printed to the database log every 60 seconds with the prefix `[Dump thread metrics]`. The metrics include information for each binlog replica such as `Secondary_id`, `Secondary_uuid`, binlog file name, and the position that each replica is reading. The metrics also include `Bytes_behind_primary` representing the distance in bytes between replication source and replica. This metric measures the lag of the replica I/O thread. That figure is different from the lag of the replica SQL applier thread, which is represented by the `seconds_behind_master` metric on the binlog replica. You can determine whether binlog replicas are catching up to the source or falling behind by checking whether the distance decreases or increases.

Optimizing binlog replication (Aurora MySQL version 2 through 2.09)

To optimize binary log replication for Aurora MySQL, you adjust the following cluster-level optimization parameters. These parameters help you to specify the right balance between latency on the binlog source instance and replication lag.

- `aurora_binlog_use_large_read_buffer`
- `aurora_binlog_read_buffer_size`
- `aurora_binlog_replication_max_yield_seconds`

> **Note**
>
> For MySQL 5.7-compatible clusters, you can use these parameters in Aurora MySQL version 2 through 2.09.*. In Aurora MySQL 2.10.0 and higher, these parameters are superseded by the binlog I/O cache optimization and you don't need to use them.

Topics

Overview of the large read buffer and max-yield optimizations

You might experience reduced binary log replication performance when the binary log dump thread accesses the Aurora cluster volume while the cluster processes a high number of transactions. You can use the parameters `aurora_binlog_use_large_read_buffer`, `aurora_binlog_replication_max_yield_seconds`, and `aurora_binlog_read_buffer_size` to help minimize this type of contention.

Suppose that you have a situation where `aurora_binlog_replication_max_yield_seconds` is set to greater than 0 and the current binlog file of the dump thread is active. In this case, the binary log dump thread waits up to a specified number of seconds for the current binlog file to be filled by transactions. This wait period avoids contention that can arise from replicating each binlog event individually. However, doing so increases the replica lag for binary log replicas. Those replicas can fall behind the source by the same number of seconds as the `aurora_binlog_replication_max_yield_seconds` setting.

The current binlog file means the binlog file that the dump thread is currently reading to perform replication. We consider that a binlog file is active when the binlog file is updating or open to be updated by incoming transactions. After Aurora MySQL fills up the active binlog file, MySQL creates and switches to a new binlog file. The old binlog file becomes inactive. It isn't updated by incoming transactions any longer.

> **Note**
>
> Before adjusting these parameters, measure your transaction latency and throughput over time. You might find that binary log replication performance is stable and has low latency even if there is occasional contention.

`aurora_binlog_use_large_read_buffer`

If this parameter is set to 1, Aurora MySQL optimizes binary log replication based on the settings of the parameters `aurora_binlog_read_buffer_size` and `aurora_binlog_replication_max_yield_seconds`. If `aurora_binlog_use_large_read_buffer` is 0, Aurora MySQL ignores the values of the `aurora_binlog_read_buffer_size` and `aurora_binlog_replication_max_yield_seconds` parameters.

`aurora_binlog_read_buffer_size`

Binary log dump threads with larger read buffer minimize the number of read I/O operations by reading more events for each I/O. The parameter `aurora_binlog_read_buffer_size` sets the read buffer size. The large read buffer can reduce binary log contention for workloads that generate a large amount of binlog data.

> **Note**
>
> This parameter only has an effect when the cluster also has the setting `aurora_binlog_use_large_read_buffer=1`.
> Increasing the size of the read buffer doesn't affect the performance of binary log replication. Binary log dump threads don't wait for updating transactions to fill up the read buffer.

`aurora_binlog_replication_max_yield_seconds`

If your workload requires low transaction latency, and you can tolerate some replication lag, you can increase the `aurora_binlog_replication_max_yield_seconds` parameter. This parameter controls the maximum yield property of binary log replication in your cluster.

> **Note**
>
> This parameter only has an effect when the cluster also has the setting `aurora_binlog_use_large_read_buffer=1`.

Aurora MySQL recognizes any change to the `aurora_binlog_replication_max_yield_seconds` parameter value immediately. You don't need to restart the DB instance. However, when you turn on this setting, the dump thread only starts to yield when the current binlog file reaches its maximum size of 128 MB and is rotated to a new file.

Related parameters

Use the following DB cluster parameters to turn on binlog optimization.

Parameter	Default	Valid Values	Description
aurora_binlog_use_large_read_buffer	1	0, 1	Switch for turning on the feature of replication improvement. When its value is 1, the binary log dump thread uses aurora_binlog_read_buffer_size for binary log replication; otherwise default buffer size (8K) is used. Not used in Aurora MySQL version 3.
aurora_binlog_read_buffer_size	5242880	8192-536870912	Read buffer size used by binary log dump thread when the parameter aurora_binlog_use_large_read_buffer is set to 1. Not used in Aurora MySQL version 3.
aurora_binlog_replication_max_yield_seconds	0	0-36000	For Aurora MySQL version 2.07.*, the maximum accepted value is 45. You can tune it to a higher value on 2.09 and later versions.

Parameter	Default	Valid Values	Description
			For version 2, this parameter works only when the parameter `aurora_binlog_use_large_read_buffer` is set to 1.

Enabling the max-yield mechanism for binary log replication

You can turn on the binary log replication max-yield optimization as follows. Doing so minimizes latency for transactions on the binlog source instance. However, you might experience higher replication lag.

To turn on the max-yield binlog optimization for an Aurora MySQL cluster

1. Create or edit a DB cluster parameter group using the following parameter settings:
   - `aurora_binlog_use_large_read_buffer`: turn on with a value of `ON` or 1.
   - `aurora_binlog_replication_max_yield_seconds`: specify a value greater than 0.
2. Associate the DB cluster parameter group with the Aurora MySQL cluster that works as the binlog source. To do so, follow the procedures in [Working with parameter groups](#).
3. Confirm that the parameter change takes effect. To do so, run the following query on the binlog source instance.

```
SELECT @@aurora_binlog_use_large_read_buffer,
 @@aurora_binlog_replication_max_yield_seconds;
```

   Your output should be similar to the following.

```
+----------------------------------------
+------------------------------------------------+
| @@aurora_binlog_use_large_read_buffer |
 @@aurora_binlog_replication_max_yield_seconds |
+---------------------------------------
+-----------------------------------------------+
|                                     1 |
  45 |
+---------------------------------------
+-----------------------------------------------+
```

Turning off the binary log replication max-yield optimization

You can turn off the binary log replication max-yield optimization as follows. Doing so minimizes replication lag. However, you might experience higher latency for transactions on the binlog source instance.

To turn off the max-yield optimization for an Aurora MySQL cluster

1. Make sure that the DB cluster parameter group associated with the Aurora MySQL cluster has `aurora_binlog_replication_max_yield_seconds` set to 0. For more information about setting configuration parameters using parameter groups, see Working with parameter groups.
2. Confirm that the parameter change takes effect. To do so, run the following query on the binlog source instance.

   ```
   SELECT @@aurora_binlog_replication_max_yield_seconds;
   ```

   Your output should be similar to the following.

   ```
   +-----------------------------------------------+
   | @@aurora_binlog_replication_max_yield_seconds |
   +-----------------------------------------------+
   |                                             0 |
   +-----------------------------------------------+
   ```

Turning off the large read buffer

You can turn off the entire large read buffer feature as follows.

To turn off the large binary log read buffer for an Aurora MySQL cluster

1. Reset the `aurora_binlog_use_large_read_buffer` to `OFF` or 0.

Make sure that the DB cluster parameter group associated with the Aurora MySQL cluster has `aurora_binlog_use_large_read_buffer` set to 0. For more information about setting configuration parameters using parameter groups, see Working with parameter groups.

2. On the binlog source instance, run the following query.

```
SELECT @@ aurora_binlog_use_large_read_buffer;
```

Your output should be similar to the following.

```
+---------------------------------------+
| @@aurora_binlog_use_large_read_buffer |
+---------------------------------------+
|                                     0 |
+---------------------------------------+
```

Setting up enhanced binlog

Enhanced binlog reduces the compute performance overhead caused by turning on binlog, which can reach up to 50% in certain cases. With enhanced binlog, this overhead can be reduced to about 13%. To reduce overhead, enhanced binlog writes the binary and transactions logs to storage in parallel, which minimizes the data written at the transaction commit time.

Using enhanced binlog also improves database recovery time after restarts and failovers by up to 99% compared to community MySQL binlog. The enhanced binlog is compatible with existing binlog-based workloads, and you interact with it the same way you interact with the community MySQL binlog.

Enhanced binlog is available on Aurora MySQL version 3.03.1 and higher.

Topics

Configuring enhanced binlog parameters

You can switch between community MySQL binlog and enhanced binlog by turning on/off the enhanced binlog parameters. The existing binlog consumers can continue to read and consume the binlog files without any gaps in the binlog file sequence.

To turn on enhanced binlog

Parameter	Default	Description
`binlog_format`	–	Set the `binlog_format` parameter to the binary logging format of your choice to turn on enhanced binlog. Make sure the `binlog_format parameter` isn't set to OFF. For more information, see [Configuring Aurora MySQL binary logging]().
`aurora_enhanced_binlog`	`0`	Set the value of this parameter to 1 in the DB cluster parameter group associated with the Aurora MySQL cluster. When you change the value of this parameter, you must reboot the writer instance when the `DBClusterParameterGroupStatus` value is shown as `pending-reboot` .
`binlog_backup`	`1`	Turn off this parameter to turn on enhanced binlog. To do so, set the value of this parameter to `0`.
`binlog_replication_globaldb`	`1`	Turn off this parameter to turn on enhanced binlog. To

Parameter	Default	Description
		do so, set the value of this parameter to `0`.

Important

You can turn off the `binlog_backup` and `binlog_replication_globaldb` parameters only when you use enhanced binlog.

To turn off the enhanced binlog

Parameter	Description
`aurora_enhanced_binlog`	Set the value of this parameter to `0` in the DB cluster parameter group associated with the Aurora MySQL cluster. Whenever you change the value of this parameter, you must reboot the writer instance when the `DBClusterParameterGroupStatus` value is shown as `pending-reboot`.
`binlog_backup`	Turn on this parameter when you turn off enhanced binlog. To do so, set the value of this parameter to 1.
`binlog_replication_globaldb`	Turn on this parameter when you turn off enhanced binlog. To do so, set the value of this parameter to 1.

To check whether enhanced binlog is turned on, use the following command in the MySQL client:

```
mysql>show status like 'aurora_enhanced_binlog';

+-------------------------+--------+
| Variable_name           | Value  |
+-------------------------+--------+
| aurora_enhanced_binlog | ACTIVE |
+------------------------+--------+
1 row in set (0.00 sec)
```

When enhanced binlog is turned on, the output shows ACTIVE for aurora_enhanced_binlog.

Other related parameters

When you turn on the enhanced binlog, the following parameters are affected:

- The max_binlog_size parameter is visible but not modifiable. It's default value 134217728 is automatically adjusted to 268435456 when enhanced binlog is turned on.
- Unlike in community MySQL binlog, the binlog_checksum doesn't act as a dynamic parameter when the enhanced binlog is turned on. For the change to this parameter to take effect, you must manually reboot the DB cluster even when the ApplyMethod is immediate.
- The value you set on the binlog_order_commits parameter has no effect on the order of the commits when enhanced binlog is turned on. The commits are always ordered without any further performance implications.

Differences between enhanced binlog and community MySQL binlog

Enhanced binlog interacts differently with clones, backups, and Aurora global database when compared to community MySQL binlog. We recommend that you understand the following differences before using enhanced binlog.

- Enhanced binlog files from the source DB cluster aren't available on a cloned DB cluster.
- Enhanced binlog files aren't included in Aurora backups. Therefore, enhanced binlog files from the source DB cluster aren't available after restoring a DB cluster despite any retention period set on it.
- When used with an Aurora global database, the enhanced binlog files of the primary DB cluster aren't replicated to the DB cluster in the secondary regions.

Examples

The following examples illustrate the differences between enhanced binlog and community MySQL binlog.

On a restored or cloned DB cluster

When enhanced binlog is turned on, the historical binlog files aren't available in the restored or cloned DB cluster. After a restore or clone operation, if binlog is turned on, the new DB cluster starts writing its own sequence of binlog files, starting from 1 (mysql-bin-changelog.000001).

To turn on enhanced binlog after a restore or clone operation, set the required DB cluster parameters on the restored or cloned DB cluster. For more information, see Configuring enhanced binlog parameters.

Example Clone or restore operation performed when enhanced binlog is turned on

Source DB Cluster:

```
mysql> show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        |
| mysql-bin-changelog.000002 |       156 | No        |
| mysql-bin-changelog.000003 |       156 | No        |
| mysql-bin-changelog.000004 |       156 | No        | --> Enhanced Binlog turned on
| mysql-bin-changelog.000005 |       156 | No        | --> Enhanced Binlog turned on
| mysql-bin-changelog.000006 |       156 | No        | --> Enhanced Binlog turned on
+----------------------------+-----------+-----------+
6 rows in set (0.00 sec)
```

On a restored or cloned DB cluster, binlog files aren't backed up when enhanced binlog is turned on. To avoid discontinuity in the binlog data, the binlog files written before turning on the enhanced binlog are also not available.

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        | --> New sequence of Binlog files
+----------------------------+-----------+-----------+
1 row in set (0.00 sec)
```

Example Clone or restore operation performed when enhanced binlog is turned off

Source DB cluster:

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        |
| mysql-bin-changelog.000002 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000003 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000004 |       156 | No        |
| mysql-bin-changelog.000005 |       156 | No        |
| mysql-bin-changelog.000006 |       156 | No        |
+----------------------------+-----------+-----------+
6 rows in set (0.00 sec)
```

On a restored or cloned DB cluster, binlog files written after turning off the enhanced binlog are available.

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000004 |       156 | No        |
| mysql-bin-changelog.000005 |       156 | No        |
| mysql-bin-changelog.000006 |       156 | No        |
+----------------------------+-----------+-----------+
1 row in set (0.00 sec)
```

On an Amazon Aurora global database

On an Amazon Aurora global database, the binlog data of the primary DB cluster isn't replicated to the secondary DB clusters. After a cross-Region failover process, the binlog data isn't available in the newly promoted primary DB cluster. If binlog is turned on, the newly promoted DB cluster starts its own sequence of binlog files, starting from 1 (mysql-bin-changelog.000001).

To turn on enhanced binlog after failover, you must set the required DB cluster parameters on the secondary DB cluster. For more information, see Configuring enhanced binlog parameters.

Example Global database failover operation is performed when enhanced binlog is turned on

Old primary DB Cluster (before failover):

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        |
| mysql-bin-changelog.000002 |       156 | No        |
| mysql-bin-changelog.000003 |       156 | No        |
| mysql-bin-changelog.000004 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000005 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000006 |       156 | No        | --> Enhanced Binlog enabled
+----------------------------+-----------+-----------+
6 rows in set (0.00 sec)
```

New primary DB cluster (after failover):

Binlog files aren't replicated to secondary regions when enhanced binlog is turned on. To avoid discontinuity in the binlog data, the binlog files written before turning on the enhanced binlog aren't available.

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        | --> Fresh sequence of Binlog
 files
+----------------------------+-----------+-----------+
1 row in set (0.00 sec)
```

Example Global database failover operation is performed when enhanced binlog is turned off

Source DB Cluster:

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000001 |       156 | No        |
| mysql-bin-changelog.000002 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000003 |       156 | No        | --> Enhanced Binlog enabled
| mysql-bin-changelog.000004 |       156 | No        |
| mysql-bin-changelog.000005 |       156 | No        |
| mysql-bin-changelog.000006 |       156 | No        |
+----------------------------+-----------+-----------+
6 rows in set (0.00 sec)
```

Restored or cloned DB cluster:

Binlog files that are written after turning off the enhanced binlog are replicated and are available in the newly promoted DB cluster.

```
mysql>show binary logs;

+----------------------------+-----------+-----------+
| Log_name                   | File_size | Encrypted |
+----------------------------+-----------+-----------+
| mysql-bin-changelog.000004 |       156 | No        |
| mysql-bin-changelog.000005 |       156 | No        |
| mysql-bin-changelog.000006 |       156 | No        |
+----------------------------+-----------+-----------+
3 rows in set (0.00 sec)
```

Amazon CloudWatch metrics for enhanced binlog

The following Amazon CloudWatch metrics are published only when enhanced binlog is turned on.

CloudWatch metric	Description	Units
ChangeLogBytesUsed	The amount of storage used by the enhanced binlog.	Bytes

CloudWatch metric	Description	Units
ChangeLogReadIOPs	The number of read I/O operations performed in the enhanced binlog within a 5-minute interval.	Count per 5 minutes
ChangeLogWriteIOPs	The number of write disk I/O operations performed in the enhanced binlog within a 5-minute interval.	Count per 5 minutes

Enhanced binlog limitations

The following limitations apply to Amazon Aurora DB clusters when enhanced binlog is turned on.

- Enhanced binlog is only supported on Aurora MySQL version3.03.1 and higher.
- The enhanced binlog files written on the primary DB cluster aren't copied to the cloned or restored DB clusters.
- When used with Amazon Aurora global database, the enhanced binlog files of the primary DB cluster aren't replicated to the secondary DB clusters. Therefore, after the failover process, the historical binlog data isn't available in the new primary DB cluster.
- The following binlog configuration parameters are ignored:
  - `binlog_group_commit_sync_delay`
  - `binlog_group_commit_sync_no_delay_count`
  - `binlog_max_flush_queue_time`
- You can't drop or rename a corrupted table in a database. To drop these tables, you can contact AWS Support.
- The binlog I/O cache is disabled when enhanced binlog is turned on. For more information, see Optimizing binary log replication.

> **Note**
>
> Enhanced binlog provides similar read performance improvements as binlog I/O cache and better write performance improvements.

- The backtrack feature is not supported. Enhanced binlog can't be turned on in a DB cluster under the following conditions:
  - DB cluster with the backtrack feature currently enabled.
  - DB cluster where the backtrack feature was previously enabled, but is now disabled.
  - DB cluster restored from a source DB cluster or a snapshot with the backtrack feature enabled.

Using GTID-based replication

The following content explains how to use global transaction identifiers (GTIDs) with binary log (binlog) replication between an Aurora MySQL cluster and an external source.

> **Note**
>
> For Aurora, you can use this feature only with Aurora MySQL clusters that use binlog replication to or from an external MySQL database. The other database might be an Amazon RDS MySQL instance, an on-premises MySQL database, or an Aurora DB cluster in a different AWS Region. To learn how to configure that kind of replication, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

If you use binlog replication and aren't familiar with GTID-based replication with MySQL, see Replication with global transaction identifiers in the MySQL documentation.

GTID-based replication is supported for Aurora MySQL version 2 and 3.

Topics

Overview of global transaction identifiers (GTIDs)

Global transaction identifiers (GTIDs) are unique identifiers generated for committed MySQL transactions. You can use GTIDs to make binlog replication simpler and easier to troubleshoot.

Note

When Aurora synchronizes data among the DB instances in a cluster, that replication mechanism doesn't involve the binary log (binlog). For Aurora MySQL, GTID-based replication only applies when you also use binlog replication to replicate into or out of an Aurora MySQL DB cluster from an external MySQL-compatible database.

MySQL uses two different types of transactions for binlog replication:

- *GTID transactions* – Transactions that are identified by a GTID.
- *Anonymous transactions* – Transactions that don't have a GTID assigned.

In a replication configuration, GTIDs are unique across all DB instances. GTIDs simplify replication configuration because when you use them, you don't have to refer to log file positions. GTIDs also make it easier to track replicated transactions and determine whether the source instance and replicas are consistent.

You typically use GTID-based replication with Aurora when replicating from an external MySQL-compatible database into an Aurora cluster. You can set up this replication configuration as part of a migration from an on-premises or Amazon RDS database into Aurora MySQL. If the external database already uses GTIDs, enabling GTID-based replication for the Aurora cluster simplifies the replication process.

You configure GTID-based replication for an Aurora MySQL cluster by first setting the relevant configuration parameters in a DB cluster parameter group. You then associate that parameter group with the cluster.

Parameters for GTID-based replication

Use the following parameters to configure GTID-based replication.

Parameter	Valid values	Description
`gtid_mode`	`OFF`, `OFF_PERMISSIVE` , `ON_PERMISSIVE` , `ON`	`OFF` specifies that new transactions are anonymous transactions (that is, don't have GTIDs), and a transaction must be anonymous to be replicated.

Parameter	Valid values	Description
		`OFF_PERMISSIVE` specifies that new transactions are anonymous transactions, but all transactions can be replicated. `ON_PERMISSIVE` specifies that new transactions are GTID transactions, but all transactions can be replicated. `ON` specifies that new transactions are GTID transactions, and a transaction must be a GTID transaction to be replicated.
`enforce_gtid_consistency`	`OFF`, `ON`, `WARN`	`OFF` allows transactions to violate GTID consistency. `ON` prevents transactions from violating GTID consistency. `WARN` allows transactions to violate GTID consistency but generates a warning when a violation occurs.

Note

In the AWS Management Console, the `gtid_mode` parameter appears as `gtid-mode`.

For GTID-based replication, use these settings for the DB cluster parameter group for your Aurora MySQL DB cluster:

- `ON` and `ON_PERMISSIVE` apply only to outgoing replication from an Aurora MySQL cluster. Both of these values cause your Aurora DB cluster to use GTIDs for transactions that are replicated to an external database. `ON` requires that the external database also use GTID-based replication. `ON_PERMISSIVE` makes GTID-based replication optional on the external database.

- `OFF_PERMISSIVE`, if set, means that your Aurora DB cluster can accept incoming replication from an external database. It can do this whether the external database uses GTID-based replication or not.
- `OFF`, if set, means that your Aurora DB cluster only accepts incoming replication from external databases that don't use GTID-based replication.

> **Tip**
>
> Incoming replication is the most common binlog replication scenario for Aurora MySQL clusters. For incoming replication, we recommend that you set the GTID mode to `OFF_PERMISSIVE`. That setting allows incoming replication from external databases regardless of the GTID settings at the replication source.

For more information about parameter groups, see Working with parameter groups.

Configuring GTID-based replication for an Aurora MySQL cluster

When GTID-based replication is enabled for an Aurora MySQL DB cluster, the GTID settings apply to both inbound and outbound binlog replication.

To enable GTID-based replication for an Aurora MySQL cluster

1. Create or edit a DB cluster parameter group using the following parameter settings:
   - `gtid_mode` – `ON` or `ON_PERMISSIVE`
   - `enforce_gtid_consistency` – `ON`
2. Associate the DB cluster parameter group with the Aurora MySQL cluster. To do so, follow the procedures in Working with parameter groups.
3. (Optional) Specify how to assign GTIDs to transactions that don't include them. To do so, call the stored procedure in mysql.rds_assign_gtids_to_anonymous_transactions (Aurora MySQL version 3).

Disabling GTID-based replication for an Aurora MySQL DB cluster

You can disable GTID-based replication for an Aurora MySQL DB cluster. Doing so means that the Aurora cluster can't perform inbound or outbound binlog replication with external databases that use GTID-based replication.

> **Note**
>
> In the following procedure, *read replica* means the replication target in an Aurora configuration with binlog replication to or from an external database. It doesn't mean the read-only Aurora Replica DB instances. For example, when an Aurora cluster accepts incoming replication from an external source, the Aurora primary instance acts as the read replica for binlog replication.

For more details about the stored procedures mentioned in this section, see Aurora MySQL stored procedures.

To disable GTID-based replication for an Aurora MySQL DB cluster

1. On the Aurora replicas, run the following procedure:

   For version 3

   ```
   CALL mysql.rds_set_source_auto_position(0);
   ```

   For version 2

   ```
   CALL mysql.rds_set_master_auto_position(0);
   ```

2. Reset the `gtid_mode` to `ON_PERMISSIVE`.

   a. Make sure that the DB cluster parameter group associated with the Aurora MySQL cluster has `gtid_mode` set to `ON_PERMISSIVE`.

      For more information about setting configuration parameters using parameter groups, see Working with parameter groups.

   b. Restart the Aurora MySQL DB cluster.

3. Reset the `gtid_mode` to `OFF_PERMISSIVE`.

   a. Make sure that the DB cluster parameter group associated with the Aurora MySQL cluster has `gtid_mode` set to `OFF_PERMISSIVE`.

   b. Restart the Aurora MySQL DB cluster.

4. Wait for all of the GTID transactions to be applied on the Aurora primary instance. To check that these are applied, do the following steps:

   a. On the Aurora primary instance, run the `SHOW MASTER STATUS` command.

      Your output should be similar to the following output.

      ```
      File                        Position
      ------------------------------------
      mysql-bin-changelog.000031      107
      ------------------------------------
      ```

      Note the file and position in your output.

   b. On each read replica, use the file and position information from its source instance in the previous step to run the following query:

      For version 3

      ```
      SELECT SOURCE_POS_WAIT('file', position);
      ```

      For version 2

      ```
      SELECT MASTER_POS_WAIT('file', position);
      ```

      For example, if the file name is `mysql-bin-changelog.000031` and the position is `107`, run the following statement:

      For version 3

      ```
      SELECT SOURCE_POS_WAIT('mysql-bin-changelog.000031', 107);
      ```

      For version 2

```
SELECT MASTER_POS_WAIT('mysql-bin-changelog.000031', 107);
```

5. Reset the GTID parameters to disable GTID-based replication.

   a. Make sure that the DB cluster parameter group associated with the Aurora MySQL cluster has the following parameter settings:

      - `gtid_mode` – `OFF`
      - `enforce_gtid_consistency` – `OFF`

   b. Restart the Aurora MySQL DB cluster.

Integrating Amazon Aurora MySQL with other AWS services

Amazon Aurora MySQL integrates with other AWS services so that you can extend your Aurora MySQL DB cluster to use additional capabilities in the AWS Cloud. Your Aurora MySQL DB cluster can use AWS services to do the following:

- Synchronously or asynchronously invoke an AWS Lambda function using the native functions `lambda_sync` or `lambda_async`. For more information, see Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.
- Load data from text or XML files stored in an Amazon Simple Storage Service (Amazon S3) bucket into your DB cluster using the `LOAD DATA FROM S3` or `LOAD XML FROM S3` command. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.
- Save data to text files stored in an Amazon S3 bucket from your DB cluster using the `SELECT INTO OUTFILE S3` command. For more information, see Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket.
- Automatically add or remove Aurora Replicas with Application Auto Scaling. For more information, see Using Amazon Aurora Auto Scaling with Aurora Replicas.
- Perform sentiment analysis with Amazon Comprehend, or a wide variety of machine learning algorithms with SageMaker. For more information, see Using Amazon Aurora machine learning.

Aurora secures the ability to access other AWS services by using AWS Identity and Access Management (IAM). You grant permission to access other AWS services by creating an IAM role with the necessary permissions, and then associating the role with your DB cluster. For details and instructions on how to permit your Aurora MySQL DB cluster to access other AWS services on your behalf, see Authorizing Amazon Aurora MySQL to access other AWS services on your behalf.

Authorizing Amazon Aurora MySQL to access other AWS services on your behalf

For your Aurora MySQL DB cluster to access other services on your behalf, create and configure an AWS Identity and Access Management (IAM) role. This role authorizes database users in your DB cluster to access other AWS services. For more information, see Setting up IAM roles to access AWS services.

You must also configure your Aurora DB cluster to allow outbound connections to the target AWS service. For more information, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

If you do so, your database users can perform these actions using other AWS services:

- Synchronously or asynchronously invoke an AWS Lambda function using the native functions `lambda_sync` or `lambda_async`. Or, asynchronously invoke an AWS Lambda function using the `mysql.lambda_async` procedure. For more information, see Invoking a Lambda function with an Aurora MySQL native function.
- Load data from text or XML files stored in an Amazon S3 bucket into your DB cluster by using the `LOAD DATA FROM S3` or `LOAD XML FROM S3` statement. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.
- Save data from your DB cluster into text files stored in an Amazon S3 bucket by using the `SELECT INTO OUTFILE S3` statement. For more information, see Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket.
- Export log data to Amazon CloudWatch Logs MySQL. For more information, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.
- Automatically add or remove Aurora Replicas with Application Auto Scaling. For more information, see Using Amazon Aurora Auto Scaling with Aurora Replicas.

Setting up IAM roles to access AWS services

To permit your Aurora DB cluster to access another AWS service, do the following:

1. Create an IAM policy that grants permission to the AWS service. For more information, see:
   - Creating an IAM policy to access Amazon S3 resources
   - Creating an IAM policy to access AWS Lambda resources
   - Creating an IAM policy to access CloudWatch Logs resources
   - Creating an IAM policy to access AWS KMS resources
2. Create an IAM role and attach the policy that you created. For more information, see Creating an IAM role to allow Amazon Aurora to access AWS services.
3. Associate that IAM role with your Aurora DB cluster. For more information, see Associating an IAM role with an Amazon Aurora MySQL DB cluster.

Creating an IAM policy to access Amazon S3 resources

Aurora can access Amazon S3 resources to either load data to or save data from an Aurora DB cluster. However, you must first create an IAM policy that provides the bucket and object permissions that allow Aurora to access Amazon S3.

The following table lists the Aurora features that can access an Amazon S3 bucket on your behalf, and the minimum required bucket and object permissions required by each feature.

Feature	Bucket permissions	Object permissions
`LOAD DATA FROM S3`	`ListBucket`	`GetObject` `GetObjectVersion`
`LOAD XML FROM S3`	`ListBucket`	`GetObject` `GetObjectVersion`

`SELECT INTO OUTFILE S3`	`ListBucket`	`AbortMultipartUpload`
		`DeleteObject`
		`GetObject`
		`ListMultipartUploadParts`
		`PutObject`

The following policy adds the permissions that might be required by Aurora to access an Amazon S3 bucket on your behalf.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAuroraToExampleBucket",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:AbortMultipartUpload",
                "s3:ListBucket",
                "s3:DeleteObject",
                "s3:GetObjectVersion",
                "s3:ListMultipartUploadParts"
            ],
            "Resource": [
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*",
                "arn:aws:s3:::DOC-EXAMPLE-BUCKET"
            ]
        }
    ]
}
```

Note

Make sure to include both entries for the `Resource` value. Aurora needs the permissions on both the bucket itself and all the objects inside the bucket.
Based on your use case, you might not need to add all of the permissions in the sample policy. Also, other permissions might be required. For example, if your Amazon S3 bucket is encrypted, you need to add `kms:Decrypt` permissions.

You can use the following steps to create an IAM policy that provides the minimum required permissions for Aurora to access an Amazon S3 bucket on your behalf. To allow Aurora to access all of your Amazon S3 buckets, you can skip these steps and use either the `AmazonS3ReadOnlyAccess` or `AmazonS3FullAccess` predefined IAM policy instead of creating your own.

To create an IAM policy to grant access to your Amazon S3 resources

1. Open the IAM Management Console.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Visual editor** tab, choose **Choose a service**, and then choose **S3**.
5. For **Actions**, choose **Expand all**, and then choose the bucket permissions and object permissions needed for the IAM policy.

Object permissions are permissions for object operations in Amazon S3, and need to be granted for objects in a bucket, not the bucket itself. For more information about permissions for object operations in Amazon S3, see Permissions for object operations.

6. Choose **Resources**, and choose **Add ARN** for **bucket**.
7. In the **Add ARN(s)** dialog box, provide the details about your resource, and choose **Add**.

   Specify the Amazon S3 bucket to allow access to. For instance, if you want to allow Aurora to access the Amazon S3 bucket named `DOC-EXAMPLE-BUCKET`, then set the Amazon Resource Name (ARN) value to `arn:aws:s3:::DOC-EXAMPLE-BUCKET`.
8. If the **object** resource is listed, choose **Add ARN** for **object**.
9. In the **Add ARN(s)** dialog box, provide the details about your resource.

   For the Amazon S3 bucket, specify the Amazon S3 bucket to allow access to. For the object, you can choose **Any** to grant permissions to any object in the bucket.

   > **Note**
   >
   > You can set **Amazon Resource Name (ARN)** to a more specific ARN value in order to allow Aurora to access only specific files or folders in an Amazon S3 bucket. For more information about how to define an access policy for Amazon S3, see Managing access permissions to your Amazon S3 resources.
10. (Optional) Choose **Add ARN** for **bucket** to add another Amazon S3 bucket to the policy, and repeat the previous steps for the bucket.

    > **Note**
    >
    > You can repeat this to add corresponding bucket permission statements to your policy for each Amazon S3 bucket that you want Aurora to access. Optionally, you can also grant access to all buckets and objects in Amazon S3.
11. Choose **Review policy**.
12. For **Name**, enter a name for your IAM policy, for example `AllowAuroraToExampleBucket`. You use this name when you create an IAM role to associate with your Aurora DB cluster. You can also add an optional **Description** value.
13. Choose **Create policy**.

14. Complete the steps in Creating an IAM role to allow Amazon Aurora to access AWS services.

Creating an IAM policy to access AWS Lambda resources

You can create an IAM policy that provides the minimum required permissions for Aurora to invoke an AWS Lambda function on your behalf.

The following policy adds the permissions required by Aurora to invoke an AWS Lambda function on your behalf.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAuroraToExampleFunction",
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource":
 "arn:aws:lambda:<region>:<123456789012>:function:<example_function>"
        }
    ]
}
```

You can use the following steps to create an IAM policy that provides the minimum required permissions for Aurora to invoke an AWS Lambda function on your behalf. To allow Aurora to invoke all of your AWS Lambda functions, you can skip these steps and use the predefined `AWSLambdaRole` policy instead of creating your own.

To create an IAM policy to grant invoke to your AWS Lambda functions

1. Open the IAM console.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Visual editor** tab, choose **Choose a service**, and then choose **Lambda**.
5. For **Actions**, choose **Expand all**, and then choose the AWS Lambda permissions needed for the IAM policy.

   Ensure that `InvokeFunction` is selected. It is the minimum required permission to enable Amazon Aurora to invoke an AWS Lambda function.

6. Choose **Resources** and choose **Add ARN** for **function**.
7. In the **Add ARN(s)** dialog box, provide the details about your resource.

   Specify the Lambda function to allow access to. For instance, if you want to allow Aurora to access a Lambda function named `example_function`, then set the ARN value to `arn:aws:lambda:::function:example_function`.

   For more information on how to define an access policy for AWS Lambda, see Authentication and access control for AWS Lambda.
8. Optionally, choose **Add additional permissions** to add another AWS Lambda function to the policy, and repeat the previous steps for the function.

   > **Note**
   >
   > You can repeat this to add corresponding function permission statements to your policy for each AWS Lambda function that you want Aurora to access.
9. Choose **Review policy**.
10. Set **Name** to a name for your IAM policy, for example `AllowAuroraToExampleFunction`. You use this name when you create an IAM role to associate with your Aurora DB cluster. You can also add an optional **Description** value.
11. Choose **Create policy**.
12. Complete the steps in Creating an IAM role to allow Amazon Aurora to access AWS services.

Creating an IAM policy to access CloudWatch Logs resources

Aurora can access CloudWatch Logs to export audit log data from an Aurora DB cluster. However, you must first create an IAM policy that provides the log group and log stream permissions that allow Aurora to access CloudWatch Logs.

The following policy adds the permissions required by Aurora to access Amazon CloudWatch Logs on your behalf, and the minimum required permissions to create log groups and export data.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "EnableCreationAndManagementOfRDSCloudwatchLogEvents",
            "Effect": "Allow",
            "Action": [
                "logs:GetLogEvents",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/rds/*:log-stream:*"
        },
        {
            "Sid": "EnableCreationAndManagementOfRDSCloudwatchLogGroupsAndStreams",
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:DescribeLogStreams",
                "logs:PutRetentionPolicy",
                "logs:CreateLogGroup"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/rds/*"
        }
    ]
}
```

You can modify the ARNs in the policy to restrict access to a specific AWS Region and account.

You can use the following steps to create an IAM policy that provides the minimum required permissions for Aurora to access CloudWatch Logs on your behalf. To allow Aurora full access to CloudWatch Logs, you can skip these steps and use the `CloudWatchLogsFullAccess` predefined IAM policy instead of creating your own. For more information, see Using identity-based policies (IAM policies) for CloudWatch Logs in the *Amazon CloudWatch User Guide.*

To create an IAM policy to grant access to your CloudWatch Logs resources

1. Open the IAM console.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Visual editor** tab, choose **Choose a service**, and then choose **CloudWatch Logs**.
5. For **Actions**, choose **Expand all** (on the right), and then choose the Amazon CloudWatch Logs permissions needed for the IAM policy.

   Ensure that the following permissions are selected:

   - `CreateLogGroup`

- `CreateLogStream`
- `DescribeLogStreams`
- `GetLogEvents`
- `PutLogEvents`
- `PutRetentionPolicy`

6. Choose **Resources** and choose **Add ARN** for **log-group**.
7. In the **Add ARN(s)** dialog box, enter the following values:

   - **Region** – An AWS Region or `*`
   - **Account** – An account number or `*`
   - **Log Group Name** – `/aws/rds/*`
8. In the **Add ARN(s)** dialog box, choose **Add**.
9. Choose **Add ARN** for **log-stream**.
10. In the **Add ARN(s)** dialog box, enter the following values:

    - **Region** – An AWS Region or `*`
    - **Account** – An account number or `*`
    - **Log Group Name** – `/aws/rds/*`
    - **Log Stream Name** – `*`
11. In the **Add ARN(s)** dialog box, choose **Add**.
12. Choose **Review policy**.
13. Set **Name** to a name for your IAM policy, for example `AmazonRDSCloudWatchLogs`. You use this name when you create an IAM role to associate with your Aurora DB cluster. You can also add an optional **Description** value.
14. Choose **Create policy**.
15. Complete the steps in Creating an IAM role to allow Amazon Aurora to access AWS services.

Creating an IAM policy to access AWS KMS resources

Aurora can access the AWS KMS keys used for encrypting their database backups. However, you must first create an IAM policy that provides the permissions that allow Aurora to access KMS keys.

The following policy adds the permissions required by Aurora to access KMS keys on your behalf.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:<region>:<123456789012>:key/<key-ID>"
        }
    ]
}
```

You can use the following steps to create an IAM policy that provides the minimum required permissions for Aurora to access KMS keys on your behalf.

To create an IAM policy to grant access to your KMS keys

1. Open the IAM console.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Visual editor** tab, choose **Choose a service**, and then choose **KMS**.
5. For **Actions**, choose **Write**, and then choose **Decrypt**.
6. Choose **Resources**, and choose **Add ARN**.
7. In the **Add ARN(s)** dialog box, enter the following values:
   - **Region** – Type the AWS Region, such as `us-west-2`.
   - **Account** – Type the user account number.
   - **Log Stream Name** – Type the KMS key identifier.
8. In the **Add ARN(s)** dialog box, choose **Add**.
9. Choose **Review policy**.
10. Set **Name** to a name for your IAM policy, for example `AmazonRDSKMSKey`. You use this name when you create an IAM role to associate with your Aurora DB cluster. You can also add an optional **Description** value.
11. Choose **Create policy**.
12. Complete the steps in Creating an IAM role to allow Amazon Aurora to access AWS services.

Creating an IAM role to allow Amazon Aurora to access AWS services

After creating an IAM policy to allow Aurora to access AWS resources, you must create an IAM role and attach the IAM policy to the new IAM role.

To create an IAM role to permit your Amazon RDS cluster to communicate with other AWS services on your behalf, take the following steps.

To create an IAM role to allow Amazon RDS to access AWS services

1. Open the IAM console.
2. In the navigation pane, choose **Roles**.
3. Choose **Create role**.
4. Under **AWS service**, choose **RDS**.
5. Under **Select your use case**, choose **RDS – Add Role to Database**.
6. Choose **Next**.
7. On the **Permissions policies** page, enter the name of your policy in the **Search** field.
8. When it appears in the list, select the policy that you defined earlier using the instructions in one of the following sections:
   - Creating an IAM policy to access Amazon S3 resources
   - Creating an IAM policy to access AWS Lambda resources
   - Creating an IAM policy to access CloudWatch Logs resources
   - Creating an IAM policy to access AWS KMS resources
9. Choose **Next**.
10. In **Role name**, enter a name for your IAM role, for example `RDSLoadFromS3`. You can also add an optional **Description** value.
11. Choose **Create Role**.
12. Complete the steps in Associating an IAM role with an Amazon Aurora MySQL DB cluster.

Associating an IAM role with an Amazon Aurora MySQL DB cluster

To permit database users in an Amazon Aurora DB cluster to access other AWS services, you associate the IAM role that you created in Creating an IAM role to allow Amazon Aurora to access

AWS services with that DB cluster. You can also have AWS create a new IAM role by associating the service directly.

> **Note**
>
> You can't associate an IAM role with an Aurora Serverless v1 DB cluster. For more information, see Using Amazon Aurora Serverless v1.
> You can associate an IAM role with an Aurora Serverless v2 DB cluster.

To associate an IAM role with a DB cluster you do two things:

1. Add the role to the list of associated roles for a DB cluster by using the RDS console, the add-role-to-db-cluster AWS CLI command, or the AddRoleToDBCluster RDS API operation.

   You can add a maximum of five IAM roles for each Aurora DB cluster.
2. Set the cluster-level parameter for the related AWS service to the ARN for the associated IAM role.

   The following table describes the cluster-level parameter names for the IAM roles used to access other AWS services.

Cluster-level parameter	Description	
`aws_default_lambda_role`	Used when invoking a Lambda function from your DB cluster.	
`aws_default_logs_role`	This parameter is no longer required for exporting log data from your DB cluster to Amazon CloudWatch Logs. Aurora MySQL now uses a service-linked role for the required permissions. For more information about service-linked roles, see Using service-linked roles for Amazon Aurora.	
`aws_default_s3_role`	Used when invoking the `LOAD DATA FROM S3`, `LOAD XML FROM S3`, or `SELECT INTO`	

Cluster-level parameter	Description
	OUTFILE S3 statement from your DB cluster. In Aurora MySQL version 2, the IAM role specified in this parameter is used if an IAM role isn't specified for aurora_load_from_s3_role or aurora_select_into_s3_role for the appropriate statement. In Aurora MySQL version 3, the IAM role specified for this parameter is always used.
aurora_load_from_s3_role	Used when invoking the LOAD DATA FROM S3 or LOAD XML FROM S3 statement from your DB cluster. If an IAM role is not specified for this parameter, the IAM role specified in aws_default_s3_role is used. In Aurora MySQL version 3, this parameter isn't available.
aurora_select_into_s3_role	Used when invoking the SELECT INTO OUTFILE S3 statement from your DB cluster. If an IAM role is not specified for this parameter, the IAM role specified in aws_default_s3_role is used. In Aurora MySQL version 3, this parameter isn't available.

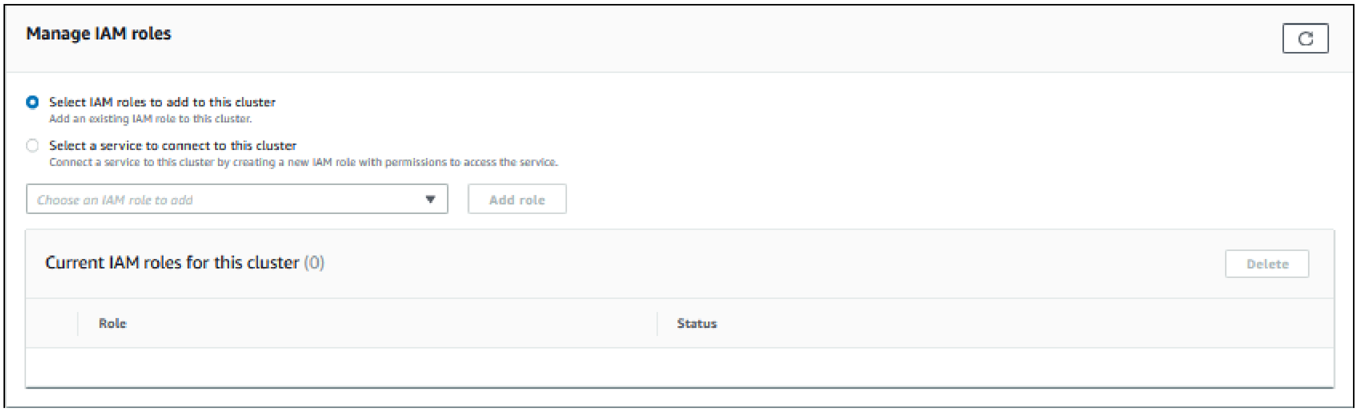
To associate an IAM role to permit your Amazon RDS cluster to communicate with other AWS services on your behalf, take the following steps.

Console

To associate an IAM role with an Aurora DB cluster using the console

1. Open the RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose the name of the Aurora DB cluster that you want to associate an IAM role with to show its details.
4. On the **Connectivity & security** tab, in the **Manage IAM roles** section, do one of the following:

   - **Select IAM roles to add to this cluster** (default)
   - **Select a service to connect to this cluster**



Manage IAM roles

● Select IAM roles to add to this cluster
Add an existing IAM role to this cluster.

○ Select a service to connect to this cluster
Connect a service to this cluster by creating a new IAM role with permissions to access the service.

Current IAM roles for this cluster (0)

Role	Status

5. To use an existing IAM role, choose it from the menu, then choose **Add role**.

   If adding the role is successful, its status shows as `Pending`, then `Available`.
6. To connect a service directly:

   a. Choose **Select a service to connect to this cluster**.

   b. Choose the service from the menu, then choose **Connect service**.

   c. For **Connect cluster to *`Service Name`***, enter the Amazon Resource Name (ARN) to use to connect to the service, then choose **Connect service**.

   AWS creates a new IAM role for connecting to the service. Its status shows as `Pending`, then `Available`.
7. (Optional) To stop associating an IAM role with a DB cluster and remove the related permission, choose the role and then choose **Delete**.

To set the cluster-level parameter for the associated IAM role

1. In the RDS console, choose **Parameter groups** in the navigation pane.
2. If you are already using a custom DB parameter group, you can select that group to use instead of creating a new DB cluster parameter group. If you are using the default DB cluster parameter group, create a new DB cluster parameter group, as described in the following steps:

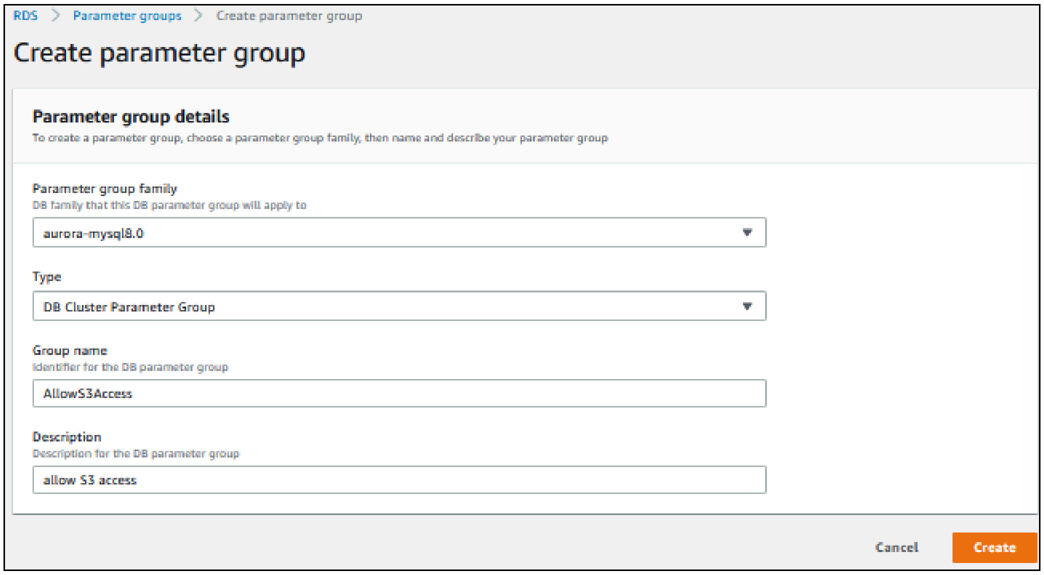
   a. Choose **Create parameter group**.

   b. For **Parameter group family**, choose `aurora-mysql8.0` for an Aurora MySQL 8.0-compatible DB cluster, or `aurora-mysql5.7` for an Aurora MySQL 5.7-compatible DB cluster.

   c. For **Type**, choose **DB Cluster Parameter Group**.

   d. For **Group name**, type the name of your new DB cluster parameter group.

   e. For **Description**, type a description for your new DB cluster parameter group.



RDS > Parameter groups > Create parameter group

Create parameter group

Parameter group details

To create a parameter group, choose a parameter group family, then name and describe your parameter group

Parameter group family
DB family that this DB parameter group will apply to
aurora-mysql8.0

Type
DB Cluster Parameter Group

Group name
Identifier for the DB parameter group
AllowS3Access

Description
Description for the DB parameter group
allow S3 access

   f. Choose **Create**.

3. On the **Parameter groups** page, select your DB cluster parameter group and choose **Edit** for **Parameter group actions**.
4. Set the appropriate cluster-level parameters to the related IAM role ARN values.

   For example, you can set just the `aws_default_s3_role` parameter to `arn:aws:iam::123456789012:role/AllowS3Access`.
5. Choose **Save changes**.

6. To change the DB cluster parameter group for your DB cluster, complete the following steps:

   a. Choose **Databases**, and then choose your Aurora DB cluster.

   b. Choose **Modify**.

   c. Scroll to **Database options** and set **DB cluster parameter group** to the DB cluster parameter group.

   d. Choose **Continue**.

   e. Verify your changes and then choose **Apply immediately**.

   f. Choose **Modify cluster**.

   g. Choose **Databases**, and then choose the primary instance for your DB cluster.

   h. For **Actions**, choose **Reboot**.

      When the instance has rebooted, your IAM role is associated with your DB cluster.

      For more information about cluster parameter groups, see Aurora MySQL configuration parameters.

CLI

To associate an IAM role with a DB cluster by using the AWS CLI

1. Call the `add-role-to-db-cluster` command from the AWS CLI to add the ARNs for your IAM roles to the DB cluster, as shown following.

```
PROMPT> aws rds add-role-to-db-cluster --db-cluster-identifier my-cluster --role-
arn arn:aws:iam::123456789012:role/AllowAuroraS3Role
PROMPT> aws rds add-role-to-db-cluster --db-cluster-identifier my-cluster --role-
arn arn:aws:iam::123456789012:role/AllowAuroraLambdaRole
```

2. If you are using the default DB cluster parameter group, create a new DB cluster parameter group. If you are already using a custom DB parameter group, you can use that group instead of creating a new DB cluster parameter group.

   To create a new DB cluster parameter group, call the `create-db-cluster-parameter-group` command from the AWS CLI, as shown following.

```
PROMPT> aws rds create-db-cluster-parameter-group  --db-cluster-parameter-group-
name AllowAWSAccess \
    --db-parameter-group-family aurora5.7 --description "Allow access to Amazon S3
 and AWS Lambda"
```

For an Aurora MySQL 5.7-compatible DB cluster, specify `aurora-mysql5.7` for `--db-parameter-group-family`. For an Aurora MySQL 8.0-compatible DB cluster, specify `aurora-mysql8.0` for `--db-parameter-group-family`.

3. Set the appropriate cluster-level parameter or parameters and the related IAM role ARN values in your DB cluster parameter group, as shown following.

```
PROMPT> aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name
 AllowAWSAccess \
    --parameters
 "ParameterName=aws_default_s3_role,ParameterValue=arn:aws:iam::123456789012:role/
AllowAuroraS3Role,method=pending-reboot" \
    --parameters
 "ParameterName=aws_default_lambda_role,ParameterValue=arn:aws:iam::123456789012:role/
AllowAuroraLambdaRole,method=pending-reboot"
```

4. Modify the DB cluster to use the new DB cluster parameter group and then reboot the cluster, as shown following.

```
PROMPT> aws rds modify-db-cluster --db-cluster-identifier my-cluster --db-cluster-
parameter-group-name AllowAWSAccess
PROMPT> aws rds reboot-db-instance --db-instance-identifier my-cluster-primary
```

When the instance has rebooted, your IAM roles are associated with your DB cluster.

For more information about cluster parameter groups, see Aurora MySQL configuration parameters.

Enabling network communication from Amazon Aurora MySQL to other AWS services

To use certain other AWS services with Amazon Aurora, the network configuration of your Aurora DB cluster must allow outbound connections to endpoints for those services. The following operations require this network configuration.

- Invoking AWS Lambda functions. To learn about this feature, see Invoking a Lambda function with an Aurora MySQL native function.

- Accessing files from Amazon S3. To learn about this feature, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket and Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket.
- Accessing AWS KMS endpoints. AWS KMS access is required to use database activity streams with Aurora MySQL. To learn about this feature, see Monitoring Amazon Aurora with Database Activity Streams.
- Accessing SageMaker endpoints. SageMaker access is required to use SageMaker machine learning with Aurora MySQL. To learn about this feature, see Using Amazon Aurora machine learning with Aurora MySQL.

Aurora returns the following error messages if it can't connect to a service endpoint.

```
ERROR 1871 (HY000): S3 API returned error: Network Connection
```

```
ERROR 1873 (HY000): Lambda API returned error: Network Connection. Unable to connect to
 endpoint
```

```
ERROR 1815 (HY000): Internal error: Unable to initialize S3Stream
```

For database activity streams using Aurora MySQL, the activity stream stops functioning if the DB cluster can't access the AWS KMS endpoint. Aurora notifies you about this issue using RDS Events.

If you encounter these messages while using the corresponding AWS services, check if your Aurora DB cluster is public or private. If your Aurora DB cluster is private, you must configure it to enable connections.

For an Aurora DB cluster to be public, it must be marked as publicly accessible. If you look at the details for the DB cluster in the AWS Management Console, **Publicly Accessible** is **Yes** if this is the case. The DB cluster must also be in an Amazon VPC public subnet. For more information about publicly accessible DB instances, see Working with a DB cluster in a VPC. For more information about public Amazon VPC subnets, see Your VPC and subnets.

If your Aurora DB cluster isn't publicly accessible and in a VPC public subnet, it is private. You might have a DB cluster that is private and want to use one of the features that requires this network configuration. If so, configure the cluster so that it can connect to Internet addresses through Network Address Translation (NAT). As an alternative for Amazon S3, Amazon SageMaker, and AWS

Lambda, you can instead configure the VPC to have a VPC endpoint for the other service associated with the DB cluster's route table, see Working with a DB cluster in a VPC. For more information about configuring NAT in your VPC, see NAT gateways. For more information about configuring VPC endpoints, see VPC endpoints. You can also create an S3 gateway endpoint to access your S3 bucket. For more information, see Gateway endpoints for Amazon S3.

You might also have to open the ephemeral ports for your network access control lists (ACLs) in the outbound rules for your VPC security group. For more information on ephemeral ports for network ACLs, see Ephemeral ports in the *Amazon Virtual Private Cloud User Guide*.

Related topics

- Integrating Aurora with other AWS services
- Managing an Amazon Aurora DB cluster

Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket

You can use the `LOAD DATA FROM S3` or `LOAD XML FROM S3` statement to load data from files stored in an Amazon S3 bucket. In Aurora MySQL, the files are first stored on the local disk, and then imported to the database. After the imports to the database are done, the local files are deleted.

> **Note**
>
> Loading data into a table from text files isn't supported for Aurora Serverless v1. It is supported for Aurora Serverless v2.

Contents

Giving Aurora access to Amazon S3

Before you can load data from an Amazon S3 bucket, you must first give your Aurora MySQL DB cluster permission to access Amazon S3.

To give Aurora MySQL access to Amazon S3

1. Create an AWS Identity and Access Management (IAM) policy that provides the bucket and object permissions that allow your Aurora MySQL DB cluster to access Amazon S3. For instructions, see Creating an IAM policy to access Amazon S3 resources.

   > **Note**
   >
   > In Aurora MySQL version 3.05 and higher, you can load objects that are encrypted using customer-managed AWS KMS keys. To do so, include the `kms:Decrypt` permission in your IAM policy. For more information, see Creating an IAM policy to access AWS KMS resources.
   > You don't need this permission to load objects that are encrypted using AWS managed keys or Amazon S3 managed keys (SSE-S3).

2. Create an IAM role, and attach the IAM policy you created in Creating an IAM policy to access Amazon S3 resources to the new IAM role. For instructions, see Creating an IAM role to allow Amazon Aurora to access AWS services.

3. Make sure the DB cluster is using a custom DB cluster parameter group.

   For more information about creating a custom DB cluster parameter group, see Creating a DB cluster parameter group.

4. For Aurora MySQL version 2, set either the `aurora_load_from_s3_role` or `aws_default_s3_role` DB cluster parameter to the Amazon Resource Name (ARN) of the

new IAM role. If an IAM role isn't specified for `aurora_load_from_s3_role`, Aurora uses the IAM role specified in `aws_default_s3_role`.

For Aurora MySQL version 3, use `aws_default_s3_role`.

If the cluster is part of an Aurora global database, set this parameter for each Aurora cluster in the global database. Although only the primary cluster in an Aurora global database can load data, another cluster might be promoted by the failover mechanism and become the primary cluster.

For more information about DB cluster parameters, see Amazon Aurora DB cluster and DB instance parameters.

5. To permit database users in an Aurora MySQL DB cluster to access Amazon S3, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with the DB cluster. For an Aurora global database, associate the role with each Aurora cluster in the global database. For information about associating an IAM role with a DB cluster, see Associating an IAM role with an Amazon Aurora MySQL DB cluster.
6. Configure your Aurora MySQL DB cluster to allow outbound connections to Amazon S3. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

   If your DB cluster isn't publicly accessible and in a VPC public subnet, it is private. You can create an S3 gateway endpoint to access your S3 bucket. For more information, see Gateway endpoints for Amazon S3.

   For an Aurora global database, enable outbound connections for each Aurora cluster in the global database.

Granting privileges to load data in Amazon Aurora MySQL

The database user that issues the `LOAD DATA FROM S3` or `LOAD XML FROM S3` statement must have a specific role or privilege to issue either statement. In Aurora MySQL version 3, you grant the `AWS_LOAD_S3_ACCESS` role. In Aurora MySQL version 2, you grant the `LOAD FROM S3` privilege. The administrative user for a DB cluster is granted the appropriate role or privilege by default. You can grant the privilege to another user by using one of the following statements.

Use the following statement for Aurora MySQL version 3:

```
GRANT AWS_LOAD_S3_ACCESS TO 'user'@'domain-or-ip-address'
```

Tip

When you use the role technique in Aurora MySQL version 3, you can also activate the role by using the `SET ROLE role_name` or `SET ROLE ALL` statement. If you aren't familiar with the MySQL 8.0 role system, you can learn more in Role-based privilege model. For more details, see Using roles in the *MySQL Reference Manual*.

This only applies to the current active session. When you reconnect, you must run the `SET ROLE` statement again to grant privileges. For more information, see SET ROLE statement in the *MySQL Reference Manual*.

You can use the `activate_all_roles_on_login` DB cluster parameter to automatically activate all roles when a user connects to a DB instance. When this parameter is set, you generally don't have to call the `SET ROLE` statement explicitly to activate a role. For more information, see activate_all_roles_on_login in the *MySQL Reference Manual*.

However, you must call `SET ROLE ALL` explicitly at the beginning of a stored procedure to activate the role, when the stored procedure is called by a different user.

Use the following statement for Aurora MySQL version 2:

```
GRANT LOAD FROM S3 ON *.* TO 'user'@'domain-or-ip-address'
```

The `AWS_LOAD_S3_ACCESS` role and `LOAD FROM S3` privilege are specific to Amazon Aurora and are not available for external MySQL databases or RDS for MySQL DB instances. If you have set up replication between an Aurora DB cluster as the replication master and a MySQL database as the replication client, then the `GRANT` statement for the role or privilege causes replication to stop with an error. You can safely skip the error to resume replication. To skip the error on an RDS for MySQL instance, use the mysql_rds_skip_repl_error procedure. To skip the error on an external MySQL database, use the slave_skip_errors system variable (Aurora MySQL version 2) or replica_skip_errors system variable (Aurora MySQL version 3).

Note

The database user must have `INSERT` privileges for the database into which it's loading data.

Specifying the path (URI) to an Amazon S3 bucket

The syntax for specifying the path (URI) to files stored on an Amazon S3 bucket is as follows.

```
s3-region://DOC-EXAMPLE-BUCKET/file-name-or-prefix
```

The path includes the following values:

- `region` (optional) – The AWS Region that contains the Amazon S3 bucket to load from. This value is optional. If you don't specify a `region` value, then Aurora loads your file from Amazon S3 in the same region as your DB cluster.
- `bucket-name` – The name of the Amazon S3 bucket that contains the data to load. Object prefixes that identify a virtual folder path are supported.
- `file-name-or-prefix` – The name of the Amazon S3 text file or XML file, or a prefix that identifies one or more text or XML files to load. You can also specify a manifest file that identifies one or more text files to load. For more information about using a manifest file to load text files from Amazon S3, see Using a manifest to specify data files to load.

To copy the URI for files in an S3 bucket

1. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.
2. In the navigation pane, choose **Buckets**, and then choose the bucket whose URI you want to copy.
3. Select the prefix or file that you want to load from S3.
4. Choose **Copy S3 URI**.

LOAD DATA FROM S3

You can use the `LOAD DATA FROM S3` statement to load data from any text file format that is supported by the MySQL LOAD DATA INFILE statement, such as text data that is comma-delimited. Compressed files are not supported.

Note

Make sure that your Aurora MySQL DB cluster allows outbound connections to S3. For more information, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

Syntax

```
LOAD DATA [FROM] S3 [FILE | PREFIX | MANIFEST] 'S3-URI'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name,...)]
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_user_var,...)]
    [SET col_name = expr,...]
```

Note

In Aurora MySQL version 3.05 and higher, the keyword `FROM` is optional.

Parameters

The `LOAD DATA FROM S3` statement uses the following required and optional parameters. You can find more details about some of these parameters in LOAD DATA Statement in the MySQL documentation.

FILE | PREFIX | MANIFEST

Identifies whether to load the data from a single file, from all files that match a given prefix, or from all files in a specified manifest. `FILE` is the default.

S3-URI

Specifies the URI for a text or manifest file to load, or an Amazon S3 prefix to use. Specify the URI using the syntax described in Specifying the path (URI) to an Amazon S3 bucket.

REPLACE | IGNORE

Determines what action to take if an input row has the same unique key values as an existing row in the database table.

- Specify `REPLACE` if you want the input row to replace the existing row in the table.
- Specify `IGNORE` if you want to discard the input row.

INTO TABLE

Identifies the name of the database table to load the input rows into.

PARTITION

Requires that all input rows be inserted into the partitions identified by the specified list of comma-separated partition names. If an input row cannot be inserted into one of the specified partitions, then the statement fails and an error is returned.

CHARACTER SET

Identifies the character set of the data in the input file.

FIELDS | COLUMNS

Identifies how the fields or columns in the input file are delimited. Fields are tab-delimited by default.

LINES

Identifies how the lines in the input file are delimited. Lines are delimited by a newline character (`'\n'`) by default.

IGNORE *number* LINES | ROWS

Specifies to ignore a certain number of lines or rows at the start of the input file. For example, you can use `IGNORE 1 LINES` to skip over an initial header line containing column names, or

IGNORE 2 ROWS to skip over the first two rows of data in the input file. If you also use PREFIX, IGNORE skips a certain number of lines or rows at the start of the first input file.

col_name_or_user_var, ...

Specifies a comma-separated list of one or more column names or user variables that identify which columns to load by name. The name of a user variable used for this purpose must match the name of an element from the text file, prefixed with @. You can employ user variables to store the corresponding field values for subsequent reuse.

For example, the following statement loads the first column from the input file into the first column of table1, and sets the value of the table_column2 column in table1 to the input value of the second column divided by 100.

```
LOAD DATA FROM S3 's3://DOC-EXAMPLE-BUCKET/data.txt'
    INTO TABLE table1
    (column1, @var1)
    SET table_column2 = @var1/100;
```

SET

Specifies a comma-separated list of assignment operations that set the values of columns in the table to values not included in the input file.

For example, the following statement sets the first two columns of table1 to the values in the first two columns from the input file, and then sets the value of the column3 in table1 to the current time stamp.

```
LOAD DATA FROM S3  's3://DOC-EXAMPLE-BUCKET/data.txt'
    INTO TABLE table1
    (column1, column2)
    SET column3 = CURRENT_TIMESTAMP;
```

You can use subqueries in the right side of SET assignments. For a subquery that returns a value to be assigned to a column, you can use only a scalar subquery. Also, you cannot use a subquery to select from the table that is being loaded.

You can't use the LOCAL keyword of the LOAD DATA FROM S3 statement if you're loading data from an Amazon S3 bucket.

Using a manifest to specify data files to load

You can use the `LOAD DATA FROM S3` statement with the `MANIFEST` keyword to specify a manifest file in JSON format that lists the text files to be loaded into a table in your DB cluster.

The following JSON schema describes the format and content of a manifest file.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "additionalProperties": false,
    "definitions": {},
    "id": "Aurora_LoadFromS3_Manifest",
    "properties": {
        "entries": {
            "additionalItems": false,
            "id": "/properties/entries",
            "items": {
                "additionalProperties": false,
                "id": "/properties/entries/items",
                "properties": {
                    "mandatory": {
                        "default": "false",
                        "id": "/properties/entries/items/properties/mandatory",
                        "type": "boolean"
                    },
                    "url": {
                        "id": "/properties/entries/items/properties/url",
                        "maxLength": 1024,
                        "minLength": 1,
                        "type": "string"
                    }
                },
                "required": [
                    "url"
                ],
                "type": "object"
            },
            "type": "array",
            "uniqueItems": true
        }
    },
    "required": [
        "entries"
    ],
    "type": "object"
}
```

Each `url` in the manifest must specify a URL with the bucket name and full object path for the file, not just a prefix. You can use a manifest to load files from different buckets, different regions, or files that do not share the same prefix. If a region is not specified in the URL, the region of the target Aurora DB cluster is used. The following example shows a manifest file that loads four files from different buckets.

```
{
  "entries": [
    {
      "url":"s3://aurora-bucket/2013-10-04-customerdata",
      "mandatory":true
    },
    {
      "url":"s3-us-west-2://aurora-bucket-usw2/2013-10-05-customerdata",
      "mandatory":true
    },
    {
      "url":"s3://aurora-bucket/2013-10-04-customerdata",
      "mandatory":false
    },
    {
      "url":"s3://aurora-bucket/2013-10-05-customerdata"
    }
  ]
}
```

The optional `mandatory` flag specifies whether `LOAD DATA FROM S3` should return an error if the file is not found. The `mandatory` flag defaults to `false`. Regardless of how `mandatory` is set, `LOAD DATA FROM S3` terminates if no files are found.

Manifest files can have any extension. The following example runs the `LOAD DATA FROM S3` statement with the manifest in the previous example, which is named **`customer.manifest`**.

```
LOAD DATA FROM S3 MANIFEST 's3-us-west-2://aurora-bucket/customer.manifest'
    INTO TABLE CUSTOMER
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    (ID, FIRSTNAME, LASTNAME, EMAIL);
```

After the statement completes, an entry for each successfully loaded file is written to the `aurora_s3_load_history` table.

Verifying loaded files using the aurora_s3_load_history table

Every successful `LOAD DATA FROM S3` statement updates the `aurora_s3_load_history` table in the `mysql` schema with an entry for each file that was loaded.

After you run the `LOAD DATA FROM S3` statement, you can verify which files were loaded by querying the `aurora_s3_load_history` table. To see the files that were loaded from one iteration of the statement, use the `WHERE` clause to filter the records on the Amazon S3 URI for the manifest file used in the statement. If you have used the same manifest file before, filter the results using the `timestamp` field.

```
select * from mysql.aurora_s3_load_history where load_prefix = 'S3_URI';
```

The following table describes the fields in the `aurora_s3_load_history` table.

Field	Description
`load_prefix`	The URI that was specified in the load statement. This URI can map to any of the following: • A single data file for a `LOAD DATA FROM S3 FILE` statement • An Amazon S3 prefix that maps to multiple data files for a `LOAD DATA FROM S3 PREFIX` statement • A single manifest file that contains the names of files to be loaded for a `LOAD DATA FROM S3 MANIFEST` statement
`file_name`	The name of a file that was loaded into Aurora from Amazon S3 using the URI identified in the `load_prefix` field.
`version_number`	The version number of the file identified by the `file_name` field that was loaded, if the Amazon S3 bucket has a version number.
`bytes_loaded`	The size of the file loaded, in bytes.

Field	Description
`load_timestamp`	The timestamp when the `LOAD DATA FROM S3` statement completed.

Examples

The following statement loads data from an Amazon S3 bucket that is in the same region as the Aurora DB cluster. The statement reads the comma-delimited data in the file `customerdata.txt` that is in the *`DOC-EXAMPLE-BUCKET`* Amazon S3 bucket, and then loads the data into the table `store-schema.customer-table`.

```
LOAD DATA FROM S3 's3://DOC-EXAMPLE-BUCKET/customerdata.csv'
    INTO TABLE store-schema.customer-table
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    (ID, FIRSTNAME, LASTNAME, ADDRESS, EMAIL, PHONE);
```

The following statement loads data from an Amazon S3 bucket that is in a different region from the Aurora DB cluster. The statement reads the comma-delimited data from all files that match the `employee-data` object prefix in the *`DOC-EXAMPLE-BUCKET`* Amazon S3 bucket in the `us-west-2` region, and then loads the data into the `employees` table.

```
LOAD DATA FROM S3 PREFIX 's3-us-west-2://DOC-EXAMPLE-BUCKET/employee_data'
    INTO TABLE employees
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    (ID, FIRSTNAME, LASTNAME, EMAIL, SALARY);
```

The following statement loads data from the files specified in a JSON manifest file named q1_sales.json into the `sales` table.

```
LOAD DATA FROM S3 MANIFEST 's3-us-west-2://DOC-EXAMPLE-BUCKET1/q1_sales.json'
    INTO TABLE sales
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    (MONTH, STORE, GROSS, NET);
```

LOAD XML FROM S3

You can use the LOAD XML FROM S3 statement to load data from XML files stored on an Amazon S3 bucket in one of three different XML formats:

- Column names as attributes of a <row> element. The attribute value identifies the contents of the table field.

```
<row column1="value1" column2="value2" .../>
```

- Column names as child elements of a <row> element. The value of the child element identifies the contents of the table field.

```
<row>
  <column1>value1</column1>
  <column2>value2</column2>
</row>
```

- Column names in the name attribute of <field> elements in a <row> element. The value of the <field> element identifies the contents of the table field.

```
<row>
  <field name='column1'>value1</field>
  <field name='column2'>value2</field>
</row>
```

Syntax

```
LOAD XML FROM S3 'S3-URI'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name,...)]
    [CHARACTER SET charset_name]
    [ROWS IDENTIFIED BY '<element-name>']
    [IGNORE number {LINES | ROWS}]
    [(field_name_or_user_var,...)]
    [SET col_name = expr,...]
```

Parameters

The `LOAD XML FROM S3` statement uses the following required and optional parameters. You can find more details about some of these parameters in [LOAD XML Statement](#) in the MySQL documentation.

FILE | PREFIX

Identifies whether to load the data from a single file, or from all files that match a given prefix. `FILE` is the default.

REPLACE | IGNORE

Determines what action to take if an input row has the same unique key values as an existing row in the database table.

- Specify `REPLACE` if you want the input row to replace the existing row in the table.
- Specify `IGNORE` if you want to discard the input row. `IGNORE` is the default.

INTO TABLE

Identifies the name of the database table to load the input rows into.

PARTITION

Requires that all input rows be inserted into the partitions identified by the specified list of comma-separated partition names. If an input row cannot be inserted into one of the specified partitions, then the statement fails and an error is returned.

CHARACTER SET

Identifies the character set of the data in the input file.

ROWS IDENTIFIED BY

Identifies the element name that identifies a row in the input file. The default is `<row>`.

IGNORE *number* LINES | ROWS

Specifies to ignore a certain number of lines or rows at the start of the input file. For example, you can use `IGNORE 1 LINES` to skip over the first line in the text file, or `IGNORE 2 ROWS` to skip over the first two rows of data in the input XML.

field_name_or_user_var, ...

Specifies a comma-separated list of one or more XML element names or user variables that identify which elements to load by name. The name of a user variable used for this purpose must match the name of an element from the XML file, prefixed with @. You can employ user variables to store the corresponding field values for subsequent reuse.

For example, the following statement loads the first column from the input file into the first column of `table1`, and sets the value of the `table_column2` column in `table1` to the input value of the second column divided by 100.

```
LOAD XML FROM S3 's3://DOC-EXAMPLE-BUCKET/data.xml'
   INTO TABLE table1
   (column1, @var1)
   SET table_column2 = @var1/100;
```

SET

Specifies a comma-separated list of assignment operations that set the values of columns in the table to values not included in the input file.

For example, the following statement sets the first two columns of `table1` to the values in the first two columns from the input file, and then sets the value of the `column3` in `table1` to the current time stamp.

```
LOAD XML FROM S3 's3://DOC-EXAMPLE-BUCKET/data.xml'
   INTO TABLE table1
   (column1, column2)
   SET column3 = CURRENT_TIMESTAMP;
```

You can use subqueries in the right side of `SET` assignments. For a subquery that returns a value to be assigned to a column, you can use only a scalar subquery. Also, you can't use a subquery to select from the table that's being loaded.

Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket

You can use the `SELECT INTO OUTFILE S3` statement to query data from an Amazon Aurora MySQL DB cluster and save it into text files stored in an Amazon S3 bucket. In Aurora MySQL, the files are first stored on the local disk, and then exported to S3. After the exports are done, the local files are deleted.

You can encrypt the Amazon S3 bucket using an Amazon S3 managed key (SSE-S3) or AWS KMS key (SSE-KMS: AWS managed key or customer managed key).

The `LOAD DATA FROM S3` statement can use files created by the `SELECT INTO OUTFILE S3` statement to load data into an Aurora DB cluster. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.

> **Note**
>
> This feature isn't supported for Aurora Serverless v1 DB clusters. It is supported for Aurora Serverless v2 DB clusters.
> You can also save DB cluster data and DB cluster snapshot data to Amazon S3 using the AWS Management Console, AWS CLI, or Amazon RDS API. For more information, see Exporting DB cluster data to Amazon S3 and Exporting DB cluster snapshot data to Amazon S3.

Contents

Giving Aurora MySQL access to Amazon S3

Before you can save data into an Amazon S3 bucket, you must first give your Aurora MySQL DB cluster permission to access Amazon S3.

To give Aurora MySQL access to Amazon S3

1. Create an AWS Identity and Access Management (IAM) policy that provides the bucket and object permissions that allow your Aurora MySQL DB cluster to access Amazon S3. For instructions, see Creating an IAM policy to access Amazon S3 resources.

   > **Note**
   >
   > In Aurora MySQL version 3.05 and higher, you can encrypt objects using AWS KMS customer managed keys. To do so, include the `kms:GenerateDataKey` permission in your IAM policy. For more information, see Creating an IAM policy to access AWS KMS resources.
   > You don't need this permission to encrypt objects using AWS managed keys or Amazon S3 managed keys (SSE-S3).

2. Create an IAM role, and attach the IAM policy you created in Creating an IAM policy to access Amazon S3 resources to the new IAM role. For instructions, see Creating an IAM role to allow Amazon Aurora to access AWS services.

3. For Aurora MySQL version 2, set either the `aurora_select_into_s3_role` or `aws_default_s3_role` DB cluster parameter to the Amazon Resource Name (ARN) of the new IAM role. If an IAM role isn't specified for `aurora_select_into_s3_role`, Aurora uses the IAM role specified in `aws_default_s3_role`.

   For Aurora MySQL version 3, use `aws_default_s3_role`.

   If the cluster is part of an Aurora global database, set this parameter for each Aurora cluster in the global database.

   For more information about DB cluster parameters, see Amazon Aurora DB cluster and DB instance parameters.

4. To permit database users in an Aurora MySQL DB cluster to access Amazon S3, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with the DB cluster.

   For an Aurora global database, associate the role with each Aurora cluster in the global database.

For information about associating an IAM role with a DB cluster, see Associating an IAM role with an Amazon Aurora MySQL DB cluster.

5. Configure your Aurora MySQL DB cluster to allow outbound connections to Amazon S3. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

   For an Aurora global database, enable outbound connections for each Aurora cluster in the global database.

Granting privileges to save data in Aurora MySQL

The database user that issues the `SELECT INTO OUTFILE S3` statement must have a specific role or privilege. In Aurora MySQL version 3, you grant the `AWS_SELECT_S3_ACCESS` role. In Aurora MySQL version 2, you grant the `SELECT INTO S3` privilege. The administrative user for a DB cluster is granted the appropriate role or privilege by default. You can grant the privilege to another user by using one of the following statements.

Use the following statement for Aurora MySQL version 3:

```
GRANT AWS_SELECT_S3_ACCESS TO 'user'@'domain-or-ip-address'
```

Tip

When you use the role technique in Aurora MySQL version 3, you can also activate the role by using the `SET ROLE` *`role_name`* or `SET ROLE ALL` statement. If you aren't familiar with the MySQL 8.0 role system, you can learn more in Role-based privilege model. For more details, see Using roles in the *MySQL Reference Manual*.
This only applies to the current active session. When you reconnect, you must run the SET ROLE statement again to grant privileges. For more information, see SET ROLE statement in the *MySQL Reference Manual*.
You can use the `activate_all_roles_on_login` DB cluster parameter to automatically activate all roles when a user connects to a DB instance. When this parameter is set, you generally don't have to call the `SET ROLE` statement explicitly to activate a role. For more information, see activate_all_roles_on_login in the *MySQL Reference Manual*.
However, you must call `SET ROLE ALL` explicitly at the beginning of a stored procedure to activate the role, when the stored procedure is called by a different user.

Use the following statement for Aurora MySQL version 2:

```
GRANT SELECT INTO S3 ON *.* TO 'user'@'domain-or-ip-address'
```

The AWS_SELECT_S3_ACCESS role and SELECT INTO S3 privilege are specific to Amazon Aurora MySQL and are not available for MySQL databases or RDS for MySQL DB instances. If you have set up replication between an Aurora MySQL DB cluster as the replication master and a MySQL database as the replication client, then the GRANT statement for the role or privilege causes replication to stop with an error. You can safely skip the error to resume replication. To skip the error on an RDS for MySQL DB instance, use the mysql_rds_skip_repl_error procedure. To skip the error on an external MySQL database, use the slave_skip_errors system variable (Aurora MySQL version 2) or replica_skip_errors system variable (Aurora MySQL version 3).

Specifying a path to an Amazon S3 bucket

The syntax for specifying a path to store the data and manifest files on an Amazon S3 bucket is similar to that used in the LOAD DATA FROM S3 PREFIX statement, as shown following.

```
s3-region://bucket-name/file-prefix
```

The path includes the following values:

- region (optional) – The AWS Region that contains the Amazon S3 bucket to save the data into. This value is optional. If you don't specify a region value, then Aurora saves your files into Amazon S3 in the same region as your DB cluster.
- bucket-name – The name of the Amazon S3 bucket to save the data into. Object prefixes that identify a virtual folder path are supported.
- file-prefix – The Amazon S3 object prefix that identifies the files to be saved in Amazon S3.

The data files created by the SELECT INTO OUTFILE S3 statement use the following path, in which *00000* represents a 5-digit, zero-based integer number.

```
s3-region://bucket-name/file-prefix.part_00000
```

For example, suppose that a SELECT INTO OUTFILE S3 statement specifies s3-us-west-2://bucket/prefix as the path in which to store data files and creates three data files. The specified Amazon S3 bucket contains the following data files.

- s3-us-west-2://bucket/prefix.part_00000
- s3-us-west-2://bucket/prefix.part_00001
- s3-us-west-2://bucket/prefix.part_00002

Creating a manifest to list data files

You can use the `SELECT INTO OUTFILE S3` statement with the `MANIFEST ON` option to create a manifest file in JSON format that lists the text files created by the statement. The `LOAD DATA FROM S3` statement can use the manifest file to load the data files back into an Aurora MySQL DB cluster. For more information about using a manifest to load data files from Amazon S3 into an Aurora MySQL DB cluster, see Using a manifest to specify data files to load.

The data files included in the manifest created by the `SELECT INTO OUTFILE S3` statement are listed in the order that they're created by the statement. For example, suppose that a `SELECT INTO OUTFILE S3` statement specified `s3-us-west-2://bucket/prefix` as the path in which to store data files and creates three data files and a manifest file. The specified Amazon S3 bucket contains a manifest file named `s3-us-west-2://bucket/prefix.manifest`, that contains the following information.

```
{
  "entries": [
    {
      "url":"s3-us-west-2://bucket/prefix.part_00000"
    },
    {
      "url":"s3-us-west-2://bucket/prefix.part_00001"
    },
    {
      "url":"s3-us-west-2://bucket/prefix.part_00002"
    }
  ]
}
```

SELECT INTO OUTFILE S3

You can use the `SELECT INTO OUTFILE S3` statement to query data from a DB cluster and save it directly into delimited text files stored in an Amazon S3 bucket.

Compressed files aren't supported. Encrypted files are supported starting in Aurora MySQL version 2.09.0.

Syntax

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
        [HIGH_PRIORITY]
        [STRAIGHT_JOIN]
        [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
        [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr ...]
    [FROM table_references
        [PARTITION partition_list]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}
        [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_condition]
    [ORDER BY {col_name | expr | position}
         [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
INTO OUTFILE S3 's3_uri'
[CHARACTER SET charset_name]
    [export_options]
    [MANIFEST {ON | OFF}]
    [OVERWRITE {ON | OFF}]
    [ENCRYPTION {ON | OFF | SSE_S3 | SSE_KMS ['cmk_id']}]

export_options:
    [FORMAT {CSV|TEXT} [HEADER]]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
]
```

Parameters

The SELECT INTO OUTFILE S3 statement uses the following required and optional parameters that are specific to Aurora.

s3-uri

Specifies the URI for an Amazon S3 prefix to use. Use the syntax described in Specifying a path to an Amazon S3 bucket.

FORMAT {CSV|TEXT} [HEADER]

Optionally saves the data in CSV format.

The TEXT option is the default and produces the existing MySQL export format.

The CSV option produces comma-separated data values. The CSV format follows the specification in RFC-4180. If you specify the optional keyword HEADER, the output file contains one header line. The labels in the header line correspond to the column names from the SELECT statement. You can use the CSV files for training data models for use with AWS ML services. For more information about using exported Aurora data with AWS ML services, see Exporting data to Amazon S3 for SageMaker model training (Advanced).

MANIFEST {ON | OFF}

Indicates whether a manifest file is created in Amazon S3. The manifest file is a JavaScript Object Notation (JSON) file that can be used to load data into an Aurora DB cluster with the LOAD DATA FROM S3 MANIFEST statement. For more information about LOAD DATA FROM S3 MANIFEST, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.

If MANIFEST ON is specified in the query, the manifest file is created in Amazon S3 after all data files have been created and uploaded. The manifest file is created using the following path:

```
s3-region://bucket-name/file-prefix.manifest
```

For more information about the format of the manifest file's contents, see Creating a manifest to list data files.

OVERWRITE {ON | OFF}

Indicates whether existing files in the specified Amazon S3 bucket are overwritten. If `OVERWRITE ON` is specified, existing files that match the file prefix in the URI specified in `s3-uri`are overwritten. Otherwise, an error occurs.

ENCRYPTION {ON | OFF | SSE_S3 | SSE_KMS ['*cmk_id*']}

Indicates whether to use server-side encryption with Amazon S3 managed keys (SSE-S3) or AWS KMS keys (SSE-KMS, including AWS managed keys and customer managed keys). The `SSE_S3` and `SSE_KMS` settings are available in Aurora MySQL version 3.05 and higher.

You can also use the `aurora_select_into_s3_encryption_default` session variable instead of the `ENCRYPTION` clause, as shown in the following example. Use either the SQL clause or the session variable, but not both.

```
set session set session aurora_select_into_s3_encryption_default={ON | OFF | SSE_S3
 | SSE_KMS};
```

The `SSE_S3` and `SSE_KMS` settings are available in Aurora MySQL version 3.05 and higher.

When you set `aurora_select_into_s3_encryption_default` to the following value:

- `OFF` – The default encryption policy of the S3 bucket is followed. The default value of `aurora_select_into_s3_encryption_default` is `OFF`.
- `ON` or `SSE_S3` – The S3 object is encrypted using Amazon S3 managed keys (SSE-S3).
- `SSE_KMS` – The S3 object is encrypted using an AWS KMS key.

  In this case, you also include the session variable `aurora_s3_default_cmk_id`, for example:

  ```
  set session aurora_select_into_s3_encryption_default={SSE_KMS};
  set session aurora_s3_default_cmk_id={NULL | 'cmk_id'};
  ```

  - When `aurora_s3_default_cmk_id` is `NULL`, the S3 object is encrypted using an AWS managed key.
  - When `aurora_s3_default_cmk_id` is a nonempty string `cmk_id`, the S3 object is encrypted using a customer managed key.

    The value of `cmk_id` can't be an empty string.

When you use the SELECT INTO OUTFILE S3 command, Aurora determines the encryption as follows:

- If the ENCRYPTION clause is present in the SQL command, Aurora relies only on the value of ENCRYPTION, and doesn't use a session variable.
- If the ENCRYPTION clause isn't present, Aurora relies on the value of the session variable.

For more information, see Using server-side encryption with Amazon S3 managed keys (SSE-S3) and Using server-side encryption withAWS KMS keys (SSE-KMS) in the *Amazon Simple Storage Service User Guide*.

You can find more details about other parameters in SELECT statement and LOAD DATA statement, in the MySQL documentation.

Considerations

The number of files written to the Amazon S3 bucket depends on the amount of data selected by the SELECT INTO OUTFILE S3 statement and the file size threshold for Aurora MySQL. The default file size threshold is 6 gigabytes (GB). If the data selected by the statement is less than the file size threshold, a single file is created; otherwise, multiple files are created. Other considerations for files created by this statement include the following:

- Aurora MySQL guarantees that rows in data files are not split across file boundaries. For multiple files, the size of every data file except the last is typically close to the file size threshold. However, occasionally staying under the file size threshold results in a row being split across two data files. In this case, Aurora MySQL creates a data file that keeps the row intact, but might be larger than the file size threshold.
- Because each SELECT statement in Aurora MySQL runs as an atomic transaction, a SELECT INTO OUTFILE S3 statement that selects a large data set might run for some time. If the statement fails for any reason, you might need to start over and issue the statement again. If the statement fails, however, files already uploaded to Amazon S3 remain in the specified Amazon S3 bucket. You can use another statement to upload the remaining data instead of starting over again.
- If the amount of data to be selected is large (more than 25 GB), we recommend that you use multiple SELECT INTO OUTFILE S3 statements to save the data to Amazon S3. Each statement should select a different portion of the data to be saved, and also specify a different file_prefix in the s3-uri parameter to use when saving the data files. Partitioning the data to be selected with multiple statements makes it easier to recover from an error in one

statement. If an error occurs for one statement, only a portion of data needs to be re-selected and uploaded to Amazon S3. Using multiple statements also helps to avoid a single long-running transaction, which can improve performance.

- If multiple `SELECT INTO OUTFILE S3` statements that use the same `file_prefix` in the `s3-uri` parameter run in parallel to select data into Amazon S3, the behavior is undefined.
- Metadata, such as table schema or file metadata, is not uploaded by Aurora MySQL to Amazon S3.
- In some cases, you might re-run a `SELECT INTO OUTFILE S3` query, such as to recover from a failure. In these cases, you must either remove any existing data files in the Amazon S3 bucket with the same file prefix specified in `s3-uri`, or include `OVERWRITE ON` in the `SELECT INTO OUTFILE S3` query.

The `SELECT INTO OUTFILE S3` statement returns a typical MySQL error number and response on success or failure. If you don't have access to the MySQL error number and response, the easiest way to determine when it's done is by specifying `MANIFEST ON` in the statement. The manifest file is the last file written by the statement. In other words, if you have a manifest file, the statement has completed.

Currently, there's no way to directly monitor the progress of the `SELECT INTO OUTFILE S3` statement while it runs. However, suppose that you're writing a large amount of data from Aurora MySQL to Amazon S3 using this statement, and you know the size of the data selected by the statement. In this case, you can estimate progress by monitoring the creation of data files in Amazon S3.

To do so, you can use the fact that a data file is created in the specified Amazon S3 bucket for about every 6 GB of data selected by the statement. Divide the size of the data selected by 6 GB to get the estimated number of data files to create. You can then estimate the progress of the statement by monitoring the number of files uploaded to Amazon S3 while the statement runs.

Examples

The following statement selects all of the data in the `employees` table and saves the data into an Amazon S3 bucket that is in a different region from the Aurora MySQL DB cluster. The statement creates data files in which each field is terminated by a comma (`,`) character and each row is terminated by a newline (`\n`) character. The statement returns an error if files that match the `sample_employee_data` file prefix exist in the specified Amazon S3 bucket.

```
SELECT * FROM employees INTO OUTFILE S3 's3-us-west-2://aurora-select-into-s3-pdx/
sample_employee_data'
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n';
```

The following statement selects all of the data in the `employees` table and saves the data into an Amazon S3 bucket that is in the same region as the Aurora MySQL DB cluster. The statement creates data files in which each field is terminated by a comma (`,`) character and each row is terminated by a newline (`\n`) character, and also a manifest file. The statement returns an error if files that match the `sample_employee_data` file prefix exist in the specified Amazon S3 bucket.

```
SELECT * FROM employees INTO OUTFILE S3 's3://aurora-select-into-s3-pdx/
sample_employee_data'
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    MANIFEST ON;
```

The following statement selects all of the data in the `employees` table and saves the data into an Amazon S3 bucket that is in a different region from the Aurora DB cluster. The statement creates data files in which each field is terminated by a comma (`,`) character and each row is terminated by a newline (`\n`) character. The statement overwrites any existing files that match the `sample_employee_data` file prefix in the specified Amazon S3 bucket.

```
SELECT * FROM employees INTO OUTFILE S3 's3-us-west-2://aurora-select-into-s3-pdx/
sample_employee_data'
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    OVERWRITE ON;
```

The following statement selects all of the data in the `employees` table and saves the data into an Amazon S3 bucket that is in the same region as the Aurora MySQL DB cluster. The statement creates data files in which each field is terminated by a comma (`,`) character and each row is terminated by a newline (`\n`) character, and also a manifest file. The statement overwrites any existing files that match the `sample_employee_data` file prefix in the specified Amazon S3 bucket.

```
SELECT * FROM employees INTO OUTFILE S3 's3://aurora-select-into-s3-pdx/
sample_employee_data'
```

```
    FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n'
    MANIFEST ON
    OVERWRITE ON;
```

Invoking a Lambda function from an Amazon Aurora MySQL DB cluster

You can invoke an AWS Lambda function from an Amazon Aurora MySQL-Compatible Edition DB cluster with the native function `lambda_sync` or `lambda_async`. Before invoking a Lambda function from an Aurora MySQL, the Aurora DB cluster must have access to Lambda. For details about granting access to Aurora MySQL, see Giving Aurora access to Lambda. For information about the `lambda_sync` and `lambda_async` stored functions, see Invoking a Lambda function with an Aurora MySQL native function.

You can also call an AWS Lambda function by using a stored procedure. However, using a stored procedure is deprecated. We strongly recommend using an Aurora MySQL native function if you are using one of the following Aurora MySQL versions:

- Aurora MySQL version 2, for MySQL 5.7-compatible clusters.
- Aurora MySQL version 3.01 and higher, for MySQL 8.0-compatible clusters. The stored procedure isn't available in Aurora MySQL version 3.

Topics

- Giving Aurora access to Lambda
- Invoking a Lambda function with an Aurora MySQL native function
- Invoking a Lambda function with an Aurora MySQL stored procedure (deprecated)

Giving Aurora access to Lambda

Before you can invoke Lambda functions from an Aurora MySQL DB cluster, make sure to first give your cluster permission to access Lambda.

To give Aurora MySQL access to Lambda

1. Create an AWS Identity and Access Management (IAM) policy that provides the permissions that allow your Aurora MySQL DB cluster to invoke Lambda functions. For instructions, see Creating an IAM policy to access AWS Lambda resources.

2. Create an IAM role, and attach the IAM policy you created in Creating an IAM policy to access AWS Lambda resources to the new IAM role. For instructions, see Creating an IAM role to allow Amazon Aurora to access AWS services.
3. Set the `aws_default_lambda_role` DB cluster parameter to the Amazon Resource Name (ARN) of the new IAM role.

   If the cluster is part of an Aurora global database, apply the same setting for each Aurora cluster in the global database.

   For more information about DB cluster parameters, see Amazon Aurora DB cluster and DB instance parameters.
4. To permit database users in an Aurora MySQL DB cluster to invoke Lambda functions, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with the DB cluster. For information about associating an IAM role with a DB cluster, see Associating an IAM role with an Amazon Aurora MySQL DB cluster.

   If the cluster is part of an Aurora global database, associate the role with each Aurora cluster in the global database.
5. Configure your Aurora MySQL DB cluster to allow outbound connections to Lambda. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

   If the cluster is part of an Aurora global database, enable outbound connections for each Aurora cluster in the global database.

Invoking a Lambda function with an Aurora MySQL native function

> **Note**
>
> You can call the native functions `lambda_sync` and `lambda_async` when you use Aurora MySQL version 2, or Aurora MySQL version 3.01 and higher. For more information about Aurora MySQL versions, see Database engine updates for Amazon Aurora MySQL.

You can invoke an AWS Lambda function from an Aurora MySQL DB cluster by calling the native functions `lambda_sync` and `lambda_async`. This approach can be useful when you want to integrate your database running on Aurora MySQL with other AWS services. For example, you might want to send a notification using Amazon Simple Notification Service (Amazon SNS) whenever a row is inserted into a specific table in your database.

Contents

Working with native functions to invoke a Lambda function

The `lambda_sync` and `lambda_async` functions are built-in, native functions that invoke a Lambda function synchronously or asynchronously. When you must know the result of the Lambda function before moving on to another action, use the synchronous function `lambda_sync`. When you don't need to know the result of the Lambda function before moving on to another action, use the asynchronous function `lambda_async`.

Granting the role in Aurora MySQL version 3

In Aurora MySQL version 3, the user invoking a native function must be granted the `AWS_LAMBDA_ACCESS` role. To grant this role to a user, connect to the DB instance as the administrative user, and run the following statement.

```
GRANT AWS_LAMBDA_ACCESS TO user@domain-or-ip-address
```

You can revoke this role by running the following statement.

```
REVOKE AWS_LAMBDA_ACCESS FROM user@domain-or-ip-address
```

Tip

When you use the role technique in Aurora MySQL version 3, you can also activate the role by using the `SET ROLE` *`role_name`* or `SET ROLE ALL` statement. If you aren't familiar with the MySQL 8.0 role system, you can learn more in Role-based privilege model. For more details, see Using roles in the *MySQL Reference Manual*.

This only applies to the current active session. When you reconnect, you must run the `SET ROLE` statement again to grant privileges. For more information, see SET ROLE statement in the *MySQL Reference Manual*.

You can use the `activate_all_roles_on_login` DB cluster parameter to automatically activate all roles when a user connects to a DB instance. When this parameter is set, you generally don't have to call the `SET ROLE` statement explicitly to activate a role. For more information, see activate_all_roles_on_login in the *MySQL Reference Manual*.

However, you must call `SET ROLE ALL` explicitly at the beginning of a stored procedure to activate the role, when the stored procedure is called by a different user.

If you get an error such as the following when you try to invoke a Lambda function, then run a `SET ROLE` statement.

```
SQL Error [1227] [42000]: Access denied; you need (at least one of) the Invoke Lambda
 privilege(s) for this operation
```

Granting the privilege in Aurora MySQL version 2

In Aurora MySQL version 2, the user invoking a native function must be granted the `INVOKE LAMBDA` privilege. To grant this privilege to a user, connect to the DB instance as the administrative user, and run the following statement.

```
GRANT INVOKE LAMBDA ON *.* TO user@domain-or-ip-address
```

You can revoke this privilege by running the following statement.

```
REVOKE INVOKE LAMBDA ON *.* FROM user@domain-or-ip-address
```

Syntax for the lambda_sync function

You invoke the `lambda_sync` function synchronously with the `RequestResponse` invocation type. The function returns the result of the Lambda invocation in a JSON payload. The function has the following syntax.

```
lambda_sync (
  lambda_function_ARN,
  JSON_payload
)
```

Parameters for the lambda_sync function

The `lambda_sync` function has the following parameters.

lambda_function_ARN

The Amazon Resource Name (ARN) of the Lambda function to invoke.

JSON_payload

The payload for the invoked Lambda function, in JSON format.

> **Note**
>
> Aurora MySQL version 3 supports the JSON parsing functions from MySQL 8.0. However, Aurora MySQL version 2 doesn't include those functions. JSON parsing isn't required when a Lambda function returns an atomic value, such as a number or a string.

Example for the lambda_sync function

The following query based on `lambda_sync` invokes the Lambda function `BasicTestLambda` synchronously using the function ARN. The payload for the function is `{"operation": "ping"}`.

```
SELECT lambda_sync(
    'arn:aws:lambda:us-east-1:123456789012:function:BasicTestLambda',
    '{"operation": "ping"}');
```

Syntax for the lambda_async function

You invoke the `lambda_async` function asynchronously with the `Event` invocation type. The function returns the result of the Lambda invocation in a JSON payload. The function has the following syntax.

```
lambda_async (
  lambda_function_ARN,
  JSON_payload
)
```

Parameters for the lambda_async function

The `lambda_async` function has the following parameters.

lambda_function_ARN

The Amazon Resource Name (ARN) of the Lambda function to invoke.

JSON_payload

The payload for the invoked Lambda function, in JSON format.

Note

Aurora MySQL version 3 supports the JSON parsing functions from MySQL 8.0. However, Aurora MySQL version 2 doesn't include those functions. JSON parsing isn't required when a Lambda function returns an atomic value, such as a number or a string.

Example for the lambda_async function

The following query based on `lambda_async` invokes the Lambda function `BasicTestLambda` asynchronously using the function ARN. The payload for the function is `{"operation": "ping"}`.

```
SELECT lambda_async(
    'arn:aws:lambda:us-east-1:123456789012:function:BasicTestLambda',
    '{"operation": "ping"}');
```

Invoking a Lambda function within a trigger

You can use triggers to call Lambda on data-modifying statements. The following example uses the `lambda_async` native function and stores the result in a variable.

```
mysql>SET @result=0;
mysql>DELIMITER //
mysql>CREATE TRIGGER myFirstTrigger
      AFTER INSERT
          ON Test_trigger FOR EACH ROW
      BEGIN
      SELECT lambda_async(
          'arn:aws:lambda:us-east-1:123456789012:function:BasicTestLambda',
          '{"operation": "ping"}')
          INTO @result;
      END; //
mysql>DELIMITER ;
```

> **Note**
>
> Triggers aren't run once per SQL statement, but once per row modified, one row at a time. When a trigger runs, the process is synchronous. The data-modifying statement only returns when the trigger completes.
>
> Be careful when invoking an AWS Lambda function from triggers on tables that experience high write traffic. `INSERT`, `UPDATE`, and `DELETE` triggers are activated per row. A write-heavy workload on a table with `INSERT`, `UPDATE`, or `DELETE` triggers results in a large number of calls to your AWS Lambda function.

Invoking a Lambda function with an Aurora MySQL stored procedure (deprecated)

You can invoke an AWS Lambda function from an Aurora MySQL DB cluster by calling the `mysql.lambda_async` procedure. This approach can be useful when you want to integrate your database running on Aurora MySQL with other AWS services. For example, you might want to send a notification using Amazon Simple Notification Service (Amazon SNS) whenever a row is inserted into a specific table in your database.

Contents

Aurora MySQL version considerations

Starting in Aurora MySQL version 2, you can use the native function method instead of these stored procedures to invoke a Lambda function. For more information about the native functions, see Working with native functions to invoke a Lambda function.

In Aurora MySQL version 2, the stored procedure `mysql.lambda_async` is no longer supported. We strongly recommend that you work with native Lambda functions instead.

In Aurora MySQL version 3, the stored procedure isn't available.

Working with the mysql.lambda_async procedure to invoke a Lambda function (deprecated)

The `mysql.lambda_async` procedure is a built-in stored procedure that invokes a Lambda function asynchronously. To use this procedure, your database user must have `EXECUTE` privilege on the `mysql.lambda_async` stored procedure.

Syntax

The `mysql.lambda_async` procedure has the following syntax.

```
CALL mysql.lambda_async (
  lambda_function_ARN,
  lambda_function_input
)
```

Parameters

The `mysql.lambda_async` procedure has the following parameters.

lambda_function_ARN

The Amazon Resource Name (ARN) of the Lambda function to invoke.

lambda_function_input

The input string, in JSON format, for the invoked Lambda function.

Examples

As a best practice, we recommend that you wrap calls to the `mysql.lambda_async` procedure in a stored procedure that can be called from different sources such as triggers or client code. This approach can help to avoid impedance mismatch issues and make it easier to invoke Lambda functions.

> **Note**
>
> Be careful when invoking an AWS Lambda function from triggers on tables that experience high write traffic. `INSERT`, `UPDATE`, and `DELETE` triggers are activated per row. A write-heavy workload on a table with `INSERT`, `UPDATE`, or `DELETE` triggers results in a large number of calls to your AWS Lambda function.
> Although calls to the `mysql.lambda_async` procedure are asynchronous, triggers are synchronous. A statement that results in a large number of trigger activations doesn't wait for the call to the AWS Lambda function to complete, but it does wait for the triggers to complete before returning control to the client.

Example Example: Invoke an AWS Lambda function to send email

The following example creates a stored procedure that you can call in your database code to send an email using a Lambda function.

AWS Lambda Function

```
import boto3

ses = boto3.client('ses')

def SES_send_email(event, context):

    return ses.send_email(
        Source=event['email_from'],
        Destination={
            'ToAddresses': [
            event['email_to'],
            ]
        },

        Message={
            'Subject': {
            'Data': event['email_subject']
            },
            'Body': {
                'Text': {
                    'Data': event['email_body']
                }
            }
        }
    )
```

Stored Procedure

```
DROP PROCEDURE IF EXISTS SES_send_email;
DELIMITER ;;
  CREATE PROCEDURE SES_send_email(IN email_from VARCHAR(255),
                                  IN email_to VARCHAR(255),
                                  IN subject VARCHAR(255),
                                  IN body TEXT) LANGUAGE SQL
  BEGIN
    CALL mysql.lambda_async(
         'arn:aws:lambda:us-west-2:123456789012:function:SES_send_email',
         CONCAT('{"email_to" : "', email_to,
             '", "email_from" : "', email_from,
             '", "email_subject" : "', subject,
             '", "email_body" : "', body, '"}')
     );
  END
  ;;
DELIMITER ;
```

Call the Stored Procedure to Invoke the AWS Lambda Function

```
mysql> call SES_send_email('example_from@amazon.com', 'example_to@amazon.com', 'Email
 subject', 'Email content');
```

Example Example: Invoke an AWS Lambda function to publish an event from a trigger

The following example creates a stored procedure that publishes an event by using Amazon SNS. The code calls the procedure from a trigger when a row is added to a table.

AWS Lambda Function

```
import boto3

sns = boto3.client('sns')

def SNS_publish_message(event, context):

    return sns.publish(
        TopicArn='arn:aws:sns:us-west-2:123456789012:Sample_Topic',
        Message=event['message'],
        Subject=event['subject'],
        MessageStructure='string'
    )
```

Stored Procedure

```
DROP PROCEDURE IF EXISTS SNS_Publish_Message;
DELIMITER ;;
CREATE PROCEDURE SNS_Publish_Message (IN subject VARCHAR(255),
                                      IN message TEXT) LANGUAGE SQL
BEGIN
  CALL mysql.lambda_async('arn:aws:lambda:us-
west-2:123456789012:function:SNS_publish_message',
     CONCAT('{ "subject" : "', subject,
            '", "message" : "', message, '" }')
     );
END
;;
DELIMITER ;
```

Table

```
CREATE TABLE 'Customer_Feedback' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'customer_name' varchar(255) NOT NULL,
  'customer_feedback' varchar(1024) NOT NULL,
  PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Trigger

```
DELIMITER ;;
CREATE TRIGGER TR_Customer_Feedback_AI
  AFTER INSERT ON Customer_Feedback
  FOR EACH ROW
BEGIN
  SELECT CONCAT('New customer feedback from ', NEW.customer_name),
 NEW.customer_feedback INTO @subject, @feedback;
  CALL SNS_Publish_Message(@subject, @feedback);
END
;;
DELIMITER ;
```

Insert a Row into the Table to Trigger the Notification

```
mysql> insert into Customer_Feedback (customer_name, customer_feedback) VALUES ('Sample
 Customer', 'Good job guys!');
```

Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs

You can configure your Aurora MySQL DB cluster to publish general, slow, audit, and error log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, and use CloudWatch to create alarms and view metrics. You can use CloudWatch Logs to store your log records in highly durable storage.

To publish logs to CloudWatch Logs, the respective logs must be enabled. Error logs are enabled by default, but you must enable the other types of logs explicitly. For information about enabling logs in MySQL, see Selecting general query and slow query log output destinations in the MySQL documentation. For more information about enabling Aurora MySQL audit logs, see Enabling Advanced Auditing.

Note

- If exporting log data is disabled, Aurora doesn't delete existing log groups or log streams. If exporting log data is disabled, existing log data remains available in CloudWatch Logs, depending on log retention, and you still incur charges for stored audit log data. You can delete log streams and log groups using the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API.

- An alternative way to publish audit logs to CloudWatch Logs is by enabling Advanced Auditing, then creating a custom DB cluster parameter group and setting the `server_audit_logs_upload` parameter to `1`. The default for the `server_audit_logs_upload` DB cluster parameter is `0`. For information on enabling Advanced Auditing, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

  If you use this alternative method, you must have an IAM role to access CloudWatch Logs and set the `aws_default_logs_role` cluster-level parameter to the ARN for this role. For information about creating the role, see Setting up IAM roles to access AWS services. However, if you have the `AWSServiceRoleForRDS` service-linked role, it provides access to CloudWatch Logs and overrides any custom-defined roles. For information about service-linked roles for Amazon RDS, see Using service-linked roles for Amazon Aurora.
- If you don't want to export audit logs to CloudWatch Logs, make sure that all methods of exporting audit logs are disabled. These methods are the AWS Management Console, the AWS CLI, the RDS API, and the `server_audit_logs_upload` parameter.
- The procedure is slightly different for Aurora Serverless v1 DB clusters than for DB clusters with provisioned or Aurora Serverless v2 DB instances. Aurora Serverless v1 clusters automatically upload all of the logs that you enable through configuration parameters.

  Therefore, you turn on or turn off log upload for Aurora Serverless v1 DB clusters by turning different log types on and off in the DB cluster parameter group. You don't modify the settings of the cluster itself through the AWS Management Console, AWS CLI, or RDS API. For information about turning on and off MySQL logs for Aurora Serverless v1 clusters, see Parameter groups for Aurora Serverless v1.

Console

You can publish Aurora MySQL logs for provisioned clusters to CloudWatch Logs with the console.

To publish Aurora MySQL logs from the console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Databases**.
3. Choose the Aurora MySQL DB cluster that you want to publish the log data for.
4. Choose **Modify**.
5. In the **Log exports** section, choose the logs that you want to start publishing to CloudWatch Logs.
6. Choose **Continue**, and then choose **Modify DB Cluster** on the summary page.

AWS CLI

You can publish Aurora MySQL logs for provisioned clusters with the AWS CLI. To do so, you run the modify-db-cluster AWS CLI command with the following options:

- `--db-cluster-identifier`—The DB cluster identifier.
- `--cloudwatch-logs-export-configuration`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

You can also publish Aurora MySQL logs by running one of the following AWS CLI commands:

- create-db-cluster
- restore-db-cluster-from-s3
- restore-db-cluster-from-snapshot
- restore-db-cluster-to-point-in-time

Run one of these AWS CLI commands with the following options:

- `--db-cluster-identifier`—The DB cluster identifier.
- `--engine`—The database engine.
- `--enable-cloudwatch-logs-exports`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

Other options might be required depending on the AWS CLI command that you run.

Example

The following command modifies an existing Aurora MySQL DB cluster to publish log files to CloudWatch Logs.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":
["error","general","slowquery","audit"]}'
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":
["error","general","slowquery","audit"]}'
```

Example

The following command creates an Aurora MySQL DB cluster to publish log files to CloudWatch Logs.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
    --db-cluster-identifier mydbcluster \
    --engine aurora \
    --enable-cloudwatch-logs-exports '["error","general","slowquery","audit"]'
```

For Windows:

```
aws rds create-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --engine aurora ^
    --enable-cloudwatch-logs-exports '["error","general","slowquery","audit"]'
```

RDS API

You can publish Aurora MySQL logs for provisioned clusters with the RDS API. To do so, you run the ModifyDBCluster operation with the following options:

- `DBClusterIdentifier`—The DB cluster identifier.
- `CloudwatchLogsExportConfiguration`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

You can also publish Aurora MySQL logs with the RDS API by running one of the following RDS API operations:

- CreateDBCluster
- RestoreDBClusterFromS3
- RestoreDBClusterFromSnapshot
- RestoreDBClusterToPointInTime

Run the RDS API operation with the following parameters:

- `DBClusterIdentifier`—The DB cluster identifier.
- `Engine`—The database engine.
- `EnableCloudwatchLogsExports`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

Other parameters might be required depending on the AWS CLI command that you run.

Monitoring log events in Amazon CloudWatch

After enabling Aurora MySQL log events, you can monitor the events in Amazon CloudWatch Logs. A new log group is automatically created for the Aurora DB cluster under the following prefix, in which *`cluster-name`* represents the DB cluster name, and *`log_type`* represents the log type.

```
/aws/rds/cluster/cluster-name/log_type
```

For example, if you configure the export function to include the slow query log for a DB cluster named `mydbcluster`, slow query data is stored in the `/aws/rds/cluster/mydbcluster/slowquery` log group.

The events from all instances in your cluster are pushed to a log group using different log streams. The behavior depends on which of the following conditions is true:

- A log group with the specified name exists.

  Aurora uses the existing log group to export log data for the cluster. To create log groups with predefined log retention periods, metric filters, and customer access, you can use automated configuration, such as AWS CloudFormation.
- A log group with the specified name doesn't exist.

  When a matching log entry is detected in the log file for the instance, Aurora MySQL creates a new log group in CloudWatch Logs automatically. The log group uses the default log retention period of **Never Expire**.

  To change the log retention period, use the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API. For more information about changing log retention periods in CloudWatch Logs, see Change log data retention in CloudWatch Logs.

To search for information within the log events for a DB cluster, use the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API. For more information about searching and filtering log data, see Searching and filtering log data.

Amazon Aurora MySQL lab mode

Aurora lab mode is used to enable Aurora features that are available in the current Aurora database version, but are not enabled by default. While Aurora lab mode features are not recommended for use in production DB clusters, you can use Aurora lab mode to enable these features for DB clusters in your development and test environments. For more information about Aurora features available when Aurora lab mode is enabled, see Aurora lab mode features.

The `aurora_lab_mode` parameter is an instance-level parameter that is in the default parameter group. The parameter is set to `0` (disabled) in the default parameter group. To enable Aurora lab mode, create a custom parameter group, set the `aurora_lab_mode` parameter to `1` (enabled) in the custom parameter group, and modify one or more DB instances in your Aurora cluster to use the custom parameter group. Then connect to the appropriate instance endpoint to try the lab mode features. For information on modifying a DB parameter group, see Modifying parameters in a DB parameter group. For information on parameter groups and Amazon Aurora, see Aurora MySQL configuration parameters.

Aurora lab mode features

The following table lists the Aurora features currently available when Aurora lab mode is enabled. You must enable Aurora lab mode before any of these features can be used.

Feature	Description
Scan Batching	Aurora MySQL scan batching speeds up in-memory, scan-oriented queries significantly. The feature boosts the performance of table full scans, index full scans, and index range scans by batch processing.
Hash Joins	This feature can improve query performance when you need to join a large amount of data by using an equijoin. You can use this feature without lab mode in Aurora MySQL version 2. For more information about using this feature, see Optimizing large Aurora MySQL join queries with hash joins.

Feature	Description
Fast DDL	This feature allows you to run an `ALTER TABLE tbl_name ADD COLUMN col_name column_definition` operation nearly instantaneously. The operation completes without requiring the table to be copied and without materially impacting other DML statements. Since it does not consume temporary storage for a table copy, it makes DDL statements practical even for large tables on small instance classes. Fast DDL is currently only supported for adding a nullable column, without a default value, at the end of a table. For more information about using this feature, see Altering tables in Amazon Aurora using Fast DDL.

Best practices with Amazon Aurora MySQL

This topic includes information on best practices and options for using or migrating data to an Amazon Aurora MySQL DB cluster. The information in this topic summarizes and reiterates some of the guidelines and procedures that you can find in Managing an Amazon Aurora DB cluster.

Contents

Determining which DB instance you are connected to

To determine which DB instance in an Aurora MySQL DB cluster a connection is connected to, check the `innodb_read_only` global variable, as shown in the following example.

```
SHOW GLOBAL VARIABLES LIKE 'innodb_read_only';
```

The `innodb_read_only` variable is set to `ON` if you are connected to a reader DB instance. This setting is `OFF` if you are connected to a writer DB instance, such as primary instance in a provisioned cluster.

This approach can be helpful if you want to add logic to your application code to balance the workload or to ensure that a write operation is using the correct connection.

Best practices for Aurora MySQL performance and scaling

You can apply the following best practices to improve the performance and scalability of your Aurora MySQL clusters.

Topics

Using T instance classes for development and testing

Amazon Aurora MySQL instances that use the `db.t2`, `db.t3`, or `db.t4g` DB instance classes are best suited for applications that do not support a high workload for an extended amount of time. The T instances are designed to provide moderate baseline performance and the capability to burst to significantly higher performance as required by your workload. They are intended for workloads that don't use the full CPU often or consistently, but occasionally need to burst. We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see Burstable performance instances.

If your Aurora cluster is larger than 40 TB, don't use the T instance classes. When your database has a large volume of data, the memory overhead for managing schema objects can exceed the capacity of a T instance.

Don't enable the MySQL Performance Schema on Amazon Aurora MySQL T instances. If the Performance Schema is enabled, the instance might run out of memory.

> **Tip**
>
> If your database is sometimes idle but at other times has a substantial workload, you can use Aurora Serverless v2 as an alternative to T instances. With Aurora Serverless v2, you define a capacity range and Aurora automatically scales your database up or down depending on the current workload. For usage details, see Using Aurora Serverless v2. For the database engine versions that you can use with Aurora Serverless v2, see Requirements and limitations for Aurora Serverless v2.

When you use a T instance as a DB instance in an Aurora MySQL DB cluster, we recommend the following:

- Use the same DB instance class for all instances in your DB cluster. For example, if you use `db.t2.medium` for your writer instance, then we recommend that you use `db.t2.medium` for your reader instances also.
- Don't adjust any memory-related configuration settings, such as `innodb_buffer_pool_size`. Aurora uses a highly tuned set of default values for memory buffers on T instances. These special defaults are needed for Aurora to run on memory-constrained instances. If you change any memory-related settings on a T instance, you are much more likely to encounter out-of-memory conditions, even if your change is intended to increase buffer sizes.
- Monitor your CPU Credit Balance (`CPUCreditBalance`) to ensure that it is at a sustainable level. That is, CPU credits are being accumulated at the same rate as they are being used.

  When you have exhausted the CPU credits for an instance, you see an immediate drop in the available CPU and an increase in the read and write latency for the instance. This situation results in a severe decrease in the overall performance of the instance.

  If your CPU credit balance is not at a sustainable level, then we recommend that you modify your DB instance to use a one of the supported R DB instance classes (scale compute).

For more information on monitoring metrics, see Viewing metrics in the Amazon RDS console.

- Monitor the replica lag (AuroraReplicaLag) between the writer instance and the reader instances.

  If a reader instance runs out of CPU credits before the writer instance does, the resulting lag can cause the reader instance to restart frequently. This result is common when an application has a heavy load of read operations distributed among reader instances, at the same time that the writer instance has a minimal load of write operations.

  If you see a sustained increase in replica lag, make sure that your CPU credit balance for the reader instances in your DB cluster is not being exhausted.

  If your CPU credit balance is not at a sustainable level, then we recommend that you modify your DB instance to use one of the supported R DB instance classes (scale compute).
- Keep the number of inserts per transaction below 1 million for DB clusters that have binary logging enabled.

  If the DB cluster parameter group for your DB cluster has the binlog_format parameter set to a value other than OFF, then your DB cluster might experience out-of-memory conditions if the DB cluster receives transactions that contain over 1 million rows to insert. You can monitor the freeable memory (FreeableMemory) metric to determine if your DB cluster is running out of available memory. You then check the write operations (VolumeWriteIOPS) metric to see if a writer instance is receiving a heavy load of write operations. If this is the case, then we recommend that you update your application to limit the number of inserts in a transaction to less than 1 million. Alternatively, you can modify your instance to use one of the supported R DB instance classes (scale compute).

Optimizing Aurora MySQL indexed join queries with asynchronous key prefetch

Aurora MySQL can use the asynchronous key prefetch (AKP) feature to improve the performance of queries that join tables across indexes. This feature improves performance by anticipating the rows needed to run queries in which a JOIN query requires use of the Batched Key Access (BKA) Join algorithm and Multi-Range Read (MRR) optimization features. For more information about BKA and MRR, see Block nested-loop and batched key access joins and Multi-range read optimization in the MySQL documentation.

To take advantage of the AKP feature, a query must use both BKA and MRR. Typically, such a query occurs when the JOIN clause of a query uses a secondary index, but also needs some columns from the primary index. For example, you can use AKP when a JOIN clause represents an equijoin on index values between a small outer and large inner table, and the index is highly selective on the larger table. AKP works in concert with BKA and MRR to perform a secondary to primary index lookup during the evaluation of the JOIN clause. AKP identifies the rows required to run the query during the evaluation of the JOIN clause. It then uses a background thread to asynchronously load the pages containing those rows into memory before running the query.

AKP is available for Aurora MySQL version 2.10 and higher, and version 3. For more information about Aurora MySQL versions, see Database engine updates for Amazon Aurora MySQL.

Enabling asynchronous key prefetch

You can enable the AKP feature by setting `aurora_use_key_prefetch`, a MySQL server variable, to on. By default, this value is set to on. However, AKP can't be enabled until you also enable the BKA Join algorithm and disable cost-based MRR functionality. To do so, you must set the following values for `optimizer_switch`, a MySQL server variable:

- Set `batched_key_access` to on. This value controls the use of the BKA Join algorithm. By default, this value is set to `off`.
- Set `mrr_cost_based` to `off`. This value controls the use of cost-based MRR functionality. By default, this value is set to on.

Currently, you can set these values only at the session level. The following example illustrates how to set these values to enable AKP for the current session by executing SET statements.

```
mysql> set @@session.aurora_use_key_prefetch=on;
mysql> set @@session.optimizer_switch='batched_key_access=on,mrr_cost_based=off';
```

Similarly, you can use SET statements to disable AKP and the BKA Join algorithm and re-enable cost-based MRR functionality for the current session, as shown in the following example.

```
mysql> set @@session.aurora_use_key_prefetch=off;
mysql> set @@session.optimizer_switch='batched_key_access=off,mrr_cost_based=on';
```

For more information about the **batched_key_access** and **mrr_cost_based** optimizer switches, see Switchable optimizations in the MySQL documentation.

Optimizing queries for asynchronous key prefetch

You can confirm whether a query can take advantage of the AKP feature. To do so, use the EXPLAIN statement to profile the query before running it. The EXPLAIN statement provides information about the execution plan to use for a specified query.

In the output for the EXPLAIN statement, the Extra column describes additional information included with the execution plan. If the AKP feature applies to a table used in the query, this column includes one of the following values:

- Using Key Prefetching
- Using join buffer (Batched Key Access with Key Prefetching)

The following example shows the use of EXPLAIN to view the execution plan for a query that can take advantage of AKP.

```
mysql> explain select sql_no_cache
    ->     ps_partkey,
    ->     sum(ps_supplycost * ps_availqty) as value
    -> from
    ->     partsupp,
    ->     supplier,
    ->     nation
    -> where
    ->     ps_suppkey = s_suppkey
    ->     and s_nationkey = n_nationkey
    ->     and n_name = 'ETHIOPIA'
    -> group by
    ->     ps_partkey having
    ->         sum(ps_supplycost * ps_availqty) > (
    ->             select
    ->                 sum(ps_supplycost * ps_availqty) * 0.0000003333
    ->             from
    ->                 partsupp,
    ->                 supplier,
    ->                 nation
    ->             where
    ->                  ps_suppkey = s_suppkey
    ->                  and s_nationkey = n_nationkey
    ->                  and n_name = 'ETHIOPIA'
    ->          )
    -> order by
    ->     value desc;
+----+-------------+----------+------+-----------------------+---------------
+---------+----------------------------------+------+----------
+--------------------------------------------------------------+
| id | select_type | table    | type | possible_keys         | key           | key_len
 | ref                              | rows | filtered | Extra
                             |
+----+-------------+----------+------+-----------------------+---------------
+---------+----------------------------------+------+----------
+--------------------------------------------------------------+
|  1 | PRIMARY     | nation   | ALL  | PRIMARY               | NULL          | NULL
 | NULL                             |   25 |   100.00 | Using where; Using temporary;
 Using filesort                |
|  1 | PRIMARY     | supplier | ref  | PRIMARY,i_s_nationkey | i_s_nationkey | 5
 | dbt3_scale_10.nation.n_nationkey | 2057 |   100.00 | Using index
                             |
|  1 | PRIMARY     | partsupp | ref  | i_ps_suppkey          | i_ps_suppkey  | 4
 | dbt3_scale_10.supplier.s_suppkey |   42 |   100.00 | Using join buffer (Batched Key
 Access with Key Prefetching) |
|  2 | SUBQUERY    | nation   | ALL  | PRIMARY               | NULL          | NULL
 | NULL                             |   25 |   100.00 | Using where
                             |
|  2 | SUBQUERY    | supplier | ref  | PRIMARY,i_s_nationkey | i_s_nationkey | 5
 | dbt3_scale_10.nation.n_nationkey | 2057 |   100.00 | Using index
                             |
|  2 | SUBQUERY    | partsupp | ref  | i_ps_suppkey          | i_ps_suppkey  | 4
 | dbt3_scale_10.supplier.s_suppkey |   42 |   100.00 | Using join buffer (Batched Key
 Access with Key Prefetching) |
+----+-------------+----------+------+-----------------------+---------------
+---------+----------------------------------+------+----------
+--------------------------------------------------------------+
6 rows in set, 1 warning (0.00 sec)
```

For more information about the EXPLAIN output format, see Extended EXPLAIN output format in the MySQL documentation.

Optimizing large Aurora MySQL join queries with hash joins

When you need to join a large amount of data by using an equijoin, a hash join can improve query performance. You can enable hash joins for Aurora MySQL.

A hash join column can be any complex expression. In a hash join column, you can compare across data types in the following ways:

- You can compare anything across the category of precise numeric data types, such as `int`, `bigint`, `numeric`, and `bit`.
- You can compare anything across the category of approximate numeric data types, such as `float` and `double`.
- You can compare items across string types if the string types have the same character set and collation.
- You can compare items with date and timestamp data types if the types are the same.

> **Note**
>
> You can't compare data types in different categories.

The following restrictions apply to hash joins for Aurora MySQL:

- Left-right outer joins aren't supported for Aurora MySQL version 2, but are supported for version 3.
- Semijoins such as subqueries aren't supported, unless the subqueries are materialized first.
- Multiple-table updates or deletes aren't supported.

  > **Note**
  >
  > Single-table updates or deletes are supported.

- BLOB and spatial data type columns can't be join columns in a hash join.

Enabling hash joins

To enable hash joins:

- Aurora MySQL version 2 – Set the DB parameter or DB cluster parameter aurora_disable_hash_join to 0. Turning off aurora_disable_hash_join sets the value of optimizer_switch to hash_join=on.
- Aurora MySQL version 3 – Set the MySQL server parameter optimizer_switch to block_nested_loop=on.

Hash joins are turned on by default in Aurora MySQL version 3 and turned off by default in Aurora MySQL version 2. The following example illustrates how to enable hash joins for Aurora MySQL version 3. You can issue the statement select @@optimizer_switch first to see what other settings are present in the SET parameter string. Updating one setting in the optimizer_switch parameter doesn't erase or modify the other settings.

```
mysql> SET optimizer_switch='block_nested_loop=on';
```

Note

For Aurora MySQL version 3, hash join support is available in all minor versions and is turned on by default.
For Aurora MySQL version 2, hash join support is available in all minor versions.
In Aurora MySQL version 2, the hash join feature is always controlled by the aurora_disable_hash_join value.

With this setting, the optimizer chooses to use a hash join based on cost, query characteristics, and resource availability. If the cost estimation is incorrect, you can force the optimizer to choose a hash join. You do so by setting hash_join_cost_based, a MySQL server variable, to off. The following example illustrates how to force the optimizer to choose a hash join.

```

mysql> SET optimizer_switch='hash_join_cost_based=off';
```

Note

This setting overrides the decisions of the cost-based optimizer. While the setting can be useful for testing and development, we recommend that you not use it in production.

Optimizing queries for hash joins

To find out whether a query can take advantage of a hash join, use the EXPLAIN statement to profile the query first. The EXPLAIN statement provides information about the execution plan to use for a specified query.

In the output for the EXPLAIN statement, the Extra column describes additional information included with the execution plan. If a hash join applies to the tables used in the query, this column includes values similar to the following:

- `Using where; Using join buffer (Hash Join Outer table` *`table1_name`*`)`
- `Using where; Using join buffer (Hash Join Inner table` *`table2_name`*`)`

The following example shows the use of EXPLAIN to view the execution plan for a hash join query.

```
mysql> explain SELECT sql_no_cache * FROM hj_small, hj_big, hj_big2
    ->     WHERE hj_small.col1 = hj_big.col1 and hj_big.col1=hj_big2.col1 ORDER BY 1;
+----+-------------+----------+------+---------------+------+---------+------+------
+----------------------------------------------------------------+
| id | select_type | table    | type | possible_keys | key  | key_len | ref  | rows |
 Extra                                                          |
+----+-------------+----------+------+---------------+------+---------+------+------
+----------------------------------------------------------------+
|  1 | SIMPLE      | hj_small | ALL  | NULL          | NULL | NULL    | NULL |    6 |
 Using temporary; Using filesort                                |
|  1 | SIMPLE      | hj_big   | ALL  | NULL          | NULL | NULL    | NULL |   10 |
 Using where; Using join buffer (Hash Join Outer table hj_big)  |
|  1 | SIMPLE      | hj_big2  | ALL  | NULL          | NULL | NULL    | NULL |   15 |
 Using where; Using join buffer (Hash Join Inner table hj_big2) |
+----+-------------+----------+------+---------------+------+---------+------+------
+----------------------------------------------------------------+
3 rows in set (0.04 sec)
```

In the output, the `Hash Join Inner table` is the table used to build hash table, and the `Hash Join Outer table` is the table that is used to probe the hash table.

For more information about the extended EXPLAIN output format, see Extended EXPLAIN Output Format in the MySQL product documentation.

In Aurora MySQL 2.08 and higher, you can use SQL hints to influence whether a query uses hash join or not, and which tables to use for the build and probe sides of the join. For details, see Aurora MySQL hints.

Using Amazon Aurora to scale reads for your MySQL database

You can use Amazon Aurora with your MySQL DB instance to take advantage of the read scaling capabilities of Amazon Aurora and expand the read workload for your MySQL DB instance. To use Aurora to read scale your MySQL DB instance, create an Aurora MySQL DB cluster and make it a read replica of your MySQL DB instance. Then connect to the Aurora MySQL cluster to process the read queries. The source database can be an RDS for MySQL DB instance, or a MySQL database running external to Amazon RDS. For more information, see Using Amazon Aurora to scale reads for your MySQL database.

Optimizing timestamp operations

When the value of the system variable `time_zone` is set to `SYSTEM`, each MySQL function call that requires a time zone calculation makes a system library call. When you run SQL statements that return or change such `TIMESTAMP` values at high concurrency, you might experience increased latency, lock contention, and CPU usage. For more information, see time_zone in the MySQL documentation.

To avoid this behavior, we recommend that you change the value of the `time_zone` DB cluster parameter to `UTC`. For more information, see Modifying parameters in a DB cluster parameter group.

While the `time_zone` parameter is dynamic (doesn't require a database server restart), the new value is used only for new connections. To make sure that all connections are updated to use the new `time_zone` value, we recommend that you recycle your application connections after updating the DB cluster parameter.

Best practices for Aurora MySQL high availability

You can apply the following best practices to improve the availability of your Aurora MySQL clusters.

Topics

- Migrating from MySQL to Amazon Aurora MySQL with reduced downtime
- Avoiding slow performance, automatic restart, and failover for Aurora MySQL DB instances

Using Amazon Aurora for Disaster Recovery with your MySQL databases

You can use Amazon Aurora with your MySQL DB instance to create an offsite backup for disaster recovery. To use Aurora for disaster recovery of your MySQL DB instance, create an Amazon Aurora DB cluster and make it a read replica of your MySQL DB instance. This applies to an RDS for MySQL DB instance, or a MySQL database running external to Amazon RDS.

> ⚠ **Important**
>
> When you set up replication between a MySQL DB instance and an Amazon Aurora MySQL DB cluster, you should monitor the replication to ensure that it remains healthy and repair it if necessary.

For instructions on how to create an Amazon Aurora MySQL DB cluster and make it a read replica of your MySQL DB instance, follow the procedure in Using Amazon Aurora to scale reads for your MySQL database.

For more information on disaster recovery models, see How to choose the best disaster recovery option for your Amazon Aurora MySQL cluster.

Migrating from MySQL to Amazon Aurora MySQL with reduced downtime

When importing data from a MySQL database that supports a live application to an Amazon Aurora MySQL DB cluster, you might want to reduce the time that service is interrupted while you migrate. To do so, you can use the procedure documented in Importing data to a MySQL or MariaDB DB instance with reduced downtime in the *Amazon Relational Database Service User Guide*. This procedure can especially help if you are working with a very large database. You can use the procedure to reduce the cost of the import by minimizing the amount of data that is passed across the network to AWS.

The procedure lists steps to transfer a copy of your database data to an Amazon EC2 instance and import the data into a new RDS for MySQL DB instance. Because Amazon Aurora is compatible with MySQL, you can instead use an Amazon Aurora DB cluster for the target Amazon RDS MySQL DB instance.

Avoiding slow performance, automatic restart, and failover for Aurora MySQL DB instances

If you're running a heavy workload or workloads that spike beyond the allocated resources of your DB instance, you can exhaust the resources on which you're running your application and Aurora database. To get metrics on your database instance such as CPU utilization, memory usage, and number of database connections used, you can refer to the metrics provided by Amazon CloudWatch, Performance Insights, and Enhanced Monitoring. For more information on monitoring your DB instance, see Monitoring metrics in an Amazon Aurora cluster.

If your workload exhausts the resources you're using, your DB instance might slow down, restart, or even fail over to another DB instance. To avoid this, monitor your resource utilization, examine the workload running on your DB instance, and make optimizations where necessary. If optimizations don't improve the instance metrics and mitigate the resource exhaustion, consider scaling up your DB instance before you reach its limits. For more information on available DB instance classes and their specifications, see Aurora DB instance classes.

Recommendations for Aurora MySQL

The following features are available in Aurora MySQL for MySQL compatibility. However, they have performance, scalability, stability, or compatibility issues in the Aurora environment. Thus, we recommend that you follow certain guidelines in your use of these features. For example, we recommend that you don't use certain features for production Aurora deployments.

Topics

Using multithreaded replication in Aurora MySQL

With multithreaded binary log replication, a SQL thread reads events from the relay log and queues them up for SQL worker threads to apply. The SQL worker threads are managed by a coordinator thread. The binary log events are applied in parallel when possible.

Multithreaded replication is supported in Aurora MySQL version 3, and in Aurora MySQL version 2.12.1 and higher.

For Aurora MySQL versions lower than 3.04, Aurora uses single-threaded replication by default when an Aurora MySQL DB cluster is used as a read replica for binary log replication.

Earlier versions of Aurora MySQL version 2 inherited several issues regarding multithreaded replication from MySQL Community Edition. For those versions, we recommend that you not use multithreaded replication in production.

If you do use multithreaded replication, we recommend that you test it thoroughly.

For more information about using replication in Amazon Aurora, see Replication with Amazon Aurora. For more information about multithreaded replication in Aurora MySQL, see Multithreaded binary log replication.

Invoking AWS Lambda functions using native MySQL functions

We recommend using the native MySQL functions `lambda_sync` and `lambda_async` to invoke Lambda functions.

If you are using the deprecated `mysql.lambda_async` procedure, we recommend that you wrap calls to the `mysql.lambda_async` procedure in a stored procedure. You can call this stored procedure from different sources, such as triggers or client code. This approach can help to avoid impedance mismatch issues and make it easier for your database programmers to invoke Lambda functions.

For more information on invoking Lambda functions from Amazon Aurora, see Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.

Avoiding XA transactions with Amazon Aurora MySQL

We recommend that you don't use eXtended Architecture (XA) transactions with Aurora MySQL, because they can cause long recovery times if the XA was in the `PREPARED` state. If you must use XA transactions with Aurora MySQL, follow these best practices:

- Don't leave an XA transaction open in the `PREPARED` state.
- Keep XA transactions as small as possible.

For more information about using XA transactions with MySQL, see XA transactions in the MySQL documentation.

Keeping foreign keys turned on during DML statements

We strongly recommend that you don't run any data definition language (DDL) statements when the `foreign_key_checks` variable is set to `0` (off).

If you need to insert or update rows that require a transient violation of foreign keys, follow these steps:

1. Set `foreign_key_checks` to `0`.
2. Make your data manipulation language (DML) changes.
3. Make sure that your completed changes don't violate any foreign key constraints.
4. Set `foreign_key_checks` to 1 (on).

In addition, follow these other best practices for foreign key constraints:

- Make sure that your client applications don't set the `foreign_key_checks` variable to `0` as a part of the `init_connect` variable.
- If a restore from a logical backup such as `mysqldump` fails or is incomplete, make sure that `foreign_key_checks` is set to 1 before starting any other operations in the same session. A logical backup sets `foreign_key_checks` to `0` when it starts.

Configuring how frequently the log buffer is flushed

In MySQL Community Edition, to make transactions durable, the InnoDB log buffer must be flushed to durable storage. You use the`innodb_flush_log_at_trx_commit` parameter to configure how frequently the log buffer is flushed to disk.

When you set the `innodb_flush_log_at_trx_commit` parameter to the default value of 1, the log buffer is flushed at each transaction commit. This setting helps to keep the database ACID compliant. We recommend that you keep the default setting of 1.

Changing `innodb_flush_log_at_trx_commit` to a nondefault value can help reduce data manipulation language (DML) latency, but sacrifices the durability of the log records. This lack of durability makes the database ACID noncompliant. We recommend that your databases be ACID compliant to avoid the risk of data loss in the event of a server restart. For more information on this parameter, see innodb_flush_log_at_trx_commit in the MySQL documentation.

In Aurora MySQL, redo log processing is offloaded to the storage layer, so no flushing to log files occurs on the DB instance. When a write is issued, redo logs are sent from the writer DB instance directly to the Aurora cluster volume. The only writes that cross the network are redo log records. No pages are ever written from the database tier.

By default, each thread committing a transaction waits for confirmation from the Aurora cluster volume. This confirmation indicates that this record and all previous redo log records are written and have achieved quorum. Persisting the log records and achieving quorum make the transaction durable, whether through autocommit or explicit commit. For more information on the Aurora storage architecture, see Amazon Aurora storage demystified.

Aurora MySQL doesn't flush logs to data files as MySQL Community Edition does. However, you can use the `innodb_flush_log_at_trx_commit` parameter to relax durability constraints when writing redo log records to the Aurora cluster volume.

For Aurora MySQL version 2:

- `innodb_flush_log_at_trx_commit` = 0 or 2 – The database doesn't wait for confirmation that the redo log records are written to the Aurora cluster volume.
- `innodb_flush_log_at_trx_commit` = 1 – The database waits for confirmation that the redo log records are written to the Aurora cluster volume.

For Aurora MySQL version 3:

- `innodb_flush_log_at_trx_commit` = 0 – The database doesn't wait for confirmation that the redo log records are written to the Aurora cluster volume.
- `innodb_flush_log_at_trx_commit` = 1 or 2 – The database waits for confirmation that the redo log records are written to the Aurora cluster volume.

Therefore, to obtain the same nondefault behavior in Aurora MySQL version 3 that you would with the value set to 0 or 2 in Aurora MySQL version 2, set the parameter to 0.

While these settings can lower DML latency to the client, they can also result in data loss in the event of a failover or restart. Therefore, we recommend that you keep the `innodb_flush_log_at_trx_commit` parameter set to the default value of 1.

While data loss can occur in both MySQL Community Edition and Aurora MySQL, behavior differs in each database because of their different architectures. These architectural differences can lead to varying degrees of data loss. To make sure that your database is ACID compliant, always set `innodb_flush_log_at_trx_commit` to 1.

> **Note**
>
> In Aurora MySQL version 3, before you can change `innodb_flush_log_at_trx_commit` to a value other than 1, you must first change the value of `innodb_trx_commit_allow_data_loss` to 1. By doing so, you acknowledge the risk of data loss.

Minimizing and troubleshooting Aurora MySQL deadlocks

Users running workloads that regularly experience constraint violations on unique secondary indexes or foreign keys, when modifying records on the same data page concurrently, might experience increased deadlocks and lock wait timeouts. These deadlocks and timeouts are because of a MySQL Community Edition bug fix.

This fix is included in MySQL Community Edition versions 5.7.26 and higher, and was backported into Aurora MySQL versions 2.10.3 and higher. The fix is necessary for enforcing *serializability*, by implementing additional locking for these types of data manipulation language (DML) operations, on changes made to records in an InnoDB table. This issue was uncovered as part of an investigation into deadlock issues introduced by a previous MySQL Community Edition bug fix.

The fix changed the internal handling for the *partial rollback* of a tuple (row) update in the InnoDB storage engine. Operations that generate constraint violations on foreign keys or unique secondary indexes cause partial rollback. This includes, but isn't limited to, concurrent `INSERT...ON DUPLICATE KEY UPDATE`, `REPLACE INTO`, and `INSERT IGNORE` statements (*upserts*).

In this context, partial rollback doesn't refer to the rollback of application-level transactions, but rather an internal InnoDB rollback of changes to a clustered index, when a constraint violation is encountered. For example, a duplicate key value is found during an upsert operation.

In a normal insert operation, InnoDB atomically creates clustered and secondary index entries for each index. If InnoDB detects a duplicate value on a unique secondary index during an upsert operation, the inserted entry in the clustered index has to be reverted (partial rollback), and the update then has to be applied to the existing duplicate row. During this internal partial rollback step, InnoDB must lock each record seen as part of the operation. The fix ensures transaction serializability by introducing additional locking after the partial rollback.

Minimizing InnoDB deadlocks

You can take the following approaches to reduce the frequency of deadlocks in your database instance. More examples can be found in the MySQL documentation.

1. To reduce the chances of deadlocks, commit transactions immediately after making a related set of changes. You can do this by breaking up large transactions (multiple row updates between commits) into smaller ones. If you're batch inserting rows, then try to reduce batch insert sizes, especially when using the upsert operations mentioned previously.

   To reduce the number of possible partial rollbacks, you can try some of the following approaches:

   a. Replace batch insert operations with inserting one row at a time. This can reduce the amount of time where locks are held by transactions that might have conflicts.

   b. Instead of using `REPLACE INTO`, rewrite the SQL statement as a multistatement transaction such as the following:

   ```
   BEGIN;
   DELETE conflicting rows;
   INSERT new rows;
   COMMIT;
   ```

   c. Instead of using `INSERT...ON DUPLICATE KEY UPDATE`, rewrite the SQL statement as a multistatement transaction such as the following:

   ```
   BEGIN;
   SELECT rows that conflict on secondary indexes;
   UPDATE conflicting rows;
   INSERT new rows;
   COMMIT;
   ```

2. Avoid long-running transactions, active or idle, that might hold onto locks. This includes interactive MySQL client sessions that might be open for an extended period with an uncommitted transaction. When optimizing transaction sizes or batch sizes, the impact can vary depending on a number of factors such as concurrency, number of duplicates, and table structure. Any changes should be implemented and tested based on your workload.

3. In some situations, deadlocks can occur when two transactions attempt to access the same datasets, either in one or multiple tables, in different orders. To prevent this, you can modify the transactions to access the data in the same order, thereby serializing the access. For example,

create a queue of transactions to be completed. This approach can help to avoid deadlocks when multiple transactions occur concurrently.

4. Adding carefully chosen indexes to your tables can improve selectivity and reduce the need to access rows, which leads to less locking.
5. If you encounter gap locking, you can modify the transaction isolation level to `READ COMMITTED` for the session or transaction to prevent it. For more information on InnoDB isolation levels and their behaviors, see Transaction isolation levels in the MySQL documentation.

> **Note**
>
> While you can take precautions to reduce the possibility of deadlocks occurring, deadlocks are an expected database behavior and can still occur. Applications should have the necessary logic to handle deadlocks when they are encountered. For example, implement retry and backing-off logic in the application. It's best to address the root cause of the issue but if a deadlock does occur, the application has the option to wait and retry.

Monitoring InnoDB deadlocks

Deadlocks can occur in MySQL when application transactions try to take table-level and row-level locks in a way that results in circular waiting. An occasional InnoDB deadlock isn't necessarily an issue, because the InnoDB storage engine detects the condition immediately and rolls back one of the transactions automatically. If you encounter deadlocks frequently, we recommend reviewing and modifying your application to alleviate performance issues and avoid deadlocks. When deadlock detection is turned on (the default), InnoDB automatically detects transaction deadlocks and rolls back a transaction or transactions to break the deadlock. InnoDB tries to pick small transactions to roll back, where the size of a transaction is determined by the number of rows inserted, updated, or deleted.

- `SHOW ENGINE` statement – The `SHOW ENGINE INNODB STATUS \G` statement contains details of the most recent deadlock encountered on the database since the last restart.
- MySQL error log – If you encounter frequent deadlocks where the output of the `SHOW ENGINE` statement is inadequate, you can turn on the innodb_print_all_deadlocks DB cluster parameter.

  When this parameter is turned on, information about all deadlocks in InnoDB user transactions is recorded in the Aurora MySQL error log.

- Amazon CloudWatch metrics – We also recommend that you proactively monitor deadlocks using the CloudWatch metric `Deadlocks`. For more information, see Instance-level metrics for Amazon Aurora.
- Amazon CloudWatch Logs – With CloudWatch Logs, you can view metrics, analyze log data, and create real-time alarms. For more information, see Monitor errors in Amazon Aurora MySQL and Amazon RDS for MySQL using Amazon CloudWatch and send notifications using Amazon SNS.

  Using CloudWatch Logs with `innodb_print_all_deadlocks` turned on, you can configure alarms to notify you when the number of deadlocks exceeds a given threshold. To define a threshold, we recommend that you observe your trends and use a value based on your normal workload.
- Performance Insights – When you use Performance Insights, you can monitor the `innodb_deadlocks` and `innodb_lock_wait_timeout` metrics. For more information on these metrics, see Non-native counters for Aurora MySQL.

Troubleshooting Amazon Aurora MySQL database performance

This topic focuses on some common Aurora MySQL DB performance issues, and how to troubleshoot or collect information to remediate these issues quickly. We divide database performance into two categories:

- Server performance – The entire database server runs slower.
- Query performance – One or more queries take longer to run.

AWS monitoring options

We recommend that you use the following AWS monitoring options to help with troubleshooting:

- Amazon CloudWatch – Amazon CloudWatch monitors your AWS resources and the applications you run on AWS in real time. You can use CloudWatch to collect and track metrics, which are variables you can measure for your resources and applications. For more information, see What is Amazon CloudWatch?.

  You can view all of the system metrics and process information for your DB instances on the AWS Management Console. You can configure your Aurora MySQL DB cluster to publish general, slow, audit, and error log data to a log group in Amazon CloudWatch Logs. This allows you to view trends, maintain logs if a host is impacted, and create a baseline for "normal" performance to easily identify anomalies or changes. For more information, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.
- Enhanced Monitoring – To enable additional Amazon CloudWatch metrics for an Aurora MySQL database, turn on Enhanced Monitoring. When you create or modify an Aurora DB cluster, select **Enable Enhanced Monitoring**. This allows Aurora to publish performance metrics to CloudWatch. Some of the key metrics available include CPU usage, database connections, storage usage, and query latency. These can help identify performance bottlenecks.

  The amount of information transferred for a DB instance is directly proportional to the defined granularity for Enhanced Monitoring. A smaller monitoring interval results in more frequent reporting of OS metrics and increases your monitoring cost. To manage costs, set different granularities for different instances in your AWS accounts. The default granularity at creation of an instance is 60 seconds. For more information, see Cost of Enhanced Monitoring.
- Performance Insights – You can view all of the database call metrics. This includes DB locks, waits, and the number of rows processed, all of which you can use for troubleshooting. When

you create or modify an Aurora DB cluster, select **Turn on Performance Insights**. By default, Performance Insights has a 7-day data retention period, but can be customized to analyze longer-term performance trends. For longer than 7-day retention, you need to upgrade to the paid tier. For more information, see Performance Insights pricing. You can set the data retention period for each Aurora DB instance separately. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.

Most common reasons for Aurora MySQL database performance issues

You can use the following steps to troubleshoot performance issues in your Aurora MySQL database. We list these steps in the logical order of investigation, but they're not intended to be linear. One discovery could jump across steps, which allow for a series of investigative paths.

1. Workload – Understand your database workload.
2. Logging – Review all of the database logs.
3. Query performance – Examine your query execution plans to see if they've changed. Code changes can cause plans to change.

Troubleshooting workload issues for Aurora MySQL databases

Database workload can be viewed as reads and writes. With an understanding of "normal" database workload, you can tune queries and the database server to meet demand as it changes. There are a number of different reasons why performance can change, so the first step is to understand what has changed.

- Has there been a major or minor version upgrade?

  A major version upgrade includes changes to the engine code, particularly in the optimizer, that can change the query execution plan. When upgrading database versions, especially major versions, it's very important that you analyze the database workload and tune accordingly. Tuning can involve optimizing and rewriting queries, or adding and updating parameter settings, depending on the results of testing. Understanding what is causing the impact will allow you to start focusing on that specific area.

  For more information, see What is new in MySQL 8.0 and Server and status variables and options added, deprecated, or removed in MySQL 8.0 in the MySQL documentation, and Comparison of Aurora MySQL version 2 and Aurora MySQL version 3.

- Has there been an increase in data being processed (row counts)?
- Are there more queries running concurrently?
- Are there schema or database changes?
- Have there been code defects or fixes?

Contents

Instance host metrics

Monitor instance host metrics such as CPU, memory, and network activity to help understand whether there has been a workload change. There are two main concepts for understanding workload changes:

- Utilization – The usage of a device, such as CPU or disk. It can be time-based or capacity-based.
  - Time-based – The amount of time that a resource is busy over a particular observation period.
  - Capacity-based – The amount of throughput that a system or component can deliver, as a percentage of its capacity.
- Saturation – The degree to which more work is required of a resource than it can process. When capacity-based usage reaches 100%, the extra work can't be processed and must be queued.

CPU usage

You can use the following tools to identify CPU usage and saturation:

- CloudWatch provides the `CPUUtilization` metric. If this reaches 100%, then the instance is saturated. However, CloudWatch metrics are averaged over 1 minute, and lack granularity.

  For more information on CloudWatch metrics, see Instance-level metrics for Amazon Aurora.
- Enhanced Monitoring provides metrics returned by the operating system `top` command. It shows load averages and the following CPU states, with 1-second granularity:
  - `Idle (%)` = Idle time
  - `IRQ (%)` = Software interrupts
  - `Nice (%)` = Nice time for processes with a niced priority.
  - `Steal (%)` = Time spent serving other tenants (virtualization related)
  - `System (%)` = System time
  - `User (%)` = User time
  - `Wait (%)` = I/O wait

  For more information on Enhanced Monitoring metrics, see OS metrics for Aurora.

Memory usage

If the system is under memory pressure, and resource consumption is reaching saturation, you should be observing a high degree of page scanning, paging, swapping, and out-of-memory errors.

You can use the following tools to identify memory usage and saturation:

CloudWatch provides the `FreeableMemory` metric, that shows how much memory can be reclaimed by flushing some of the OS caches and the current free memory.

For more information on CloudWatch metrics, see Instance-level metrics for Amazon Aurora.

Enhanced Monitoring provides the following metrics that can help you identify memory usage issues:

- `Buffers (KB)` – The amount of memory used for buffering I/O requests before writing to the storage device, in kilobytes.
- `Cached (KB)` – The amount of memory used for caching file system–based I/O.
- `Free (KB)` – The amount of unassigned memory, in kilobytes.
- `Swap` – Cached, Free, and Total.

For example, if you see that your DB instance uses Swap memory, then the total amount of memory for your workload is larger than your instance currently has available. We recommend increasing the size of your DB instance or tuning your workload to use less memory.

For more information on Enhanced Monitoring metrics, see OS metrics for Aurora.

For more detailed information on using the Performance Schema and sys schema to determine which connections and components are using memory, see Troubleshooting memory usage issues for Aurora MySQL databases.

Network throughput

CloudWatch provides the following metrics for total network throughput, all averaged over 1 minute:

- NetworkReceiveThroughput – The amount of network throughput received from clients by each instance in the Aurora DB cluster.
- NetworkTransmitThroughput – The amount of network throughput sent to clients by each instance in the Aurora DB cluster.
- NetworkThroughput – The amount of network throughput both received from and transmitted to clients by each instance in the Aurora DB cluster.
- StorageNetworkReceiveThroughput – The amount of network throughput received from the Aurora storage subsystem by each instance in the DB cluster.
- StorageNetworkTransmitThroughput – The amount of network throughput sent to the Aurora storage subsystem by each instance in the Aurora DB cluster.
- StorageNetworkThroughput – The amount of network throughput received from and sent to the Aurora storage subsystem by each instance in the Aurora DB cluster.

For more information on CloudWatch metrics, see Instance-level metrics for Amazon Aurora.

Enhanced Monitoring provides the network received (**RX**) and transmitted (**TX**) graphs, with up to 1-second granularity.

For more information on Enhanced Monitoring metrics, see OS metrics for Aurora.

Database metrics

Examine the following CloudWatch metrics for workload changes:

- `BlockedTransactions` – The average number of transactions in the database that are blocked per second.
- `BufferCacheHitRatio` – The percentage of requests that are served by the buffer cache.
- `CommitThroughput` – The average number of commit operations per second.
- `DatabaseConnections` – The number of client network connections to the database instance.
- `Deadlocks` – The average number of deadlocks in the database per second.
- `DMLThroughput` – The average number of inserts, updates, and deletes per second.
- `ResultSetCacheHitRatio` – The percentage of requests that are served by the query cache.
- `RollbackSegmentHistoryListLength` – The undo logs that record committed transactions with delete-marked records.
- `RowLockTime` – The total time spent acquiring row locks for InnoDB tables.
- `SelectThroughput` – The average number of select queries per second.

For more information on CloudWatch metrics, see Instance-level metrics for Amazon Aurora.

Consider the following questions when examining the workload:

1. Were there recent changes in DB instance class, for example reducing the instance size from 8xlarge to 4xlarge, or changing from db.r5 to db.r6?
2. Can you create a clone and reproduce the issue, or is it happening only on that one instance?
3. Is there server resource exhaustion, high CPU or memory exhaustion? If yes, this could mean that additional hardware is required.
4. Are one or more queries taking longer?
5. Are the changes caused by an upgrade, especially a major version upgrade? If yes, then compare the pre- and post-upgrade metrics.
6. Are there changes in the number of reader DB instances?
7. Have you enabled general, audit, or binary logging? For more information, see Logging for Aurora MySQL databases.
8. Did you enable, disable, or change your use of binary log (binlog) replication?
9. Are there any long-running transactions holding large numbers of row locks? Examine the InnoDB history list length (HLL) for indications of long-running transactions.

   For more information, see The InnoDB history list length increased significantly and the blog post Why is my SELECT query running slowly on my Amazon Aurora MySQL DB cluster?.

   a. If a large HLL is caused by a write transaction, it means that UNDO logs are accumulating (not being cleaned regularly). In a large write transaction, this accumulation can grow quickly. In MySQL, UNDO is stored in the SYSTEM tablespace. The SYSTEM tablespace is not shrinkable. The UNDO log might cause the SYSTEM tablespace to grow to several GB, or even TB. After the purge, release the allocated space by taking a logical backup (dump) of the data, then import the dump to a new DB instance.
   b. If a large HLL is caused by a read transaction (long-running query), it can mean that the query is using a large amount of temporary space. Release the temporary space by rebooting. Examine Performance Insights DB metrics for any changes in the Temp section, such as created_tmp_tables. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.

10. Can you split long-running transactions into smaller ones that modify fewer rows?
11. Are there any changes in blocked transactions or increases in deadlocks? Examine Performance Insights DB metrics for any changes in status variables in the Locks section, such as innodb_row_lock_time, innodb_row_lock_waits, and innodb_dead_locks. Use 1-minute or 5-minute intervals.
12. Are there increased wait events? Examine Performance Insights wait events and wait types using 1-minute or 5-minute intervals. Analyze the top wait events and see whether they are correlated to workload changes or database contention. For example, buf_pool mutex indicates buffer pool contention. For more information, see Tuning Aurora MySQL with wait events.

Troubleshooting memory usage issues for Aurora MySQL databases

While CloudWatch, Enhanced Monitoring, and Performance Insights provide a good overview of memory usage at the operating system level, such as how much memory the database process is using, they don't allow you to break down what connections or components within the engine might be causing this memory usage.

To troubleshoot this, you can use the Performance Schema and sys schema. In Aurora MySQL version 3, memory instrumentation is enabled by default when the Performance Schema is enabled. In Aurora MySQL version 2, only memory instrumentation for Performance Schema memory usage is enabled by default. For information on tables available in the Performance Schema to track memory usage and enabling Performance Schema memory instrumentation, see Memory summary tables in the MySQL documentation. For more information on using the Performance Schema with Performance Insights, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

While detailed information is available in the Performance Schema to track current memory usage, the MySQL sys schema has views on top of Performance Schema tables that you can use to quickly pinpoint where memory is being used.

In the `sys` schema, the following views are available to track memory usage by connection, component, and query.

View	Description
memory_by_host_by_current_bytes	Provides information on engine memory usage by host. This can be useful for identifying which application servers or client hosts are consuming memory.
memory_by_thread_by_current_bytes	Provides information on engine memory usage by thread ID. The thread ID in MySQL can be a client connection or a background thread. You can map thread IDs to MySQL connection IDs by using the sys.processlist view or performance_schema.threads table.
memory_by_user_by_current_bytes	Provides information on engine memory usage by user. This can be useful for identifying which user accounts or clients are consuming memory.
memory_global_by_current_bytes	Provides information on engine memory usage by engine component. This can be useful for identifying memory usage globally by engine buffers or components. For example, you might see the `memory/innodb/buf_buf_pool` event for the InnoDB buffer pool, or the `memory/sql/Prepared_statement::main_mem_root` event for prepared statements.
memory_global_total	Provides an overview of total tracked memory usage in the database engine.

In Aurora MySQL version 3.05 and higher, you can also track maximum memory usage by statement digest in the Performance Schema statement summary tables. The statement summary tables contain normalized statement digests and aggregated statistics on their execution. The `MAX_TOTAL_MEMORY` column can help you identify maximum memory used by query digest since the statistics were last reset, or since the database instance was restarted. This can be useful in identifying specific queries that might be consuming a lot of memory.

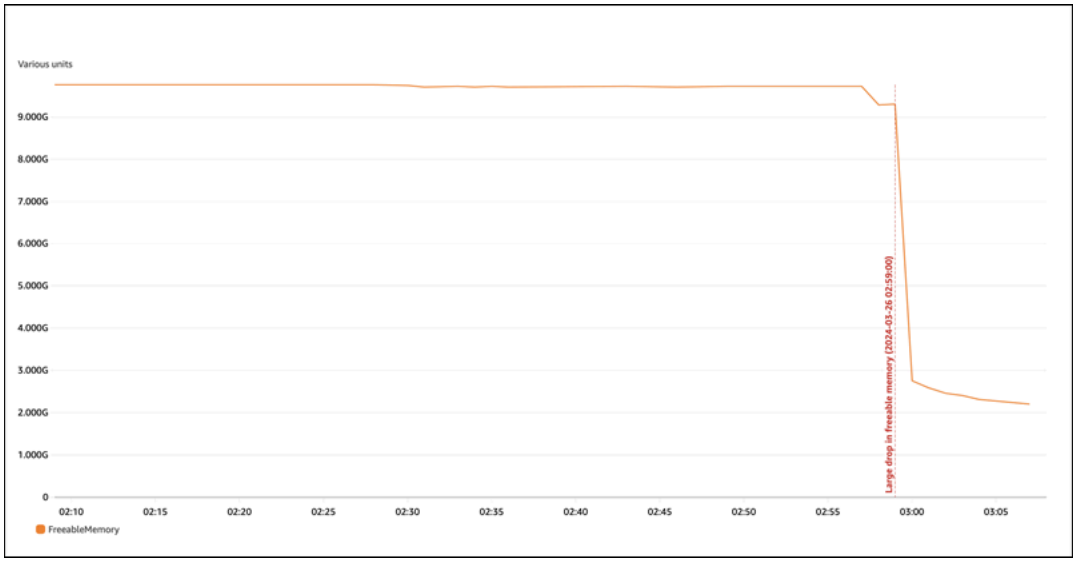
> **Note**
>
> The Performance Schema and `sys` schema show you the current memory usage on the server, and the high-water marks for memory consumed per connection and engine component. Because the Performance Schema is maintained in memory, information is reset when the DB instance restarts. To maintain a history over time, we recommend that you configure retrieval and storage of this data outside of the Performance Schema.

Topics

- Example 1: Continuous high memory usage
- Example 2: Transient memory spikes

Example 1: Continuous high memory usage

Looking globally at `FreeableMemory` in CloudWatch, we can see that memory usage greatly increased at 2024-03-26 02:59 UTC.



This doesn't tell us the whole picture. To determine which component is using the most memory, you can log into the database and look at `sys.memory_global_by_current_bytes`. This table contains a list of memory events that MySQL tracks, along with information on memory allocation per event. Each memory tracking event starts with `memory/%`, followed by other information on which engine component/feature the event is associated with.

For example, `memory/performance_schema/%` is for memory events related to the Performance Schema, `memory/innodb/%` is for InnoDB, and so on. For more information on event naming conventions, see [Performance Schema instrument naming conventions]() in the MySQL documentation.

From the following query, we can find the likely culprit based on `current_alloc`, but we can also see many `memory/performance_schema/%` events.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes LIMIT 10;

+-----------------------------------------------------------------------------
+---------------+---------------+--------------------+------------+------------
+-----------------+
| event_name                                                                    |
 current_count | current_alloc | current_avg_alloc | high_count | high_alloc |
 high_avg_alloc |
+-----------------------------------------------------------------------------
+---------------+---------------+--------------------+------------+------------
+-----------------+
| memory/sql/Prepared_statement::main_mem_root                                  |
 512817 | 4.91 GiB      | 10.04 KiB         |     512823 | 4.91 GiB   | 10.04 KiB
  |
| memory/performance_schema/prepared_statements_instances                       |
   252 | 488.25 MiB    | 1.94 MiB          |        252 | 488.25 MiB | 1.94 MiB        |
| memory/innodb/hash0hash                                                       |
     4 | 79.07 MiB     | 19.77 MiB         |          4 | 79.07 MiB  | 19.77 MiB       |
| memory/performance_schema/events_errors_summary_by_thread_by_error            |
  1028 | 52.27 MiB     | 52.06 KiB         |       1028 | 52.27 MiB  | 52.06 KiB       |
| memory/performance_schema/events_statements_summary_by_thread_by_event_name  |
     4 | 47.25 MiB     | 11.81 MiB         |          4 | 47.25 MiB  | 11.81 MiB       |
| memory/performance_schema/events_statements_summary_by_digest                 |
     1 | 40.28 MiB     | 40.28 MiB         |          1 | 40.28 MiB  | 40.28 MiB       |
| memory/performance_schema/memory_summary_by_thread_by_event_name              |
     4 | 31.64 MiB     | 7.91 MiB          |          4 | 31.64 MiB  | 7.91 MiB        |
| memory/innodb/memory                                                          |
 15227 | 27.44 MiB     | 1.85 KiB          |      20619 | 33.33 MiB  | 1.66 KiB        |
| memory/sql/String::value                                                  |
 74411 | 21.85 MiB     |  307 bytes        |      76867 | 25.54 MiB  |  348 bytes     |
| memory/sql/TABLE                                                          |
  8381 | 21.03 MiB     | 2.57 KiB          |       8381 | 21.03 MiB  | 2.57 KiB       |
+-------------------------------------------------------------------------------
+---------------+---------------+------------------+------------+------------
+----------------+
10 rows in set (0.02 sec)
```

We mentioned previously that the Performance Schema is stored in memory, which means that it's also tracked in the performance_schema memory instrumentation.

Note

If you find that the Performance Schema is using a lot of memory, and want to limit its memory usage, you can tune database parameters based on your requirements. For more information, see The Performance Schema memory-allocation model in the MySQL documentation.

For readability, you can rerun the same query but exclude Performance Schema events. The output shows the following:

- The main memory consumer is memory/sql/Prepared_statement::main_mem_root.
- The current_alloc column tells us that MySQL has 4.91 GiB currently allocated to this event.
- The high_alloc column tells us that 4.91 GiB is the high-water mark of current_alloc since the stats were last reset or since the server restarted. This means that memory/sql/Prepared_statement::main_mem_root is at its highest value.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes WHERE event_name NOT LIKE
 'memory/performance_schema/%' LIMIT 10;

+-----------------------------------------------+---------------+---------------
+-------------------+------------+------------+----------------+
| event_name                                    | current_count | current_alloc |
 current_avg_alloc | high_count | high_alloc | high_avg_alloc |
+-----------------------------------------------+---------------+---------------
+-------------------+------------+------------+----------------+
| memory/sql/Prepared_statement::main_mem_root  |        512817 | 4.91 GiB      | 10.04
 KiB         |     512823 | 4.91 GiB   | 10.04 KiB      |
| memory/innodb/hash0hash                       |             4 | 79.07 MiB     | 19.77
 MiB         |          4 | 79.07 MiB  | 19.77 MiB      |
| memory/innodb/memory                          |         17096 | 31.68 MiB     | 1.90
 KiB          |      22498 | 37.60 MiB  | 1.71 KiB       |
| memory/sql/String::value                      |        122277 | 27.94 MiB     |  239
 bytes        |     124699 | 29.47 MiB  |  247 bytes     |
| memory/sql/TABLE                              |          9927 | 24.67 MiB     | 2.55
 KiB          |       9929 | 24.68 MiB  | 2.55 KiB       |
| memory/innodb/lock0lock                       |          8888 | 19.71 MiB     | 2.27
 KiB          |       8888 | 19.71 MiB  | 2.27 KiB       |
| memory/sql/Prepared_statement::infrastructure |        257623 | 16.24 MiB     |   66
 bytes        |     257631 | 16.24 MiB  |   66 bytes     |
| memory/mysys/KEY_CACHE                        |             3 | 16.00 MiB     | 5.33
 MiB          |          3 | 16.00 MiB  | 5.33 MiB       |
| memory/innodb/sync0arr                        |             3 | 7.03 MiB      | 2.34
 MiB          |          3 | 7.03 MiB   | 2.34 MiB       |
| memory/sql/THD::main_mem_root                 |           815 | 6.56 MiB      | 8.24
 KiB          |        849 | 7.19 MiB   | 8.67 KiB       |
+-----------------------------------------------+---------------+----------------
+-------------------+------------+------------+-----------------+
10 rows in set (0.06 sec)
```

From the name of the event, we can tell that this memory is being used for prepared statements. If you want to see which connections are using this memory, you can check memory_by_thread_by_current_bytes.

In the following example, each connection has approximately 7 MiB allocated, with a high-water mark of approximately 6.29 MiB (`current_max_alloc`). This makes sense, because the example is using `sysbench` with 80 tables and 800 connections with prepared statements. If you want to reduce memory usage in this scenario, you can optimize your application's usage of prepared statements to reduce memory consumption.

```
mysql> SELECT * FROM sys.memory_by_thread_by_current_bytes;

+-----------+-------------------------------------------+--------------------
+-------------------+-------------------+-------------------+-----------------+
| thread_id | user                                      | current_count_used |
 current_allocated | current_avg_alloc | current_max_alloc | total_allocated |
+-----------+-------------------------------------------+--------------------
+-------------------+-------------------+-------------------+-----------------+
|        46 | rdsadmin@localhost                        |                405 | 8.47 MiB
          | 21.42 KiB         | 8.00 MiB          | 155.86 MiB      |
|        61 | reinvent@10.0.4.4                         |               1749 | 6.72 MiB
          | 3.93 KiB          | 6.29 MiB          | 14.24 MiB       |
|       101 | reinvent@10.0.4.4                         |               1845 | 6.71 MiB
          | 3.72 KiB          | 6.29 MiB          | 14.50 MiB       |
|        55 | reinvent@10.0.4.4                         |               1674 | 6.68 MiB
          | 4.09 KiB          | 6.29 MiB          | 14.13 MiB       |
|        57 | reinvent@10.0.4.4                         |               1416 | 6.66 MiB
          | 4.82 KiB          | 6.29 MiB          | 13.52 MiB       |
|       112 | reinvent@10.0.4.4                         |               1759 | 6.66 MiB
          | 3.88 KiB          | 6.29 MiB          | 14.17 MiB       |
|        66 | reinvent@10.0.4.4                         |               1428 | 6.64 MiB
          | 4.76 KiB          | 6.29 MiB          | 13.47 MiB       |
|        75 | reinvent@10.0.4.4                         |               1389 | 6.62 MiB
          | 4.88 KiB          | 6.29 MiB          | 13.40 MiB       |
|       116 | reinvent@10.0.4.4                         |               1333 | 6.61 MiB
          | 5.08 KiB          | 6.29 MiB          | 13.21 MiB       |
|        90 | reinvent@10.0.4.4                         |               1448 | 6.59 MiB
          | 4.66 KiB          | 6.29 MiB          | 13.58 MiB       |
|        98 | reinvent@10.0.4.4                         |               1440 | 6.57 MiB
          | 4.67 KiB          | 6.29 MiB          | 13.52 MiB       |
|        94 | reinvent@10.0.4.4                         |               1433 | 6.57 MiB
          | 4.69 KiB          | 6.29 MiB          | 13.49 MiB       |
|        62 | reinvent@10.0.4.4                         |               1323 | 6.55 MiB
          | 5.07 KiB          | 6.29 MiB          | 13.48 MiB       |
|        87 | reinvent@10.0.4.4                         |               1323 | 6.55 MiB
          | 5.07 KiB          | 6.29 MiB          | 13.25 MiB       |
|        99 | reinvent@10.0.4.4                         |               1346 | 6.54 MiB
          | 4.98 KiB          | 6.29 MiB          | 13.24 MiB       |
|       105 | reinvent@10.0.4.4                         |               1347 | 6.54 MiB
          | 4.97 KiB          | 6.29 MiB          | 13.34 MiB       |
|        73 | reinvent@10.0.4.4                         |               1335 | 6.54 MiB
          | 5.02 KiB          | 6.29 MiB          | 13.23 MiB       |
|        54 | reinvent@10.0.4.4                         |               1510 | 6.53 MiB
          | 4.43 KiB          | 6.29 MiB          | 13.49 MiB       |
.
                                                                    .
.
                                                                    .
.
                                                                    .
|       812 | reinvent@10.0.4.4                         |               1259 | 6.38 MiB
          | 5.19 KiB          | 6.29 MiB          | 13.05 MiB       |
|       214 | reinvent@10.0.4.4                         |                1279 | 6.38 MiB
          | 5.10 KiB          | 6.29 MiB          | 12.90 MiB       |
|       325 | reinvent@10.0.4.4                         |                1254 | 6.38 MiB
          | 5.21 KiB          | 6.29 MiB          | 12.99 MiB       |
|       705 | reinvent@10.0.4.4                         |                1273 | 6.37 MiB
          | 5.13 KiB          | 6.29 MiB          | 13.03 MiB       |
|       530 | reinvent@10.0.4.4                         |                1268 | 6.37 MiB
          | 5.15 KiB          | 6.29 MiB          | 12.92 MiB       |
|       307 | reinvent@10.0.4.4                         |                1263 | 6.37 MiB
          | 5.17 KiB          | 6.29 MiB          | 12.87 MiB       |
|       738 | reinvent@10.0.4.4                         |                1260 | 6.37 MiB
          | 5.18 KiB          | 6.29 MiB          | 13.00 MiB       |
|       819 | reinvent@10.0.4.4                         |                1252 | 6.37 MiB
          | 5.21 KiB          | 6.29 MiB          | 13.01 MiB       |
|        31 | innodb/srv_purge_thread                   |               17810 | 3.14 MiB
          |  184 bytes        | 2.40 MiB          | 205.69 MiB      |
|        38 | rdsadmin@localhost                        |                 599 | 1.76 MiB
          | 3.01 KiB          | 1.00 MiB          | 25.58 MiB       |
|         1 | sql/main                                  |                3756 | 1.32 MiB
          |  367 bytes        | 355.78 KiB        | 6.19 MiB        |
|       854 | rdsadmin@localhost                        |                  46 | 1.08 MiB
          | 23.98 KiB         | 1.00 MiB          | 5.10 MiB        |
|        30 | innodb/clone_gtid_thread                  |                1596 | 573.14
 KiB        |  367 bytes        | 254.91 KiB        | 970.69 KiB      |
|        40 | rdsadmin@localhost                        |                 235 | 245.19
 KiB        | 1.04 KiB          | 128.88 KiB        | 808.64 KiB      |
|       853 | rdsadmin@localhost                        |                  96 | 94.63
 KiB         | 1009 bytes        | 29.73 KiB         | 422.45 KiB      |
|        36 | rdsadmin@localhost                        |                  33 | 36.29
 KiB         | 1.10 KiB          | 16.08 KiB         | 74.15 MiB       |
|        33 | sql/event_scheduler                       |                   3 | 16.27
 KiB         | 5.42 KiB          | 16.04 KiB         | 16.27 KiB       |
|        35 | sql/compress_gtid_table                   |                   8 | 14.20
 KiB         | 1.77 KiB          | 8.05 KiB          | 18.62 KiB       |
|        25 | innodb/fts_optimize_thread                |                  12 | 1.86 KiB
          |  158 bytes        |  648 bytes        | 1.98 KiB        |
|        23 | innodb/srv_master_thread                  |                  11 | 1.23 KiB
          |  114 bytes        |  361 bytes        | 24.40 KiB       |
|        24 | innodb/dict_stats_thread                  |                  11 | 1.23 KiB
          |  114 bytes        |  361 bytes        | 1.35 KiB        |
|         5 | innodb/io_read_thread                     |                   1 |  144
 bytes        |  144 bytes        |  144 bytes        |  144 bytes      |
|         6 | innodb/io_read_thread                     |                   1 |  144
 bytes        |  144 bytes        |  144 bytes        |  144 bytes      |
|         2 | sql/aws_oscar_log_level_monitor           |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|         4 | innodb/io_ibuf_thread                     |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|         7 | innodb/io_write_thread                    |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|         8 | innodb/io_write_thread                    |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|         9 | innodb/io_write_thread                    |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        10 | innodb/io_write_thread                    |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        11 | innodb/srv_lra_thread                     |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        12 | innodb/srv_akp_thread                     |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        18 | innodb/srv_lock_timeout_thread            |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |  248 bytes      |
|        19 | innodb/srv_error_monitor_thread           |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        20 | innodb/srv_monitor_thread                 |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        21 | innodb/buf_resize_thread                  |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        22 | innodb/btr_search_sys_toggle_thread       |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        32 | innodb/dict_persist_metadata_table_thread |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
|        34 | sql/signal_handler                        |                  0 |    0
 bytes        |    0 bytes       |    0 bytes       |    0 bytes      |
+-----------+-------------------------------------------+--------------------
+-------------------+-------------------+-------------------+-----------------+
831 rows in set (2.48 sec)
```

As mentioned earlier, the thread ID (`thd_id`) value here can refer to server background threads or database connections. If you want to map thread ID values to database connection IDs, you can use the `performance_schema.threads` table or the `sys.processlist` view, where `conn_id` is the connection ID.

```
mysql> SELECT thd_id,conn_id,user,db,command,state,time,last_wait FROM sys.processlist
 WHERE user='reinvent@10.0.4.4';
```

```
+--------+---------+-------------------+----------+---------+----------------+------
+-------------------------------------------------+
| thd_id | conn_id | user              | db       | command | state          | time |
 last_wait                                       |
+--------+---------+-------------------+----------+---------+----------------+------
+-------------------------------------------------+
|    590 |     562 | reinvent@10.0.4.4 | sysbench | Execute | closing tables |    0 |
 wait/io/redo_log_flush                          |
|    578 |     550 | reinvent@10.0.4.4 | sysbench | Sleep   | NULL           |    0 |
 idle                                            |
|    579 |     551 | reinvent@10.0.4.4 | sysbench | Execute | closing tables |    0 |
 wait/io/redo_log_flush                          |
|    580 |     552 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    581 |     553 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    582 |     554 | reinvent@10.0.4.4 | sysbench | Sleep   | NULL           |    0 |
 idle                                            |
|    583 |     555 | reinvent@10.0.4.4 | sysbench | Sleep   | NULL           |    0 |
 idle                                            |
|    584 |     556 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    585 |     557 | reinvent@10.0.4.4 | sysbench | Execute | closing tables |    0 |
 wait/io/redo_log_flush                          |
|    586 |     558 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    587 |     559 | reinvent@10.0.4.4 | sysbench | Execute | closing tables |    0 |
 wait/io/redo_log_flush                          |
.
                                               .
.
                                               .
.
                                               .
|    323 |     295 | reinvent@10.0.4.4 | sysbench | Sleep   | NULL           |    0 |
 idle                                            |
|    324 |     296 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    325 |     297 | reinvent@10.0.4.4 | sysbench | Execute | closing tables |    0 |
 wait/io/redo_log_flush                          |
|    326 |     298 | reinvent@10.0.4.4 | sysbench | Execute | updating       |    0 |
 wait/io/table/sql/handler                       |
|    438 |     410 | reinvent@10.0.4.4 | sysbench | Execute | System lock    |    0 |
 wait/lock/table/sql/handler                     |
|     280 |       252 | reinvent@10.0.4.4 | sysbench | Sleep    | starting        |    0 |
 wait/io/socket/sql/client_connection              |
|      98 |        70 | reinvent@10.0.4.4 | sysbench | Query    | freeing items   |    0 |
 NULL                                              |
+---------+---------+-------------------+----------+---------+----------------+------
+--------------------------------------------------+
804 rows in set (5.51 sec)
```

Now we stop the `sysbench` workload, which closes the connections and released the memory. Checking the events again, we can confirm that memory is released, but `high_alloc` still tells us what the high-water mark is. The `high_alloc` column can be very useful in identifying short spikes in memory usage, where you might not be able to immediately identify usage from `current_alloc`, which shows only currently allocated memory.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes WHERE event_name='memory/sql/
Prepared_statement::main_mem_root' LIMIT 10;

+----------------------------------------------+---------------+---------------
+-------------------+------------+------------+----------------+
| event_name                                   | current_count | current_alloc |
 current_avg_alloc | high_count | high_alloc | high_avg_alloc |
+----------------------------------------------+---------------+---------------
+-------------------+------------+------------+----------------+
| memory/sql/Prepared_statement::main_mem_root |            17 | 253.80 KiB    | 14.93
 KiB          |     512823 | 4.91 GiB   | 10.04 KiB      |
+----------------------------------------------+---------------+---------------
+-------------------+------------+------------+----------------+
1 row in set (0.00 sec)
```

If you want to reset `high_alloc`, you can truncate the `performance_schema` memory summary tables, but this resets all memory instrumentation. For more information, see [Performance Schema general table characteristics]() in the MySQL documentation.

In the following example, we can see that `high_alloc` is reset after truncation.

```
mysql> TRUNCATE `performance_schema`.`memory_summary_global_by_event_name`;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM sys.memory_global_by_current_bytes WHERE event_name='memory/sql/
Prepared_statement::main_mem_root' LIMIT 10;
```

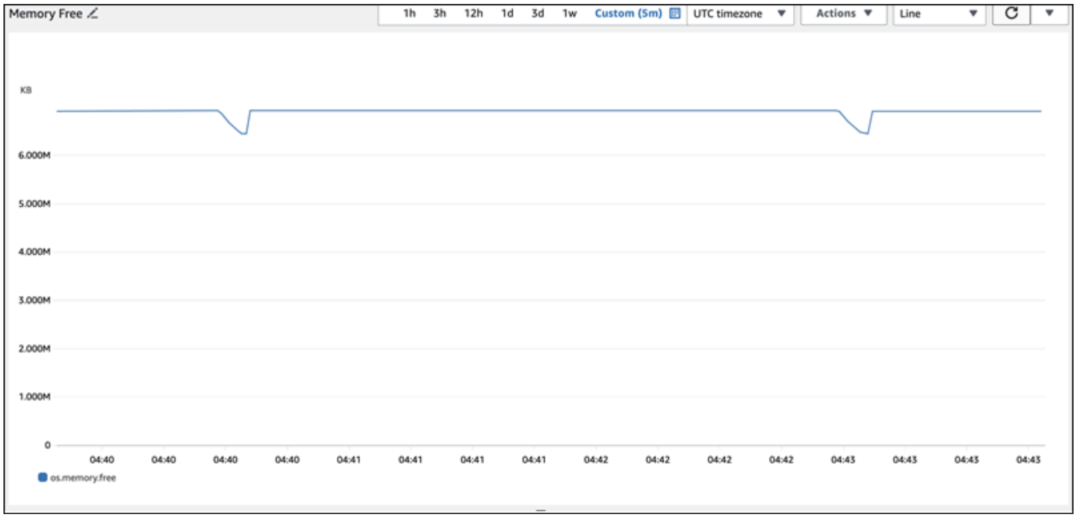
```
+-----------------------------------------------+---------------+---------------
+-----------------+-----------+-----------+---------------+
| event_name                                    | current_count | current_alloc |
 current_avg_alloc | high_count | high_alloc | high_avg_alloc |
+-----------------------------------------------+---------------+---------------
+-----------------+-----------+-----------+---------------+
| memory/sql/Prepared_statement::main_mem_root |            17 | 253.80 KiB    | 14.93
 KiB          |          17 | 253.80 KiB | 14.93 KiB      |
+-----------------------------------------------+---------------+---------------
+-----------------+-----------+-----------+---------------+
1 row in set (0.00 sec)
```

Example 2: Transient memory spikes

Another common occurrence is short spikes in memory usage on a database server. These can be periodic drops in freeable memory that are difficult to troubleshoot using `current_alloc` in `sys.memory_global_by_current_bytes`, because the memory has already been freed.

> **Note**
>
> If Performance Schema statistics have been reset, or the database instance has been restarted, this information won't be available in `sys` or `performance_schema`. To retain this information, we recommend that you configure external metrics collection.

The following graph of the `os.memory.free` metric in Enhanced Monitoring shows brief 7-second spikes in memory usage. Enhanced Monitoring allows you to monitor at intervals as short as 1 second, which is perfect for catching transient spikes like these.



To help diagnose the cause of the memory usage here, we can use a combination of `high_alloc` in the `sys` memory summary views and [Performance Schema statement summary tables](#) to try to identify offending sessions and connections.

As expected, because memory usage isn't currently high, we can't see any major offenders in the `sys` schema view under `current_alloc`.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes LIMIT 10;

+-----------------------------------------------------------------------------
+---------------+---------------+-------------------+------------+------------
+----------------+
| event_name                                                                 |
 current_count | current_alloc | current_avg_alloc | high_count | high_alloc |
 high_avg_alloc |
+-----------------------------------------------------------------------------
+---------------+---------------+-------------------+------------+------------
+----------------+
| memory/innodb/hash0hash                                                    |
     4 | 79.07 MiB     | 19.77 MiB         |          4 | 79.07 MiB  | 19.77 MiB      |
| memory/innodb/os0event                                                     |
 439372 | 60.34 MiB     |  144 bytes        |     439372 | 60.34 MiB  |  144 bytes
 |
| memory/performance_schema/events_statements_summary_by_digest              |
     1 | 40.28 MiB     | 40.28 MiB         |          1 | 40.28 MiB  | 40.28 MiB      |
| memory/mysys/KEY_CACHE                                                     |
     3 | 16.00 MiB     | 5.33 MiB          |          3 | 16.00 MiB  | 5.33 MiB       |
| memory/performance_schema/events_statements_history_long                   |
     1 | 14.34 MiB     | 14.34 MiB         |          1 | 14.34 MiB  | 14.34 MiB      |
| memory/performance_schema/events_errors_summary_by_thread_by_error         |
   257 | 13.07 MiB     | 52.06 KiB         |        257 | 13.07 MiB  | 52.06 KiB      |
| memory/performance_schema/events_statements_summary_by_thread_by_event_name |
     1 | 11.81 MiB     | 11.81 MiB         |          1 | 11.81 MiB  | 11.81 MiB      |
| memory/performance_schema/events_statements_summary_by_digest.digest_text   |
     1 | 9.77 MiB      | 9.77 MiB          |          1 | 9.77 MiB   | 9.77 MiB       |
| memory/performance_schema/events_statements_history_long.digest_text       |
     1 | 9.77 MiB      | 9.77 MiB          |          1 | 9.77 MiB   | 9.77 MiB       |
| memory/performance_schema/events_statements_history_long.sql_text          |
     1 | 9.77 MiB      | 9.77 MiB          |          1 | 9.77 MiB   | 9.77 MiB       |
+-----------------------------------------------------------------------------
+---------------+---------------+-------------------+------------+------------
+----------------+
10 rows in set (0.01 sec)
```

Expanding the view to order by high_alloc, we can now see that the memory/temptable/physical_ram component is a very good candidate here. At its highest, it consumed 515.00 MiB.

As its name suggests, memory/temptable/physical_ram instruments memory usage for the TEMP storage engine in MySQL, which was introduced in MySQL 8.0. For more information on how MySQL uses temporary tables, see Internal temporary table use in MySQL in the MySQL documentation.

Note

We're using the sys.x$memory_global_by_current_bytes view in this example.

```
mysql> SELECT event_name, format_bytes(current_alloc) AS "currently allocated",
 sys.format_bytes(high_alloc) AS "high-water mark"
FROM sys.x$memory_global_by_current_bytes ORDER BY high_alloc DESC LIMIT 10;

+-----------------------------------------------------------------------------
+---------------------+-----------------+
| event_name                                                                  |
 currently allocated | high-water mark |
+-----------------------------------------------------------------------------
+---------------------+-----------------+
| memory/temptable/physical_ram                                               | 4.00
 MiB            | 515.00 MiB      |
| memory/innodb/hash0hash                                                     | 79.07
 MiB           | 79.07 MiB       |
| memory/innodb/os0event                                                      | 63.95
 MiB           | 63.95 MiB       |
| memory/performance_schema/events_statements_summary_by_digest               | 40.28
 MiB           | 40.28 MiB       |
| memory/mysys/KEY_CACHE                                                      | 16.00
 MiB           | 16.00 MiB       |
| memory/performance_schema/events_statements_history_long                    | 14.34
 MiB           | 14.34 MiB       |
| memory/performance_schema/events_errors_summary_by_thread_by_error          | 13.07
 MiB           | 13.07 MiB       |
| memory/performance_schema/events_statements_summary_by_thread_by_event_name | 11.81
 MiB           | 11.81 MiB       |
| memory/performance_schema/events_statements_summary_by_digest.digest_text   | 9.77
 MiB            | 9.77 MiB        |
| memory/performance_schema/events_statements_history_long.sql_text          | 9.77
 MiB            | 9.77 MiB        |
+-----------------------------------------------------------------------------
+---------------------+-----------------+
10 rows in set (0.00 sec)
```

In Example 1: Continuous high memory usage, we checked the current memory usage for each connection to determine which connection is responsible for using the memory in question. In this example, the memory is already freed, so checking the memory usage for current connections isn't useful.

To dig deeper and find the offending statements, users, and hosts, we use the Performance Schema. The Performance Schema contains multiple statement summary tables that are sliced by different dimensions such as event name, statement digest, host, thread, and user. Each view will allow you dig deeper into where certain statements are being run and what they are doing. This section is focused on MAX_TOTAL_MEMORY, but you can find more information on all of the columns available in the Performance Schema statement summary tables documentation.

```
mysql> SHOW TABLES IN performance_schema LIKE 'events_statements_summary_%';

+------------------------------------------------------------+
| Tables_in_performance_schema (events_statements_summary_%) |
+------------------------------------------------------------+
| events_statements_summary_by_account_by_event_name         |
| events_statements_summary_by_digest                        |
| events_statements_summary_by_host_by_event_name            |
| events_statements_summary_by_program                       |
| events_statements_summary_by_thread_by_event_name          |
| events_statements_summary_by_user_by_event_name            |
| events_statements_summary_global_by_event_name             |
+------------------------------------------------------------+
7 rows in set (0.00 sec)
```

First we check events_statements_summary_by_digest to see MAX_TOTAL_MEMORY.

From this we can see the following:

- The query with digest 20676ce4a690592ff05debcffcbc26faeb76f22005e7628364d7a498769d0c4a seems to be a good candidate for this memory usage. The MAX_TOTAL_MEMORY is 537450710, which matches the high-water mark we saw for the `memory/temptable/physical_ram` event in `sys.x$memory_global_by_current_bytes`.

- It has been run four times (`COUNT_STAR`), first at 2024-03-26 04:08:34.943256, and last at 2024-03-26 04:43:06.998310.

```
mysql> SELECT SCHEMA_NAME,DIGEST,COUNT_STAR,MAX_TOTAL_MEMORY,FIRST_SEEN,LAST_SEEN
FROM performance_schema.events_statements_summary_by_digest ORDER BY MAX_TOTAL_MEMORY
 DESC LIMIT 5;

+-------------+-------------------------------------------------------------------
+------------+------------------+----------------------------
+----------------------------+
| SCHEMA_NAME | DIGEST                                                           |
 COUNT_STAR | MAX_TOTAL_MEMORY | FIRST_SEEN                 | LAST_SEEN
  |
+-------------+-------------------------------------------------------------------
+------------+------------------+----------------------------
+----------------------------+
| sysbench    | 20676ce4a690592ff05debcffcbc26faeb76f22005e7628364d7a498769d0c4a |
     4 |        537450710 | 2024-03-26 04:08:34.943256 | 2024-03-26 04:43:06.998310 |
| NULL        | f158282ea0313fefd0a4778f6e9b92fc7d1e839af59ebd8c5eea35e12732c45d |
     4 |          3636413 | 2024-03-26 04:29:32.712348 | 2024-03-26 04:36:26.269329 |
| NULL        | 0046bc5f642c586b8a9afd6ce1ab70612dc5b1fd2408fa8677f370c1b0ca3213 |
     2 |          3459965 | 2024-03-26 04:31:37.674008 | 2024-03-26 04:32:09.410718 |
| NULL        | 8924f01bba3c55324701716c7b50071a60b9ceaf17108c71fd064c20c4ab14db |
     1 |          3290981 | 2024-03-26 04:31:49.751506 | 2024-03-26 04:31:49.751506 |
| NULL        | 90142bbcb50a744fcec03a1aa336b2169761597ea06d85c7f6ab03b5a4e1d841 |
     1 |          3131729 | 2024-03-26 04:15:09.719557 | 2024-03-26 04:15:09.719557 |
+-------------+-------------------------------------------------------------------
+------------+------------------+----------------------------
+----------------------------+
5 rows in set (0.00 sec)
```

Now that we know the offending digest, we can get more details such as the query text, the user who ran it, and where it was run. Based on the digest text returned, we can see that this is a common table expression (CTE) that creates four temporary tables and performs four table scans, which is very inefficient.

```
mysql> SELECT
 SCHEMA_NAME,DIGEST_TEXT,QUERY_SAMPLE_TEXT,MAX_TOTAL_MEMORY,SUM_ROWS_SENT,SUM_ROWS_EXAMINED,SUM
FROM performance_schema.events_statements_summary_by_digest
```

```
WHERE DIGEST='20676ce4a690592ff05debcffcbc26faeb76f22005e7628364d7a498769d0c4a'\G;

*************************** 1. row ***************************
           SCHEMA_NAME: sysbench
           DIGEST_TEXT: WITH RECURSIVE `cte` ( `n` ) AS ( SELECT ? FROM `sbtest1` UNION
 ALL SELECT `id` + ? FROM `sbtest1` ) SELECT * FROM `cte`
     QUERY_SAMPLE_TEXT: WITH RECURSIVE cte (n) AS (   SELECT 1  from sbtest1 UNION ALL
  SELECT id + 1 FROM sbtest1) SELECT * FROM cte
      MAX_TOTAL_MEMORY: 537450710
         SUM_ROWS_SENT: 80000000
     SUM_ROWS_EXAMINED: 80000000
SUM_CREATED_TMP_TABLES: 4
     SUM_NO_INDEX_USED: 4
1 row in set (0.01 sec)
```

For more information on the `events_statements_summary_by_digest` table and other Performance Schema statement summary tables, see Statement summary tables in the MySQL documentation.

You can also run an EXPLAIN or EXPLAIN ANALYZE statement to see more details.

Note

EXPLAIN ANALYZE can provide more information than EXPLAIN, but it also runs the query, so be careful.

```
-- EXPLAIN
mysql> EXPLAIN WITH RECURSIVE cte (n) AS (SELECT 1  FROM sbtest1 UNION ALL SELECT id +
 1 FROM sbtest1) SELECT * FROM cte;

+----+-------------+------------+------------+-------+---------------+------+---------
+------+----------+----------+-------------+
| id | select_type | table      | partitions | type  | possible_keys | key  | key_len |
 ref  | rows     | filtered | Extra       |
+----+-------------+------------+------------+-------+---------------+------+---------
+------+----------+----------+-------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL          | NULL | NULL    |
 NULL | 19221520 |   100.00 | NULL        |
|  2 | DERIVED     | sbtest1    | NULL       | index | NULL          | k_1  | 4       |
 NULL |  9610760 |   100.00 | Using index |
|  3 | UNION        | sbtest1    | NULL       | index | NULL           | k_1  | 4        |
 NULL |  9610760 |   100.00 | Using index |
+----+-------------+------------+------------+-------+---------------+------+---------
+------+---------+----------+-------------+
3 rows in set, 1 warning (0.00 sec)

-- EXPLAIN format=tree
mysql> EXPLAIN format=tree WITH RECURSIVE cte (n) AS (SELECT 1 FROM sbtest1 UNION ALL
 SELECT id + 1 FROM sbtest1) SELECT * FROM cte\G;

*************************** 1. row ***************************
EXPLAIN: -> Table scan on cte  (cost=4.11e+6..4.35e+6 rows=19.2e+6)
    -> Materialize union CTE cte  (cost=4.11e+6..4.11e+6 rows=19.2e+6)
        -> Index scan on sbtest1 using k_1  (cost=1.09e+6 rows=9.61e+6)
        -> Index scan on sbtest1 using k_1  (cost=1.09e+6 rows=9.61e+6)
1 row in set (0.00 sec)

-- EXPLAIN ANALYZE
mysql> EXPLAIN ANALYZE WITH RECURSIVE cte (n) AS (SELECT 1 from sbtest1 UNION ALL
 SELECT id + 1 FROM sbtest1) SELECT * FROM cte\G;

*************************** 1. row ***************************
EXPLAIN: -> Table scan on cte  (cost=4.11e+6..4.35e+6 rows=19.2e+6) (actual
 time=6666..9201 rows=20e+6 loops=1)
    -> Materialize union CTE cte  (cost=4.11e+6..4.11e+6 rows=19.2e+6) (actual
 time=6666..6666 rows=20e+6 loops=1)
        -> Covering index scan on sbtest1 using k_1  (cost=1.09e+6 rows=9.61e+6)
 (actual time=0.0365..2006 rows=10e+6 loops=1)
        -> Covering index scan on sbtest1 using k_1  (cost=1.09e+6 rows=9.61e+6)
 (actual time=0.0311..2494 rows=10e+6 loops=1)
1 row in set (10.53 sec)
```

But who ran it? We can see in the Performance Schema that the `destructive_operator` user had `MAX_TOTAL_MEMORY` of 537450710, which again matches the previous results.

Note

The Performance Schema is stored in memory, so should not be relied upon as the sole source for auditing. If you need to maintain a history of statements run, and from which users, we recommend that enable audit logging. If you also need to maintain information on memory usage, we recommend that you configure monitoring to export and store these values.

```
mysql> SELECT USER,EVENT_NAME,COUNT_STAR,MAX_TOTAL_MEMORY FROM
 performance_schema.events_statements_summary_by_user_by_event_name
ORDER BY MAX_CONTROLLED_MEMORY DESC LIMIT 5;

+----------------------+---------------------------+------------+------------------+
| USER                 | EVENT_NAME                | COUNT_STAR | MAX_TOTAL_MEMORY |
+----------------------+---------------------------+------------+------------------+
| destructive_operator | statement/sql/select      |          4 |        537450710 |
| rdsadmin             | statement/sql/select      |       4172 |          3290981 |
| rdsadmin             | statement/sql/show_tables |          2 |          3615821 |
| rdsadmin             | statement/sql/show_fields |          2 |          3459965 |
| rdsadmin             | statement/sql/show_status |         75 |          1914976 |
+----------------------+---------------------------+------------+------------------+
5 rows in set (0.00 sec)

mysql> SELECT HOST,EVENT_NAME,COUNT_STAR,MAX_TOTAL_MEMORY FROM
 performance_schema.events_statements_summary_by_host_by_event_name
WHERE HOST != 'localhost' AND COUNT_STAR>0 ORDER BY MAX_CONTROLLED_MEMORY DESC LIMIT 5;

+------------+----------------------+------------+------------------+
| HOST       | EVENT_NAME           | COUNT_STAR | MAX_TOTAL_MEMORY |
+------------+----------------------+------------+------------------+
| 10.0.8.231 | statement/sql/select |          4 |        537450710 |
+------------+----------------------+------------+------------------+
1 row in set (0.00 sec)
```

Troubleshooting out-of-memory issues for Aurora MySQL databases

The Aurora MySQL `aurora_oom_response` instance-level parameter can enable the DB instance to monitor the system memory and estimate the memory consumed by various statements and connections. If the system runs low on memory, it can perform a list of actions to attempt to release that memory. It does so in an attempt to avoid a database restart due to out-of-memory (OOM) issues. The instance-level parameter takes a string of comma-separated actions that a DB instance performs when its memory is low. The `aurora_oom_response` parameter is supported for Aurora MySQL versions 2 and 3.

The following values, and combinations of them, can be used for the `aurora_oom_response` parameter. An empty string means that no action is taken, and effectively turns off the feature, leaving the database prone to OOM restarts.

- `decline` – Declines new queries when the DB instance is low on memory.
- `kill_connect` – Closes database connections that are consuming a large amount of memory, and ends current transactions and Data Definition Language (DDL) statements. This response isn't supported for Aurora MySQL version 2.

  For more information, see KILL statement in the MySQL documentation.
- `kill_query` – Ends queries in descending order of memory consumption until the instance memory surfaces above the low threshold. DDL statements aren't ended.

  For more information, see KILL statement in the MySQL documentation.
- `print` – Only prints the queries that are consuming a large amount of memory.
- `tune` – Tunes the internal table caches to release some memory back to the system. Aurora MySQL decreases the memory used for caches such as `table_open_cache` and `table_definition_cache` in low-memory conditions. Eventually, Aurora MySQL sets their memory usage back to normal when the system is no longer low on memory.

  For more information, see table_open_cache and table_definition_cache in the MySQL documentation.
- `tune_buffer_pool` – Decreases the size of the buffer pool to release some memory and make it available for the database server to process connections. This response is supported for Aurora MySQL version 3.06 and higher.

  You must pair `tune_buffer_pool` with either `kill_query` or `kill_connect` in the `aurora_oom_response` parameter value. If not, buffer pool resizing won't happen, even when you include `tune_buffer_pool` in the parameter value.

In Aurora MySQL versions lower than 3.06, for DB instance classes with memory less than or equal to 4 GiB, when the instance is under memory pressure, the default actions include `print`, `tune`, `decline`, and `kill_query`. For DB instance classes with memory greater than 4 GiB, the parameter value is empty by default (disabled).

In Aurora MySQL version 3.06 and higher, for DB instance classes with memory less than or equal to 4 GiB, Aurora MySQL also closes the top memory-consuming connections (`kill_connect`). For DB instance classes with memory greater than 4 GiB, the default parameter value is `print`.

If you frequently run into out-of-memory issues, memory usage can be monitored using memory summary tables when `performance_schema` is enabled.

Logging for Aurora MySQL databases

Aurora MySQL logs provide essential information about database activity and errors. By enabling these logs, you can identify and troubleshoot issues, understand database performance, and audit database activity. We recommend that you enable these logs for all of your Aurora MySQL DB instances to ensure optimal performance and availability of the databases. The following types of logging can be enabled. Each log contains specific information that can lead to uncovering impacts to database processing.

- Error – Aurora MySQL writes to the error log only on startup, shutdown, and when it encounters errors. A DB instance can go hours or days without new entries being written to the error log. If you see no recent entries, it's because the server didn't encounter an error that would result in a log entry. Error logging is enabled by default. For more information, see Aurora MySQL error logs.
- General – The general log provides detailed information about database activity, including all SQL statements executed by the database engine. For more information on enabling general logging and setting logging parameters, see Aurora MySQL slow query and general logs, and The general query log in the MySQL documentation.

  > **Note**
  >
  > General logs can grow to be very large and consume your storage. For more information, see Log rotation and retention for Aurora MySQL.

- Slow query – The slow query log consists of SQL statements that take more than long_query_time seconds to run and require at least min_examined_row_limit rows to be examined. You can use the slow query log to find queries that take a long time to run and are therefore candidates for optimization.

  The default value for `long_query_time` is 10 seconds. We recommend that you start with a high value to identify the slowest queries, then work your way down for fine tuning.

You can also use related parameters, such as `log_slow_admin_statements` and `log_queries_not_using_indexes`. Compare `rows_examined` with `rows_returned`. If `rows_examined` is much greater than `rows_returned`, then those queries can potentially be blocking.

In Aurora MySQL version 3, you can enable `log_slow_extra` to obtain more details. For more information, see Slow query log contents in the MySQL documentation. You can also modify `long_query_time` at the session level for debugging query execution interactively, which is especially useful if `log_slow_extra` is enabled globally.

For more information on enabling slow query logging and setting logging parameters, see Aurora MySQL slow query and general logs, and The slow query log in the MySQL documentation.

- Audit – The audit log monitors and logs database activity. Audit logging for Aurora MySQL is called Advanced Auditing. To enable Advanced Auditing, you set certain DB cluster parameters. For more information, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.
- Binary – The binary log (binlog) contains events that describe database changes, such as table creation operations and changes to table data. It also contains events for statements that potentially could have made changes (for example, a DELETE that matched no rows), unless row-based logging is used. The binary log also contains information about how long each statement took that updated data.

  Running a server with binary logging enabled makes performance slightly slower. However, the benefits of the binary log in enabling you to set up replication and for restore operations generally outweigh this minor performance decrease.

  > **Note**
  >
  > Aurora MySQL doesn't require binary logging for restore operations.

  For more information on enabling binary logging and setting the binlog format, see Configuring Aurora MySQL binary logging, and The binary log in the MySQL documentation.

You can publish the error, general, slow, query, and audit logs to Amazon CloudWatch Logs. For more information, see Publishing database logs to Amazon CloudWatch Logs.

Another useful tool for summarizing slow, general, and binary log files is pt-query-digest.

Troubleshooting query performance for Aurora MySQL databases

MySQL provides query optimizer control through system variables that affect how query plans are evaluated, switchable optimizations, optimizer and index hints, and the optimizer cost model. These data points can be helpful not only while comparing different MySQL environments, but also to compare previous query execution plans with current execution plans, and to understand the overall execution of a MySQL query at any point.

Query performance depends on many factors, including the execution plan, table schema and size, statistics, resources, indexes, and parameter configuration. Query tuning requires identifying bottlenecks and optimizing the execution path.

- Find the execution plan for the query and check whether the query is using appropriate indexes. You can optimize your query by using `EXPLAIN` and reviewing the details of each plan.
- Aurora MySQL version 3 (compatible with MySQL 8.0 Community Edition) uses an `EXPLAIN ANALYZE` statement. The `EXPLAIN ANALYZE` statement is a profiling tool that shows where MySQL spends time on your query and why. With `EXPLAIN ANALYZE`, Aurora MySQL plans, prepares, and runs the query while counting rows and measuring the time spent at various points of the execution plan. When the query completes, `EXPLAIN ANALYZE` prints the plan and its measurements instead of the query result.
- Keep your schema statistics updated by using the `ANALYZE` statement. The query optimizer can sometimes choose poor execution plans because of outdated statistics. This can lead to poor performance of a query because of inaccurate cardinality estimates of both tables and indexes. The `last_update` column of the innodb_table_stats table shows the last time your schema statistics were updated, which is a good indicator of "staleness."
- Other issues can occur, such as distribution skew of data, that aren't taken into account for table cardinality. For more information, see Estimating ANALYZE TABLE complexity for InnoDB tables and Histogram statistics in MySQL in the MySQL documentation.

Understanding the time spent by queries

The following are ways to determine the time spent by queries:

- Profiling
- Performance Schema

- Query optimizer

Profiling

By default, profiling is disabled. Enable profiling, then run the slow query and review its profile.

```
SET profiling = 1;
Run your query.
SHOW PROFILE;
```

1. Identify the stage where the most time is spent. According to General thread states in the MySQL documentation, reading and processing rows for a SELECT statement is often the longest-running state over the lifetime of a given query. You can use the EXPLAIN statement to understand how MySQL runs this query.
2. Review the slow query log to evaluate rows_examined and rows_sent to make sure that the workload is similar in each environment. For more information, see Logging for Aurora MySQL databases.
3. Run the following command for tables that are part of the identified query:

   ```
   SHOW TABLE STATUS\G;
   ```

4. Capture the following outputs before and after running the query on each environment:

   ```
   SHOW GLOBAL STATUS;
   ```

5. Run the following commands on each environment to see if there are any other query/session influencing the performance of this sample query.

   ```
   SHOW FULL PROCESSLIST;

   SHOW ENGINE INNODB STATUS\G;
   ```

   Sometimes, when resources on the server are busy, it impacts every other operation on the server, including queries. You can also capture information periodically when queries are run or set up a cron job to capture information at useful intervals.

Performance Schema

The Performance Schema provides useful information about server runtime performance, while having minimal impact on that performance. This is different from the `information_schema`, which provides schema information about the DB instance. For more information, see Turning on the Performance Schema for Performance Insights on Aurora MySQL.

Query optimizer trace

To understand why a particular query plan was chosen for execution, you can set up `optimizer_trace` to access the MySQL query optimizer.

Run an optimizer trace to show extensive information on all the paths available to the optimizer and its choice.

```
SET SESSION OPTIMIZER_TRACE="enabled=on";
SET optimizer_trace_offset=-5, optimizer_trace_limit=5;

-- Run your query.
SELECT * FROM table WHERE x = 1 AND y = 'A';

-- After the query completes:
SELECT * FROM information_schema.OPTIMIZER_TRACE;
SET SESSION OPTIMIZER_TRACE="enabled=off";
```

Reviewing query optimizer settings

Aurora MySQL version 3 (compatible with MySQL 8.0 Community Edition) has many optimizer-related changes compared with Aurora MySQL version 2 (compatible with MySQL 5.7 Community Edition). If you have some custom values for the `optimizer_switch`, we recommend that you review the differences in the defaults and set `optimizer_switch` values that work best for your workload. We also recommend that you test the options available for Aurora MySQL version 3 to examine how your queries perform.

> **Note**
>
> Aurora MySQL version 3 uses the community default value of 20 for the innodb_stats_persistent_sample_pages parameter.

You can use the following command to show the optimizer_switch values:

```
SELECT @@optimizer_switch\G;
```

The following table shows the default optimizer_switch values for Aurora MySQL versions 2 and 3.

Setting	Aurora MySQL version 2	Aurora MySQL version 3
batched_key_access	off	off
block_nested_loop	on	on
condition_fanout_filter	on	on
derived_condition_pushdown	–	on
derived_merge	on	on
duplicateweedout	on	on
engine_condition_pushdown	on	on
firstmatch	on	on
hash_join	off	on
hash_join_cost_based	on	–
hypergraph_optimizer	–	off
index_condition_pushdown	on	on
index_merge	on	on
index_merge_intersection	on	on
index_merge_sort_union	on	on
index_merge_union	on	on

Setting	Aurora MySQL version 2	Aurora MySQL version 3
loosescan	on	on
materialization	on	on
mrr	on	on
mrr_cost_based	on	on
prefer_ordering_index	on	on
semijoin	on	on
skip_scan	–	on
subquery_materialization_cost_based	on	on
subquery_to_derived	–	off
use_index_extensions	on	on
use_invisible_indexes	–	off

For more information, see Switchable optimizations (MySQL 5.7) and Switchable optimizations (MySQL 8.0) in the MySQL documentation.

Amazon Aurora MySQL reference

This reference includes information about Aurora MySQL parameters, status variables, and general SQL extensions or differences from the community MySQL database engine.

Topics

Aurora MySQL configuration parameters

You manage your Amazon Aurora MySQL DB cluster in the same way that you manage other Amazon RDS DB instances, by using parameters in a DB parameter group. Amazon Aurora differs from other DB engines in that you have a DB cluster that contains multiple DB instances. As a result, some of the parameters that you use to manage your Aurora MySQL DB cluster apply to the entire cluster. Other parameters apply only to a particular DB instance in the DB cluster.

To manage cluster-level parameters, use DB cluster parameter groups. To manage instance-level parameters, use DB parameter groups. Each DB instance in an Aurora MySQL DB cluster is compatible with the MySQL database engine. However, you apply some of the MySQL database engine parameters at the cluster level, and you manage these parameters using DB cluster parameter groups. You can't find cluster-level parameters in the DB parameter group for an instance in an Aurora DB cluster. A list of cluster-level parameters appears later in this topic.

You can manage both cluster-level and instance-level parameters using the AWS Management Console, the AWS CLI, or the Amazon RDS API. You use separate commands for managing cluster-level parameters and instance-level parameters. For example, you can use the modify-db-cluster-parameter-group CLI command to manage cluster-level parameters in a DB cluster parameter group. You can use the modify-db-parameter-group CLI command to manage instance-level parameters in a DB parameter group for a DB instance in a DB cluster.

You can view both cluster-level and instance-level parameters in the console, or by using the CLI or RDS API. For example, you can use the describe-db-cluster-parameters AWS CLI command to view cluster-level parameters in a DB cluster parameter group. You can use the describe-db-parameters CLI command to view instance-level parameters in a DB parameter group for a DB instance in a DB cluster.

> **Note**
>
> Each default parameter group contains the default values for all parameters in the parameter group. If the parameter has "engine default" for this value, see the version-specific MySQL or PostgreSQL documentation for the actual default value.
> Unless otherwise noted, parameters listed in the following tables are valid for Aurora MySQL versions 2 and 3.

For more information about DB parameter groups, see Working with parameter groups. For rules and restrictions for Aurora Serverless v1 clusters, see Parameter groups for Aurora Serverless v1.

Topics

Cluster-level parameters

The following table shows all of the parameters that apply to the entire Aurora MySQL DB cluster.

Parameter name	Modifiable	Notes
`aurora_binlog_read_buffer_size`	Yes	Only affects clusters that use binary log (binlog) replication. For information about binlog replication, see Replication between Aurora and MySQL or between Aurora and another Aurora

Parameter name	Modifiable	Notes
		DB cluster (binary log replication). Removed from Aurora MySQL version 3.
`aurora_binlog_replication_max_yield_seconds`	Yes	Only affects clusters that use binary log (binlog) replication. For information about binlog replication, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).
`aurora_binlog_replication_sec_index_parallel_workers`	Yes	Sets the total number of parallel threads available to apply secondary index changes when replicating transactions for large tables with more than one secondary index. The parameter is set to `0` (disabled) by default. This parameter is available in Aurora MySQL version 306 and higher. For more information, see Optimizing binary log replication.
`aurora_binlog_use_large_read_buffer`	Yes	Only affects clusters that use binary log (binlog) replication. For information about binlog replication, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication). Removed from Aurora MySQL version 3.

Parameter name	Modifiable	Notes
aurora_disable_hash_join	Yes	Set this parameter to ON to turn off hash join optimization in Aurora MySQL version 2.09 or higher. It isn't supported for version 3. For more information, see Working with parallel query for Amazon Aurora MySQL.
aurora_enable_replica_log_compression	Yes	For more information, see Performance considerations for Amazon Aurora MySQL replication. Doesn't apply to clusters that are part of an Aurora global database. Removed from Aurora MySQL version 3.
aurora_enable_repl_bin_log_filtering	Yes	For more information, see Performance considerations for Amazon Aurora MySQL replication. Doesn't apply to clusters that are part of an Aurora global database. Removed from Aurora MySQL version 3.
aurora_enable_staggered_replica_restart	Yes	This setting is available in Aurora MySQL version 3, but it isn't used.
aurora_enable_zdr	Yes	This setting is turned on by default in Aurora MySQL 2.10 and higher. For more information, see Zero-downtime restart (ZDR) for Amazon Aurora MySQL.
aurora_enhanced_binlog	Yes	Set the value of this parameter to 1 to turn on the enhanced binlog in Aurora MySQL version 3.03.1 and higher. For more information, see Setting up enhanced binlog.

Parameter name	Modifiable	Notes
aurora_jemalloc_background_thread	Yes	Use this parameter to enable a background thread to perform memory maintenance operations. The allowed values are 0 (disabled) and 1 (enabled). The default value is 0. This parameter applies to Aurora MySQL version 3.05 and higher.
aurora_jemalloc_dirty_decay_ms	Yes	Use this parameter to retain freed memory for a certain amount of time (in milliseconds). Retaining memory allows for faster reuse. The allowed values are 0–18446744073709551615 . The default value (0) returns all memory to the operating system as freeable memory. This parameter applies to Aurora MySQL version 3.05 and higher.
aurora_jemalloc_tcache_enabled	Yes	Use this parameter to serve small memory requests (up to 32 KiB) in a thread local cache, bypassing the memory arenas. The allowed values are 0 (disabled) and 1 (enabled). The default value is 1. This parameter applies to Aurora MySQL version 3.05 and higher.

Parameter name	Modifiable	Notes
`aurora_load_from_s3_role`	Yes	For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket. Currently not available in Aurora MySQL version 3. Use `aws_default_s3_role` .
`aurora_mask_password_hashes_type`	Yes	This setting is turned on by default in Aurora MySQL 2.11 and higher. Use this setting to mask Aurora MySQL password hashes in the slow query and audit logs. The allowed values are `0` and 1 (default). When set to 1, passwords are logged as `<secret>`. When set to `0`, passwords are logged as hash (#) values.
`aurora_select_into_s3_role`	Yes	For more information, see Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket. Currently not available in Aurora MySQL version 3. Use `aws_default_s3_role` .
`authentication_kerberos_caseins_cmp`	Yes	Controls case-insensitive username comparison for the `authentication_kerberos` plugin. Set it to `true` for case-insensitive comparison. By default, case-sensitive comparison is used (`false`). For more information, see Using Kerberos authentication for Aurora MySQL. This parameter is available in Aurora MySQL version 3.03 and higher.

Parameter name	Modifiable	Notes
`auto_increment_increment`	Yes	
`auto_increment_offset`	Yes	
`aws_default_lambda_role`	Yes	For more information, see Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.
`aws_default_s3_role`	Yes	Used when invoking the `LOAD DATA FROM S3`, `LOAD XML FROM S3`, or `SELECT INTO OUTFILE S3` statement from your DB cluster. In Aurora MySQL version 2, the IAM role specified in this parameter is used if an IAM role isn't specified for `aurora_load_from_s3_role` or `aurora_select_into_s3_role` for the appropriate statement. In Aurora MySQL version 3, the IAM role specified for this parameter is always used. For more information, see Associating an IAM role with an Amazon Aurora MySQL DB cluster.
`binlog_backup`	Yes	Set the value of this parameter to 0 to turn on the enhanced binlog in Aurora MySQL version 3.03.1 and higher. You can turn off this parameter only when you use enhanced binlog. For more information, see Setting up enhanced binlog.

Parameter name	Modifiable	Notes
`binlog_checksum`	Yes	The AWS CLI and RDS API report a value of `None` if this parameter isn't set. In that case, Aurora MySQL uses the engine default value, which is `CRC32`. This is different from the explicit setting of `NONE`, which turns off the checksum.
`binlog-do-db`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_format`	Yes	For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).
`binlog_group_commit_sync_delay`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_group_commit_sync_no_delay_count`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog-ignore-db`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_replication_globaldb`	Yes	Set the value of this parameter to 0 to turn on the enhanced binlog in Aurora MySQL version 3.03.1 and higher. You can turn off this parameter only when you use enhanced binlog. For more information, see Setting up enhanced binlog.
`binlog_row_image`	No	

Parameter name	Modifiable	Notes
`binlog_row_metadata`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_row_value_options`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_rows_query_log_events`	Yes	
`binlog_transaction_compression`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_transaction_compression_level_zstd`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_transaction_dependency_history_size`	Yes	This parameter sets an upper limit on the number of row hashes that are kept in memory and used for looking up the transaction that last modified a given row. After this number of hashes has been reached, the history is purged. This parameter applies to Aurora MySQL version 2.12 and higher, and version 3.
`binlog_transaction_dependency_tracking`	Yes	This parameter applies to Aurora MySQL version 3.
`character-set-client-handshake`	Yes	
`character_set_client`	Yes	
`character_set_connection`	Yes	
`character_set_database`	Yes	

Parameter name	Modifiable	Notes
`character_set_filesystem`	Yes	
`character_set_results`	Yes	
`character_set_server`	Yes	
`collation_connection`	Yes	
`collation_server`	Yes	
`completion_type`	Yes	
`default_storage_engine`	No	Aurora MySQL clusters use the InnoDB storage engine for all of your data.
`enforce_gtid_consistency`	Sometimes	Modifiable in Aurora MySQL version 2 and higher.
`event_scheduler`	Yes	Indicates the status of the Event Scheduler. Modifiable only at the cluster level in Aurora MySQL version 3.
`gtid-mode`	Sometimes	Modifiable in Aurora MySQL version 2 and higher.
`information_schema_stats_expiry`	Yes	The number of seconds after which the MySQL database server fetches data from the storage engine and replaces the data in the cache. The allowed values are `0–31536000`. This parameter applies to Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`init_connect`	Yes	The command to be run by the server for each client that connects. Use double quotes (") for settings to avoid connection failures, for example: `SET optimizer_switch="hash_join =off"` In Aurora MySQL version 3, this parameter doesn't apply for users who have the `CONNECTION_ADMIN` privilege. This includes the Aurora master user. For more information, see Role-based privilege model.
`innodb_adaptive_hash_index`	Yes	You can modify this parameter at the DB cluster level in Aurora MySQL versions 2 and 3. The Adaptive Hash Index isn't supported on reader DB instances.

Parameter name	Modifiable	Notes
innodb_aurora_instant_alter_column_allowed	Yes	Controls whether the INSTANT algorithm can be used for ALTER COLUMN operations at the global level. The allowed values are the following: - 0 – The INSTANT algorithm isn't allowed for ALTER COLUMN operations (OFF). Reverts to other algorithms. - 1 – The INSTANT algorithm is allowed for ALTER COLUMN operations (ON). This is the default value. For more information, see Column Operations in the MySQL documentation. This parameter applies to Aurora MySQL version 3.05 and higher.
innodb_autoinc_lock_mode	Yes	
innodb_checksums	No	Removed from Aurora MySQL version 3.
innodb_cmp_per_index_enabled	Yes	
innodb_commit_concurrency	Yes	
innodb_data_home_dir	No	Aurora MySQL uses managed instances where you don't access the file system directly.

Parameter name	Modifiable	Notes
`innodb_deadlock_detect`	Yes	This option is used to disable deadlock detection in Aurora MySQL version 2.11 and higher and version 3. On high-concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock. Consult the MySQL documentation for more information on this parameter.
`innodb_default_row_format`	Yes	This parameter defines the default row format for InnoDB tables (including user-created InnoDB temporary tables). It applies to Aurora MySQL versions 2 and 3. Its value can be `DYNAMIC`, `COMPACT`, or `REDUNDANT`.
`innodb_file_per_table`	Yes	This parameter affects how table storage is organized. For more information, see Storage scaling.
`innodb_flush_log_at_trx_commit`	Yes	We highly recommend that you use the default value of 1. In Aurora MySQL version 3, before you can set this parameter to a value other than 1, you must set the value of `innodb_trx_commit_allow_data_loss` to 1. For more information, see Configuring how frequently the log buffer is flushed.

Parameter name	Modifiable	Notes
innodb_ft_max_token_size	Yes	
innodb_ft_min_token_size	Yes	
innodb_ft_num_word_optimize	Yes	
innodb_ft_sort_pll_degree	Yes	
innodb_online_alter_log_max_size	Yes	
innodb_optimize_fulltext_only	Yes	
innodb_page_size	No	
innodb_print_all_deadlocks	Yes	When turned on, records information about all InnoDB deadlocks in the Aurora MySQL error log. For more information, see Minimizing and troubleshooting Aurora MySQL deadlocks.
innodb_purge_batch_size	Yes	
innodb_purge_threads	Yes	
innodb_rollback_on_timeout	Yes	
innodb_rollback_segments	Yes	
innodb_spin_wait_delay	Yes	
innodb_strict_mode	Yes	
innodb_support_xa	Yes	Removed from Aurora MySQL version 3.

Parameter name	Modifiable	Notes
innodb_sync_array_size	Yes	
innodb_sync_spin_loops	Yes	
innodb_stats_include_delete_marked	Yes	When this parameter is enabled, InnoDB includes delete-marked records when calculating persistent optimizer statistics. This parameter applies to Aurora MySQL version 2.12 and higher, and version 3.
innodb_table_locks	Yes	
innodb_trx_commit_allow_data_loss	Yes	In Aurora MySQL version 3, set the value of this parameter to 1 so that you can change the value of innodb_flush_log_at_trx_commit . The default value of innodb_trx_commit_allow_data_loss is 0. For more information, see Configuring how frequently the log buffer is flushed.
innodb_undo_directory	No	Aurora MySQL uses managed instances where you don't access the file system directly.

Parameter name	Modifiable	Notes
internal_tmp_disk_storage_engine	Yes	Controls which in-memory storage engine is used for internal temporary tables. Allowed values are INNODB and MYISAM. This parameter applies to Aurora MySQL version 2.
internal_tmp_mem_storage_engine	Yes	Controls which in-memory storage engine is used for internal temporary tables. Allowed values are MEMORY and TempTable . This parameter applies to Aurora MySQL version 3.
key_buffer_size	Yes	Key cache for MyISAM tables. For more information, see keycache->cache_lock mutex.
lc_time_names	Yes	
log_error_suppression_list	Yes	Specifies a list of error codes that aren't logged in the MySQL error log. This allows you to ignore certain noncritical error conditions to help keep your error logs clean. For more information, see log_error_suppression_list in the MySQL documentation. This parameter applies to Aurora MySQL version 3.03 and higher.

Parameter name	Modifiable	Notes
`low_priority_updates`	Yes	`INSERT`, `UPDATE`, `DELETE`, and `LOCK TABLE WRITE` operations wait until there's no pending `SELECT` operation . This parameter affects only storage engines that use only table-level locking (MyISAM, MEMORY, MERGE). This parameter applies to Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`lower_case_table_names`	Yes (Aurora MySQL version 2) Only at cluster creation time (Aurora MySQL version 3)	In Aurora MySQL version 2.10 and higher 2.x versions, make sure to reboot all reader instances after changing this setting and rebooting the writer instance. For details, see [Rebooting an Aurora cluster with read availability](#). In Aurora MySQL version 3, the value of this parameter is set permanently at the time the cluster is created. If you use a nondefault value for this option, set up your Aurora MySQL version 3 custom parameter group before upgrading, and specify the parameter group during the snapshot restore operation that creates the version 3 cluster. With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. For more information on the methods that you can use, see [Major version upgrades](#).
`master-info-repository`	Yes	Removed from Aurora MySQL version 3.
`master_verify_checksum`	Yes	Aurora MySQL version 2. Use `source_verify_checksum` in Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`max_delayed_threads`	Yes	Sets the maximum number of threads to handle `INSERT DELAYED` statements. This parameter applies to Aurora MySQL version 3.
`max_error_count`	Yes	The maximum number of error, warning, and note messages to be stored for display. This parameter applies to Aurora MySQL version 3.
`max_execution_time`	Yes	The timeout for running `SELECT` statements, in milliseconds. The value can be from `0`–`18446744073709551615`. When set to `0`, there is no timeout. For more information, see max_execution_time in the MySQL documentation.
`min_examined_row_limit`	Yes	Use this parameter to prevent queries that examine fewer than the specified number of rows from being logged. This parameter applies to Aurora MySQL version 3.
`partial_revokes`	No	This parameter applies to Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`preload_buffer_size`	Yes	The size of the buffer that's allocated when preloading indexes. This parameter applies to Aurora MySQL version 3.
`query_cache_type`	Yes	Removed from Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`read_only`	Yes	When this parameter is turned on, the server permits no updates except from those performed by replica threads. For Aurora MySQL version 2, valid values are the following: • `0` – `OFF` • `1` – `ON` • `{TrueIfReplica}` – `ON` for read replicas. This is the default value. • `{TrueIfClusterReplica}` – `ON` for replica clusters such as cross-Region read replicas, secondary clusters in an Aurora global database, and blue/green deployments. For Aurora MySQL version 3, valid values are the following: • `0` – `OFF`. This is the default value. • `1` – `ON` • `{TrueIfClusterReplica}` – `ON` for replica clusters such as cross-Region read replicas, secondary clusters in an Aurora global database, and blue/green deployments. In Aurora MySQL version 3, this parameter doesn't apply for users who have the `CONNECTION_ADMIN` privilege. This includes the Aurora

Parameter name	Modifiable	Notes
		master user. For more information, see Role-based privilege model.
`relay-log-space-limit`	Yes	This parameter applies to Aurora MySQL version 3.
`replica_parallel_type`	Yes	This parameter enables parallel execution on the replica of all uncommitted threads already in the prepare phase, without violating consistency. It applies to Aurora MySQL version 3. In Aurora MySQL version 3.03.* and lower, the default value is DATABASE. In Aurora MySQL version 3.04 and higher, the default value is LOGICAL_CLOCK.
`replica_preserve_commit_order`	Yes	This parameter applies to Aurora MySQL version 3.
`replica_transaction_retries`	Yes	This parameter applies to Aurora MySQL version 3.
`replica_type_conversions`	Yes	This parameter determines the type conversions used on replicas. The allowed values are: `ALL_LOSSY` , `ALL_NON_LOSSY` , `ALL_SIGNED` , and `ALL_UNSIGNED` . For more information, see Replication with differing table definitions on source and replica in the MySQL documentation. This parameter applies to Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`replicate-do-db`	Yes	This parameter applies to Aurora MySQL version 3.
`replicate-do-table`	Yes	This parameter applies to Aurora MySQL version 3.
`replicate-ignore-db`	Yes	This parameter applies to Aurora MySQL version 3.
`replicate-ignore-table`	Yes	This parameter applies to Aurora MySQL version 3.
`replicate-wild-do-table`	Yes	This parameter applies to Aurora MySQL version 3.
`replicate-wild-ignore-table`	Yes	This parameter applies to Aurora MySQL version 3.
`require_secure_transport`	Yes	This parameter applies to Aurora MySQL version 2 and 3. For more information, see Using TLS with Aurora MySQL DB clusters.
`rpl_read_size`	Yes	This parameter applies to Aurora MySQL version 3.
`server_audit_events`	Yes	
`server_audit_excl_users`	Yes	
`server_audit_incl_users`	Yes	
`server_audit_logging`	Yes	For instructions on uploading the logs to Amazon CloudWatch Logs, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.

Parameter name	Modifiable	Notes
`server_audit_logs_upload`	Yes	You can publish audit logs to CloudWatch Logs by enabling Advanced Auditing and setting this parameter to 1. The default for the `server_audit_logs_upload` parameter is `0`. For more information, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.
`server_id`	No	
`skip-character-set-client-handshake`	Yes	
`skip_name_resolve`	No	
`slave-skip-errors`	Yes	Only applies to Aurora MySQL version 2 clusters, with MySQL 5.7 compatibility.
`source_verify_checksum`	Yes	Aurora MySQL version 3
`sync_frm`	Yes	Removed from Aurora MySQL version 3.
`thread_cache_size`	Yes	The number of threads to be cached. This parameter applies to Aurora MySQL versions 2 and 3.

Parameter name	Modifiable	Notes
time_zone	Yes	By default, the time zone for an Aurora DB cluster is Universal Time Coordinated (UTC). You can set the time zone for instances in your DB cluster to the local time zone for your application instead. For more information, see Local time zone for Amazon Aurora DB clusters.
tls_version	Yes	For more information, see TLS versions for Aurora MySQL.

Instance-level parameters

The following table shows all of the parameters that apply to a specific DB instance in an Aurora MySQL DB cluster.

Parameter name	Modifiable	Notes
activate_all_roles_on_login	Yes	This parameter applies to Aurora MySQL version 3.
allow-suspicious-udfs	No	
aurora_disable_hash_join	Yes	Set this parameter to ON to turn off hash join optimization in Aurora MySQL version 2.09 or higher. It isn't supported for version 3. For more information, see Working with parallel query for Amazon Aurora MySQL.
aurora_lab_mode	Yes	For more information, see Amazon Aurora MySQL lab mode. Removed from Aurora MySQL version 3.
aurora_oom_response	Yes	This parameter is supported for Aurora MySQL versions 2 and 3. For more

Parameter name	Modifiable	Notes
		information, see Troubleshooting out-of-memory issues for Aurora MySQL databases.
`aurora_parallel_query`	Yes	Set to `ON` to turn on parallel query in Aurora MySQL version 2.09 or higher. The old `aurora_pq` parameter isn't used in these versions. For more information, see Working with parallel query for Amazon Aurora MySQL.
`aurora_pq`	Yes	Set to `OFF` to turn off parallel query for specific DB instances in Aurora MySQL versions before 2.09. In version 2.09 or higher, turn parallel query on and off with `aurora_parallel_query` instead. For more information, see Working with parallel query for Amazon Aurora MySQL.
`aurora_read_replica_read_committed`	Yes	Enables `READ COMMITTED` isolation level for Aurora Replicas and changes the isolation behavior to reduce purge lag during long-running queries. Enable this setting only if you understand the behavior changes and how they affect your query results. For example, this setting uses less-strict isolation than the MySQL default. When it's enabled, long-running queries might see more than one copy of the same row because Aurora reorganizes the table data while the query is running. For more information, see Aurora MySQL isolation levels.

Parameter name	Modifiable	Notes
`aurora_tmptable_enable_per_table_limit`	Yes	Determines whether the `tmp_table_size` parameter controls the maximum size of in-memory temporary tables created by the `TempTable` storage engine in Aurora MySQL version 3.04 and higher. For more information, see Limiting the size of internal, in-memory temporary tables.
`aurora_use_vector_instructions`	Yes	When this parameter is enabled, Aurora MySQL uses optimized vector processing instructions provided by modern CPUs to improve performance on I/O-intensive workloads. This setting is enabled by default in Aurora MySQL version 3.05 and higher.
`autocommit`	Yes	
`automatic_sp_privileges`	Yes	
`back_log`	Yes	
`basedir`	No	Aurora MySQL uses managed instances where you don't access the file system directly.
`binlog_cache_size`	Yes	
`binlog_max_flush_queue_time`	Yes	
`binlog_order_commits`	Yes	

Parameter name	Modifiable	Notes
`binlog_stmt_cache_size`	Yes	
`binlog_transaction_compression`	Yes	This parameter applies to Aurora MySQL version 3.
`binlog_transaction_compression_level_zstd`	Yes	This parameter applies to Aurora MySQL version 3.
`bulk_insert_buffer_size`	Yes	
`concurrent_insert`	Yes	
`connect_timeout`	Yes	
`core-file`	No	Aurora MySQL uses managed instances where you don't access the file system directly.
`datadir`	No	Aurora MySQL uses managed instances where you don't access the file system directly.
`default_authentication_plugin`	No	This parameter applies to Aurora MySQL version 3.
`default_time_zone`	No	
`default_tmp_storage_engine`	Yes	The default storage engine for temporary tables.
`default_week_format`	Yes	
`delay_key_write`	Yes	
`delayed_insert_limit`	Yes	
`delayed_insert_timeout`	Yes	

Parameter name	Modifiable	Notes
delayed_queue_size	Yes	
div_precision_increment	Yes	
end_markers_in_json	Yes	
eq_range_index_dive_limit	Yes	
event_scheduler	Sometimes	Indicates the status of the Event Scheduler. Modifiable only at the cluster level in Aurora MySQL version 3.
explicit_defaults_for_timestamp	Yes	
flush	No	
flush_time	Yes	
ft_boolean_syntax	No	
ft_max_word_len	Yes	
ft_min_word_len	Yes	
ft_query_expansion_limit	Yes	
ft_stopword_file	Yes	
general_log	Yes	For instructions on uploading the logs to CloudWatch Logs, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.
general_log_file	No	Aurora MySQL uses managed instances where you don't access the file system directly.

Parameter name	Modifiable	Notes
`group_concat_max_len`	Yes	
`host_cache_size`	Yes	
`init_connect`	Yes	The command to be run by the server for each client that connects. Use double quotes (") for settings to avoid connection failures, for example: `SET optimizer_switch="hash_join=off"` In Aurora MySQL version 3, this parameter doesn't apply for users who have the `CONNECTION_ADMIN` privilege, including the Aurora master user. For more information, see Role-based privilege model.
`innodb_adaptive_hash_index`	Yes	You can modify this parameter at the DB instance level in Aurora MySQL version 2. It's modifiable only at the DB cluster level in Aurora MySQL version 3. The Adaptive Hash Index isn't supported on reader DB instances.
`innodb_adaptive_max_sleep_delay`	Yes	Modifying this parameter has no effect because `innodb_thread_concurrency` is always 0 for Aurora.

Parameter name	Modifiable	Notes
innodb_aurora_max_partitions_for_range	Yes	In some cases where persisted statistics aren't available, you can use this parameter to improve the performance of row count estimations on partitioned tables. You can set it to a value from 0–8192, where the value determines the number of partitions to check during row count estimation. The default value is 0, which estimates using all of the partitions, consistent with default MySQL behavior. This parameter is available for Aurora MySQL version 3.03.1 and higher.
innodb_autoextend_increment	Yes	
innodb_buffer_pool_dump_at_shutdown	No	
innodb_buffer_pool_dump_now	No	
innodb_buffer_pool_filename	No	
innodb_buffer_pool_load_abort	No	
innodb_buffer_pool_load_at_startup	No	
innodb_buffer_pool_load_now	No	

Parameter name	Modifiable	Notes
innodb_buffer_pool_size	Yes	The default value is represented by a formula. For details about how the DBInstanceClassMemory value in the formula is calculated, see DB parameter formula variables.
innodb_change_buffer_max_size	No	Aurora MySQL doesn't use the InnoDB change buffer at all.
innodb_compression_failure_threshold_pct	Yes	
innodb_compression_level	Yes	
innodb_compression_pad_pct_max	Yes	
innodb_concurrency_tickets	Yes	Modifying this parameter has no effect, because innodb_thread_concurrency is always 0 for Aurora.
innodb_deadlock_detect	Yes	This option is used to disable deadlock detection in Aurora MySQL version 2.11 and higher and version 3. On high-concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock. Consult the MySQL documentation for more information on this parameter.
innodb_file_format	Yes	Removed from Aurora MySQL version 3.
innodb_flushing_avg_loops	No	

Parameter name	Modifiable	Notes
`innodb_force_load_corrupted`	No	
`innodb_ft_aux_table`	Yes	
`innodb_ft_cache_size`	Yes	
`innodb_ft_enable_stopword`	Yes	
`innodb_ft_server_stopword_table`	Yes	
`innodb_ft_user_stopword_table`	Yes	
`innodb_large_prefix`	Yes	Removed from Aurora MySQL version 3.
`innodb_lock_wait_timeout`	Yes	
`innodb_log_compressed_pages`	No	
`innodb_lru_scan_depth`	Yes	
`innodb_max_purge_lag`	Yes	
`innodb_max_purge_lag_delay`	Yes	
`innodb_monitor_disable`	Yes	
`innodb_monitor_enable`	Yes	
`innodb_monitor_reset`	Yes	
`innodb_monitor_reset_all`	Yes	
`innodb_old_blocks_pct`	Yes	

Parameter name	Modifiable	Notes
`innodb_old_blocks_time`	Yes	
`innodb_open_files`	Yes	
`innodb_print_all_deadlocks`	Yes	When turned on, records information about all InnoDB deadlocks in the Aurora MySQL error log. For more information, see Minimizing and troubleshooting Aurora MySQL deadlocks.
`innodb_random_read_ahead`	Yes	
`innodb_read_ahead_threshold`	Yes	
`innodb_read_io_threads`	No	
`innodb_read_only`	No	Aurora MySQL manages the read-only and read/write state of DB instances based on the type of cluster. For example, a provisioned cluster has one read/write DB instance (the *primary instance*) and any other instances in the cluster are read-only (the Aurora Replicas).
`innodb_replication_delay`	Yes	
`innodb_sort_buffer_size`	Yes	
`innodb_stats_auto_recalc`	Yes	
`innodb_stats_method`	Yes	
`innodb_stats_on_metadata`	Yes	
`innodb_stats_persistent`	Yes	

Parameter name	Modifiable	Notes
`innodb_stats_persistent_sample_pages`	Yes	
`innodb_stats_transient_sample_pages`	Yes	
`innodb_thread_concurrency`	No	
`innodb_thread_sleep_delay`	Yes	Modifying this parameter has no effect because `innodb_thread_concurrency` is always 0 for Aurora.
`interactive_timeout`	Yes	Aurora evaluates the minimum value of `interactive_timeout` and `wait_timeout`. It then uses that minimum as the timeout to end all idle sessions, both interactive and noninteractive.
`internal_tmp_disk_storage_engine`	Yes	Controls which in-memory storage engine is used for internal temporary tables. Allowed values are `INNODB` and `MYISAM`. This parameter applies to Aurora MySQL version 2.
`internal_tmp_mem_storage_engine`	Yes	Controls which in-memory storage engine is used for internal temporary tables. Allowed values are `MEMORY` and `TempTable`. This parameter applies to Aurora MySQL version 3.
`join_buffer_size`	Yes	

Parameter name	Modifiable	Notes
keep_files_on_create	Yes	
key_buffer_size	Yes	Key cache for MyISAM tables. For more information, see keycache->cache_lock mutex.
key_cache_age_threshold	Yes	
key_cache_block_size	Yes	
key_cache_division_limit	Yes	
local_infile	Yes	
lock_wait_timeout	Yes	
log-bin	No	Setting binlog_format to STATEMENT , MIXED, or ROW automatically sets log-bin to ON. Setting binlog_format to OFF automatically sets log-bin to OFF. For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).
log_bin_trust_function_creators	Yes	
log_bin_use_v1_row_events	Yes	Removed from Aurora MySQL version 3.
log_error	No	

Parameter name	Modifiable	Notes
`log_error_suppression_list`	Yes	Specifies a list of error codes that aren't logged in the MySQL error log. This allows you to ignore certain noncritical error conditions to help keep your error logs clean. For more information, see log_error_suppression_list in the MySQL documentation. This parameter applies to Aurora MySQL version 3.03 and higher.
`log_output`	Yes	
`log_queries_not_using_indexes`	Yes	
`log_slave_updates`	No	Aurora MySQL version 2. Use `log_replica_updates` in Aurora MySQL version 3.
`log_replica_updates`	No	Aurora MySQL version 3
`log_throttle_queries_not_using_indexes`	Yes	
`log_warnings`	Yes	Removed from Aurora MySQL version 3.
`long_query_time`	Yes	

Parameter name	Modifiable	Notes
`low_priority_updates`	Yes	`INSERT`, `UPDATE`, `DELETE`, and `LOCK TABLE WRITE` operations wait until there's no pending `SELECT` operation . This parameter affects only storage engines that use only table-level locking (MyISAM, MEMORY, MERGE). This parameter applies to Aurora MySQL version 3.
`max_allowed_packet`	Yes	
`max_binlog_cache_size`	Yes	
`max_binlog_size`	No	
`max_binlog_stmt_cache_size`	Yes	
`max_connect_errors`	Yes	
`max_connections`	Yes	The default value is represented by a formula. For details about how the `DBInstanceClassMemory` value in the formula is calculated, see DB parameter formula variables. For the default values depending on the instance class, see Maximum connections to an Aurora MySQL DB instance.
`max_delayed_threads`	Yes	Sets the maximum number of threads to handle `INSERT DELAYED` statements. This parameter applies to Aurora MySQL version 3.

Parameter name	Modifiable	Notes
`max_error_count`	Yes	The maximum number of error, warning, and note messages to be stored for display. This parameter applies to Aurora MySQL version 3.
`max_execution_time`	Yes	The timeout for running `SELECT` statements, in milliseconds. The value can be from `0`–`18446744073709551615`. When set to `0`, there is no timeout. For more information, see max_execution_time in the MySQL documentation.
`max_heap_table_size`	Yes	
`max_insert_delayed_threads`	Yes	
`max_join_size`	Yes	
`max_length_for_sort_data`	Yes	Removed from Aurora MySQL version 3.
`max_prepared_stmt_count`	Yes	
`max_seeks_for_key`	Yes	
`max_sort_length`	Yes	
`max_sp_recursion_depth`	Yes	
`max_tmp_tables`	Yes	Removed from Aurora MySQL version 3.
`max_user_connections`	Yes	

Parameter name	Modifiable	Notes
max_write_lock_count	Yes	
metadata_locks_cache_size	Yes	Removed from Aurora MySQL version 3.
min_examined_row_limit	Yes	Use this parameter to prevent queries that examine fewer than the specified number of rows from being logged. This parameter applies to Aurora MySQL version 3.
myisam_data_pointer_size	Yes	
myisam_max_sort_file_size	Yes	
myisam_mmap_size	Yes	
myisam_sort_buffer_size	Yes	
myisam_stats_method	Yes	
myisam_use_mmap	Yes	
net_buffer_length	Yes	
net_read_timeout	Yes	
net_retry_count	Yes	
net_write_timeout	Yes	
old-style-user-limits	Yes	
old_passwords	Yes	Removed from Aurora MySQL version 3.
optimizer_prune_level	Yes	

Parameter name	Modifiable	Notes
optimizer_search_depth	Yes	
optimizer_switch	Yes	For information about Aurora MySQL features that use this switch, see Best practices with Amazon Aurora MySQL.
optimizer_trace	Yes	
optimizer_trace_features	Yes	
optimizer_trace_limit	Yes	
optimizer_trace_max_mem_size	Yes	
optimizer_trace_offset	Yes	
performance-schema-consumer-events-waits-current	Yes	
performance-schema-instrument	Yes	
performance_schema	Yes	
performance_schema_accounts_size	Yes	
performance_schema_consumer_global_instrumentation	Yes	
performance_schema_consumer_thread_instrumentation	Yes	
performance_schema_consumer_events_stages_current	Yes	

Parameter name	Modifiable	Notes
performance_schema_consumer_events_stages_history	Yes	
performance_schema_consumer_events_stages_history_long	Yes	
performance_schema_consumer_events_statements_current	Yes	
performance_schema_consumer_events_statements_history	Yes	
performance_schema_consumer_events_statements_history_long	Yes	
performance_schema_consumer_events_waits_history	Yes	
performance_schema_consumer_events_waits_history_long	Yes	
performance_schema_consumer_statements_digest	Yes	
performance_schema_digests_size	Yes	
performance_schema_events_stages_history_long_size	Yes	
performance_schema_events_stages_history_size	Yes	

Parameter name	Modifiable	Notes
`performance_schema_events_statements_history_long_size`	Yes	
`performance_schema_events_statements_history_size`	Yes	
`performance_schema_events_transactions_history_long_size`	Yes	
`performance_schema_events_transactions_history_size`	Yes	
`performance_schema_events_waits_history_long_size`	Yes	
`performance_schema_events_waits_history_size`	Yes	
`performance_schema_hosts_size`	Yes	
`performance_schema_max_cond_classes`	Yes	
`performance_schema_max_cond_instances`	Yes	
`performance_schema_max_digest_length`	Yes	
`performance_schema_max_file_classes`	Yes	
`performance_schema_max_file_handles`	Yes	

Parameter name	Modifiable	Notes
`performance_schema_max_file_instances`	Yes	
`performance_schema_max_index_stat`	Yes	
`performance_schema_max_memory_classes`	Yes	
`performance_schema_max_metadata_locks`	Yes	
`performance_schema_max_mutex_classes`	Yes	
`performance_schema_max_mutex_instances`	Yes	
`performance_schema_max_prepared_statements_instances`	Yes	
`performance_schema_max_program_instances`	Yes	
`performance_schema_max_rwlock_classes`	Yes	
`performance_schema_max_rwlock_instances`	Yes	
`performance_schema_max_socket_classes`	Yes	
`performance_schema_max_socket_instances`	Yes	

Parameter name	Modifiable	Notes
`performance_schema_max_sql_text_length`	Yes	
`performance_schema_max_stage_classes`	Yes	
`performance_schema_max_statement_classes`	Yes	
`performance_schema_max_statement_stack`	Yes	
`performance_schema_max_table_handles`	Yes	
`performance_schema_max_table_instances`	Yes	
`performance_schema_max_table_lock_stat`	Yes	
`performance_schema_max_thread_classes`	Yes	
`performance_schema_max_thread_instances`	Yes	
`performance_schema_session_connect_attrs_size`	Yes	
`performance_schema_setup_actors_size`	Yes	
`performance_schema_setup_objects_size`	Yes	

Parameter name	Modifiable	Notes
`performance_schema_show_processlist`	Yes	This parameter determines which `SHOW PROCESSLIST` implementation to use: - The default implementation iterates across active threads from within the thread manager while holding a global mutex. This can cause slow performance, especially on busy systems. - The alternative `SHOW PROCESSLIST` implementation is based on the Performance Schema `processlist` table. This implementation queries active thread data from the Performance Schema rather than the thread manager and doesn't require a mutex. This parameter applies to Aurora MySQL version 2.12 and higher, and version 3.
`performance_schema_users_size`	Yes	
`pid_file`	No	
`plugin_dir`	No	Aurora MySQL uses managed instances where you don't access the file system directly.
`port`	No	Aurora MySQL manages the connection properties and enforces consistent settings for all DB instances in a cluster.

Parameter name	Modifiable	Notes
`preload_buffer_size`	Yes	The size of the buffer that's allocated when preloading indexes. This parameter applies to Aurora MySQL version 3.
`profiling_history_size`	Yes	
`query_alloc_block_size`	Yes	
`query_cache_limit`	Yes	Removed from Aurora MySQL version 3.
`query_cache_min_res_unit`	Yes	Removed from Aurora MySQL version 3.
`query_cache_size`	Yes	The default value is represented by a formula. For details about how the `DBInstanceClassMemory` value in the formula is calculated, see DB parameter formula variables. Removed from Aurora MySQL version 3.
`query_cache_type`	Yes	Removed from Aurora MySQL version 3.
`query_cache_wlock_invalidate`	Yes	Removed from Aurora MySQL version 3.
`query_prealloc_size`	Yes	
`range_alloc_block_size`	Yes	
`read_buffer_size`	Yes	

Parameter name	Modifiable	Notes
`read_only`	Yes	When this parameter is turned on, the server permits no updates except from those performed by replica threads. For Aurora MySQL version 2, valid values are the following: • `0` – `OFF` • `1` – `ON` • `{TrueIfReplica}` – `ON` for read replicas. This is the default value. • `{TrueIfClusterReplica}` – `ON` for instances in replica clusters such as cross-Region read replicas, secondary clusters in an Aurora global database, and blue/green deployments. We recommend that you use the DB cluster parameter group in Aurora MySQL version 2 to make sure that the `read_only` parameter is applied to new writer instances on failover. **Note** Reader instances are always read only, because Aurora MySQL sets `innodb_read_only` to 1 on all readers. Therefore, `read_only` is redundant on reader instances.

Parameter name	Modifiable	Notes
		Removed at the instance level from Aurora MySQL version 3.
`read_rnd_buffer_size`	Yes	
`relay-log`	No	
`relay_log_info_repository`	Yes	Removed from Aurora MySQL version 3.
`relay_log_recovery`	No	
`replica_checkpoint_group`	Yes	Aurora MySQL version 3
`replica_checkpoint_period`	Yes	Aurora MySQL version 3
`replica_parallel_workers`	Yes	Aurora MySQL version 3
`replica_pending_jobs_size_max`	Yes	Aurora MySQL version 3
`replica_skip_errors`	Yes	Aurora MySQL version 3
`replica_sql_verify_checksum`	Yes	Aurora MySQL version 3
`safe-user-create`	Yes	
`secure_auth`	Yes	This parameter is always turned on in Aurora MySQL version 2. Trying to turn it off generates an error. Removed from Aurora MySQL version 3.
`secure_file_priv`	No	Aurora MySQL uses managed instances where you don't access the file system directly.

Parameter name	Modifiable	Notes
`show_create_table_verbosity`	Yes	Enabling this variable causes SHOW_CREATE_TABLE to display the `ROW_FORMAT` regardless of whether it's the default format. This parameter applies to Aurora MySQL version 2.12 and higher, and version 3.
`skip-slave-start`	No	
`skip_external_locking`	No	
`skip_show_database`	Yes	
`slave_checkpoint_group`	Yes	Aurora MySQL version 2. Use `replica_checkpoint_group` in Aurora MySQL version 3.
`slave_checkpoint_period`	Yes	Aurora MySQL version 2. Use `replica_checkpoint_period` in Aurora MySQL version 3.
`slave_parallel_workers`	Yes	Aurora MySQL version 2. Use `replica_parallel_workers` in Aurora MySQL version 3.
`slave_pending_jobs_size_max`	Yes	Aurora MySQL version 2. Use `replica_pending_jobs_size_max` in Aurora MySQL version 3.
`slave_sql_verify_checksum`	Yes	Aurora MySQL version 2. Use `replica_sql_verify_checksum` in Aurora MySQL version 3.
`slow_launch_time`	Yes	

Parameter name	Modifiable	Notes
slow_query_log	Yes	For instructions on uploading the logs to CloudWatch Logs, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.
slow_query_log_file	No	Aurora MySQL uses managed instances where you don't access the file system directly.
socket	No	
sort_buffer_size	Yes	
sql_mode	Yes	
sql_select_limit	Yes	
stored_program_cache	Yes	
sync_binlog	No	
sync_master_info	Yes	
sync_source_info	Yes	This parameter applies to Aurora MySQL version 3.
sync_relay_log	Yes	Removed from Aurora MySQL version 3.
sync_relay_log_info	Yes	
sysdate-is-now	Yes	
table_cache_element_entry_ttl	No	

Parameter name	Modifiable	Notes
`table_definition_cache`	Yes	The default value is represented by a formula. For details about how the `DBInstanceClassMemory` value in the formula is calculated, see DB parameter formula variables.
`table_open_cache`	Yes	The default value is represented by a formula. For details about how the `DBInstanceClassMemory` value in the formula is calculated, see DB parameter formula variables.
`table_open_cache_instances`	Yes	
`temp-pool`	Yes	Removed from Aurora MySQL version 3.
`temptable_max_mmap`	Yes	This parameter applies to Aurora MySQL version 3. For details, see New temporary table behavior in Aurora MySQL version 3.
`temptable_max_ram`	Yes	This parameter applies to Aurora MySQL version 3. For details, see New temporary table behavior in Aurora MySQL version 3.
`temptable_use_mmap`	Yes	This parameter applies to Aurora MySQL version 3. For details, see New temporary table behavior in Aurora MySQL version 3.
`thread_cache_size`	Yes	The number of threads to be cached. This parameter applies to Aurora MySQL versions 2 and 3.

Parameter name	Modifiable	Notes
thread_handling	No	
thread_stack	Yes	
timed_mutexes	Yes	
tmp_table_size	Yes	Defines the maximum size of internal in-memory temporary tables created by the MEMORY storage engine in Aurora MySQL version 3. In Aurora MySQL version 3.04 and higher, defines the maximum size of internal in-memory temporary tables created by the TempTable storage engine when aurora_tmptable_enable_per_table_limit is ON. For more information, see Limiting the size of internal, in-memory temporary tables.
tmpdir	No	Aurora MySQL uses managed instances where you don't access the file system directly.
transaction_alloc_block_size	Yes	
transaction_isolation	Yes	This parameter applies to Aurora MySQL version 3. It replaces tx_isolation .
transaction_prealloc_size	Yes	

Parameter name	Modifiable	Notes
`tx_isolation`	Yes	Removed from Aurora MySQL version 3. It is replaced by `transaction_isolation` .
`updatable_views_with_limit`	Yes	
`validate-password`	No	
`validate_password_dictionary_file`	No	
`validate_password_length`	No	
`validate_password_mixed_case_count`	No	
`validate_password_number_count`	No	
`validate_password_policy`	No	
`validate_password_special_char_count`	No	
`wait_timeout`	Yes	Aurora evaluates the minimum value of `interactive_timeout` and `wait_timeout` . It then uses that minimum as the timeout to end all idle sessions, both interactive and noninteractive.

MySQL parameters that don't apply to Aurora MySQL

Because of architectural differences between Aurora MySQL and MySQL, some MySQL parameters don't apply to Aurora MySQL.

The following MySQL parameters don't apply to Aurora MySQL. This list isn't exhaustive.

- activate_all_roles_on_login – This parameter doesn't apply to Aurora MySQL version 2. It is available in Aurora MySQL version 3.
- big_tables
- bind_address
- character_sets_dir
- innodb_adaptive_flushing
- innodb_adaptive_flushing_lwm
- innodb_buffer_pool_chunk_size
- innodb_buffer_pool_instances
- innodb_change_buffering
- innodb_checksum_algorithm
- innodb_data_file_path
- innodb_dedicated_server
- innodb_doublewrite
- innodb_flush_log_at_timeout – This parameter doesn't apply to Aurora MySQL. For more information, see Configuring how frequently the log buffer is flushed.
- innodb_flush_method
- innodb_flush_neighbors
- innodb_io_capacity
- innodb_io_capacity_max
- innodb_log_buffer_size
- innodb_log_file_size
- innodb_log_files_in_group
- innodb_log_spin_cpu_abs_lwm
- innodb_log_spin_cpu_pct_hwm
- innodb_log_writer_threads
- innodb_max_dirty_pages_pct
- innodb_numa_interleave
- innodb_page_size
- innodb_redo_log_capacity

- innodb_redo_log_encrypt
- innodb_undo_log_encrypt
- innodb_undo_log_truncate
- innodb_undo_logs
- innodb_undo_tablespaces
- innodb_use_native_aio
- innodb_write_io_threads

Aurora MySQL global status variables

You can find the current values for Aurora MySQL global status variables by using a statement such as the following:

```
show global status like '%aurora%';
```

The following table describes the global status variables that Aurora MySQL uses.

Name	Description
AuroraDb_commits	The total number of commits since the last restart.
AuroraDb_commit_latency	The aggregate commit latency since the last restart.
AuroraDb_ddl_stmt_duration	The aggregate DDL latency since the last restart.
AuroraDb_select_stmt_duration	The aggregate SELECT statement latency since the last restart.
AuroraDb_insert_stmt_duration	The aggregate INSERT statement latency since the last restart.
AuroraDb_update_stmt_duration	The aggregate UPDATE statement latency since the last restart.

Name	Description
`AuroraDb_delete_stmt_duration`	The aggregate `DELETE` statement latency since the last restart.
`Aurora_binlog_io_cache_allocated`	The number of bytes allocated to the binlog I/O cache.
`Aurora_binlog_io_cache_read_requests`	The number of read requests made to the binlog I/O cache.
`Aurora_binlog_io_cache_reads`	The number of read requests that were served from the binlog I/O cache.
`Aurora_enhanced_binlog`	Indicates whether enhanced binlog is enabled or disabled for this DB instance. For more information, see Setting up enhanced binlog.
`Aurora_external_connection_count`	The number of database connections to the DB instance, excluding RDS service connectio ns used for database health checks.
`Aurora_fast_insert_cache_hits`	A counter that's incremented when the cached cursor is successfully retrieved and verified. For more information on the fast insert cache, see Amazon Aurora MySQL performance enhancements.
`Aurora_fast_insert_cache_misses`	A counter that's incremented when the cached cursor is no longer valid and Aurora performs a normal index traversal. For more information on the fast insert cache, see Amazon Aurora MySQL performance enhancements.
`Aurora_fts_cache_memory_used`	The amount of memory in bytes that the InnoDB full-text search system is using. This variable applies to Aurora MySQL version 3.07 and higher.

Name	Description
`Aurora_fwd_master_dml_stmt_count`	The total number of DML statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 2.
`Aurora_fwd_master_dml_stmt_duration`	The total duration of DML statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 2.
`Aurora_fwd_master_errors_rpc_timeout`	The number of times a forwarded connection failed to be established on the writer.
`Aurora_fwd_master_errors_session_limit`	The number of forwarded queries that get rejected due to `session full` on the writer.
`Aurora_fwd_master_errors_session_timeout`	The number of times a forwarding session is ended due to a timeout on the writer.
`Aurora_fwd_master_open_sessions`	The number of forwarded sessions on the writer DB instance. This variable applies to Aurora MySQL version 2.
`Aurora_fwd_master_select_stmt_count`	The total number of `SELECT` statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 2.
`Aurora_fwd_master_select_stmt_duration`	The total duration of `SELECT` statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 2.
`Aurora_fwd_writer_dml_stmt_count`	The total number of DML statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 3.
`Aurora_fwd_writer_dml_stmt_duration`	The total duration of DML statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 3.

Name	Description
`Aurora_fwd_writer_errors_rpc_timeout`	The number of times a forwarded connection failed to be established on the writer.
`Aurora_fwd_writer_errors_session_limit`	The number of forwarded queries that get rejected due to `session full` on the writer.
`Aurora_fwd_writer_errors_session_timeout`	The number of times a forwarding session is ended due to a timeout on the writer.
`Aurora_fwd_writer_open_sessions`	The number of forwarded sessions on the writer DB instance. This variable applies to Aurora MySQL version 3.
`Aurora_fwd_writer_select_stmt_count`	The total number of `SELECT` statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 3.
`Aurora_fwd_writer_select_stmt_duration`	The total duration of `SELECT` statements forwarded to this writer DB instance. This variable applies to Aurora MySQL version 3.
`Aurora_lockmgr_buffer_pool_memory_used`	The amount of buffer pool memory in bytes that the Aurora MySQL lock manager is using.
`Aurora_lockmgr_memory_used`	The amount of memory in bytes that the Aurora MySQL lock manager is using.
`Aurora_ml_actual_request_cnt`	The aggregate request count that Aurora MySQLmakes to the Aurora machine learning services across all queries run by users of the DB instance. For more information, see Using Amazon Aurora machine learning with Aurora MySQL.

Name	Description
`Aurora_ml_actual_response_cnt`	The aggregate response count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance. For more informati on, see Using Amazon Aurora machine learning with Aurora MySQL.
`Aurora_ml_cache_hit_cnt`	The aggregate internal cache hit count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance. For more information, see Using Amazon Aurora machine learning with Aurora MySQL.
`Aurora_ml_logical_request_cnt`	The number of logical requests that the DB instance has evaluated to be sent to the Aurora machine learning services since the last status reset. Depending on whether batching has been used, this value can be higher than `Aurora_ml_actual_request_cnt` . For more information, see Using Amazon Aurora machine learning with Aurora MySQL.
`Aurora_ml_logical_response_cnt`	The aggregate response count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance. For more informati on, see Using Amazon Aurora machine learning with Aurora MySQL.
`Aurora_ml_retry_request_cnt`	The number of retried requests that the DB instance has sent to the Aurora machine learning services since the last status reset. For more information, see Using Amazon Aurora machine learning with Aurora MySQL.

Name	Description
`Aurora_ml_single_request_cnt`	The aggregate count of Aurora machine learning functions that are evaluated by non-batch mode across all queries run by users of the DB instance. For more information, see [Using Amazon Aurora machine learning with Aurora MySQL]().
`aurora_oom_avoidance_recovery_state`	Indicates whether Aurora out-of-memory (OOM) avoidance recovery is in the `ACTIVE` or `INACTIVE` state for this DB instance.
`aurora_oom_reserved_mem_enter_kb`	Represents the threshold for entering the `RESERVED` state in Aurora's OOM handling mechanism. When the available memory on the server falls below this threshold, `aurora_oom_status` changes to `RESERVED`, indicating that the server is approaching a critical level of memory usage.
`aurora_oom_reserved_mem_exit_kb`	Represents the threshold for exiting the `RESERVED` state in Aurora's OOM handling mechanism. When the available memory on the server rises above this threshold, `aurora_oom_status` reverts to `NORMAL`, indicating that the server has returned to a more stable state with sufficient memory resources.

Name	Description
`aurora_oom_status`	Represents the current OOM status of this DB instance. When the value is `NORMAL`, it indicates that there are sufficient memory resources. If the value changes to `RESERVED`, it indicates that the server has low available memory. Actions are taken based on the `aurora_oom_response` parameter configuration. For more information, see Troubleshooting out-of-memory issues for Aurora MySQL databases.
`Aurora_pq_bytes_returned`	The number of bytes for the tuple data structures transmitted to the head node during parallel queries. Divide by 16,384 to compare against `Aurora_pq_pages_pushed_down` .
`Aurora_pq_max_concurrent_requests`	The maximum number of parallel query sessions that can run concurrently on this Aurora DB instance. This is a fixed number that depends on the AWS DB instance class.
`Aurora_pq_pages_pushed_down`	The number of data pages (each with a fixed size of 16 KiB) where parallel query avoided a network transmission to the head node.
`Aurora_pq_request_attempted`	The number of parallel query sessions requested. This value might represent more than one session per query, depending on SQL constructs such as subqueries and joins.
`Aurora_pq_request_executed`	The number of parallel query sessions run successfully.

Name	Description
`Aurora_pq_request_failed`	The number of parallel query sessions that returned an error to the client. In some cases, a request for a parallel query might fail, for example due to a problem in the storage layer. In these cases, the query part that failed is retried using the nonparallel query mechanism . If the retried query also fails, an error is returned to the client and this counter is incremented.
`Aurora_pq_request_in_progress`	The number of parallel query sessions currently in progress. This number applies to the particular Aurora DB instance that you are connected to, not the entire Aurora DB cluster. To see if a DB instance is close to its concurrency limit, compare this value to `Aurora_pq_max_concurrent_requests` .
`Aurora_pq_request_not_chosen`	The number of times parallel query wasn't chosen to satisfy a query. This value is the sum of several other more granular counters. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_below_min_rows`	The number of times parallel query wasn't chosen due to the number of rows in the table. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_column_bit`	The number of parallel query requests that use the nonparallel query processing path because of an unsupported data type in the list of projected columns.

Name	Description
Aurora_pq_request_not_chosen_column_geometry	The number of parallel query requests that use the nonparallel query processing path because the table has columns with the GEOMETRY data type. For information about Aurora MySQL versions that remove this limitation, see Upgrading parallel query clusters to Aurora MySQL version 3.
Aurora_pq_request_not_chosen_column_lob	The number of parallel query requests that use the nonparallel query processing path because the table has columns with a LOB data type, or VARCHAR columns that are stored externally due to the declared length. For information about Aurora MySQL versions that remove this limitation, see Upgrading parallel query clusters to Aurora MySQL version 3.
Aurora_pq_request_not_chosen_column_virtual	The number of parallel query requests that use the nonparallel query processing path because the table contains a virtual column.
Aurora_pq_request_not_chosen_custom_charset	The number of parallel query requests that use the nonparallel query processing path because the table has columns with a custom character set.
Aurora_pq_request_not_chosen_fast_ddl	The number of parallel query requests that use the nonparallel query processing path because the table is currently being altered by a fast DDL ALTER statement.

Name	Description
`Aurora_pq_request_not_chosen_few_pages_outside_buffer_pool`	The number of times parallel query wasn't chosen, even though less than 95 percent of the table data was in the buffer pool, because there wasn't enough unbuffered table data to make parallel query worthwhile.
`Aurora_pq_request_not_chosen_full_text_index`	The number of parallel query requests that use the nonparallel query processing path because the table has full-text indexes.
`Aurora_pq_request_not_chosen_high_buffer_pool_pct`	The number of times parallel query wasn't chosen because a high percentage of the table data (currently, greater than 95 percent) was already in the buffer pool. In these cases, the optimizer determines that reading the data from the buffer pool is more efficient . An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_index_hint`	The number of parallel query requests that use the nonparallel query processing path because the query includes an index hint.
`Aurora_pq_request_not_chosen_innodb_table_format`	The number of parallel query requests that use the nonparallel query processing path because the table uses an unsupported InnoDB row format. Aurora parallel query only applies to the `COMPACT`, `REDUNDANT` , and `DYNAMIC` row formats.

Name	Description
`Aurora_pq_request_not_chosen_long_trx`	The number of parallel query requests that used the nonparallel query processing path, due to the query being started inside a long-running transaction. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_no_where_clause`	The number of parallel query requests that use the nonparallel query processing path because the query doesn't include any `WHERE` clause.
`Aurora_pq_request_not_chosen_range_scan`	The number of parallel query requests that use the nonparallel query processing path because the query uses a range scan on an index.
`Aurora_pq_request_not_chosen_row_length_too_long`	The number of parallel query requests that use the nonparallel query processing path because the total combined length of all the columns is too long.
`Aurora_pq_request_not_chosen_small_table`	The number of times parallel query wasn't chosen due to the overall size of the table, as determined by number of rows and average row length. An `EXPLAIN` statement can increment this counter even though the query isn't actually performed.
`Aurora_pq_request_not_chosen_temporary_table`	The number of parallel query requests that use the nonparallel query processing path because the query refers to temporary tables that use the unsupported `MyISAM` or `memory` table types.

Name	Description
`Aurora_pq_request_not_chosen_tx_isolation`	The number of parallel query requests that use the nonparallel query processing path because query uses an unsupported transaction isolation level. On reader DB instances, parallel query only applies to the `REPEATABLE READ` and `READ COMMITTED` isolation levels.
`Aurora_pq_request_not_chosen_update_delete_stmts`	The number of parallel query requests that use the nonparallel query processing path because the query is part of an `UPDATE` or `DELETE` statement.
`Aurora_pq_request_not_chosen_unsupported_access`	The number of parallel query requests that use the nonparallel query processing path because the `WHERE` clause doesn't meet the criteria for parallel query. This result can occur if the query doesn't require a data-intensive scan, or if the query is a `DELETE` or `UPDATE` statement.
`Aurora_pq_request_not_chosen_unsupported_storage_type`	The number of parallel query requests that use the nonparallel query processing path because the Aurora MySQL DB cluster isn't using a supported Aurora cluster storage configuration. For more information, see Limitations. This parameter applies to Aurora MySQL version 3.04 and higher.
`Aurora_pq_request_throttled`	The number of times parallel query wasn't chosen due to the maximum number of concurrent parallel queries already running on a particular Aurora DB instance.

Name	Description
`Aurora_repl_bytes_received`	Number of bytes replicated to an Aurora MySQL reader database instance since the last restart. For more information, see Replication with Amazon Aurora MySQL.
`Aurora_reserved_mem_exceeded_incidents`	The number of times since the last restart that the engine has exceeded reserved memory limits. If `aurora_oom_response` is configured, this threshold defines when out-of-memory (OOM) avoidance activities are triggered. For more information on the Aurora MySQL OOM response, see Troubleshooting out-of-memory issues for Aurora MySQL databases.
`Aurora_thread_pool_thread_count`	The current number of threads in the Aurora thread pool. For more information on the thread pool in Aurora MySQL, see Thread pool.
`Aurora_tmz_version`	Denotes the current version of the time zone information used by the DB cluster. The values follow the Internet Assigned Numbers Authority (IANA) format: `YYYYsuffix`, for example `2022a` and `2023c`. This parameter applies to Aurora MySQL version 2.12 and higher, and version 3.04 and higher.
`Aurora_zdr_oom_threshold`	Represents the memory threshold, in kilobytes (KB), for an Aurora DB instance to initiate a zero downtime restart (ZDR) to recover from potential memory-related issues.

Name	Description
`server_aurora_das_running`	Indicates whether Database Activity Streams (DAS) are enabled or disabled on this DB instance. For more information, see Monitoring Amazon Aurora with Database Activity Streams.

MySQL status variables that don't apply to Aurora MySQL

Because of architectural differences between Aurora MySQL and MySQL, some MySQL status variables don't apply to Aurora MySQL.

The following MySQL status variables don't apply to Aurora MySQL. This list isn't exhaustive.

- `innodb_buffer_pool_bytes_dirty`
- `innodb_buffer_pool_pages_dirty`
- `innodb_buffer_pool_pages_flushed`

Aurora MySQL version 3 removes the following status variables that were in Aurora MySQL version 2:

- `AuroraDb_lockmgr_bitmaps0_in_use`
- `AuroraDb_lockmgr_bitmaps1_in_use`
- `AuroraDb_lockmgr_bitmaps_mem_used`
- `AuroraDb_thread_deadlocks`
- `available_alter_table_log_entries`
- `Aurora_lockmgr_memory_used`
- `Aurora_missing_history_on_replica_incidents`
- `Aurora_new_lock_manager_lock_release_cnt`
- `Aurora_new_lock_manager_lock_release_total_duration_micro`
- `Aurora_new_lock_manager_lock_timeout_cnt`
- `Aurora_total_op_memory`

- `Aurora_total_op_temp_space`
- `Aurora_used_alter_table_log_entries`
- `Aurora_using_new_lock_manager`
- `Aurora_volume_bytes_allocated`
- `Aurora_volume_bytes_left_extent`
- `Aurora_volume_bytes_left_total`
- `Com_alter_db_upgrade`
- `Compression`
- `External_threads_connected`
- `Innodb_available_undo_logs`
- `Last_query_cost`
- `Last_query_partial_plans`
- `Slave_heartbeat_period`
- `Slave_last_heartbeat`
- `Slave_received_heartbeats`
- `Slave_retried_transactions`
- `Slave_running`
- `Time_since_zero_connections`

These MySQL status variables are available in Aurora MySQL version 2, but they aren't available in Aurora MySQL version 3:

- `Innodb_redo_log_enabled`
- `Innodb_undo_tablespaces_total`
- `Innodb_undo_tablespaces_implicit`
- `Innodb_undo_tablespaces_explicit`
- `Innodb_undo_tablespaces_active`

Aurora MySQL wait events

The following are some common wait events for Aurora MySQL.

Note

For information on tuning Aurora MySQL performance using wait events, see Tuning Aurora MySQL with wait events.

For information about the naming conventions used in MySQL wait events, see Performance Schema instrument naming conventions in the MySQL documentation.

cpu

The number of active connections that are ready to run is consistently higher than the number of vCPUs. For more information, see cpu.

io/aurora_redo_log_flush

A session is persisting data to Aurora storage. Typically, this wait event is for a write I/O operation in Aurora MySQL. For more information, see io/aurora_redo_log_flush.

io/aurora_respond_to_client

Query processing has completed and results are being returned to the application client for the following Aurora MySQL versions: 2.10.2 and higher 2.10 versions, 2.09.3 and higher 2.09 versions, and 2.07.7 and higher 2.07 versions. Compare the network bandwidth of the DB instance class with the size of the result set being returned. Also, check client-side response times. If the client is unresponsive and can't process the TCP packets, packet drops and TCP retransmissions can occur. This situation negatively affects network bandwidth. In versions lower than 2.10.2, 2.09.3, and 2.07.7, the wait event erroneously includes idle time. To learn how to tune your database when this wait is prominent, see io/aurora_respond_to_client.

io/file/csv/data

Threads are writing to tables in comma-separated value (CSV) format. Check your CSV table usage. A typical cause of this event is setting `log_output` on a table.

io/file/sql/binlog

A thread is waiting on a binary log (binlog) file that is being written to disk.

io/redo_log_flush

A session is persisting data to Aurora storage. Typically, this wait event is for a write I/O operation in Aurora MySQL. For more information, see io/redo_log_flush.

io/socket/sql/client_connection

The mysqld program is busy creating threads to handle incoming new client connections. For more information, see io/socket/sql/client_connection.

io/table/sql/handler

The engine is waiting for access to a table. This event occurs regardless of whether the data is cached in the buffer pool or accessed on disk. For more information, see io/table/sql/handler.

lock/table/sql/handler

This wait event is a table lock wait event handler. For more information about atom and molecule events in the Performance Schema, see Performance Schema atom and molecule events in the MySQL documentation.

synch/cond/innodb/row_lock_wait

Multiple data manipulation language (DML) statements are accessing the same database rows at the same time. For more information, see synch/cond/innodb/row_lock_wait.

synch/cond/innodb/row_lock_wait_cond

Multiple DML statements are accessing the same database rows at the same time. For more information, see synch/cond/innodb/row_lock_wait_cond.

synch/cond/sql/MDL_context::COND_wait_status

Threads are waiting on a table metadata lock. The engine uses this type of lock to manage concurrent access to a database schema and to ensure data consistency. For more information, see Optimizing locking operations in the MySQL documentation. To learn how to tune your database when this event is prominent, see synch/cond/sql/MDL_context::COND_wait_status.

synch/cond/sql/MYSQL_BIN_LOG::COND_done

You have turned on binary logging. There might be a high commit throughput, large number transactions committing, or replicas reading binlogs. Consider using multirow statements or bundling statements into one transaction. In Aurora, use global databases instead of binary log replication, or use the aurora_binlog_* parameters.

synch/mutex/innodb/aurora_lock_thread_slot_futex

Multiple DML statements are accessing the same database rows at the same time. For more information, see synch/mutex/innodb/aurora_lock_thread_slot_futex.

synch/mutex/innodb/buf_pool_mutex

The buffer pool isn't large enough to hold the working data set. Or the workload accesses pages from a specific table, which leads to contention in the buffer pool. For more information, see synch/mutex/innodb/buf_pool_mutex.

synch/mutex/innodb/fil_system_mutex

The process is waiting for access to the tablespace memory cache. For more information, see synch/mutex/innodb/fil_system_mutex.

synch/mutex/innodb/trx_sys_mutex

Operations are checking, updating, deleting, or adding transaction IDs in InnoDB in a consistent or controlled manner. These operations require a `trx_sys` mutex call, which is tracked by Performance Schema instrumentation. Operations include management of the transaction system when the database starts or shuts down, rollbacks, undo cleanups, row read access, and buffer pool loads. High database load with a large number of transactions results in the frequent appearance of this wait event. For more information, see synch/mutex/innodb/trx_sys_mutex.

synch/mutex/mysys/KEY_CACHE::cache_lock

The `keycache->cache_lock` mutex controls access to the key cache for MyISAM tables. While Aurora MySQL doesn't allow usage of MyISAM tables to store persistent data, they are used to store internal temporary tables. Consider checking the `created_tmp_tables` or `created_tmp_disk_tables` status counters, because in certain situations, temporary tables are written to disk when they no longer fit in memory.

synch/mutex/sql/FILE_AS_TABLE::LOCK_offsets

The engine acquires this mutex when opening or creating a table metadata file. When this wait event occurs with excessive frequency, the number of tables being created or opened has spiked.

synch/mutex/sql/FILE_AS_TABLE::LOCK_shim_lists

The engine acquires this mutex while performing operations such as `reset_size`, `detach_contents`, or `add_contents` on the internal structure that keeps track of opened tables. The mutex synchronizes access to the list contents. When this wait event occurs with high frequency, it indicates a sudden change in the set of tables that were previously accessed. The engine needs to access new tables or let go of the context related to previously accessed tables.

synch/mutex/sql/LOCK_open

The number of tables that your sessions are opening exceeds the size of the table definition cache or the table open cache. Increase the size of these caches. For more information, see How MySQL opens and closes tables.

synch/mutex/sql/LOCK_table_cache

The number of tables that your sessions are opening exceeds the size of the table definition cache or the table open cache. Increase the size of these caches. For more information, see How MySQL opens and closes tables.

synch/mutex/sql/LOG

In this wait event, there are threads waiting on a log lock. For example, a thread might wait for a lock to write to the slow query log file.

synch/mutex/sql/MYSQL_BIN_LOG::LOCK_commit

In this wait event, there is a thread that is waiting to acquire a lock with the intention of committing to the binary log. Binary logging contention can occur on databases with a very high change rate. Depending on your version of MySQL, there are certain locks being used to protect the consistency and durability of the binary log. In RDS for MySQL, binary logs are used for replication and the automated backup process. In Aurora MySQL, binary logs are not needed for native replication or backups. They are disabled by default but can be enabled and used for external replication or change data capture. For more information, see The binary log in the MySQL documentation.

sync/mutex/sql/MYSQL_BIN_LOG::LOCK_dump_thread_metrics_collection

If binary logging is turned on, the engine acquires this mutex when it prints active dump threads metrics to the engine error log and to the internal operations map.

sync/mutex/sql/MYSQL_BIN_LOG::LOCK_inactive_binlogs_map

If binary logging is turned on, the engine acquires this mutex when it adds to, deletes from, or searches through the list of binlog files behind the latest one.

sync/mutex/sql/MYSQL_BIN_LOG::LOCK_io_cache

If binary logging is turned on, the engine acquires this mutex during Aurora binlog IO cache operations: allocate, resize, free, write, read, purge, and access cache info. If this event occurs frequently, the engine is accessing the cache where binlog events are stored. To reduce wait times, reduce commits. Try grouping multiple statements into a single transaction.

synch/mutex/sql/MYSQL_BIN_LOG::LOCK_log

You have turned on binary logging. There might be high commit throughput, many transactions committing, or replicas reading binlogs. Consider using multirow statements or bundling statements into one transaction. In Aurora, use global databases instead of binary log replication or use the `aurora_binlog_*` parameters.

synch/mutex/sql/SERVER_THREAD::LOCK_sync

The mutex `SERVER_THREAD::LOCK_sync` is acquired during the scheduling, processing, or launching of threads for file writes. The excessive occurrence of this wait event indicates increased write activity in the database.

synch/mutex/sql/TABLESPACES:lock

The engine acquires the `TABLESPACES:lock` mutex during the following tablespace operations: create, delete, truncate, and extend. The excessive occurrence of this wait event indicates a high frequency of tablespace operations. An example is loading a large amount of data into the database.

synch/rwlock/innodb/dict

In this wait event, there are threads waiting on an rwlock held on the InnoDB data dictionary.

synch/rwlock/innodb/dict_operation_lock

In this wait event, there are threads holding locks on InnoDB data dictionary operations.

synch/rwlock/innodb/dict sys RW lock

A high number of concurrent data control language statements (DCLs) in data definition language code (DDLs) are triggered at the same time. Reduce the application's dependency on DDLs during regular application activity.

synch/rwlock/innodb/index_tree_rw_lock

A large number of similar data manipulation language (DML) statements are accessing the same database object at the same time. Try using multirow statements. Also, spread the workload over different database objects. For example, implement partitioning.

synch/sxlock/innodb/dict_operation_lock

A high number of concurrent data control language statements (DCLs) in data definition language code (DDLs) are triggered at the same time. Reduce the application's dependency on DDLs during regular application activity.

synch/sxlock/innodb/dict_sys_lock

A high number of concurrent data control language statements (DCLs) in data definition language code (DDLs) are triggered at the same time. Reduce the application's dependency on DDLs during regular application activity.

synch/sxlock/innodb/hash_table_locks

The session couldn't find pages in the buffer pool. The engine either needs to read a file or modify the least-recently used (LRU) list for the buffer pool. Consider increasing the buffer cache size and improving access paths for the relevant queries.

synch/sxlock/innodb/index_tree_rw_lock

Many similar data manipulation language (DML) statements are accessing the same database object at the same time. Try using multirow statements. Also, spread the workload over different database objects. For example, implement partitioning.

For more information on troubleshooting synch wait events, see Why is my MySQL DB instance showing a high number of active sessions waiting on SYNCH wait events in Performance Insights?.

Aurora MySQL thread states

The following are some common thread states for Aurora MySQL.

checking permissions

The thread is checking whether the server has the required privileges to run the statement.

checking query cache for query

The server is checking whether the current query is present in the query cache.

cleaned up

This is the final state of a connection whose work is complete but which hasn't been closed by the client. The best solution is to explicitly close the connection in code. Or you can set a lower value for `wait_timeout` in your parameter group.

closing tables

The thread is flushing the changed table data to disk and closing the used tables. If this isn't a fast operation, verify the network bandwidth consumption metrics against the instance class network bandwidth. Also, check that the parameter values for `table_open_cache` and `table_definition_cache` parameter allow for enough tables to be simultaneously open so that the engine doesn't need to open and close tables frequently. These parameters influence the memory consumption on the instance.

converting HEAP to MyISAM

The query is converting a temporary table from in-memory to on-disk. This conversion is necessary because the temporary tables created by MySQL in the intermediate steps of query processing grew too big for memory. Check the values of `tmp_table_size` and `max_heap_table_size`. In later versions, this thread state name is `converting HEAP to ondisk`.

converting HEAP to ondisk

The thread is converting an internal temporary table from an in-memory table to an on-disk table.

copy to tmp table

The thread is processing an `ALTER TABLE` statement. This state occurs after the table with the new structure has been created but before rows are copied into it. For a thread in this state, you can use the Performance Schema to obtain information about the progress of the copy operation.

creating sort index

Aurora MySQL is performing a sort because it can't use an existing index to satisfy the `ORDER BY` or `GROUP BY` clause of a query. For more information, see creating sort index.

creating table

The thread is creating a permanent or temporary table.

delayed commit ok done

An asynchronous commit in Aurora MySQL has received an acknowledgement and is complete.

delayed commit ok initiated

The Aurora MySQL thread has started the async commit process but is waiting for acknowledgement. This is usually the genuine commit time of a transaction.

delayed send ok done

An Aurora MySQL worker thread that is tied to a connection can be freed while a response is sent to the client. The thread can begin other work. The state `delayed send ok` means that the asynchronous acknowledgement to the client completed.

delayed send ok initiated

An Aurora MySQL worker thread has sent a response asynchronously to a client and is now free to do work for other connections. The transaction has started an async commit process that hasn't yet been acknowledged.

executing

The thread has begun running a statement.

freeing items

The thread has run a command. Some freeing of items done during this state involves the query cache. This state is usually followed by cleaning up.

init

This state occurs before the initialization of `ALTER TABLE`, `DELETE`, `INSERT`, `SELECT`, or `UPDATE` statements. Actions in this state include flushing the binary log or InnoDB log, and some cleanup of the query cache.

master has sent all binlog to slave

The primary node has finished its part of the replication. The thread is waiting for more queries to run so that it can write to the binary log (binlog).

opening tables

The thread is trying to open a table. This operation is fast unless an `ALTER TABLE` or a `LOCK TABLE` statement needs to finish, or it exceeds the value of `table_open_cache`.

optimizing

The server is performing initial optimizations for a query.

preparing

This state occurs during query optimization.

query end

This state occurs after processing a query but before the freeing items state.

removing duplicates

Aurora MySQL couldn't optimize a `DISTINCT` operation in the early stage of a query. Aurora MySQL must remove all duplicated rows before sending the result to the client.

searching rows for update

The thread is finding all matching rows before updating them. This stage is necessary if the `UPDATE` is changing the index that the engine uses to find the rows.

sending binlog event to slave

The thread read an event from the binary log and is sending it to the replica.

sending cached result to client

The server is taking the result of a query from the query cache and sending it to the client.

sending data

The thread is reading and processing rows for a `SELECT` statement but hasn't yet started sending data to the client. The process is identifying which pages contain the results necessary to satisfy the query. For more information, see sending data.

sending to client

The server is writing a packet to the client. In earlier MySQL versions, this wait event was labeled `writing to net`.

starting

This is the first stage at the beginning of statement execution.

statistics

The server is calculating statistics to develop a query execution plan. If a thread is in this state for a long time, the server is probably disk-bound while performing other work.

storing result in query cache

The server is storing the result of a query in the query cache.

system lock

The thread has called `mysql_lock_tables`, but the thread state hasn't been updated since the call. This general state occurs for many reasons.

update

The thread is preparing to start updating the table.

updating

The thread is searching for rows and is updating them.

user lock

The thread issued a `GET_LOCK` call. The thread either requested an advisory lock and is waiting for it, or is planning to request it.

waiting for more updates

The primary node has finished its part of the replication. The thread is waiting for more queries to run so that it can write to the binary log (binlog).

waiting for schema metadata lock

This is a wait for a metadata lock.

waiting for stored function metadata lock

This is a wait for a metadata lock.

waiting for stored procedure metadata lock

This is a wait for a metadata lock.

waiting for table flush

The thread is executing `FLUSH TABLES` and is waiting for all threads to close their tables. Or the thread received notification that the underlying structure for a table changed, so it must reopen the table to get the new structure. To reopen the table, the thread must wait until all other threads have closed the table. This notification takes place if another thread has used one of the following statements on the table: `FLUSH TABLES`, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE`, or `OPTIMIZE TABLE`.

waiting for table level lock

One session is holding a lock on a table while another session tries to acquire the same lock on the same table.

waiting for table metadata lock

Aurora MySQL uses metadata locking to manage concurrent access to database objects and to ensure data consistency. In this wait event, one session is holding a metadata lock on a table while another session tries to acquire the same lock on the same table. When the

Performance Schema is enabled, this thread state is reported as the wait event `synch/cond/sql/MDL_context::COND_wait_status`.

writing to net

The server is writing a packet to the network. In later MySQL versions, this wait event is labeled `Sending to client`.

Aurora MySQL isolation levels

Learn how DB instances in an Aurora MySQL cluster implement the database property of isolation. This topic explains how the Aurora MySQL default behavior balances between strict consistency and high performance. You can use this information to help you decide when to change the default settings based on the traits of your workload.

Available isolation levels for writer instances

You can use the isolation levels `REPEATABLE READ`, `READ COMMITTED`, `READ UNCOMMITTED`, and `SERIALIZABLE` on the primary instance of an Aurora MySQL DB cluster. These isolation levels work the same in Aurora MySQL as in RDS for MySQL.

REPEATABLE READ isolation level for reader instances

By default, Aurora MySQL DB instances that are configured as read-only Aurora Replicas always use the `REPEATABLE READ` isolation level. These DB instances ignore any `SET TRANSACTION ISOLATION LEVEL` statements and continue using the `REPEATABLE READ` isolation level.

You can't set the isolation level for reader DB instances using DB parameters or DB cluster parameters.

READ COMMITTED isolation level for reader instances

If your application includes a write-intensive workload on the primary instance and long-running queries on the Aurora Replicas, you might experience substantial purge lag. *Purge lag* happens when internal garbage collection is blocked by long-running queries. The symptom that you see is a high value for `history list length` in the output from the `SHOW ENGINE INNODB STATUS` command. You can monitor this value using the `RollbackSegmentHistoryListLength` metric in CloudWatch. Substantial purge lag can reduce the effectiveness of secondary indexes, decrease overall query performance, and lead to wasted storage space.

If you experience such issues, you can set an Aurora MySQL session-level configuration setting, `aurora_read_replica_read_committed`, to use the `READ COMMITTED` isolation level on Aurora Replicas. When you apply this setting, you can help reduce slowdowns and wasted space that can result from performing long-running queries at the same time as transactions that modify your tables.

We recommend making sure that you understand the specific Aurora MySQL behavior of the `READ COMMITTED` isolation before using this setting. The Aurora Replica `READ COMMITTED` behavior complies with the ANSI SQL standard. However, the isolation is less strict than typical MySQL `READ COMMITTED` behavior that you might be familiar with. Therefore, you might see different query results under `READ COMMITTED` on an Aurora MySQL read replica than you might see for the same query under `READ COMMITTED` on the Aurora MySQL primary instance or on RDS for MySQL. You might consider using the `aurora_read_replica_read_committed` setting for such cases as a comprehensive report that scans a very large database. In contrast, you might avoid it for short queries with small result sets, where precision and repeatability are important.

The `READ COMMITTED` isolation level isn't available for sessions within a secondary cluster in an Aurora global database that use the write forwarding feature. For information about write forwarding, see Using write forwarding in an Amazon Aurora global database.

Using READ COMMITTED for readers

To use the `READ COMMITTED` isolation level for Aurora Replicas, set the `aurora_read_replica_read_committed` configuration setting to `ON`. Use this setting at the session level while connected to a specific Aurora Replica. To do so, run the following SQL commands.

```
set session aurora_read_replica_read_committed = ON;
set session transaction isolation level read committed;
```

You might use this configuration setting temporarily to perform interactive, one-time queries. You might also want to run a reporting or data analysis application that benefits from the `READ COMMITTED` isolation level, while leaving the default setting unchanged for other applications.

When the `aurora_read_replica_read_committed` setting is turned on, use the `SET TRANSACTION ISOLATION LEVEL` command to specify the isolation level for the appropriate transactions.

```
set transaction isolation level read committed;
```

Differences in READ COMMITTED behavior on Aurora replicas

The `aurora_read_replica_read_committed` setting makes the `READ COMMITTED` isolation level available for an Aurora Replica, with consistency behavior that is optimized for long-running transactions. The `READ COMMITTED` isolation level on Aurora Replicas has less strict isolation than on Aurora primary instances. For that reason, enable this setting only on Aurora Replicas where you know that your queries can accept the possibility of certain types of inconsistent results.

Your queries can experience certain kinds of read anomalies when the `aurora_read_replica_read_committed` setting is turned on. Two kinds of anomalies are especially important to understand and handle in your application code. A *non-repeatable read* occurs when another transaction commits while your query is running. A long-running query can see different data at the start of the query than it sees at the end. A *phantom read* occurs when other transactions cause existing rows to be reorganized while your query is running, and one or more rows are read twice by your query.

Your queries might experience inconsistent row counts as a result of phantom reads. Your queries might also return incomplete or inconsistent results due to non-repeatable reads. For example, suppose that a join operation refers to tables that are concurrently modified by SQL statements such as `INSERT` or `DELETE`. In this case, the join query might read a row from one table but not the corresponding row from another table.

The ANSI SQL standard allows both of these behaviors for the `READ COMMITTED` isolation level. However, those behaviors are different than the typical MySQL implementation of `READ COMMITTED`. Thus, before enabling the `aurora_read_replica_read_committed` setting, check any existing SQL code to verify if it operates as expected under the looser consistency model.

Row counts and other results might not be strongly consistent under the `READ COMMITTED` isolation level while this setting is enabled. Thus, you typically enable the setting only while running analytic queries that aggregate large amounts of data and don't require absolute precision. If you don't have these kinds of long-running queries alongside a write-intensive workload, you probably don't need the `aurora_read_replica_read_committed` setting. Without the combination of long-running queries and a write-intensive workload, you're unlikely to encounter issues with the length of the history list.

Example Queries showing isolation behavior for READ COMMITTED on Aurora Replicas

The following example shows how `READ COMMITTED` queries on an Aurora Replica might return non-repeatable results if transactions modify the associated tables at the same time. The table

BIG_TABLE contains 1 million rows before any queries start. Other data manipulation language (DML) statements add, remove, or change rows while they are running.

The queries on the Aurora primary instance under the READ COMMITTED isolation level produce predictable results. However, the overhead of keeping the consistent read view for the lifetime of every long-running query can lead to expensive garbage collection later.

The queries on the Aurora Replica under the READ COMMITTED isolation level are optimized to minimize this garbage collection overhead. The tradeoff is that the results might vary depending on whether the queries retrieve rows that are added, removed, or reorganized by transactions that are committed while the query is running. The queries are allowed to consider these rows, but aren't required to. For demonstration purposes, the queries check only the number of rows in the table by using the COUNT(*) function.

Time	DML statement on Aurora primary instance	Query on Aurora primary instance with READ COMMITTED	Query on Aurora Replica with READ COMMITTED
T1	`INSERT INTO big_table SELECT * FROM other_table LIMIT 1000000; COMMIT;`		
T2		**Q1:** `SELECT COUNT(*) FROM big_table;`	**Q2:** `SELECT COUNT(*) FROM big_table;`
T3	`INSERT INTO big_table (c1, c2) VALUES (1, 'one more row'); COMMIT;`		
T4		If Q1 finishes now, result is 1,000,000.	If Q2 finishes now, result is 1,000,000 or 1,000,001.

Time	DML statement on Aurora primary instance	Query on Aurora primary instance with READ COMMITTED	Query on Aurora Replica with READ COMMITTED
T5	`DELETE FROM big_table LIMIT 2; COMMIT;`		
T6		If Q1 finishes now, result is 1,000,000.	If Q2 finishes now, result is 1,000,000 or 1,000,001 or 999,999 or 999,998.
T7	`UPDATE big_table SET c2 = CONCAT(c2 ,c2,c2); COMMIT;`		
T8		If Q1 finishes now, result is 1,000,000.	If Q2 finishes now, result is 1,000,000 or 1,000,001 or 999,999, or possibly some higher number.
T9		**Q3:** `SELECT COUNT(*) FROM big_table;`	**Q4:** `SELECT COUNT(*) FROM big_table;`
T10		If Q3 finishes now, result is 999,999.	If Q4 finishes now, result is 999,999.

Time	DML statement on Aurora primary instance	Query on Aurora primary instance with READ COMMITTED	Query on Aurora Replica with READ COMMITTED
T11		**Q5:** `SELECT COUNT(*) FROM parent_table p JOIN child_table c ON (p.id = c.id) WHERE p.id = 1000;`	**Q6:** `SELECT COUNT(*) FROM parent_table p JOIN child_table c ON (p.id = c.id) WHERE p.id = 1000;`
T12	`INSERT INTO parent_table (id, s) VALUES (1000, 'hello'); INSERT INTO child_table (id, s) VALUES (1000, 'world'); COMMIT;`		
T13		If Q5 finishes now, result is 0.	If Q6 finishes now, result is 0 or 1.

If the queries finish quickly, before any other transactions perform DML statements and commit, the results are predictable and the same between the primary instance and the Aurora Replica. Let's examine the differences in behavior in detail, starting with the first query.

The results for Q1 are highly predictable because `READ COMMITTED` on the primary instance uses a strong consistency model similar to the `REPEATABLE READ` isolation level.

The results for Q2 might vary depending on what transactions are committed while that query is running. For example, suppose that other transactions perform DML statements and commit while the queries are running. In this case, the query on the Aurora Replica with the `READ COMMITTED` isolation level might or might not take the changes into account. The row counts aren't predictable in the same way as under the REPEATABLE READ isolation level. They also aren't as predictable as queries running under the READ COMMITTED isolation level on the primary instance, or on an RDS for MySQL instance.

The UPDATE statement at T7 doesn't actually change the number of rows in the table. However, by changing the length of a variable-length column, this statement can cause rows to be reorganized internally. A long-running READ COMMITTED transaction might see the old version of a row, and later within the same query see the new version of the same row. The query can also skip both the old and new versions of the row, so the row count might be different than expected.

The results of Q5 and Q6 might be identical or slightly different. Query Q6 on the Aurora Replica under READ COMMITTED is able to see, but is not required to see, the new rows that are committed while the query is running. It might also see the row from one table, but not from the other table. If the join query doesn't find a matching row in both tables, it returns a count of zero. If the query does find both the new rows in PARENT_TABLE and CHILD_TABLE, the query returns a count of one. In a long-running query, the lookups from the joined tables might happen at widely separated times.

Note

These differences in behavior depend on the timing of when transactions are committed and when the queries process the underlying table rows. Thus, you're most likely to see such differences in report queries that take minutes or hours and that run on Aurora clusters processing OLTP transactions at the same time. These are the kinds of mixed workloads that benefit the most from the READ COMMITTED isolation level on Aurora Replicas.

Aurora MySQL hints

You can use SQL hints with Aurora MySQL queries to fine-tune performance. You can also use hints to prevent execution plans for important queries from changing because of unpredictable conditions.

Tip

To verify the effect that a hint has on a query, examine the query plan produced by the EXPLAIN statement. Compare the query plans with and without the hint.

In Aurora MySQL version 3, you can use all the hints that are available in MySQL Community Edition 8.0. For more information about these hints, see Optimizer Hints in the *MySQL Reference Manual*.

The following hints are available in Aurora MySQL version 2. These hints apply to queries that use the hash join feature in Aurora MySQL version 2, especially queries that use parallel query optimization.

PQ, NO_PQ

Specifies whether to force the optimizer to use parallel query on a per-table or per-query basis.

PQ forces the optimizer to use parallel query for specified tables or the whole query (block). NO_PQ prevents the optimizer from using parallel query for specified tables or the whole query (block).

This hint is available in Aurora MySQL version 2.11 and higher. The following examples show you how to use this hint.

> **Note**
>
> Specifying a table name forces the optimizer to apply the PQ/NO_PQ hint only on those select tables. Not specifying a table name forces the PQ/NO_PQ hint on all tables affected by the query block.

```
EXPLAIN SELECT /*+ PQ() */ f1, f2
    FROM num1 t1 WHERE f1 > 10 and f2 < 100;

EXPLAIN SELECT /*+ PQ(t1) */ f1, f2
    FROM num1 t1 WHERE f1 > 10 and f2 < 100;

EXPLAIN SELECT /*+ PQ(t1,t2) */ f1, f2
    FROM num1 t1, num1 t2 WHERE t1.f1 = t2.f21;

EXPLAIN SELECT /*+ NO_PQ() */ f1, f2
    FROM num1 t1 WHERE f1 > 10 and f2 < 100;

EXPLAIN SELECT /*+ NO_PQ(t1) */ f1, f2
    FROM num1 t1 WHERE f1 > 10 and f2 < 100;
```

```
EXPLAIN SELECT /*+ NO_PQ(t1,t2) */ f1, f2
    FROM num1 t1, num1 t2 WHERE t1.f1 = t2.f21;
```

HASH_JOIN, NO_HASH_JOIN

Turns on or off the ability of the parallel query optimizer to choose whether to use the hash join optimization method for a query. `HASH_JOIN` lets the optimizer use hash join if that mechanism is more efficient. `NO_HASH_JOIN` prevents the optimizer from using hash join for the query. This hint is available in Aurora MySQL version 2.08 and higher. It has no effect in Aurora MySQL version 3.

The following examples show you how to use this hint.

```
EXPLAIN SELECT/*+ HASH_JOIN(t2) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;

EXPLAIN SELECT /*+ NO_HASH_JOIN(t2) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;
```

HASH_JOIN_PROBING, NO_HASH_JOIN_PROBING

In a hash join query, specifies whether to use the specified table for the probe side of the join. The query tests if column values from the build table exist in the probe table, instead of reading the entire contents of the probe table. You can use `HASH_JOIN_PROBING` and `HASH_JOIN_BUILDING` to specify how hash join queries are processed without reordering the tables within the query text. This hint is available in Aurora MySQL version 2.08 and higher. It has no effect in Aurora MySQL version 3.

The following examples show how to use this hint. Specifying the `HASH_JOIN_PROBING` hint for the table T2 has the same effect as specifying `NO_HASH_JOIN_PROBING` for the table `T1`.

```
EXPLAIN SELECT /*+ HASH_JOIN(t2) HASH_JOIN_PROBING(t2) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;

EXPLAIN SELECT /*+ HASH_JOIN(t2) NO_HASH_JOIN_PROBING(t1) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;
```

HASH_JOIN_BUILDING, NO_HASH_JOIN_BUILDING

In a hash join query, specifies whether to use the specified table for the build side of the join. The query processes all the rows from this table to build the list of column values to cross-reference with the other table. You can use HASH_JOIN_PROBING and HASH_JOIN_BUILDING to specify how hash join queries are processed without reordering the tables within the query text. This hint is available in Aurora MySQL version 2.08 and higher. It has no effect in Aurora MySQL version 3.

The following example shows you how to use this hint. Specifying the HASH_JOIN_BUILDING hint for the table T2 has the same effect as specifying NO_HASH_JOIN_BUILDING for the table T1.

```
EXPLAIN SELECT /*+ HASH_JOIN(t2) HASH_JOIN_BUILDING(t2) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;

EXPLAIN SELECT /*+ HASH_JOIN(t2) NO_HASH_JOIN_BUILDING(t1) */ f1, f2
  FROM t1, t2 WHERE t1.f1 = t2.f1;
```

JOIN_FIXED_ORDER

Specifies that tables in the query are joined based on the order they are listed in the query. It is useful with queries involving three or more tables. It is intended as a replacement for the MySQL STRAIGHT_JOIN hint and is equivalent to the MySQL JOIN_FIXED_ORDER hint. This hint is available in Aurora MySQL version 2.08 and higher.

The following example shows you how to use this hint.

```
EXPLAIN SELECT /*+ JOIN_FIXED_ORDER() */ f1, f2
  FROM t1 JOIN t2 USING (id) JOIN t3 USING (id) JOIN t4 USING (id);
```

JOIN_ORDER

Specifies the join order for the tables in the query. It is useful with queries involving three or more tables. It is equivalent to the MySQL JOIN_ORDER hint. This hint is available in Aurora MySQL version 2.08 and higher.

The following example shows you how to use this hint.

```
EXPLAIN SELECT /*+ JOIN_ORDER (t4, t2, t1, t3) */ f1, f2
  FROM t1 JOIN t2 USING (id) JOIN t3 USING (id) JOIN t4 USING (id);
```

JOIN_PREFIX

Specifies the tables to put first in the join order. It is useful with queries involving three or more tables. It is equivalent to the MySQL JOIN_PREFIX hint. This hint is available in Aurora MySQL version 2.08 and higher.

The following example shows you how to use this hint.

```
EXPLAIN SELECT /*+ JOIN_PREFIX (t4, t2) */ f1, f2
  FROM t1 JOIN t2 USING (id) JOIN t3 USING (id) JOIN t4 USING (id);
```

JOIN_SUFFIX

Specifies the tables to put last in the join order. It is useful with queries involving three or more tables. It is equivalent to the MySQL JOIN_SUFFIX hint. This hint is available in Aurora MySQL version 2.08 and higher.

The following example shows you how to use this hint.

```
EXPLAIN SELECT /*+ JOIN_SUFFIX (t1) */ f1, f2
  FROM t1 JOIN t2 USING (id) JOIN t3 USING (id) JOIN t4 USING (id);
```

For information about using hash join queries, see Optimizing large Aurora MySQL join queries with hash joins.

Aurora MySQL stored procedures

You can manage your Aurora MySQL DB cluster by calling built-in stored procedures.

Topics

- Configuring
- Ending a session or query
- Logging
- Managing the Global Status History
- Replicating

Configuring

The following stored procedures set and show configuration parameters, such as for binary log file retention.

Topics

- mysql.rds_set_configuration
- mysql.rds_show_configuration

mysql.rds_set_configuration

Specifies the number of hours to retain binary logs or the number of seconds to delay replication.

Syntax

```
CALL mysql.rds_set_configuration(name,value);
```

Parameters

name

The name of the configuration parameter to set.

value

The value of the configuration parameter.

Usage notes

The `mysql.rds_set_configuration` procedure supports the following configuration parameters:

- binlog retention hours

The configuration parameters are stored permanently and survive any DB instance reboot or failover.

binlog retention hours

The `binlog retention hours` parameter is used to specify the number of hours to retain binary log files. Amazon Aurora normally purges a binary log as soon as possible, but the binary log might still be required for replication with a MySQL database external to Aurora.

The default value of `binlog retention hours` is `NULL`. For Aurora MySQL, `NULL` means binary logs are cleaned up lazily. Aurora MySQL binary logs might remain in the system for a certain period, which is usually not longer than a day.

To specify the number of hours to retain binary logs on a DB cluster, use the `mysql.rds_set_configuration` stored procedure and specify a period with enough time for replication to occur, as shown in the following example.

```
call mysql.rds_set_configuration('binlog retention hours', 24);
```

> **Note**
>
> You can't use the value `0` for `binlog retention hours`.

For Aurora MySQL version 2.11.0 and higher and version 3 DB clusters, the maximum `binlog retention hours` value is 2160 (90 days).

After you set the retention period, monitor storage usage for the DB instance to make sure that the retained binary logs don't take up too much storage.

mysql.rds_show_configuration

The number of hours that binary logs are retained.

Syntax

```
CALL mysql.rds_show_configuration;
```

Usage notes

To verify the number of hours that Amazon RDS retains binary logs, use the `mysql.rds_show_configuration` stored procedure.

Examples

The following example displays the retention period:

```
call mysql.rds_show_configuration;
                name                         value     description
                binlog retention hours       24        binlog retention hours specifies
 the duration in hours before binary logs are automatically deleted.
```

Ending a session or query

The following stored procedures end a session or query.

Topics

- mysql.rds_kill
- mysql.rds_kill_query

mysql.rds_kill

Ends a connection to the MySQL server.

Syntax

```
CALL mysql.rds_kill(processID);
```

Parameters

processID

The identity of the connection thread to be ended.

Usage notes

Each connection to the MySQL server runs in a separate thread. To end a connection, use the `mysql.rds_kill` procedure and pass in the thread ID of that connection. To obtain the thread ID, use the MySQL SHOW PROCESSLIST command.

Examples

The following example ends a connection with a thread ID of 4243:

```
CALL mysql.rds_kill(4243);
```

mysql.rds_kill_query

Ends a query running against the MySQL server.

Syntax

```
CALL mysql.rds_kill_query(processID);
```

Parameters

processID

The identity of the process or thread that is running the query to be ended.

Usage notes

To stop a query running against the MySQL server, use the mysql_rds_kill_query procedure and pass in the connection ID of the thread that is running the query. The procedure then terminates the connection.

To obtain the ID, query the MySQL INFORMATION_SCHEMA PROCESSLIST table or use the MySQL SHOW PROCESSLIST command. The value in the ID column from SHOW PROCESSLIST or SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST is the *processID*.

Examples

The following example stops a query with a query thread ID of 230040:

```
CALL mysql.rds_kill_query(230040);
```

Logging

The following stored procedures rotate MySQL logs to backup tables. For more information, see Aurora MySQL database log files.

Topics

- mysql.rds_rotate_general_log
- mysql.rds_rotate_slow_log

mysql.rds_rotate_general_log

Rotates the `mysql.general_log` table to a backup table.

Syntax

```
CALL mysql.rds_rotate_general_log;
```

Usage notes

You can rotate the `mysql.general_log` table to a backup table by calling the `mysql.rds_rotate_general_log` procedure. When log tables are rotated, the current log table is copied to a backup log table and the entries in the current log table are removed. If a backup log table already exists, then it is deleted before the current log table is copied to the backup. You can query the backup log table if needed. The backup log table for the `mysql.general_log` table is named `mysql.general_log_backup`.

You can run this procedure only when the `log_output` parameter is set to `TABLE`.

mysql.rds_rotate_slow_log

Rotates the `mysql.slow_log` table to a backup table.

Syntax

```
CALL mysql.rds_rotate_slow_log;
```

Usage notes

You can rotate the `mysql.slow_log` table to a backup table by calling the `mysql.rds_rotate_slow_log` procedure. When log tables are rotated, the current log table is copied to a backup log table and the entries in the current log table are removed. If a backup log table already exists, then it is deleted before the current log table is copied to the backup.

You can query the backup log table if needed. The backup log table for the `mysql.slow_log` table is named `mysql.slow_log_backup`.

Managing the Global Status History

Amazon RDS provides a set of procedures that take snapshots of the values of status variables over time and write them to a table, along with any changes since the last snapshot. This infrastructure is called Global Status History. For more information, see Managing the Global Status History.

The following stored procedures manage how the Global Status History is collected and maintained.

Topics

mysql.rds_collect_global_status_history

Takes a snapshot on demand for the Global Status History.

Syntax

```
CALL mysql.rds_collect_global_status_history;
```

mysql.rds_disable_gsh_collector

Turns off snapshots taken by the Global Status History.

Syntax

```
CALL mysql.rds_disable_gsh_collector;
```

mysql.rds_disable_gsh_rotation

Turns off rotation of the `mysql.global_status_history` table.

Syntax

```
CALL mysql.rds_disable_gsh_rotation;
```

mysql.rds_enable_gsh_collector

Turns on the Global Status History to take default snapshots at intervals specified by `rds_set_gsh_collector`.

Syntax

```
CALL mysql.rds_enable_gsh_collector;
```

mysql.rds_enable_gsh_rotation

Turns on rotation of the contents of the `mysql.global_status_history` table to `mysql.global_status_history_old` at intervals specified by `rds_set_gsh_rotation`.

Syntax

```
CALL mysql.rds_enable_gsh_rotation;
```

mysql.rds_rotate_global_status_history

Rotates the contents of the `mysql.global_status_history` table to `mysql.global_status_history_old` on demand.

Syntax

```
CALL mysql.rds_rotate_global_status_history;
```

mysql.rds_set_gsh_collector

Specifies the interval, in minutes, between snapshots taken by the Global Status History.

Syntax

```
CALL mysql.rds_set_gsh_collector(intervalPeriod);
```

Parameters

intervalPeriod

The interval, in minutes, between snapshots. Default value is 5.

mysql.rds_set_gsh_rotation

Specifies the interval, in days, between rotations of the `mysql.global_status_history` table.

Syntax

```
CALL mysql.rds_set_gsh_rotation(intervalPeriod);
```

Parameters

intervalPeriod

The interval, in days, between table rotations. Default value is 7.

Replicating

You can call the following stored procedures while connected to the primary instance in an Aurora MySQL cluster. These procedures control how transactions are replicated from an external database into Aurora MySQL, or from Aurora MySQL to an external database. To learn how to use replication based on global transaction identifiers (GTIDs) with Aurora MySQL, see Using GTID-based replication.

Topics

- mysql.rds_start_replication_until_gtid (Aurora MySQL version 3)
- mysql.rds_stop_replication

mysql.rds_assign_gtids_to_anonymous_transactions (Aurora MySQL version 3)

Configures the ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS option of the CHANGE REPLICATION SOURCE TO statement. It makes the replication channel assign a GTID to replicated transactions that don't have one. That way, you can perform binary log replication from a source that doesn't use GTID-based replication to a replica that does. For more information, see CHANGE REPLICATION SOURCE TO Statement and Replication From a Source Without GTIDs to a Replica With GTIDs in the *MySQL Reference Manual*.

Syntax

```
CALL mysql.rds_assign_gtids_to_anonymous_transactions(gtid_option);
```

Parameters

gtid_option

String value. The allowed values are OFF, LOCAL, or a specified UUID.

Usage notes

This procedure has the same effect as issuing the statement CHANGE REPLICATION SOURCE TO ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = *gtid_option* in community MySQL.

GTID must be turned to ON for *gtid_option* to be set to LOCAL or a specific UUID.

The default is OFF, meaning that the feature isn't used.

LOCAL assigns a GTID including the replica's own UUID (the server_uuid setting).

Passing a parameter that is a UUID assigns a GTID that includes the specified UUID, such as the server_uuid setting for the replication source server.

Examples

To turn off this feature:

```
mysql> call mysql.rds_assign_gtids_to_anonymous_transactions('OFF');
```

```
+-------------------------------------------------------------+
| Message  |
+-------------------------------------------------------------+
| ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS has been set to: OFF |
+-------------------------------------------------------------+
1 row in set (0.07 sec)
```

To use the replica's own UUID:

```
mysql> call mysql.rds_assign_gtids_to_anonymous_transactions('LOCAL');
+---------------------------------------------------------------+
| Message  |
+---------------------------------------------------------------+
| ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS has been set to: LOCAL |
+---------------------------------------------------------------+
1 row in set (0.07 sec)
```

To use a specified UUID:

```
mysql> call mysql.rds_assign_gtids_to_anonymous_transactions('317a4760-
f3dd-3b74-8e45-0615ed29de0e');
+----------------------------------------------------------------------------------------------
+
| Message |
+----------------------------------------------------------------------------------------------
+
| ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS has been set to: 317a4760-
f3dd-3b74-8e45-0615ed29de0e |
+----------------------------------------------------------------------------------------------
+
1 row in set (0.07 sec)
```

mysql.rds_disable_session_binlog (Aurora MySQL version 2)

Turns off binary logging for the current session by setting the `sql_log_bin` variable to `OFF`.

Syntax

```
CALL mysql.rds_disable_session_binlog;
```

Parameters

None

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

For Aurora, this procedure is supported for Aurora MySQL version 2.12 and higher MySQL 5.7–compatible versions.

> **Note**
>
> In Aurora MySQL version 3, you can use the following command to disable binary logging for the current session if you have the `SESSION_VARIABLES_ADMIN` privilege:
>
> ```
> SET SESSION sql_log_bin = OFF;
> ```

mysql.rds_enable_session_binlog (Aurora MySQL version 2)

Turns on binary logging for the current session by setting the `sql_log_bin` variable to `ON`.

Syntax

```
CALL mysql.rds_enable_session_binlog;
```

Parameters

None

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

For Aurora, this procedure is supported for Aurora MySQL version 2.12 and higher MySQL 5.7–compatible versions.

> **Note**
>
> In Aurora MySQL version 3, you can use the following command to enable binary logging for the current session if you have the `SESSION_VARIABLES_ADMIN` privilege:

```
SET SESSION sql_log_bin = ON;
```

mysql.rds_gtid_purged (Aurora MySQL version 3)

Sets the global value of the system variable `gtid_purged` to a given global transaction identifier (GTID) set. The `gtid_purged` system variable is a GTID set that consists of the GTIDs of all transactions that have been committed on the server, but don't exist in any binary log file on the server.

To allow compatibility with MySQL 8.0, there are two ways to set the value of `gtid_purged`:

- Replace the value of `gtid_purged` with your specified GTID set.
- Append your specified GTID set to the GTID set that `gtid_purged` already contains.

Syntax

To replace the value of `gtid_purged` with your specified GTID set:

```
CALL mysql.rds_gtid_purged (gtid_set);
```

To append the value of `gtid_purged` to your specified GTID set:

```
CALL mysql.rds_gtid_purged (+gtid_set);
```

Parameters

`gtid_set`

The value of *`gtid_set`* must be a superset of the current value of `gtid_purged`, and can't intersect with `gtid_subtract(gtid_executed,gtid_purged)`. That is, the new GTID set must include any GTIDs that were already in `gtid_purged`, and can't include any GTIDs in `gtid_executed` that haven't yet been purged. The *`gtid_set`* parameter also can't include any GTIDs that are in the global `gtid_owned` set, the GTIDs for transactions that are currently being processed on the server.

Usage notes

The master user must run the `mysql.rds_gtid_purged` procedure.

This procedure is supported for Aurora MySQL version 3.04 and higher.

Examples

The following example assigns the GTID `3E11FA47-71CA-11E1-9E33-C80AA9429562:23` to the `gtid_purged` global variable.

```
CALL mysql.rds_gtid_purged('3E11FA47-71CA-11E1-9E33-C80AA9429562:23');
```

mysql.rds_import_binlog_ssl_material

Imports the certificate authority certificate, client certificate, and client key into an Aurora MySQL DB cluster. The information is required for SSL communication and encrypted replication.

> **Note**
>
> Currently, this procedure is supported for Aurora MySQL version 2: 2.09.2, 2.10.0, 2.10.1, and 2.11.0; and version 3: 3.01.1 and higher.

Syntax

```
CALL mysql.rds_import_binlog_ssl_material (
  ssl_material
);
```

Parameters

`ssl_material`

JSON payload that contains the contents of the following .pem format files for a MySQL client:

- "ssl_ca":"*`Certificate authority certificate`*"
- "ssl_cert":"*`Client certificate`*"
- "ssl_key":"*`Client key`*"

Usage notes

Prepare for encrypted replication before you run this procedure:

- If you don't have SSL enabled on the external MySQL source database instance and don't have a client key and client certificate prepared, enable SSL on the MySQL database server and generate the required client key and client certificate.
- If SSL is enabled on the external source database instance, supply a client key and certificate for the Aurora MySQL DB cluster. If you don't have these, generate a new key and certificate for the Aurora MySQL DB cluster. To sign the client certificate, you must have the certificate authority key you used to configure SSL on the external MySQL source database instance.

For more information, see [Creating SSL certificates and keys using openssl](#) in the MySQL documentation.

> ⚠ **Important**
>
> After you prepare for encrypted replication, use an SSL connection to run this procedure. The client key must not be transferred across an insecure connection.

This procedure imports SSL information from an external MySQL database into an Aurora MySQL DB cluster. The SSL information is in .pem format files that contain the SSL information for the Aurora MySQL DB cluster. During encrypted replication, the Aurora MySQL DB cluster acts a client to the MySQL database server. The certificates and keys for the Aurora MySQL client are in files in .pem format.

You can copy the information from these files into the `ssl_material` parameter in the correct JSON payload. To support encrypted replication, import this SSL information into the Aurora MySQL DB cluster.

The JSON payload must be in the following format.

```
'{"ssl_ca":"-----BEGIN CERTIFICATE-----
ssl_ca_pem_body_code
-----END CERTIFICATE-----\n","ssl_cert":"-----BEGIN CERTIFICATE-----
ssl_cert_pem_body_code
-----END CERTIFICATE-----\n","ssl_key":"-----BEGIN RSA PRIVATE KEY-----
ssl_key_pem_body_code
-----END RSA PRIVATE KEY-----\n"}'
```

Examples

The following example imports SSL information into an Aurora MySQL. In .pem format files, the body code typically is longer than the body code shown in the example.

```
call mysql.rds_import_binlog_ssl_material(
'{"ssl_ca":"-----BEGIN CERTIFICATE-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END CERTIFICATE-----\n","ssl_cert":"-----BEGIN CERTIFICATE-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END CERTIFICATE-----\n","ssl_key":"-----BEGIN RSA PRIVATE KEY-----
AAAAB3NzaC1yc2EAAAADAQABAAABAQClKsfkNkuSevGj3eYhCe53pcjqP3maAhDFcvBS7O6V
hz2ItxCih+PnDSUaw+WNQn/mZphTk/a/gU8jEzoOWbkM4yxyb/wB96xbiFveSFJuOp/d6RJhJOI0iBXr
lsLnBItntckiJ7FbtxJMXLvvwJryDUilBMTjYtwB+QhYXUMOzce5Pjz5/i8SeJtjnV3iAoG/cQk+0FzZ
qaeJAAHco+CY/5WrUBkrHmFJr6HcXkvJdWPkYQS3xqC0+FmUZofz221CBt5IMucxXPkX4rWi+z7wB3Rb
BQoQzd8v7yeb7OzlPnWOyN0qFU0XA246RA8QFYiCNYwI3f05p6KLxEXAMPLE
-----END RSA PRIVATE KEY-----\n"}');
```

mysql.rds_next_master_log (Aurora MySQL version 2)

Changes the source database instance log position to the start of the next binary log on the source database instance. Use this procedure only if you are receiving replication I/O error 1236 on a read replica.

Syntax

```
CALL mysql.rds_next_master_log(
curr_master_log
);
```

Parameters

curr_master_log

The index of the current master log file. For example, if the current file is named mysql-bin-changelog.012345, then the index is 12345. To determine the current master log file name, run the SHOW REPLICA STATUS command and view the Master_Log_File field.

> **Note**
>
> Previous versions of MySQL used SHOW SLAVE STATUS instead of SHOW REPLICA STATUS. If you are using a MySQL version before 8.0.23, then use SHOW SLAVE STATUS.

Usage notes

The master user must run the mysql.rds_next_master_log procedure.

> **Warning**
>
> Call mysql.rds_next_master_log only if replication fails after a failover of a Multi-AZ DB instance that is the replication source, and the Last_IO_Errno field of SHOW REPLICA STATUS reports I/O error 1236.
> Calling mysql.rds_next_master_log can result in data loss in the read replica if transactions in the source instance were not written to the binary log on disk before the failover event occurred.

Examples

Assume replication fails on an Aurora MySQL read replica. Running SHOW REPLICA STATUS\G on the read replica returns the following result:

```
*************************** 1. row ***************************
             Replica_IO_State:
                  Source_Host: myhost.XXXXXXXXXXXXXXX.rr-rrrr-1.rds.amazonaws.com
                  Source_User: MasterUser
                  Source_Port: 3306
```

```
                Connect_Retry: 10
              Source_Log_File: mysql-bin-changelog.012345
          Read_Source_Log_Pos: 1219393
               Relay_Log_File: relaylog.012340
                Relay_Log_Pos: 30223388
        Relay_Source_Log_File: mysql-bin-changelog.012345
           Replica_IO_Running: No
          Replica_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Source_Log_Pos: 30223232
              Relay_Log_Space: 5248928866
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Source_SSL_Allowed: No
           Source_SSL_CA_File:
           Source_SSL_CA_Path:
              Source_SSL_Cert:
            Source_SSL_Cipher:
               Source_SSL_Key:
        Seconds_Behind_Master: NULL
Source_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 1236
                Last_IO_Error: Got fatal error 1236 from master when reading data from
 binary log: 'Client requested master to start replication from impossible position;
 the first event 'mysql-bin-changelog.013406' at 1219393, the last event read from
 '/rdsdbdata/log/binlog/mysql-bin-changelog.012345' at 4, the last byte read from '/
rdsdbdata/log/binlog/mysql-bin-changelog.012345' at 4.'
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids:
             Source_Server_Id: 67285976
```

The `Last_IO_Errno` field shows that the instance is receiving I/O error 1236. The `Master_Log_File` field shows that the file name is `mysql-bin-changelog.012345`, which means that the log file index is 12345. To resolve the error, you can call mysql.rds_next_master_log with the following parameter:

```
CALL mysql.rds_next_master_log(12345);
```

Note

Previous versions of MySQL used SHOW SLAVE STATUS instead of SHOW REPLICA STATUS. If you are using a MySQL version before 8.0.23, then use SHOW SLAVE STATUS.

mysql.rds_next_source_log (Aurora MySQL version 3)

Changes the source database instance log position to the start of the next binary log on the source database instance. Use this procedure only if you are receiving replication I/O error 1236 on a read replica.

Syntax

```
CALL mysql.rds_next_source_log(
curr_source_log
);
```

Parameters

curr_source_log

The index of the current source log file. For example, if the current file is named mysql-bin-changelog.012345, then the index is 12345. To determine the current source log file name, run the SHOW REPLICA STATUS command and view the Source_Log_File field.

Usage notes

The master user must run the mysql.rds_next_source_log procedure.

Warning

Call `mysql.rds_next_source_log` only if replication fails after a failover of a Multi-AZ DB instance that is the replication source, and the `Last_IO_Errno` field of `SHOW REPLICA STATUS` reports I/O error 1236.

Calling `mysql.rds_next_source_log` can result in data loss in the read replica if transactions in the source instance were not written to the binary log on disk before the failover event occurred.

Examples

Assume replication fails on an Aurora MySQL read replica. Running `SHOW REPLICA STATUS\G` on the read replica returns the following result:

```
*************************** 1. row ***************************
             Replica_IO_State:
                  Source_Host: myhost.XXXXXXXXXXXXXXX.rr-rrrr-1.rds.amazonaws.com
                  Source_User: MasterUser
                  Source_Port: 3306
                Connect_Retry: 10
              Source_Log_File: mysql-bin-changelog.012345
          Read_Source_Log_Pos: 1219393
               Relay_Log_File: relaylog.012340
                Relay_Log_Pos: 30223388
        Relay_Source_Log_File: mysql-bin-changelog.012345
           Replica_IO_Running: No
          Replica_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Source_Log_Pos: 30223232
              Relay_Log_Space: 5248928866
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Source_SSL_Allowed: No
           Source_SSL_CA_File:
           Source_SSL_CA_Path:
              Source_SSL_Cert:
            Source_SSL_Cipher:
               Source_SSL_Key:
        Seconds_Behind_Source: NULL
Source_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 1236
                Last_IO_Error: Got fatal error 1236 from source when reading data from
 binary log: 'Client requested source to start replication from impossible position;
 the first event 'mysql-bin-changelog.013406' at 1219393, the last event read from
 '/rdsdbdata/log/binlog/mysql-bin-changelog.012345' at 4, the last byte read from '/
rdsdbdata/log/binlog/mysql-bin-changelog.012345' at 4.'
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids:
             Source_Server_Id: 67285976
```

The `Last_IO_Errno` field shows that the instance is receiving I/O error 1236. The `Source_Log_File` field shows that the file name is `mysql-bin-changelog.012345`, which means that the log file index is `12345`. To resolve the error, you can call `mysql.rds_next_source_log` with the following parameter:

```
CALL mysql.rds_next_source_log(12345);
```

mysql.rds_remove_binlog_ssl_material

Removes the certificate authority certificate, client certificate, and client key for SSL communication and encrypted replication. This information is imported by using [mysql.rds_import_binlog_ssl_material](#).

Syntax

```
CALL mysql.rds_remove_binlog_ssl_material;
```

mysql.rds_reset_external_master (Aurora MySQL version 2)

Reconfigures an Aurora MySQL DB instance to no longer be a read replica of an instance of MySQL running external to Amazon RDS.

Important

To run this procedure, `autocommit` must be enabled. To enable it, set the `autocommit` parameter to 1. For information about modifying parameters, see Modifying parameters in a DB parameter group.

Syntax

```
CALL mysql.rds_reset_external_master;
```

Usage notes

The master user must run the `mysql.rds_reset_external_master` procedure. This procedure must be run on the MySQL DB instance to be removed as a read replica of a MySQL instance running external to Amazon RDS.

Note

We offer these stored procedures primarily to enable replication with MySQL instances running external to Amazon RDS. We recommend that you use Aurora Replicas to manage replication within an Aurora MySQL DB cluster when possible. For information about managing replication in Aurora MySQL DB clusters, see Using Aurora Replicas.

For more information about using replication to import data from an instance of MySQL running external to Aurora MySQL, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

mysql.rds_reset_external_source (Aurora MySQL version 3)

Reconfigures an Aurora MySQL DB instance to no longer be a read replica of an instance of MySQL running external to Amazon RDS.

Important

To run this procedure, autocommit must be enabled. To enable it, set the autocommit parameter to 1. For information about modifying parameters, see Modifying parameters in a DB parameter group.

Syntax

```
CALL mysql.rds_reset_external_source;
```

Usage notes

The master user must run the mysql.rds_reset_external_source procedure. This procedure must be run on the MySQL DB instance to be removed as a read replica of a MySQL instance running external to Amazon RDS.

Note

We offer these stored procedures primarily to enable replication with MySQL instances running external to Amazon RDS. We recommend that you use Aurora Replicas to manage replication within an Aurora MySQL DB cluster when possible. For information about managing replication in Aurora MySQL DB clusters, see Using Aurora Replicas.

mysql.rds_set_binlog_source_ssl (Aurora MySQL version 3)

Enables SOURCE_SSL encryption for binlog replication. For more information, see CHANGE REPLICATION SOURCE TO statement in the MySQL documentation.

Syntax

```
CALL mysql.rds_set_binlog_source_ssl(mode);
```

Parameters

mode

A value that indicates whether SOURCE_SSL encryption is enabled:

- `0` – `SOURCE_SSL` encryption is disabled. The default is `0`.
- 1 – `SOURCE_SSL` encryption is enabled. You can configure encryption using SSL or TLS.

Usage notes

This procedure is supported for Aurora MySQL version 3.06 and higher.

mysql.rds_set_external_master (Aurora MySQL version 2)

Configures an Aurora MySQL DB instance to be a read replica of an instance of MySQL running external to Amazon RDS.

The `mysql.rds_set_external_master` procedure is deprecated and will be removed in a future release. Use `mysql.rds_set_external_source` instead.

> **Important**
>
> To run this procedure, `autocommit` must be enabled. To enable it, set the `autocommit` parameter to 1. For information about modifying parameters, see Modifying parameters in a DB parameter group.

Syntax

```
CALL mysql.rds_set_external_master (
  host_name
  , host_port
  , replication_user_name
  , replication_user_password
  , mysql_binary_log_file_name
  , mysql_binary_log_file_location
  , ssl_encryption
);
```

Parameters

`host_name`

The host name or IP address of the MySQL instance running external to Amazon RDS to become the source database instance.

host_port

The port used by the MySQL instance running external to Amazon RDS to be configured as the source database instance. If your network configuration includes Secure Shell (SSH) port replication that converts the port number, specify the port number that is exposed by SSH.

replication_user_name

The ID of a user with `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the MySQL instance running external to Amazon RDS. We recommend that you provide an account that is used solely for replication with the external instance.

replication_user_password

The password of the user ID specified in `replication_user_name`.

mysql_binary_log_file_name

The name of the binary log on the source database instance that contains the replication information.

mysql_binary_log_file_location

The location in the `mysql_binary_log_file_name` binary log at which replication starts reading the replication information.

You can determine the binlog file name and location by running `SHOW MASTER STATUS` on the source database instance.

ssl_encryption

A value that specifies whether Secure Socket Layer (SSL) encryption is used on the replication connection. 1 specifies to use SSL encryption, 0 specifies to not use encryption. The default is 0.

> **Note**
>
> The `MASTER_SSL_VERIFY_SERVER_CERT` option isn't supported. This option is set to 0, which means that the connection is encrypted, but the certificates aren't verified.

Usage notes

The master user must run the `mysql.rds_set_external_master` procedure. This procedure must be run on the MySQL DB instance to be configured as the read replica of a MySQL instance running external to Amazon RDS.

Before you run `mysql.rds_set_external_master`, you must configure the instance of MySQL running external to Amazon RDS to be a source database instance. To connect to the MySQL instance running external to Amazon RDS, you must specify `replication_user_name` and `replication_user_password` values that indicate a replication user that has `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the external instance of MySQL.

To configure an external instance of MySQL as a source database instance

1. Using the MySQL client of your choice, connect to the external instance of MySQL and create a user account to be used for replication. The following is an example.

   MySQL 5.7

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED BY 'password';
   ```

   MySQL 8.0

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED WITH mysql_native_password BY
    'password';
   ```

   > **Note**
   >
   > Specify a password other than the prompt shown here as a security best practice.

2. On the external instance of MySQL, grant `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges to your replication user. The following example grants `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges on all databases for the 'repl_user' user for your domain.

   MySQL 5.7

   ```
   GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com'
    IDENTIFIED BY 'password';
   ```

   MySQL 8.0

   ```
   GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com';
   ```

To use encrypted replication, configure source database instance to use SSL connections. Also, import the certificate authority certificate, client certificate, and client key into the DB instance or DB cluster using the mysql.rds_import_binlog_ssl_material procedure.

> **Note**
>
> We offer these stored procedures primarily to enable replication with MySQL instances running external to Amazon RDS. We recommend that you use Aurora Replicas to manage replication within an Aurora MySQL DB cluster when possible. For information about managing replication in Aurora MySQL DB clusters, see Using Aurora Replicas.

After calling `mysql.rds_set_external_master` to configure an Amazon RDS DB instance as a read replica, you can call mysql.rds_start_replication on the read replica to start the replication process. You can call mysql.rds_reset_external_master (Aurora MySQL version 2) to remove the read replica configuration.

When `mysql.rds_set_external_master` is called, Amazon RDS records the time, user, and an action of `set master` in the `mysql.rds_history` and `mysql.rds_replication_status` tables.

Examples

When run on a MySQL DB instance, the following example configures the DB instance to be a read replica of an instance of MySQL running external to Amazon RDS.

```
call mysql.rds_set_external_master(
  'Externaldb.some.com',
  3306,
  'repl_user',
  'password',
  'mysql-bin-changelog.0777',
  120,
  0);
```

mysql.rds_set_external_master_with_auto_position (Aurora MySQL version 2)

Configures an Aurora MySQL primary instance to accept incoming replication from an external MySQL instance. This procedure also configures replication based on global transaction identifiers (GTIDs).

This procedure doesn't configure delayed replication, because Aurora MySQL doesn't support delayed replication.

Syntax

```
CALL mysql.rds_set_external_master_with_auto_position (
  host_name
  , host_port
  , replication_user_name
  , replication_user_password
  , ssl_encryption
);
```

Parameters

`host_name`

The host name or IP address of the MySQL instance running external to Aurora to become the replication master.

`host_port`

The port used by the MySQL instance running external to Aurora to be configured as the replication master. If your network configuration includes Secure Shell (SSH) port replication that converts the port number, specify the port number that is exposed by SSH.

`replication_user_name`

The ID of a user with `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the MySQL instance running external to Aurora. We recommend that you provide an account that is used solely for replication with the external instance.

`replication_user_password`

The password of the user ID specified in `replication_user_name`.

`ssl_encryption`

This option isn't currently implemented. The default is 0.

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

The master user must run the mysql.rds_set_external_master_with_auto_position procedure. The master user runs this procedure on the primary instance of an Aurora MySQL DB cluster that acts as a replication target. This can be the replication target of an external MySQL DB instance or an Aurora MySQL DB cluster.

This procedure is supported for Aurora MySQL version 2. For Aurora MySQL version 3, use the procedure mysql.rds_set_external_source_with_auto_position (Aurora MySQL version 3) instead.

Before you run mysql.rds_set_external_master_with_auto_position, configure the external MySQL DB instance to be a replication master. To connect to the external MySQL instance, specify values for replication_user_name and replication_user_password. These values must indicate a replication user that has REPLICATION CLIENT and REPLICATION SLAVE permissions on the external MySQL instance.

To configure an external MySQL instance as a replication master

1. Using the MySQL client of your choice, connect to the external MySQL instance and create a user account to be used for replication. The following is an example.

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED BY 'SomePassW0rd'
   ```

2. On the external MySQL instance, grant REPLICATION CLIENT and REPLICATION SLAVE privileges to your replication user. The following example grants REPLICATION CLIENT and REPLICATION SLAVE privileges on all databases for the 'repl_user' user for your domain.

   ```
   GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com'
   IDENTIFIED BY 'SomePassW0rd'
   ```

When you call mysql.rds_set_external_master_with_auto_position, Amazon RDS records certain information. This information is the time, the user, and an action of "set master" in the mysql.rds_history and mysql.rds_replication_status tables.

To skip a specific GTID-based transaction that is known to cause a problem, you can use the mysql.rds_skip_transaction_with_gtid stored procedure. For more information about working with GTID-based replication, see Using GTID-based replication.

Examples

When run on an Aurora primary instance, the following example configures the Aurora cluster to act as a read replica of an instance of MySQL running external to Aurora.

```
call mysql.rds_set_external_master_with_auto_position(
  'Externaldb.some.com',
  3306,
  'repl_user'@'mydomain.com',
  'SomePassW0rd');
```

mysql.rds_set_external_source (Aurora MySQL version 3)

Configures an Aurora MySQL DB instance to be a read replica of an instance of MySQL running external to Amazon RDS.

> **Important**
>
> To run this procedure, `autocommit` must be enabled. To enable it, set the `autocommit` parameter to 1. For information about modifying parameters, see Modifying parameters in a DB parameter group.

Syntax

```
CALL mysql.rds_set_external_source (
  host_name
  , host_port
  , replication_user_name
  , replication_user_password
  , mysql_binary_log_file_name
  , mysql_binary_log_file_location
  , ssl_encryption
);
```

Parameters

`host_name`

The host name or IP address of the MySQL instance running external to Amazon RDS to become the source database instance.

host_port

The port used by the MySQL instance running external to Amazon RDS to be configured as the source database instance. If your network configuration includes Secure Shell (SSH) port replication that converts the port number, specify the port number that is exposed by SSH.

replication_user_name

The ID of a user with `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the MySQL instance running external to Amazon RDS. We recommend that you provide an account that is used solely for replication with the external instance.

replication_user_password

The password of the user ID specified in `replication_user_name`.

mysql_binary_log_file_name

The name of the binary log on the source database instance that contains the replication information.

mysql_binary_log_file_location

The location in the `mysql_binary_log_file_name` binary log at which replication starts reading the replication information.

You can determine the binlog file name and location by running `SHOW MASTER STATUS` on the source database instance.

ssl_encryption

A value that specifies whether Secure Socket Layer (SSL) encryption is used on the replication connection. 1 specifies to use SSL encryption, 0 specifies to not use encryption. The default is 0.

> **Note**
>
> You must have imported a custom SSL certificate using mysql.rds_import_binlog_ssl_material to enable this option. If you haven't imported an custom SSL certificate, then set this parameter to 0 and use mysql.rds_set_binlog_source_ssl (Aurora MySQL version 3) to enable SSL for binary log replication.
> The `MASTER_SSL_VERIFY_SERVER_CERT` option isn't supported. This option is set to 0, which means that the connection is encrypted, but the certificates aren't verified.

Usage notes

The master user must run the `mysql.rds_set_external_source` procedure. This procedure must be run on the Aurora MySQL DB instance to be configured as the read replica of a MySQL instance running external to Amazon RDS.

Before you run `mysql.rds_set_external_source`, you must configure the instance of MySQL running external to Amazon RDS to be a source database instance. To connect to the MySQL instance running external to Amazon RDS, you must specify `replication_user_name` and `replication_user_password` values that indicate a replication user that has `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the external instance of MySQL.

To configure an external instance of MySQL as a source database instance

1. Using the MySQL client of your choice, connect to the external instance of MySQL and create a user account to be used for replication. The following is an example.

   MySQL 5.7

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED BY 'password';
   ```

   MySQL 8.0

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED WITH mysql_native_password BY
    'password';
   ```

   > **Note**
   >
   > Specify a password other than the prompt shown here as a security best practice.

2. On the external instance of MySQL, grant `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges to your replication user. The following example grants `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges on all databases for the 'repl_user' user for your domain.

   MySQL 5.7

   ```
   GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com'
    IDENTIFIED BY 'password';
   ```

MySQL 8.0

```
GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com';
```

To use encrypted replication, configure source database instance to use SSL connections. Also, import the certificate authority certificate, client certificate, and client key into the DB instance or DB cluster using the mysql.rds_import_binlog_ssl_material procedure.

> **Note**
>
> We offer these stored procedures primarily to enable replication with MySQL instances running external to Amazon RDS. We recommend that you use Aurora Replicas to manage replication within an Aurora MySQL DB cluster when possible. For information about managing replication in Aurora MySQL DB clusters, see Using Aurora Replicas.

After calling `mysql.rds_set_external_source` to configure an Aurora MySQL DB instance as a read replica, you can call mysql.rds_start_replication on the read replica to start the replication process. You can call mysql.rds_reset_external_source to remove the read replica configuration.

When `mysql.rds_set_external_source` is called, Amazon RDS records the time, user, and an action of `set master` in the `mysql.rds_history` and `mysql.rds_replication_status` tables.

Examples

When run on an Aurora MySQL DB instance, the following example configures the DB instance to be a read replica of an instance of MySQL running external to Amazon RDS.

```
call mysql.rds_set_external_source(
  'Externaldb.some.com',
  3306,
  'repl_user',
  'password',
  'mysql-bin-changelog.0777',
  120,
  0);
```

mysql.rds_set_external_source_with_auto_position (Aurora MySQL version 3)

Configures an Aurora MySQL primary instance to accept incoming replication from an external MySQL instance. This procedure also configures replication based on global transaction identifiers (GTIDs).

Syntax

```
CALL mysql.rds_set_external_source_with_auto_position (
  host_name
  , host_port
  , replication_user_name
  , replication_user_password
  , ssl_encryption
);
```

Parameters

host_name

The host name or IP address of the MySQL instance running external to Aurora to become the replication source.

host_port

The port used by the MySQL instance running external to Aurora to be configured as the replication source. If your network configuration includes Secure Shell (SSH) port replication that converts the port number, specify the port number that is exposed by SSH.

replication_user_name

The ID of a user with `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the MySQL instance running external to Aurora. We recommend that you provide an account that is used solely for replication with the external instance.

replication_user_password

The password of the user ID specified in `replication_user_name`.

ssl_encryption

This option isn't currently implemented. The default is 0.

Note

Use mysql.rds_set_binlog_source_ssl (Aurora MySQL version 3) to enable SSL for binary log replication.

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

The administrative user must run the `mysql.rds_set_external_source_with_auto_position` procedure. The administrative user runs this procedure on the primary instance of an Aurora MySQL DB cluster that acts as a replication target. This can be the replication target of an external MySQL DB instance or an Aurora MySQL DB cluster.

This procedure is supported for Aurora MySQL version 3. This procedure doesn't configure delayed replication, because Aurora MySQL doesn't support delayed replication.

Before you run `mysql.rds_set_external_source_with_auto_position`, configure the external MySQL DB instance to be a replication source. To connect to the external MySQL instance, specify values for `replication_user_name` and `replication_user_password`. These values must indicate a replication user that has `REPLICATION CLIENT` and `REPLICATION SLAVE` permissions on the external MySQL instance.

To configure an external MySQL instance as a replication source

1. Using the MySQL client of your choice, connect to the external MySQL instance and create a user account to be used for replication. The following is an example.

   ```
   CREATE USER 'repl_user'@'mydomain.com' IDENTIFIED BY 'SomePassW0rd'
   ```

2. On the external MySQL instance, grant `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges to your replication user. The following example grants `REPLICATION CLIENT` and `REPLICATION SLAVE` privileges on all databases for the `'repl_user'` user for your domain.

   ```
   GRANT REPLICATION CLIENT, REPLICATION SLAVE ON *.* TO 'repl_user'@'mydomain.com'
   IDENTIFIED BY 'SomePassW0rd'
   ```

When you call `mysql.rds_set_external_source_with_auto_position`, Amazon RDS records certain information. This information is the time, the user, and an action of `"set master"` in the `mysql.rds_history` and `mysql.rds_replication_status` tables.

To skip a specific GTID-based transaction that is known to cause a problem, you can use the mysql.rds_skip_transaction_with_gtid/> stored procedure. For more information about working with GTID-based replication, see Using GTID-based replication.

Examples

When run on an Aurora primary instance, the following example configures the Aurora cluster to act as a read replica of an instance of MySQL running external to Aurora.

```
call mysql.rds_set_external_source_with_auto_position(
  'Externaldb.some.com',
  3306,
  'repl_user'@'mydomain.com',
  'SomePassW0rd');
```

mysql.rds_set_master_auto_position (Aurora MySQL version 2)

Sets the replication mode to be based on either binary log file positions or on global transaction identifiers (GTIDs).

Syntax

```
CALL mysql.rds_set_master_auto_position (
auto_position_mode
);
```

Parameters

`auto_position_mode`

A value that indicates whether to use log file position replication or GTID-based replication:

- `0` – Use the replication method based on binary log file position. The default is `0`.
- 1 – Use the GTID-based replication method.

Usage notes

The master user must run the `mysql.rds_set_master_auto_position` procedure.

This procedure is supported for Aurora MySQL version 2.

mysql.rds_set_read_only (Aurora MySQL version 3)

Turns `read_only` mode on or off globally for the DB instance.

Syntax

```
CALL mysql.rds_set_read_only(mode);
```

Parameters

mode

A value that indicates whether `read_only` mode is on or off globally for the DB instance:

- `0` – `OFF`. The default is `0`.
- `1` – `ON`

Usage notes

The `mysql.rds_set_read_only` stored procedure modifies only the `read_only` parameter. The `innodb_read_only` parameter can't be changed on reader DB instances.

The `read_only` parameter change doesn't persist on rebooting. To make permanent changes to `read_only`, you must use the `read_only` DB cluster parameter.

This procedure is supported for Aurora MySQL version 3.06 and higher.

mysql.rds_set_session_binlog_format (Aurora MySQL version 2)

Sets the binary log format for the current session.

Syntax

```
CALL mysql.rds_set_session_binlog_format(format);
```

Parameters

format

A value that indicates the binary log format for the current session:

- STATEMENT – The replication source writes events to the binary log based on SQL statements.
- ROW – The replication source writes events to the binary log that indicate changes to individual table rows.
- MIXED – Logging is generally based on SQL statements, but switches to rows under certain conditions. For more information, see Mixed Binary Logging Format in the MySQL documentation.

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

To use this stored procedure, you must have binary logging configured for the current session.

For Aurora, this procedure is supported for Aurora MySQL version 2.12 and higher MySQL 5.7–compatible versions.

mysql.rds_set_source_auto_position (Aurora MySQL version 3)

Sets the replication mode to be based on either binary log file positions or on global transaction identifiers (GTIDs).

Syntax

```
CALL mysql.rds_set_source_auto_position (auto_position_mode);
```

Parameters

auto_position_mode

A value that indicates whether to use log file position replication or GTID-based replication:

- 0 – Use the replication method based on binary log file position. The default is 0.
- 1 – Use the GTID-based replication method.

Usage notes

For an Aurora MySQL DB cluster, you call this stored procedure while connected to the primary instance.

The administrative user must run the `mysql.rds_set_source_auto_position` procedure.

mysql.rds_skip_transaction_with_gtid (Aurora MySQL version 2 and 3)

Skips replication of a transaction with the specified global transaction identifier (GTID) on an Aurora primary instance.

You can use this procedure for disaster recovery when a specific GTID transaction is known to cause a problem. Use this stored procedure to skip the problematic transaction. Examples of problematic transactions include transactions that disable replication, delete important data, or cause the DB instance to become unavailable.

Syntax

```
CALL mysql.rds_skip_transaction_with_gtid (
gtid_to_skip
);
```

Parameters

`gtid_to_skip`

The GTID of the replication transaction to skip.

Usage notes

The master user must run the `mysql.rds_skip_transaction_with_gtid` procedure.

This procedure is supported for Aurora MySQL version 2 and 3.

Examples

The following example skips replication of the transaction with the GTID `3E11FA47-71CA-11E1-9E33-C80AA9429562:23`.

```
CALL mysql.rds_skip_transaction_with_gtid('3E11FA47-71CA-11E1-9E33-C80AA9429562:23');
```

mysql.rds_skip_repl_error

Skips and deletes a replication error on a MySQL DB read replica.

Syntax

```
CALL mysql.rds_skip_repl_error;
```

Usage notes

The master user must run the `mysql.rds_skip_repl_error` procedure on a read replica. For more information about this procedure, see Skipping the current replication error.

To determine if there are errors, run the MySQL `SHOW REPLICA STATUS\G` command. If a replication error isn't critical, you can run `mysql.rds_skip_repl_error` to skip the error. If there are multiple errors, `mysql.rds_skip_repl_error` deletes the first error, then warns that others are present. You can then use `SHOW REPLICA STATUS\G` to determine the correct course of action for the next error. For information about the values returned, see SHOW REPLICA STATUS statement in the MySQL documentation.

> **Note**
>
> Previous versions of MySQL used `SHOW SLAVE STATUS` instead of `SHOW REPLICA STATUS`. If you are using a MySQL version before 8.0.23, then use `SHOW SLAVE STATUS`.

For more information about addressing replication errors with Aurora MySQL, see Diagnosing and resolving a MySQL read replication failure.

Replication stopped error

When you call the `mysql.rds_skip_repl_error` procedure, you might receive an error message stating that the replica is down or disabled.

This error message appears if you run the procedure on the primary instance instead of the read replica. You must run this procedure on the read replica for the procedure to work.

This error message might also appear if you run the procedure on the read replica, but replication can't be restarted successfully.

If you need to skip a large number of errors, the replication lag can increase beyond the default retention period for binary log (binlog) files. In this case, you might encounter a fatal error due to binlog files being purged before they have been replayed on the read replica. This purge causes replication to stop, and you can no longer call the `mysql.rds_skip_repl_error` command to skip replication errors.

You can mitigate this issue by increasing the number of hours that binlog files are retained on your source database instance. After you have increased the binlog retention time, you can restart replication and call the `mysql.rds_skip_repl_error` command as needed.

To set the binlog retention time, use the [mysql.rds_set_configuration](#) procedure and specify a configuration parameter of `'binlog retention hours'` along with the number of hours to retain binlog files on the DB cluster. The following example sets the retention period for binlog files to 48 hours.

```
CALL mysql.rds_set_configuration('binlog retention hours', 48);
```

mysql.rds_start_replication

Initiates replication from an Aurora MySQL DB cluster.

> **Note**
>
> You can use the mysql.rds_start_replication_until (Aurora MySQL version 3) or mysql.rds_start_replication_until_gtid (Aurora MySQL version 3) stored procedure to initiate replication from an Aurora MySQL DB instance and stop replication at the specified binary log file location.

Syntax

```
CALL mysql.rds_start_replication;
```

Usage notes

The master user must run the `mysql.rds_start_replication` procedure.

To import data from an instance of MySQL external to Amazon RDS, call `mysql.rds_start_replication` on the read replica to start the replication process after you call `mysql.rds_set_external_master`or `mysql.rds_set_external_source` to build the replication configuration. For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

To export data to an instance of MySQL external to Amazon RDS, call `mysql.rds_start_replication` and `mysql.rds_stop_replication` on the read replica to control some replication actions, such as purging binary logs. For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

You can also call `mysql.rds_start_replication` on the read replica to restart any replication process that you previously stopped by calling `mysql.rds_stop_replication`. For more information, see Replication stopped error.

mysql.rds_start_replication_until (Aurora MySQL version 3)

Initiates replication from an Aurora MySQL DB cluster and stops replication at the specified binary log file location.

Syntax

```
CALL mysql.rds_start_replication_until (
replication_log_file
  , replication_stop_point
);
```

Parameters

`replication_log_file`

The name of the binary log on the source database instance that contains the replication information.

`replication_stop_point`

The location in the `replication_log_file` binary log at which replication will stop.

Usage notes

The master user must run the `mysql.rds_start_replication_until` procedure.

This procedure is supported for Aurora MySQL version 3.04 and higher.

The mysql.rds_start_replication_until stored procedure isn't supported for managed replication, which includes the following:

- Replicating Amazon Aurora MySQL DB clusters across AWS Regions
- Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica

The file name specified for the replication_log_file parameter must match the source database instance binlog file name.

When the replication_stop_point parameter specifies a stop location that is in the past, replication is stopped immediately.

Examples

The following example initiates replication and replicates changes until it reaches location 120 in the mysql-bin-changelog.000777 binary log file.

```
call mysql.rds_start_replication_until(
  'mysql-bin-changelog.000777',
  120);
```

mysql.rds_start_replication_until_gtid (Aurora MySQL version 3)

Initiates replication from an Aurora MySQL DB cluster and stops replication immediately after the specified global transaction identifier (GTID).

Syntax

```
CALL mysql.rds_start_replication_until_gtid(gtid);
```

Parameters

gtid

The GTID after which replication is to stop.

Usage notes

The master user must run the mysql.rds_start_replication_until_gtid procedure.

This procedure is supported for Aurora MySQL version 3.04 and higher.

The `mysql.rds_start_replication_until_gtid` stored procedure isn't supported for managed replication, which includes the following:

- Replicating Amazon Aurora MySQL DB clusters across AWS Regions
- Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica

When the `gtid` parameter specifies a transaction that has already been run by the replica, replication is stopped immediately.

Examples

The following example initiates replication and replicates changes until it reaches GTID `3E11FA47-71CA-11E1-9E33-C80AA9429562:23`.

```
call mysql.rds_start_replication_until_gtid('3E11FA47-71CA-11E1-9E33-C80AA9429562:23');
```

mysql.rds_stop_replication

Stops replication from a MySQL DB instance.

Syntax

```
CALL mysql.rds_stop_replication;
```

Usage notes

The master user must run the `mysql.rds_stop_replication` procedure.

If you are configuring replication to import data from an instance of MySQL running external to Amazon RDS, you call `mysql.rds_stop_replication` on the read replica to stop the replication process after the import has completed. For more information, see Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

If you are configuring replication to export data to an instance of MySQL external to Amazon RDS, you call `mysql.rds_start_replication` and `mysql.rds_stop_replication` on the read replica to control some replication actions, such as purging binary logs. For more information, see

Replication between Aurora and MySQL or between Aurora and another Aurora DB cluster (binary log replication).

The `mysql.rds_stop_replication` stored procedure isn't supported for managed replication, which includes the following:

- Replicating Amazon Aurora MySQL DB clusters across AWS Regions
- Migrating data from an RDS for MySQL DB instance to an Amazon Aurora MySQL DB cluster by using an Aurora read replica

Aurora MySQL–specific information_schema tables

Aurora MySQL has certain `information_schema` tables that are specific to Aurora.

information_schema.aurora_global_db_instance_status

The `information_schema.aurora_global_db_instance_status` table contains information about the status of all DB instances in a global database's primary and secondary DB clusters. The following table shows the columns that you can use. The remaining columns are for Aurora internal use only.

> **Note**
>
> This information schema table is only available with Aurora MySQL version 3.04.0 and higher global databases.

Column	Data type	Description
SERVER_ID	varchar(100)	The identifier of the DB instance.
SESSION_ID	varchar(100)	A unique identifier for the current session. A value of `MASTER_SESSION_ID` identifies the Writer (primary) DB instance.

Column	Data type	Description
AWS_REGION	varchar(100)	The AWS Region in which this global database instance runs. For a list of Regions, see Region availability.
DURABLE_LSN	bigint unsigned	The log sequence number (LSN) made durable in storage. A log sequence number (LSN) is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
HIGHEST_LSN_RCVD	bigint unsigned	The highest LSN received by the DB instance from the writer DB instance.
OLDEST_READ_VIEW_TRX_ID	bigint unsigned	The ID of the oldest transacti on that the writer DB instance can purge to.
OLDEST_READ_VIEW_LSN	bigint unsigned	The oldest LSN used by the DB instance to read from storage.

Column	Data type	Description
VISIBILITY_LAG_IN_MSEC	float(10,0) unsigned	For readers in the primary DB cluster, how far this DB instance is lagging behind the writer DB instance in milliseconds. For readers in a secondary DB cluster, how far this DB instance is lagging behind the secondary volume in milliseconds.

information_schema.aurora_global_db_status

The `information_schema.aurora_global_db_status` table contains information about various aspects of Aurora global database lag, specifically, lag of the underlying Aurora storage (so called durability lag) and lag between the recovery point objective (RPO). The following table shows the columns that you can use. The remaining columns are for Aurora internal use only.

Note

This information schema table is only available with Aurora MySQL version 3.04.0 and higher global databases.

Column	Data type	Description
AWS_REGION	varchar(100)	The AWS Region in which this global database instance runs. For a list of Regions, see Region availability.
HIGHEST_LSN_WRITTEN	bigint unsigned	The highest log sequence number (LSN) that currently exists on this DB cluster. A log sequence number (LSN) is a unique sequential number

Column	Data type	Description
		that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
DURABILITY_LAG_IN_MILLISECONDS	float(10,0) unsigned	The difference in the timestamp values between the `HIGHEST_LSN_WRITTEN` on a secondary DB cluster and the `HIGHEST_LSN_WRITTEN` on the primary DB cluster. This value is always 0 on the primary DB cluster of the Aurora global database.

Column	Data type	Description
RPO_LAG_IN_MILLISECONDS	float(10,0) unsigned	The recovery point objective (RPO) lag. The RPO lag is the time it takes for the most recent user transaction COMMIT to be stored on a secondary DB cluster after it's been stored on the primary DB cluster of the Aurora global database. This value is always 0 on the primary DB cluster of the Aurora global database. In simple terms, this metric calculates the recovery point objective for each Aurora MySQL DB cluster in the Aurora global database, that is, how much data might be lost if there were an outage. As with lag, RPO is measured in time.
LAST_LAG_CALCULATION_TIMESTAMP	datetime	The timestamp that specifies when values were last calculated for DURABILITY_LAG_IN_MILLISECONDS and RPO_LAG_IN_MILLISECONDS . A time value such as 1970-01-01 00:00:00+00 means this is the primary DB cluster.

Column	Data type	Description
OLDEST_READ_VIEW_TRX_ID	bigint unsigned	The ID of the oldest transaction that the writer DB instance can purge to.

information_schema.replica_host_status

The `information_schema.replica_host_status` table contains replication information. The columns that you can use are shown in the following table. The remaining columns are for Aurora internal use only.

Column	Data type	Description
CPU	double	The CPU percentage usage of the replica host.
IS_CURRENT	tinyint	Whether the replica is current.
LAST_UPDATE_TIMESTAMP	datetime(6)	The time the last update occurred. Used to determine whether a record is stale.
REPLICA_LAG_IN_MILLISECONDS	double	The replica lag in milliseconds.
SERVER_ID	varchar(100)	The ID of the database server.
SESSION_ID	varchar(100)	The ID of the database session. Used to determine whether a DB instance is a writer or reader instance.

Note

When a replica instance falls behind, the information queried from its `information_schema.replica_host_status` table might be outdated. In this situation, we recommend that you query from the writer instance instead.
While the `mysql.ro_replica_status` table has similar information, we don't recommend that you use it.

information_schema.aurora_forwarding_processlist

The `information_schema.aurora_forwarding_processlist` table contains information about processes involved in write forwarding.

The contents of this table are visible only on the writer DB instance for a DB cluster with global or in-cluster write forwarding turned on. An empty result set is returned on reader DB instances.

Field	Data type	Description
ID	bigint	The identifier of the connection on the writer DB instance. This identifier is the same value displayed in the `Id` column of the `SHOW PROCESSLIST` statement and returned by the `CONNECTION_ID()` function within the thread.
USER	varchar(32)	The MySQL user that issued the statement.
HOST	varchar(255)	The MySQL client that issued the statement. For forwarded statements, this field shows the application client host address that established the connection on the forwarding reader DB instance.
DB	varchar(64)	The default database for the thread.
COMMAND	varchar(16)	The type of command the thread is executing on behalf of the client, or `Sleep` if the session is idle. For descriptions

Field	Data type	Description
		of thread commands, see the MySQL documentation on Thread Command Values in the MySQL documentation.
TIME	int	The time in seconds that the thread has been in its current state.
STATE	varchar(64)	An action, event, or state that indicates what the thread is doing. For descriptions of state values, see General Thread States in the MySQL documentation.
INFO	longtext	The statement that the thread is executing, or `NULL` if it isn't executing a statement. The statement might be the one sent to the server, or an innermost statement if the statement executes other statements.
IS_FORWARDED	bigint	Indicates whether the thread is forwarded from a reader DB instance.
REPLICA_SESSION_ID	bigint	The connection identifier on the Aurora Replica. This identifier is the same value displayed in the `Id` column of the `SHOW PROCESSLIST` statement on the forwarding Aurora reader DB instance.
REPLICA_INSTANCE_IDENTIFIER	varchar(64)	The DB instance identifier of the forwarding thread.
REPLICA_CLUSTER_NAME	varchar(64)	The DB cluster identifier of the forwarding thread. For in-cluster write forwarding, this identifier is the same DB cluster as the writer DB instance.
REPLICA_REGION	varchar(64)	The AWS Region from which the forwarding thread originates. For in-cluster write forwarding, this Region is the same AWS Region as the writer DB instance.

Database engine updates for Amazon Aurora MySQL

Amazon Aurora releases updates regularly. Updates are applied to Aurora DB clusters during system maintenance windows. The timing when updates are applied depends on the region and maintenance window setting for the DB cluster, as well as the type of update.

Amazon Aurora releases are made available to all AWS Regions over the course of multiple days. Some Regions might temporarily show an engine version that isn't available in a different Region yet.

Updates are applied to all instances in a DB cluster at the same time. An update requires a database restart on all instances in a DB cluster, so you experience 20 to 30 seconds of downtime, after which you can resume using your DB cluster or clusters. You can view or change your maintenance window settings from the AWS Management Console.

For details about the Aurora MySQL versions that are supported by Amazon Aurora, see the *Release Notes for Aurora MySQL*.

Following, you can learn how to choose the right Aurora MySQL version for your cluster, how to specify the version when you create or upgrade a cluster, and the procedures to upgrade a cluster from one version to another with minimal interruption.

Topics

Aurora MySQL version numbers and special versions

Although Aurora MySQL-Compatible Edition is compatible with the MySQL database engines, Aurora MySQL includes features and bug fixes that are specific to particular Aurora MySQL versions. Application developers can check the Aurora MySQL version in their applications by using SQL. Database administrators can check and specify Aurora MySQL versions when creating or upgrading Aurora MySQL DB clusters and DB instances.

Topics

- Checking or specifying Aurora MySQL engine versions through AWS
- Checking Aurora MySQL versions using SQL
- Aurora MySQL long-term support (LTS) releases
- Aurora MySQL beta releases

Checking or specifying Aurora MySQL engine versions through AWS

When you perform administrative tasks using the AWS Management Console, AWS CLI, or RDS API, you specify the Aurora MySQL version in a descriptive alphanumeric format.

Starting with Aurora MySQL version 2, Aurora engine versions have the following syntax.

```
mysql-major-version.mysql_aurora.aurora-mysql-version
```

The *`mysql-major-version-`* portion is `5.7` or `8.0`. This value represents the version of the client protocol and general level of MySQL feature support for the corresponding Aurora MySQL version.

The *`aurora-mysql-version`* is a dotted value with three parts: the Aurora MySQL major version, the Aurora MySQL minor version, and the patch level. The major version is 2 or 3. Those values represent Aurora MySQL compatible with MySQL 5.7 or 8.0, respectively. The minor version represents the feature release within the 2.x or 3.x series. The patch level begins at `0` for each minor version, and represents the set of subsequent bug fixes that apply to the minor version. Occasionally, a new feature is incorporated into a minor version but not made visible immediately. In these cases, the feature undergoes fine-tuning and is made public in a later patch level.

All 2.x Aurora MySQL engine versions are wire-compatible with Community MySQL 5.7.12. All 3.x Aurora MySQL engine versions are wire-compatible with MySQL 8.0.23 onwards. You can refer to release notes of the specific 3.x version to find the corresponding MySQL compatible version.

For example, the engine versions for Aurora MySQL 3.02.0 and 2.11.2 are the following.

```
8.0.mysql_aurora.3.02.0
5.7.mysql_aurora.2.11.2
```

Note

There isn't a one-to-one correspondence between community MySQL versions and the Aurora MySQL 2.x versions. For Aurora MySQL version 3, there is a more direct mapping.

To check which bug fixes and new features are in a particular Aurora MySQL release, see Database engine updates for Amazon Aurora MySQL version 3 and Database engine updates for Amazon Aurora MySQL version 2 in the *Release Notes for Aurora MySQL*. For a chronological list of new features and releases, see Document history. To check the minimum version required for a security-related fix, see Security vulnerabilities fixed in Aurora MySQLin the *Release Notes for Aurora MySQL*.

You specify the Aurora MySQL engine version in some AWS CLI commands and RDS API operations. For example, you specify the `--engine-version` option when you run the AWS CLI commands create-db-cluster and modify-db-cluster. You specify the `EngineVersion` parameter when you run the RDS API operations CreateDBCluster and ModifyDBCluster.

In Aurora MySQL version 2 and higher, the engine version in the AWS Management Console also includes the Aurora version. Upgrading the cluster changes the displayed value. This change helps you to specify and check the precise Aurora MySQL versions, without the need to connect to the cluster or run any SQL commands.

Tip

For Aurora clusters managed through AWS CloudFormation, this change in the `EngineVersion` setting can trigger actions by AWS CloudFormation. For information about how AWS CloudFormation treats changes to the `EngineVersion` setting, see the AWS CloudFormation documentation.

Checking Aurora MySQL versions using SQL

The Aurora version numbers that you can retrieve in your application using SQL queries use the format *<major version>*.*<minor version>*.*<patch version>*. You can get this version number for any DB instance in your Aurora MySQL cluster by querying the `AURORA_VERSION` system variable. To get this version number, use one of the following queries.

```
select aurora_version();
select @@aurora_version;
```

Those queries produce output similar to the following.

```
mysql> select aurora_version(), @@aurora_version;
+------------------+------------------+
| aurora_version() | @@aurora_version |
+------------------+------------------+
| 2.11.1           | 2.11.1           |
+------------------+------------------+
```

The version numbers that the console, CLI, and RDS API return by using the techniques described in Checking or specifying Aurora MySQL engine versions through AWS are typically more descriptive.

Aurora MySQL long-term support (LTS) releases

Each new Aurora MySQL version remains available for a certain amount of time for you to use when you create or upgrade a DB cluster. After this period, you must upgrade any clusters that use that version. You can manually upgrade your cluster before the support period ends, or Aurora can automatically upgrade it for you when its Aurora MySQL version is no longer supported.

Aurora designates certain Aurora MySQL versions as long-term support (LTS) releases. DB clusters that use LTS releases can stay on the same version longer and undergo fewer upgrade cycles than clusters that use non-LTS releases. Aurora supports each LTS release for at least three years after that release becomes available. When a DB cluster that's on an LTS release is required to upgrade, Aurora upgrades it to the next LTS release. That way, the cluster doesn't need to be upgraded again for a long time.

During the lifetime of an Aurora MySQL LTS release, new patch levels introduce fixes to important issues. The patch levels don't include any new features. You can choose whether to apply such patches to DB clusters running the LTS release. For certain critical fixes, Amazon might perform a managed upgrade to a patch level within the same LTS release. Such managed upgrades are performed automatically within the cluster maintenance window.

We recommend that you upgrade to the latest release, instead of using the LTS release, for most of your Aurora MySQL clusters. Doing so takes advantage of Aurora as a managed service and gives you access to the latest features and bug fixes. The LTS releases are intended for clusters with the following characteristics:

- You can't afford downtime on your Aurora MySQL application for upgrades outside of rare occurrences for critical patches.
- The testing cycle for the cluster and associated applications takes a long time for each update to the Aurora MySQL database engine.

- The database version for your Aurora MySQL cluster has all the DB engine features and bug fixes that your application needs.

The current LTS release for Aurora MySQL is the following:

- Aurora MySQL version 3.04.*. For more details about the LTS version, see Database engine updates for Amazon Aurora MySQL version 3 in the *Release Notes for Aurora MySQL*.

> **Note**
>
> We recommend that you don't set the `AutoMinorVersionUpgrade` parameter to `true` (or enable **Auto minor version upgrade** in the AWS Management Console) for LTS versions. Doing so could lead to your DB cluster being upgraded to a non-LTS version such as 3.05.2.

Aurora MySQL beta releases

An Aurora MySQL beta release is an early, security fix–only release in a limited number of AWS Regions. These fixes are later deployed more broadly across all Regions with the next patch release.

The numbering for a beta release is similar to an Aurora MySQL minor version, but with an extra fourth digit, for example 2.12.0.1 or 3.05.0.1.

For more information, see Database engine updates for Amazon Aurora MySQL version 2 and Database engine updates for Amazon Aurora MySQL version 3 in the *Release Notes for Aurora MySQL*.

Preparing for Amazon Aurora MySQL-Compatible Edition version 2 end of standard support

Amazon Aurora MySQL-Compatible Edition version 2 (with MySQL 5.7 compatibility) is planned to reach the end of standard support on October 31, 2024. We recommend that you upgrade all clusters running Aurora MySQL version 2 to the default Aurora MySQL version 3 (with MySQL 8.0 compatibility) or higher before Aurora MySQL version 2 reaches the end of its standard support period. On October 31, 2024, Amazon RDS will automatically enroll your databases into Amazon RDS Extended Support. If you're running Amazon Aurora MySQL version 2 (with MySQL 5.7 compatibility) in an Aurora Serverless version 1 cluster, this doesn't apply to you. If you want to upgrade your Aurora Serverless version 1 clusters to Aurora MySQL version 3, see Upgrade path for Aurora Serverless v1 DB clusters.

You can find upcoming end-of-support dates for Aurora major versions in Amazon Aurora versions.

If you have clusters running Aurora MySQL version 2, you will receive periodic notices with the latest information about how to conduct an upgrade as we get closer to the end of standard support date. We will update this page periodically with the latest information.

End of standard support timeline

1. Now through October 31, 2024 – You can upgrade clusters from Aurora MySQL version 2 (with MySQL 5.7 compatibility) to Aurora MySQL version 3 (with MySQL 8.0 compatibility).
2. October 31, 2024 – On this date, Aurora MySQL version 2 will reach the end of standard support and Amazon RDS automatically enrolls your clusters into Amazon RDS Extended Support.

We will automatically enroll you in RDS Extended Support. For more information, see Using Amazon RDS Extended Support.

Finding clusters affected by this end-of-life process

To find clusters affected by this end-of-life process, use the following procedures.

> ⚠ **Important**
>
> Be sure to perform these instructions in every AWS Region and for each AWS account where your resources are located.

Console

To find an Aurora MySQL version 2 cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. In the **Filter by databases** box, enter **5.7**.
4. Check for Aurora MySQL in the engine column.

AWS CLI

To find clusters affected by this end-of-life process using the AWS CLI, call the describe-db-clusters command. You can use the sample script following.

Example

```
aws rds describe-db-clusters --include-share --query 'DBClusters[?(Engine==`aurora-
mysql` && contains(EngineVersion,`5.7.mysql_aurora`))].{EngineVersion:EngineVersion,
 DBClusterIdentifier:DBClusterIdentifier, EngineMode:EngineMode}' --output table
 --region us-east-1

        +---------------------------------------------------------------+
        |                      DescribeDBClusters                       |
        +---------------------+---------------+-------------------------+
        |         DBCI        |      EM       |           EV            |
        +---------------------+---------------+-------------------------+
        |    aurora-mysql2    |  provisioned  | 5.7.mysql_aurora.2.11.3 |
        | aurora-serverlessv1 |   serverless  | 5.7.mysql_aurora.2.11.3 |
        +---------------------+---------------+-------------------------+
```

RDS API

To find Aurora MySQL DB clusters running Aurora MySQL version 2, use the RDS DescribeDBClusters API operation with the following required parameters:

- `DescribeDBClusters`
  - Filters.Filter.N
    - Name
      - engine
    - Values.Value.N
      - ['aurora']

Amazon RDS Extended Support

You can use Amazon RDS Extended Support over community MySQL 5.7 at no charge until the end of support date, October 31, 2024. On October 31, 2024, Amazon RDS automatically enrolls your databases into RDS Extended Support for Aurora MySQL version 2. RDS Extended Support for Aurora is a paid service that provides up to 28 additional months of support for Aurora MySQL version 2 until the end of RDS Extended Support in February 2027. RDS Extended Support will only be offered for Aurora MySQL minor versions 2.11 and 2.12. To use Amazon Aurora MySQL version 2 past the end of standard support, plan to run your databases on one of these minor versions before October 31, 2024.

For more information about RDS Extended Support, such as charges and other considerations, see Using Amazon RDS Extended Support.

Performing an upgrade

Upgrading between major versions requires more extensive planning and testing than for a minor version. The process can take substantial time. We want to look at the upgrade as a three-step process, with activities before the upgrade, for the upgrade, and after the upgrade.

Before the upgrade:

Before the upgrade, we recommend that you check for application compatibility, performance, maintenance procedures, and similar considerations for the upgraded cluster, thereby confirming that post-upgrade your applications will work as expected. Here are five recommendations that will help provide you a better upgrade experience.

- First, it's critical to understand How the Aurora MySQL in-place major version upgrade works.
- Next, explore the upgrade techniques that are available when Upgrading from Aurora MySQL version 2 to version 3.
- To help you decide the right time and approach to upgrade, you can learn the differences between Aurora MySQL version 3 and your current environment with Comparison of Aurora MySQL version 2 and Aurora MySQL version 3.
- After you've decided on the option that's convenient and works best, try a mock in-place upgrade on a cloned cluster, using Planning a major version upgrade for an Aurora MySQL cluster. The pre-checker can run and determine if the your database can be upgraded successfully, and if there is any application incompatibility issue post-upgrade as well as performance, maintenance procedures, and similar considerations.

  Review the upgrade checklist blog part 1 and part 2.
- Not all kinds or versions of Aurora MySQL clusters can use the in-place upgrade mechanism. For more information, see Aurora MySQL major version upgrade paths.

If you have any questions or concerns, the AWS Support Team is available on the community forums and Premium Support.

Doing the upgrade:

You can use one of the following upgrade techniques. The amount of downtime your system will experience depends on the technique chosen.

- **Blue/Green Deployments** – For situations where the top priority is to reduce application downtime, you can use Amazon RDS Blue/Green Deployments for performing the major version upgrade in provisioned Amazon Aurora DB clusters. A blue/green deployment creates a staging environment that copies the production environment. You can make certain changes to the Aurora DB cluster in the green (staging) environment without affecting production workloads. The switchover typically takes under a minute with no data loss. For more information, see Overview of Amazon RDS Blue/Green Deployments for Aurora. This minimizes downtime, but requires you to run additional resources while performing the upgrade.
- **In-place upgrades** – You can perform an in-place upgrade where Aurora automatically performs a precheck process for you, takes the cluster offline, backs up your cluster, performs the upgrade, and puts your cluster back online. An in-place major version upgrade can be performed in a few clicks, and doesn't involve other coordination or failovers with other clusters, but does involve downtime. For more information, see How to perform an in-place upgrade
- **Snapshot restore** – You can upgrade your Aurora MySQL version 2 cluster by restoring from an Aurora MySQL version 2 snapshot into an Aurora MySQL version 3 cluster. To do this, you should follow the process for taking a snapshot and restoring from it. This process involves database interruption because you're restoring from a snapshot.

After the upgrade:

After the upgrade, you need to closely monitor your system (application and database) and make fine-tuning changes if necessary. Following the pre-upgrade steps closely will minimize the required changes needed. For more information, see Troubleshooting Amazon Aurora MySQL database performance.

To learn more about the methods, planning, testing, and troubleshooting of Aurora MySQL major version upgrades, be sure to thoroughly read Upgrading the major version of an Amazon Aurora MySQL DB cluster, including Troubleshooting for Aurora MySQL in-place upgrade. Also, note that some instance types aren't supported for Aurora MySQL version 3. For more information, see Aurora DB instance classes.

Upgrade path for Aurora Serverless v1 DB clusters

Upgrading between major versions requires more extensive planning and testing than for a minor version. The process can take substantial time. We want to look at the upgrade as a three-step process, with activities before the upgrade, for the upgrade, and after the upgrade.

Aurora MySQL version 2 (with MySQL 5.7 compatibility) will continue to receive standard support for Aurora Serverless v1 clusters.

If you want to upgrade to Amazon Aurora MySQL 3 (with MySQL 8.0 compatibility) and continue running Aurora Serverless, you can use Amazon Aurora Serverless v2. To understand the differences between Aurora Serverless v1 and Aurora Serverless v2, see Comparison of Aurora Serverless v2 and Aurora Serverless v1.

Upgrade to Aurora Serverless v2: You can upgrade an Aurora Serverless v1 cluster to Aurora Serverless v2. For more information, see Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2.

Preparing for Amazon Aurora MySQL-Compatible Edition version 1 end of life

Amazon Aurora MySQL-Compatible Edition version 1 (with MySQL 5.6 compatibility) is planned to reach end of life on February 28, 2023. Amazon advises that you upgrade all clusters (provisioned and Aurora Serverless) running Aurora MySQL version 1 to Aurora MySQL version 2 (with MySQL 5.7 compatibility) or Aurora MySQL version 3 (with MySQL 8.0 compatibility). Do this before Aurora MySQL version 1 reaches the end of its support period.

For Aurora provisioned DB clusters, you can complete upgrades from Aurora MySQL version 1 to Aurora MySQL version 2 by several methods. You can find instructions for the in-place upgrade mechanism in How to perform an in-place upgrade. Another way to complete the upgrade is to take a snapshot of an Aurora MySQL version 1 cluster and restore the snapshot to an Aurora MySQL version 2 cluster. Or you can follow a multistep process that runs the old and new clusters side by side. For more details about each method, see Upgrading the major version of an Amazon Aurora MySQL DB cluster.

For Aurora Serverless v1 DB clusters, you can perform an in-place upgrade from Aurora MySQL version 1 to Aurora MySQL version 2. For more details about this method, see Modifying an Aurora Serverless v1 DB cluster.

For Aurora provisioned DB clusters, you can complete upgrades from Aurora MySQL version 1 to Aurora MySQL version 3 by using a two-stage upgrade process:

1. Upgrade from Aurora MySQL version 1 to Aurora MySQL version 2 using the methods described preceding.
2. Upgrade from Aurora MySQL version 2 to Aurora MySQL version 3 using the same methods as for upgrading from version 1 to version 2. For more details, see Upgrading from Aurora MySQL version 2 to version 3. Note the Feature differences between Aurora MySQL version 2 and 3.

You can find upcoming end-of-life dates for Aurora major versions in Amazon Aurora versions. Amazon automatically upgrades any clusters that you don't upgrade yourself before the end-of-life date. After the end-of-life date, these automatic upgrades to the subsequent major version occur during a scheduled maintenance window for clusters.

The following are additional milestones for upgrading Aurora MySQL version 1 clusters (provisioned and Aurora Serverless) that are reaching end of life. For each, the start time is 00:00 Universal Coordinated Time (UTC).

1. Now through February 28, 2023 – You can at any time start upgrades of Aurora MySQL version 1 (with MySQL 5.6 compatibility) clusters to Aurora MySQL version 2 (with MySQL 5.7 compatibility). From Aurora MySQL version 2, you can do a further upgrade to Aurora MySQL version 3 (with MySQL 8.0 compatibility) for Aurora provisioned DB clusters.
2. January 16, 2023 – After this time, you can't create new Aurora MySQL version 1 clusters or instances from either the AWS Management Console or the AWS Command Line Interface (AWS CLI). You also can't add new secondary Regions to an Aurora global database. This might affect your ability to recover from an unplanned outage as outlined in Recovering an Amazon Aurora global database from an unplanned outage, because you can't complete steps 5 and 6 after this time. You will also be unable to create a new cross-Region read replica running Aurora MySQL version 1. You can still do the following for existing Aurora MySQL version 1 clusters until February 28, 2023:
   - Restore a snapshot taken of an Aurora MySQL version 1 cluster to the same version as the original snapshot cluster.
   - Add read replicas (not applicable for Aurora Serverless DB clusters).
   - Change instance configuration.
   - Perform point-in-time restore.
   - Create clones of existing version 1 clusters.

- Create a new cross-Region read replica running Aurora MySQL version 2 or higher.

3. February 28, 2023 – After this time, we plan to automatically upgrade Aurora MySQL version 1 clusters to the default version of Aurora MySQL version 2 within a scheduled maintenance window that follows. Restoring Aurora MySQL version 1 DB snapshots results in an automatic upgrade of the restored cluster to the default version of Aurora MySQL version 2 at that time.

Upgrading between major versions requires more extensive planning and testing than for a minor version. The process can take substantial time.

For situations where the top priority is to reduce downtime, you can also use blue/green deployments for performing the major version upgrade in provisioned Amazon Aurora DB clusters. A blue/green deployment creates a staging environment that copies the production environment. You can make changes to the Aurora DB cluster in the green (staging) environment without affecting production workloads. The switchover typically takes under a minute with no data loss and no need for application changes. For more information, see Overview of Amazon RDS Blue/Green Deployments for Aurora.

After the upgrade is finished, you also might have follow-up work to do. For example, you might need to follow up due to differences in SQL compatibility, the way certain MySQL-related features work, or parameter settings between the old and new versions.

To learn more about the methods, planning, testing, and troubleshooting of Aurora MySQL major version upgrades, be sure to thoroughly read Upgrading the major version of an Amazon Aurora MySQL DB cluster.

Finding clusters affected by this end-of-life process

To find clusters affected by this end-of-life process, use the following procedures.

> ⚠ **Important**
>
> Be sure to perform these instructions in every AWS Region and for each AWS account where your resources are located.

Console

To find an Aurora MySQL version 1 cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. In the **Filter by databases** box, enter **5.6**.
4. Check for Aurora MySQL in the engine column.

AWS CLI

To find clusters affected by this end-of-life process using the AWS CLI, call the describe-db-clusters command. You can use the sample script following.

Example

```
aws rds describe-db-clusters --include-share --query 'DBClusters[?Engine==`aurora`].
{EV:EngineVersion, DBCI:DBClusterIdentifier, EM:EngineMode}' --output table --region
 us-east-1

        +--------------------------------------------+
        |             DescribeDBClusters             |
        +---------------+--------------+-------------+
        |     DBCI      |     EM       |    EV       |
        +---------------+--------------+-------------+
        |  my-database-1|  serverless  |  5.6.10a    |
        +---------------+--------------+-------------+
```

RDS API

To find Aurora MySQL DB clusters running Aurora MySQL version 1, use the RDS DescribeDBClusters API operation with the following required parameters:

- `DescribeDBClusters`
  - Filters.Filter.N
    - Name
      - engine

  - Values.Value.N
    - ['aurora']

Upgrading Amazon Aurora MySQL DB clusters

You can upgrade an Aurora MySQL DB cluster to get bug fixes, new Aurora MySQL features, or to change to an entirely new version of the underlying database engine. The following sections show how.

> **Note**
>
> The type of upgrade that you do depends on how much downtime you can afford for your cluster, how much verification testing you plan to do, how important the specific bug fixes or new features are for your use case, and whether you plan to do frequent small upgrades or occasional upgrades that skip several intermediate versions. For each upgrade, you can change the major version, the minor version, and the patch level for your cluster. If you aren't familiar with the distinction between Aurora MySQL major versions, minor versions, and patch levels, you can read the background information at Aurora MySQL version numbers and special versions.

> **Tip**
>
> You can minimize the downtime required for a DB cluster upgrade by using a blue/green deployment. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.

Topics

Upgrading the minor version or patch level of an Aurora MySQL DB cluster

You can use the following methods to upgrade the minor version of a DB cluster or to patch a DB cluster:

- Upgrading Aurora MySQL by modifying the engine version (for Aurora MySQL version 2 and 3)
- Enabling automatic upgrades between minor Aurora MySQL versions

For information about how zero-downtime patching can reduce interruptions during the upgrade process, see Using zero-downtime patching.

Before performing a minor version upgrade

We recommend that you perform the following actions to reduce the downtime during a minor version upgrade:

- The Aurora DB cluster maintenance should be performed during a period of low traffic. Use Performance Insights to identify these time periods in order to configure the maintenance windows correctly. For more information on Performance Insights, see Monitoring DB load with Performance Insights on Amazon RDS. For more information on DB cluster maintenance window, Adjusting the preferred DB cluster maintenance window.
- Use AWS SDKs that support exponential backoff and jitter as a best practice. For more information, see Exponential Backoff And Jitter.

Minor version upgrade prechecks for Aurora MySQL

When you start a minor version upgrade, Amazon Aurora runs prechecks automatically.

These prechecks are mandatory. You can't choose to skip them. The prechecks provide the following benefits:

- They enable you to avoid unplanned downtime during the upgrade.
- If there are incompatibilities, Amazon Aurora prevents the upgrade and provides a log for you to learn about them. You can then use the log to prepare your database for the upgrade by reducing the incompatibilities. For detailed information about removing incompatibilities, see Preparing your installation for upgrade in the MySQL documentation.

The prechecks run before the DB instance is stopped for the upgrade, meaning that they don't cause any downtime when they run. If the prechecks find an incompatibility, Aurora automatically cancels the upgrade before the DB instance is stopped. Aurora also generates an event for the incompatibility. For more information about Amazon Aurora events, see Working with Amazon RDS event notification.

Aurora records detailed information about each incompatibility in the log file `PrePatchCompatibility.log`. In most cases, the log entry includes a link to the MySQL documentation for correcting the incompatibility. For more information about viewing log files, see Viewing and listing database log files.

Due to the nature of the prechecks, they analyze the objects in your database. This analysis results in resource consumption and increases the time for the upgrade to complete.

Upgrading Aurora MySQL by modifying the engine version

Upgrading the minor version of an Aurora MySQL DB cluster applies additional fixes and new features to an existing cluster.

This kind of upgrade applies to Aurora MySQL clusters where the original version and the upgraded version both have the same Aurora MySQL major version, either 2 or 3. The process is fast and straightforward because it doesn't involve any conversion for the Aurora MySQL metadata or reorganization of your table data.

You perform this kind of upgrade by modifying the engine version of the DB cluster using the AWS Management Console, AWS CLI, or the RDS API. For example, if your cluster is running Aurora MySQL 2.x, choose a higher 2.x version.

If you're performing a minor upgrade on an Aurora global database, upgrade all of the secondary clusters before you upgrade the primary cluster.

> **Note**
>
> To perform a minor version upgrade to Aurora MySQL version 3.03.* or higher, or version 2.12.*, use the following process:
>
> 1. Remove all secondary Regions from the global cluster. Follow the steps in Removing a cluster from an Amazon Aurora global database.
> 2. Upgrade the engine version of the primary Region to version 3.03.* or higher, or version 2.12.*, as applicable. Follow the steps in To modify the engine version of a DB cluster.
> 3. Add secondary Regions to the global cluster. Follow the steps in Adding an AWS Region to an Amazon Aurora global database.

To modify the engine version of a DB cluster

- **By using the console** – Modify the properties of your cluster. In the **Modify DB cluster** window, change the Aurora MySQL engine version in the **DB engine version** box. If you aren't familiar with the general procedure for modifying a cluster, follow the instructions at Modifying the DB cluster by using the console, CLI, and API.
- **By using the AWS CLI** – Call the modify-db-cluster AWS CLI command, and specify the name of your DB cluster for the `--db-cluster-identifier` option and the engine version for the `--engine-version` option.

  For example, to upgrade to Aurora MySQL version 2.12.1, set the `--engine-version` option to `5.7.mysql_aurora.2.12.1`. Specify the `--apply-immediately` option to immediately update the engine version for your DB cluster.
- **By using the RDS API** – Call the ModifyDBCluster API operation, and specify the name of your DB cluster for the `DBClusterIdentifier` parameter and the engine version for the `EngineVersion` parameter. Set the `ApplyImmediately` parameter to `true` to immediately update the engine version for your DB cluster.

Enabling automatic upgrades between minor Aurora MySQL versions

For an Amazon Aurora MySQL DB cluster, you can specify that Aurora upgrades the DB cluster automatically to new minor versions. You do so by setting the `AutoMinorVersionUpgrade` property (**Auto minor version upgrade** in the AWS Management Console) of the DB cluster.

Automatic upgrades occur during the maintenance window. If the individual DB instances in the DB cluster have different maintenance windows from the cluster maintenance window, then the cluster maintenance window takes precedence.

Automatic minor version upgrade doesn't apply to the following kinds of Aurora MySQL clusters:

- Clusters that are part of an Aurora global database
- Clusters that have cross-Region replicas

The outage duration varies depending on workload, cluster size, the amount of binary log data, and if Aurora can use the zero-downtime patching (ZDP) feature. Aurora restarts the database cluster, so you might experience a short period of unavailability before resuming use of your cluster. In particular, the amount of binary log data affects recovery time. The DB instance processes the binary log data during recovery. Thus, a high volume of binary log data increases recovery time.

Note

Aurora only performs automatic upgrades if all DB instances in your DB cluster have the `AutoMinorVersionUpgrade` setting enabled. For information on how to set it, and how it works when applied at the cluster and instance levels, see [Automatic minor version upgrades for Aurora DB clusters](#).

Then if an upgrade path exists for the DB cluster's instances to a minor DB engine version that has `AutoUpgrade` set to true, the upgrade will take place. The `AutoUpgrade` setting is dynamic, and is set by RDS.

Auto minor version upgrades are performed to the default minor version.

You can use a CLI command such as the following to check the status of the `AutoMinorVersionUpgrade` setting for all of the DB instances in your Aurora MySQL clusters.

```
aws rds describe-db-instances \
  --query '*[].
{DBClusterIdentifier:DBClusterIdentifier,DBInstanceIdentifier:DBInstanceIdentifier,AutoMinorVer
```

That command produces output similar to the following:

```
[
  {
      "DBInstanceIdentifier": "db-t2-medium-instance",
      "DBClusterIdentifier": "cluster-57-2020-06-03-6411",
      "AutoMinorVersionUpgrade": true
  },
  {
      "DBInstanceIdentifier": "db-t2-small-original-size",
      "DBClusterIdentifier": "cluster-57-2020-06-03-6411",
      "AutoMinorVersionUpgrade": false
  },
  {
      "DBInstanceIdentifier": "instance-2020-05-01-2332",
      "DBClusterIdentifier": "cluster-57-2020-05-01-4615",
      "AutoMinorVersionUpgrade": true
  },
... output omitted ...
```

In this example, **Enable auto minor version upgrade** is turned off for the DB cluster `cluster-57-2020-06-03-6411`, because it's turned off for one of the DB instances in the cluster.

Using zero-downtime patching

Performing upgrades for Aurora MySQL DB clusters involves the possibility of an outage when the database is shut down and while it's being upgraded. By default, if you start the upgrade while the database is busy, you lose all the connections and transactions that the DB cluster is processing. If you wait until the database is idle to perform the upgrade, you might have to wait a long time.

The zero-downtime patching (ZDP) feature attempts, on a best-effort basis, to preserve client connections through an Aurora MySQL upgrade. If ZDP completes successfully, application sessions are preserved and the database engine restarts while the upgrade is in progress. The database engine restart can cause a drop in throughput lasting for a few seconds to approximately one minute.

ZDP doesn't apply to the following:

- Operating system (OS) patches and upgrades
- Major version upgrades

ZDP is available for all supported Aurora MySQL versions and DB instance classes.

ZDP isn't supported for Aurora Serverless v1 or Aurora global databases.

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see Using T instance classes for development and testing.

You can see metrics of important attributes during ZDP in the MySQL error log. You can also see information about when Aurora MySQL uses ZDP or chooses not to use ZDP on the **Events** page in the AWS Management Console.

In Aurora MySQL version 2.10 and higher and version 3, Aurora can perform a zero-downtime patch whether or not binary log replication is enabled. If binary log replication is enabled, Aurora MySQL automatically drops the connection to the binlog target during a ZDP operation. Aurora

MySQL automatically reconnects to the binlog target and resumes replication after the restart finishes.

ZDP also works in combination with the reboot enhancements in Aurora MySQL 2.10 and higher. Patching the writer DB instance automatically patches readers at the same time. After performing the patch, Aurora restores the connections on both the writer and reader DB instances. Before Aurora MySQL 2.10, ZDP applies only to the writer DB instance of a cluster.

ZDP might not complete successfully under the following conditions:

- Long-running queries or transactions are in progress. If Aurora can perform ZDP in this case, any open transactions are canceled.
- Temporary tables or table locks are in use, for example while data definition language (DDL) statements run. If Aurora can perform ZDP in this case, any open transactions are canceled.
- Pending parameter changes exist.

If no suitable time window for performing ZDP becomes available because of one or more of these conditions, patching reverts to the standard behavior.

> **Note**
>
> For Aurora MySQL version 2 lower than 2.11.0 and version 3 lower than 3.04.0, ZDP might not complete successfully when there are open Secure Socket Layer (SSL) or Transport Layer Security (TLS) connections.

Although connections remain intact following a successful ZDP operation, some variables and features are reinitialized. The following kinds of information aren't preserved through a restart caused by zero-downtime patching:

- Global variables. Aurora restores session variables, but it doesn't restore global variables after the restart.
- Status variables. In particular, the uptime value reported by the engine status is reset after a restart that uses the ZDR or ZDP mechanisms.
- `LAST_INSERT_ID`.
- In-memory `auto_increment` state for tables. The in-memory auto-increment state is reinitialized. For more information about auto-increment values, see MySQL Reference Manual.

- Diagnostic information from INFORMATION_SCHEMA and PERFORMANCE_SCHEMA tables. This diagnostic information also appears in the output of commands such as SHOW PROFILE and SHOW PROFILES.

The following activities related to zero-downtime restart are reported on the **Events** page:

- Attempting to upgrade the database with zero downtime.
- Attempting to upgrade the database with zero downtime finished. The event reports how long the process took. The event also reports how many connections were preserved during the restart and how many connections were dropped. You can consult the database error log to see more details about what happened during the restart.

Alternative blue/green upgrade technique

In some situations, your top priority is to perform an immediate switchover from the old cluster to an upgraded one. In such situations, you can use a multistep process that runs the old and new clusters side-by-side. Here, you replicate data from the old cluster to the new one until you are ready for the new cluster to take over. For details, see Using Amazon RDS Blue/Green Deployments for database updates.

Upgrading the major version of an Amazon Aurora MySQL DB cluster

In an Aurora MySQL version number such as 2.12.1, the 2 represents the major version. Aurora MySQL version 2 is compatible with MySQL 5.7. Aurora MySQL version 3 is compatible with MySQL 8.0.

Upgrading between major versions requires more extensive planning and testing than for a minor version. The process can take substantial time. After the upgrade is finished, you also might have followup work to do. For example, this might occur because of differences in SQL compatibility or the way certain MySQL-related features work. Or it might occur because of differing parameter settings between the old and new versions.

Contents

Upgrading from Aurora MySQL version 2 to version 3

If you have a MySQL 5.7–compatible cluster and want to upgrade it to a MySQL–8.0 compatible cluster, you can do so by running an upgrade process on the cluster itself. This kind of upgrade is an *in-place upgrade*, in contrast to upgrades that you do by creating a new cluster. This technique keeps the same endpoint and other characteristics of the cluster. The upgrade is relatively fast because it doesn't require copying all your data to a new cluster volume. This stability helps to minimize any configuration changes in your applications. It also helps to reduce the amount of testing for the upgraded cluster. This is because the number of DB instances and their instance classes all stay the same.

The in-place upgrade mechanism involves shutting down your DB cluster while the operation takes place. Aurora performs a clean shutdown and completes outstanding operations such as transaction rollback and undo purge. For more information, see How the Aurora MySQL in-place major version upgrade works.

The in-place upgrade method is convenient, because it is simple to perform and minimizes configuration changes to associated applications. For example, an in-place upgrade preserves the endpoints and set of DB instances for your cluster. However, the time needed for an in-place upgrade can vary depending on the properties of your schema and how busy the cluster is. Thus, depending on the needs for your cluster, you can choose among the upgrade techniques:

- In-place upgrade
- Blue/Green Deployment
- Snapshot restore

> **Note**
>
> If you use the AWS CLI or RDS API for the snapshot restore upgrade method, you must run a subsequent operation to create a writer DB instance in the restored DB cluster.

For general information about Aurora MySQL version 3 and its new features, see Aurora MySQL version 3 compatible with MySQL 8.0.

For details about planning an upgrade, see Planning a major version upgrade for an Aurora MySQL cluster and How to perform an in-place upgrade.

Planning a major version upgrade for an Aurora MySQL cluster

To help you decide the right time and approach to upgrade, you can learn the differences between Aurora MySQL version 3 and your current environment:

- If you're converting from RDS for MySQL 8.0 or MySQL 8.0 Community Edition, see Comparison of Aurora MySQL version 3 and MySQL 8.0 Community Edition.
- If you're upgrading from Aurora MySQL version 2, RDS for MySQL 5.7, or community MySQL 5.7, see Comparison of Aurora MySQL version 2 and Aurora MySQL version 3.
- Create new MySQL 8.0-compatible versions of any custom parameter groups. Apply any necessary custom parameter values to the new parameter groups. Consult Parameter changes for Aurora MySQL version 3 to learn about parameter changes.
- Review your Aurora MySQL version 2 database schema and object definitions for the usage of new reserved keywords introduced in MySQL 8.0 Community Edition. Do so before you upgrade. For more information, see MySQL 8.0 New Keywords and Reserved Words in the MySQL documentation.

You can also find more MySQL-specific upgrade considerations and tips in Changes in MySQL 8.0 in the *MySQL Reference Manual*. For example, you can use the command `mysqlcheck --check-upgrade` to analyze your existing Aurora MySQL databases and identify potential upgrade issues.

> **Note**
>
> We recommend using larger DB instance classes when upgrading to Aurora MySQL version 3 using the in-place upgrade or snapshot restore technique. Examples are db.r5.24xlarge and db.r6g.16xlarge. This helps the upgrade process to complete faster by using the majority of available CPU capacity on the DB instance. You can change to the DB instance class that you want after the major version upgrade is complete.

After you finish the upgrade itself, you can follow the post-upgrade procedures in Post-upgrade cleanup for Aurora MySQL version 3. Finally, test your application's functionality and performance.

If you're converting from RDS from MySQL or community MySQL, follow the migration procedure explained in Migrating data to an Amazon Aurora MySQL DB cluster. In some cases, you might use binary log replication to synchronize your data with an Aurora MySQL version 3 cluster as part of the migration. If so, the source system must run a version that's compatible with your target DB cluster.

To make sure that your applications and administration procedures work smoothly after upgrading a cluster between major versions, do some advance planning and preparation. To see what sorts of management code to update for your AWS CLI scripts or RDS API–based applications, see How in-place upgrades affect the parameter groups for a cluster. Also see Changes to cluster properties between Aurora MySQL versions.

To learn what issues that you might encounter during the upgrade, see Troubleshooting for Aurora MySQL in-place upgrade. For issues that might cause the upgrade to take a long time, you can test those conditions in advance and correct them.

> **Note**
>
> An in-place upgrade involves shutting down your DB cluster while the operation takes place. Aurora MySQL performs a clean shutdown and completes outstanding operations such as undo purge. An upgrade might take a long time if there many undo records to purge. We recommend performing the upgrade only after the history list length (HLL) is

low. A generally acceptable value for the HLL is 100,000 or less. For more information, see this blog post.

Simulating the upgrade by cloning your DB cluster

You can check application compatibility, performance, maintenance procedures, and similar considerations for the upgraded cluster. To do so, you can perform a simulation of the upgrade before doing the real upgrade. This technique can be especially useful for production clusters. Here, it's important to minimize downtime and have the upgraded cluster ready to go as soon as the upgrade has finished.

Use the following steps:

1. Create a clone of the original cluster. Follow the procedure in Cloning a volume for an Amazon Aurora DB cluster.
2. Set up a similar set of writer and reader DB instances as in the original cluster.
3. Perform an in-place upgrade of the cloned cluster. Follow the procedure in How to perform an in-place upgrade.

   Start the upgrade immediately after creating the clone. That way, the cluster volume is still identical to the state of the original cluster. If the clone sits idle before you do the upgrade, Aurora performs database cleanup processes in the background. In that case, the upgrade of the clone isn't an accurate simulation of upgrading the original cluster.
4. Test application compatibility, performance, administration procedures, and so on, using the cloned cluster.
5. If you encounter any issues, adjust your upgrade plans to account for them. For example, adapt any application code to be compatible with the feature set of the higher version. Estimate how long the upgrade is likely to take based on the amount of data in your cluster. You might also choose to schedule the upgrade for a time when the cluster isn't busy.
6. After you're satisfied that your applications and workload work properly with the test cluster, you can perform the in-place upgrade for your production cluster.
7. Work to minimize the total downtime of your cluster during a major version upgrade. To do so, make sure that the workload on the cluster is low or zero at the time of the upgrade. In particular, make sure that there are no long running transactions in progress when you start the upgrade.

Using the blue-green upgrade technique

You can also create a blue/green deployment that runs the old and new clusters side-by-side. Here, you replicate data from the old cluster to the new one until you are ready for the new cluster to take over. For details, see Using Amazon RDS Blue/Green Deployments for database updates.

Major version upgrade prechecks for Aurora MySQL

MySQL 8.0 includes a number of incompatibilities with MySQL 5.7. These incompatibilities can cause problems during an upgrade from Aurora MySQL version 2 to version 3. Some preparation might be required on your database for the upgrade to be successful.

When you start an upgrade from Aurora MySQL version 2 to version 3, Amazon Aurora runs prechecks automatically to detect these incompatibilities.

These prechecks are mandatory. You can't choose to skip them. The prechecks provide the following benefits:

- They enable you to avoid unplanned downtime during the upgrade.
- If there are incompatibilities, Amazon Aurora prevents the upgrade and provides a log for you to learn about them. You can then use the log to prepare your database for the upgrade to version 3 by reducing the incompatibilities. For detailed information about removing incompatibilities, see Preparing your installation for upgrade in the MySQL documentation and Upgrading to MySQL 8.0? Here is what you need to know... on the MySQL Server Blog.

  For more information about upgrading to MySQL 8.0, see Upgrading MySQL in the MySQL documentation.

The prechecks include some that are included with MySQL and some that were created specifically by the Aurora team. For information about the prechecks provided by MySQL, see Upgrade checker utility.

The prechecks run before the DB instance is stopped for the upgrade, meaning that they don't cause any downtime when they run. If the prechecks find an incompatibility, Aurora automatically cancels the upgrade before the DB instance is stopped. Aurora also generates an event for the incompatibility. For more information about Amazon Aurora events, see Working with Amazon RDS event notification.

Aurora records detailed information about each incompatibility in the log file `PrePatchCompatibility.log`. In most cases, the log entry includes a link to the MySQL documentation for correcting the incompatibility. For more information about viewing log files, see Viewing and listing database log files.

Due to the nature of the prechecks, they analyze the objects in your database. This analysis results in resource consumption and increases the time for the upgrade to complete.

Community MySQL upgrade prechecks

The following is a general list of incompatibilities between MySQL 5.7 and 8.0:

- Your MySQL 5.7–compatible DB cluster must not use features that aren't supported in MySQL 8.0.

  For more information, see Features removed in MySQL 8.0 in the MySQL documentation.
- There must be no keyword or reserved word violations. Some keywords might be reserved in MySQL 8.0 that were not reserved previously.

  For more information, see Keywords and reserved words in the MySQL documentation.
- For improved Unicode support, consider converting objects that use the `utf8mb3` charset to use the `utf8mb4` charset. The `utf8mb3` character set is deprecated. Also, consider using `utf8mb4` for character set references instead of `utf8`, because currently `utf8` is an alias for the `utf8mb3` charset.

  For more information, see The utf8mb3 character set (3-byte UTF-8 unicode encoding) in the MySQL documentation.
- There must be no InnoDB tables with a nondefault row format.
- There must be no `ZEROFILL` or `display` length type attributes.
- There must be no partitioned table that uses a storage engine that does not have native partitioning support.
- There must be no tables in the MySQL 5.7 `mysql` system database that have the same name as a table used by the MySQL 8.0 data dictionary.
- There must be no tables that use obsolete data types or functions.
- There must be no foreign key constraint names longer than 64 characters.
- There must be no obsolete SQL modes defined in your `sql_mode` system variable setting.
- There must be no tables or stored procedures with individual `ENUM` or `SET` column elements that exceed 255 characters in length.

- There must be no table partitions that reside in shared InnoDB tablespaces.
- There must be no circular references in tablespace data file paths.
- There must be no queries and stored program definitions that use `ASC` or `DESC` qualifiers for `GROUP BY` clauses.
- There must be no removed system variables, and system variables must use the new default values for MySQL 8.0.
- There must be no zero (`0`) date, datetime, or timestamp values.
- There must be no schema inconsistencies resulting from file removal or corruption.
- There must be no table names that contain the `FTS` character string.
- There must be no InnoDB tables that belong to a different engine.
- There must be no table or schema names that are invalid for MySQL 5.7.

For more information about upgrading to MySQL 8.0, see Upgrading MySQL in the MySQL documentation.

Aurora MySQL upgrade prechecks

Aurora MySQL has its own specific requirements when upgrading from version 2 to version 3:

- There must be no deprecated SQL syntax, such as `SQL_CACHE`, `SQL_NO_CACHE`, and `QUERY_CACHE`, in views, routines, triggers, and events.
- There must be no `FTS_DOC_ID` column present on any table without the `FTS` index.
- There must be no column definition mismatch between the InnoDB data dictionary and the actual table definition.
- All database and table names must be lowercase when the `lower_case_table_names` parameter is set to `1`.
- Events and triggers must not have a missing or empty definer or an invalid creation context.
- All trigger names in a database must be unique.
- DDL recovery and Fast DDL aren't supported in Aurora MySQL version 3. There must be no artifacts in databases related to these features.
- Tables with the `REDUNDANT` or `COMPACT` row format can't have indexes larger than 767 bytes.
- The prefix length of indexes defined on `tiny` text columns can't exceed 255 bytes. With the `utf8mb4` character set, this limits the prefix length supported to 63 characters.

A larger prefix length was allowed in MySQL 5.7 using the `innodb_large_prefix` parameter. This parameter is deprecated in MySQL 8.0.

- There must be no InnoDB metadata inconsistency in the `mysql.host` table.
- There must be no column data type mismatch in system tables.
- There must be no XA transactions in the `prepared` state.
- Column names in views can't be longer than 64 characters.
- Special characters in stored procedures can't be inconsistent.
- Tables can't have data file path inconsistency.

Aurora MySQL major version upgrade paths

Not all kinds or versions of Aurora MySQL clusters can use the in-place upgrade mechanism. You can learn the appropriate upgrade path for each Aurora MySQL cluster by consulting the following table.

Type of Aurora MySQL DB cluster	Can it use in-place upgrade?	Action
Aurora MySQL provisioned cluster, 2.0 or higher	Yes	In-place upgrade is supported for 5.7-compatible Aurora MySQL clusters. For information about upgrading to Aurora MySQL version 3, see Planning a major version upgrade for an Aurora MySQL cluster and How to perform an in-place upgrade.
Aurora MySQL provisioned cluster, 3.01.0 or higher	N/A	Use a minor version upgrade procedure to upgrade between Aurora MySQL version 3 versions.
Aurora Serverless v1 cluster	N/A	Currently, Aurora Serverless v1 is supported for Aurora MySQL only on version 2.

Type of Aurora MySQL DB cluster	Can it use in-place upgrade?	Action
Aurora Serverless v2 cluster	N/A	Currently, Aurora Serverless v2 is supported for Aurora MySQL only on version 3.
Cluster in an Aurora global database	Yes	To upgrade Aurora MySQL from version 2 to version 3, follow the procedure for doing an in-place upgrade for clusters in an Aurora global database. Perform the upgrade on the global cluster. Aurora upgrades the primary cluster and all the secondary clusters in the global database at the same time. If you use the AWS CLI or RDS API, call the `modify-global-cluster` command or `ModifyGlobalCluster` operation instead of `modify-db-cluster` or `ModifyDBCluster` . You can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. For more information, see Major version upgrades.
Parallel query cluster	Yes	You can perform an in-place upgrade. In this case, choose 2.09.1 or higher for the Aurora MySQL version.
Cluster that is the target of binary log replication	Maybe	If the binary log replication is from an Aurora MySQL cluster, you can perform an in-place upgrade. You can't perform the upgrade if the binary log replication is from an RDS for MySQL or an on-premises MySQL DB instance. In that case, you can upgrade using the snapshot restore mechanism.

Type of Aurora MySQL DB cluster	Can it use in-place upgrade?	Action
Cluster with zero DB instances	No	Using the AWS CLI or the RDS API, you can create an Aurora MySQL cluster without any attached DB instances. In the same way, you can also remove all DB instances from an Aurora MySQL cluster while leaving the data in the cluster volume intact. While a cluster has zero DB instances, you can't perform an in-place upgrade. The upgrade mechanism requires a writer instance in the cluster to perform conversions on the system tables, data files, and so on. In this case, use the AWS CLI or the RDS API to create a writer instance for the cluster. Then you can perform an in-place upgrade.
Cluster with backtrack enabled	Yes	You can perform an in-place upgrade for an Aurora MySQL cluster that uses the backtrack feature. However, after the upgrade, you can't backtrack the cluster to a time before the upgrade.

How the Aurora MySQL in-place major version upgrade works

Aurora MySQL performs a major version upgrade as a multistage process. You can check the current status of an upgrade. Some of the upgrade steps also provide progress information. As each stage begins, Aurora MySQL records an event. You can examine events as they occur on the **Events** page in the RDS console. For more information about working with events, see Working with Amazon RDS event notification.

> ⚠ **Important**
>
> Once the process begins, it runs until the upgrade either succeeds or fails. You can't cancel the upgrade while it's underway. If the upgrade fails, Aurora rolls back all the changes and your cluster has the same engine version, metadata, and so on as before.

The upgrade process consists of these stages:

1. Aurora performs a series of prechecks before beginning the upgrade process. Your cluster keeps running while Aurora does these checks. For example, the cluster can't have any XA transactions in the prepared state or be processing any data definition language (DDL) statements. For example, you might need to shut down applications that are submitting certain kinds of SQL statements. Or you might simply wait until certain long-running statements are finished. Then try the upgrade again. Some checks test for conditions that don't prevent the upgrade but might make the upgrade take a long time.

   If Aurora detects that any required conditions aren't met, modify the conditions identified in the event details. Follow the guidance in Troubleshooting for Aurora MySQL in-place upgrade. If Aurora detects conditions that might cause a slow upgrade, plan to monitor the upgrade over an extended period.
2. Aurora takes your cluster offline. Then Aurora performs a similar set of tests as in the previous stage, to confirm that no new issues arose during the shutdown process. If Aurora detects any conditions at this point that would prevent the upgrade, Aurora cancels the upgrade and brings the cluster back online. In this case, confirm when the conditions no longer apply and start the upgrade again.
3. Aurora creates a snapshot of your cluster volume. Suppose that you discover compatibility or other kinds of issues after the upgrade is finished. Or suppose that you want to perform testing using both the original and upgraded clusters. In such cases, you can restore from this snapshot to create a new cluster with the original engine version and the original data.

   > **Tip**
   >
   > This snapshot is a manual snapshot. However, Aurora can create it and continue with the upgrade process even if you have reached your quota for manual snapshots. This snapshot remains permanently (if needed) until you delete it. After you finish all post-upgrade testing, you can delete this snapshot to minimize storage charges.
4. Aurora clones your cluster volume. Cloning is a fast operation that doesn't involve copying the actual table data. If Aurora encounters an issue during the upgrade, it reverts to the original data from the cloned cluster volume and brings the cluster back online. The temporary cloned volume during the upgrade isn't subject to the usual limit on the number of clones for a single cluster volume.

5. Aurora performs a clean shutdown for the writer DB instance. During the clean shutdown, progress events are recorded every 15 minutes for the following operations. You can examine events as they occur on the **Events** page in the RDS console.
   - Aurora purges the undo records for old versions of rows.
   - Aurora rolls back any uncommitted transactions.
6. Aurora upgrades the engine version on the writer DB instance:
   - Aurora installs the binary for the new engine version on the writer DB instance.
   - Aurora uses the writer DB instance to upgrade your data to MySQL 5.7-compatible format. During this stage, Aurora modifies the system tables and performs other conversions that affect the data in your cluster volume. In particular, Aurora upgrades the partition metadata in the system tables to be compatible with the MySQL 5.7 partition format. This stage can take a long time if the tables in your cluster have a large number of partitions.

     If any errors occur during this stage, you can find the details in the MySQL error logs. After this stage starts, if the upgrade process fails for any reason, Aurora restores the original data from the cloned cluster volume.
7. Aurora upgrades the engine version on the reader DB instances.
8. The upgrade process is completed. Aurora records a final event to indicate that the upgrade process completed successfully. Now your DB cluster is running the new major version.

Blue/Green Deployments

In some situations, your top priority is to perform an immediate switchover from the old cluster to an upgraded one. In such situations, you can use a multistep process that runs the old and new clusters side-by-side. Here, you replicate data from the old cluster to the new one until you are ready for the new cluster to take over. For details, see Using Amazon RDS Blue/Green Deployments for database updates.

How to perform an in-place upgrade

We recommend that you review the background material in How the Aurora MySQL in-place major version upgrade works.

Perform any preupgrade planning and testing, as described in Planning a major version upgrade for an Aurora MySQL cluster.

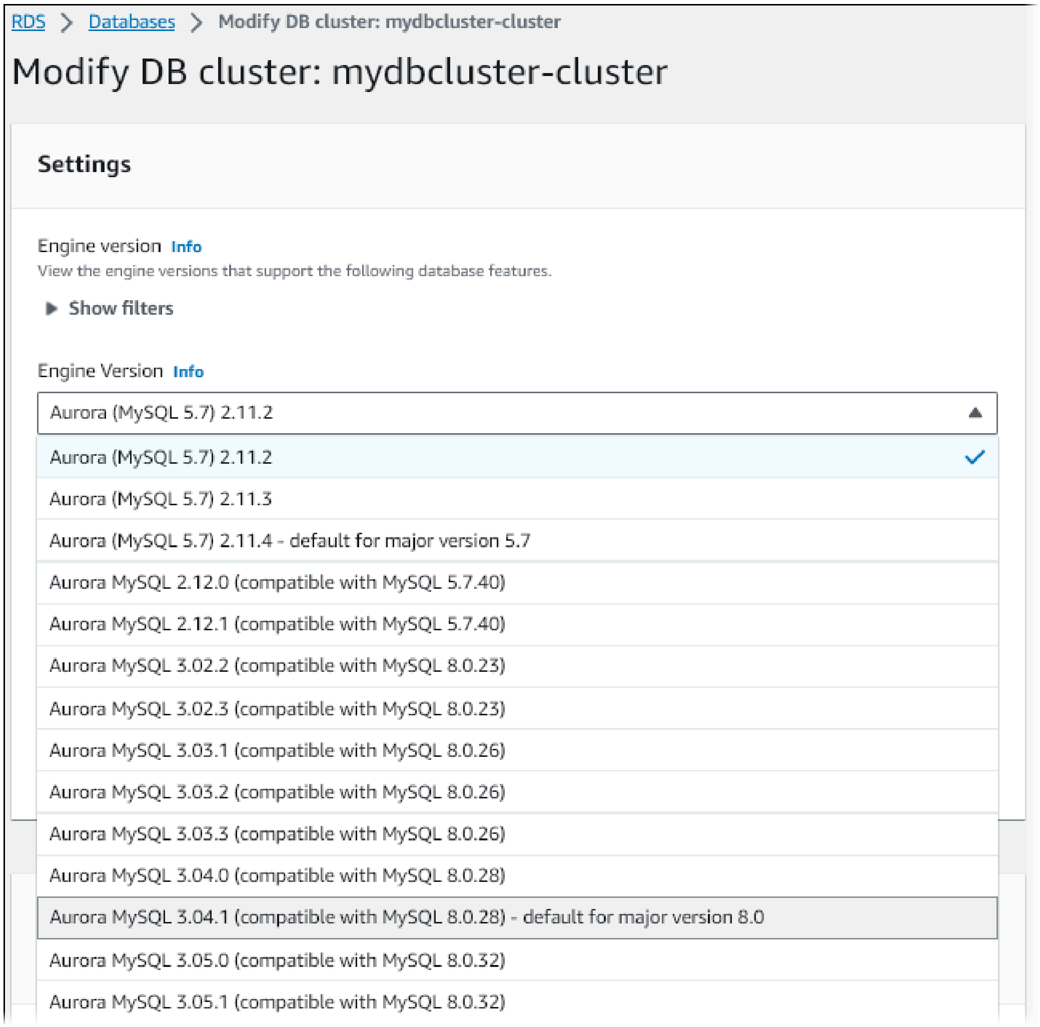
Console

The following example upgrades the `mydbcluster-cluster` DB cluster to Aurora MySQL version 3.04.1.

To upgrade the major version of an Aurora MySQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. If you used a custom parameter group for the original DB cluster, create a corresponding parameter group compatible with the new major version. Make any necessary adjustments to the configuration parameters in that new parameter group. For more information, see How in-place upgrades affect the parameter groups for a cluster.
3. In the navigation pane, choose **Databases**.
4. In the list, choose the DB cluster that you want to modify.
5. Choose **Modify**.
6. For **Version**, choose a new Aurora MySQL major version.

   We generally recommend using the latest minor version of the major version. Here, we choose the current default version.



RDS > Databases > Modify DB cluster: mydbcluster-cluster

Modify DB cluster: mydbcluster-cluster

Settings

Engine version Info

View the engine versions that support the following database features.

▶ Show filters

Engine Version Info

Aurora (MySQL 5.7) 2.11.2

- Aurora (MySQL 5.7) 2.11.2 ✓
- Aurora (MySQL 5.7) 2.11.3
- Aurora (MySQL 5.7) 2.11.4 - default for major version 5.7
- Aurora MySQL 2.12.0 (compatible with MySQL 5.7.40)
- Aurora MySQL 2.12.1 (compatible with MySQL 5.7.40)
- Aurora MySQL 3.02.2 (compatible with MySQL 8.0.23)
- Aurora MySQL 3.02.3 (compatible with MySQL 8.0.23)
- Aurora MySQL 3.03.1 (compatible with MySQL 8.0.26)
- Aurora MySQL 3.03.2 (compatible with MySQL 8.0.26)
- Aurora MySQL 3.03.3 (compatible with MySQL 8.0.26)
- Aurora MySQL 3.04.0 (compatible with MySQL 8.0.28)
- Aurora MySQL 3.04.1 (compatible with MySQL 8.0.28) - default for major version 8.0
- Aurora MySQL 3.05.0 (compatible with MySQL 8.0.32)
- Aurora MySQL 3.05.1 (compatible with MySQL 8.0.32)

7. Choose **Continue**.

8. On the next page, specify when to perform the upgrade. Choose **During the next scheduled maintenance window** or **Immediately**.

9. (Optional) Periodically examine the **Events** page in the RDS console during the upgrade. Doing so helps you to monitor the progress of the upgrade and identify any issues. If the upgrade encounters any issues, consult Troubleshooting for Aurora MySQL in-place upgrade for the steps to take.

10. If you created a new parameter group at the start of this procedure, associate the custom parameter group with your upgraded cluster. For more information, see How in-place upgrades affect the parameter groups for a cluster.

> **Note**
>
> Performing this step requires you to restart the cluster again to apply the new parameter group.

11. (Optional) After you complete any post-upgrade testing, delete the manual snapshot that Aurora created at the beginning of the upgrade.

AWS CLI

To upgrade the major version of an Aurora MySQL DB cluster, use the AWS CLI modify-db-cluster command with the following required parameters:

- `--db-cluster-identifier`
- `--engine-version`
- `--allow-major-version-upgrade`
- `--apply-immediately` or `--no-apply-immediately`

If your cluster uses any custom parameter groups, also include one or both of the following options:

- `--db-cluster-parameter-group-name`, if the cluster uses a custom cluster parameter group
- `--db-instance-parameter-group-name`, if any instances in the cluster use a custom DB parameter group

The following example upgrades the `sample-cluster` DB cluster to Aurora MySQL version 3.04.1. The upgrade happens immediately, instead of waiting for the next maintenance window.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
          --db-cluster-identifier sample-cluster \
          --engine-version 8.0.mysql_aurora.3.04.1 \
          --allow-major-version-upgrade \
          --apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
          --db-cluster-identifier sample-cluster ^
          --engine-version 8.0.mysql_aurora.3.04.1 ^
          --allow-major-version-upgrade ^
          --apply-immediately
```

You can combine other CLI commands with `modify-db-cluster` to create an automated end-to-end process for performing and verifying upgrades. For more information and examples, see Aurora MySQL in-place upgrade tutorial.

> **Note**
>
> If your cluster is part of an Aurora global database, the in-place upgrade procedure is slightly different. You call the modify-global-cluster command operation instead of `modify-db-cluster`. For more information, see In-place major upgrades for global databases.

RDS API

To upgrade the major version of an Aurora MySQL DB cluster, use the RDS API operation ModifyDBCluster with the following required parameters:

- `DBClusterIdentifier`
- `Engine`
- `EngineVersion`
- `AllowMajorVersionUpgrade`
- `ApplyImmediately` (set to `true` or `false`)

> **Note**
>
> If your cluster is part of an Aurora global database, the in-place upgrade procedure is slightly different. You call the ModifyGlobalCluster operation instead of `ModifyDBCluster`. For more information, see In-place major upgrades for global databases.

How in-place upgrades affect the parameter groups for a cluster

Aurora parameter groups have different sets of configuration settings for clusters that are compatible with MySQL 5.7 or 8.0. When you perform an in-place upgrade, the upgraded cluster and all its instances must use the corresponding cluster and instance parameter groups:

Your cluster and instances might use the default 5.7-compatible parameter groups. If so, the upgraded cluster and instance start with the default 8.0-compatible parameter groups. If your cluster and instances use any custom parameter groups, make sure to create corresponding or 8.0-compatible parameter groups. Also make sure to specify those during the upgrade process.

> **Note**
>
> For most parameter settings, you can choose the custom parameter group at two points. These are when you create the cluster or associate the parameter group with the cluster later.
> However, if you use a nondefault setting for the `lower_case_table_names` parameter, you must set up the custom parameter group with this setting in advance. Then specify the parameter group when you perform the snapshot restore to create the cluster. Any change to the `lower_case_table_names` parameter has no effect after the cluster is created.
> We recommend that you use the same setting for `lower_case_table_names` when you upgrade from Aurora MySQL version 2 to version 3.
> With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. For more information on the methods that you can use, see Major version upgrades.

> **Important**
>
> If you specify any custom parameter group during the upgrade process, make sure to manually reboot the cluster after the upgrade finishes. Doing so makes the cluster begin using your custom parameter settings.

Changes to cluster properties between Aurora MySQL versions

When you upgrade from Aurora MySQL version 2 to version 3, make sure to check any applications or scripts that you use to set up or manage Aurora MySQL clusters and DB instances.

Also, change your code that manipulates parameter groups to account for the fact that the default parameter group names are different for 5.7- and 8.0-compatible clusters. The default parameter group names for Aurora MySQL version 2 and 3 clusters are `default.aurora-mysql5.7` and `default.aurora-mysql8.0`, respectively.

For example, you might have code like the following that applies to your cluster before an upgrade.

```
# Check the default parameter values for MySQL 5.7–compatible clusters.
aws rds describe-db-parameters --db-parameter-group-name default.aurora-mysql5.7 --
region us-east-1
```

After upgrading the major version of the cluster, modify that code as follows.

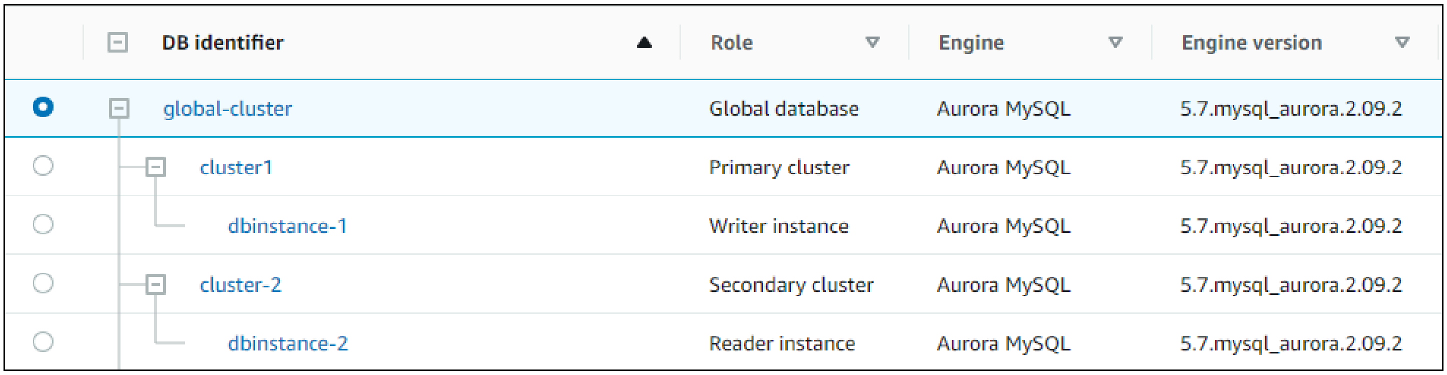
```
# Check the default parameter values for MySQL 8.0–compatible clusters.
aws rds describe-db-parameters --db-parameter-group-name default.aurora-mysql8.0 --
region us-east-1
```

In-place major upgrades for global databases

For an Aurora global database, you upgrade the global database cluster. Aurora automatically upgrades all of the clusters at the same time and makes sure that they all run the same engine version. This requirement is because any changes to system tables, data file formats, and so on, are automatically replicated to all the secondary clusters.

Follow the instructions in How the Aurora MySQL in-place major version upgrade works. When you specify what to upgrade, make sure to choose the global database cluster instead of one of the clusters it contains.

If you use the AWS Management Console, choose the item with the role **Global database**.



	⊟ DB identifier ▲	Role ▽	Engine ▽	Engine version ▽
●	⊟ global-cluster	Global database	Aurora MySQL	5.7.mysql_aurora.2.09.2
○	⊟ cluster1	Primary cluster	Aurora MySQL	5.7.mysql_aurora.2.09.2
○	dbinstance-1	Writer instance	Aurora MySQL	5.7.mysql_aurora.2.09.2
○	⊟ cluster-2	Secondary cluster	Aurora MySQL	5.7.mysql_aurora.2.09.2
○	dbinstance-2	Reader instance	Aurora MySQL	5.7.mysql_aurora.2.09.2

If you use the AWS CLI or RDS API, start the upgrade process by calling the modify-global-cluster command or ModifyGlobalCluster operation. You use one of these instead of `modify-db-cluster` or `ModifyDBCluster`.

> **Note**
>
> You can't specify a custom parameter group for the global database cluster while you're performing a major version upgrade of that Aurora global database. Create your custom parameter groups in each Region of the global cluster. Then apply them manually to the Regional clusters after the upgrade.

To upgrade the major version of an Aurora MySQL global database cluster by using the AWS CLI, use the modify-global-cluster command with the following required parameters:

- `--global-cluster-identifier`
- `--engine aurora-mysql`
- `--engine-version`
- `--allow-major-version-upgrade`

The following example upgrades the global database cluster to Aurora MySQL version 2.10.2.

Example

For Linux, macOS, or Unix:

```
aws rds modify-global-cluster \
          --global-cluster-identifier global_cluster_identifier \
          --engine aurora-mysql \
          --engine-version 5.7.mysql_aurora.2.10.2 \
          --allow-major-version-upgrade
```

For Windows:

```
aws rds modify-global-cluster ^
          --global-cluster-identifier global_cluster_identifier ^
          --engine aurora-mysql ^
          --engine-version 5.7.mysql_aurora.2.10.2 ^
          --allow-major-version-upgrade
```

Backtrack considerations

If the cluster that you upgraded had the Backtrack feature enabled, you can't backtrack the upgraded cluster to a time that's before the upgrade.

Aurora MySQL in-place upgrade tutorial

The following Linux examples show how you might perform the general steps of the in-place upgrade procedure using the AWS CLI.

This first example creates an Aurora DB cluster that's running a 2.x version of Aurora MySQL. The cluster includes a writer DB instance and a reader DB instance. The `wait db-instance-available` command pauses until the writer DB instance is available. That's the point when the cluster is ready to be upgraded.

```
aws rds create-db-cluster --db-cluster-identifier mynewdbcluster --engine aurora-mysql
 \
  --db-cluster-version 5.7.mysql_aurora.2.10.2
...
aws rds create-db-instance --db-instance-identifier mynewdbcluster-instance1 \
  --db-cluster-identifier mynewdbcluster --db-instance-class db.t4g.medium --engine
 aurora-mysql
...
aws rds wait db-instance-available --db-instance-identifier mynewdbcluster-instance1
```

The Aurora MySQL 3.x versions that you can upgrade the cluster to depend on the 2.x version that the cluster is currently running and on the AWS Region where the cluster is located. The first command, with `--output text`, just shows the available target version. The second command shows the full JSON output of the response. In that response, you can see details such as the `aurora-mysql` value that you use for the `engine` parameter. You can also see the fact that upgrading to 3.02.0 represents a major version upgrade.

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster \
  --query '*[].{EngineVersion:EngineVersion}' --output text
5.7.mysql_aurora.2.10.2

aws rds describe-db-engine-versions --engine aurora-mysql --engine-version
 5.7.mysql_aurora.2.10.2 \
  --query '*[].[ValidUpgradeTarget]'
...
{
    "Engine": "aurora-mysql",
    "EngineVersion": "8.0.mysql_aurora.3.02.0",
    "Description": "Aurora MySQL 3.02.0 (compatible with MySQL 8.0.23)",
    "AutoUpgrade": false,
    "IsMajorVersionUpgrade": true,
    "SupportedEngineModes": [
        "provisioned"
    ],
    "SupportsParallelQuery": true,
    "SupportsGlobalDatabases": true,
    "SupportsBabelfish": false
},
...
```

This example shows how if you enter a target version number that isn't a valid upgrade target for the cluster, Aurora doesn't perform the upgrade. Aurora also doesn't perform a major version upgrade unless you include the `--allow-major-version-upgrade` parameter. That way, you can't accidentally perform an upgrade that has the potential to require extensive testing and changes to your application code.

```
aws rds modify-db-cluster --db-cluster-identifier mynewdbcluster \
  --engine-version 5.7.mysql_aurora.2.09.2 --apply-immediately
An error occurred (InvalidParameterCombination) when calling the ModifyDBCluster
 operation: Cannot find upgrade target from 5.7.mysql_aurora.2.10.2 with requested
 version 5.7.mysql_aurora.2.09.2.

aws rds modify-db-cluster --db-cluster-identifier mynewdbcluster \
  --engine-version 8.0.mysql_aurora.3.02.0 --region us-east-1 --apply-immediately
An error occurred (InvalidParameterCombination) when calling the ModifyDBCluster
 operation: The AllowMajorVersionUpgrade flag must be present when upgrading to a new
 major version.

aws rds modify-db-cluster --db-cluster-identifier mynewdbcluster \
  --engine-version 8.0.mysql_aurora.3.02.0 --apply-immediately --allow-major-version-
upgrade
{
  "DBClusterIdentifier": "mynewdbcluster",
  "Status": "available",
  "Engine": "aurora-mysql",
  "EngineVersion": "5.7.mysql_aurora.2.10.2"
}
```

It takes a few moments for the status of the cluster and associated DB instances to change to `upgrading`. The version numbers for the cluster and DB instances only change when the upgrade is finished. Again, you can use the `wait db-instance-available` command for the writer DB instance to wait until the upgrade is finished before proceeding.

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster \
  --query '*[].[Status,EngineVersion]' --output text
upgrading 5.7.mysql_aurora.2.10.2

aws rds describe-db-instances --db-instance-identifier mynewdbcluster-instance1 \
  --query '*[].
{DBInstanceIdentifier:DBInstanceIdentifier,DBInstanceStatus:DBInstanceStatus} | [0]'
{
    "DBInstanceIdentifier": "mynewdbcluster-instance1",
    "DBInstanceStatus": "upgrading"
}

aws rds wait db-instance-available --db-instance-identifier mynewdbcluster-instance1
```

At this point, the version number for the cluster matches the one that was specified for the upgrade.

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster \
  --query '*[].[EngineVersion]' --output text

8.0.mysql_aurora.3.02.0
```

The preceding example did an immediate upgrade by specifying the `--apply-immediately` parameter. To let the upgrade happen at a convenient time when the cluster isn't expected to be busy, you can specify the `--no-apply-immediately` parameter. Doing so makes the upgrade start during the next maintenance window for the cluster. The maintenance window defines the period during which maintenance operations can start. A long-running operation might not finish during the maintenance window. Thus, you don't need to define a larger maintenance window even if you expect that the upgrade might take a long time.

The following example upgrades a cluster that's initially running Aurora MySQL version 2.10.2. In the `describe-db-engine-versions` output, the `False` and `True` values represent the `IsMajorVersionUpgrade` property. From version 2.10.2, you can upgrade to some other 2.*

versions. Those upgrades aren't considered major version upgrades and so don't require an in-place upgrade. In-place upgrade is only available for upgrades to the 3.* versions that are shown in the list.

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster \
  --query '*[].{EngineVersion:EngineVersion}' --output text
5.7.mysql_aurora.2.10.2

aws rds describe-db-engine-versions --engine aurora-mysql --engine-version
 5.7.mysql_aurora.2.10.2 \
  --query '*[].[ValidUpgradeTarget]|[0][0]|[*].[EngineVersion,IsMajorVersionUpgrade]'
 --output text

5.7.mysql_aurora.2.10.3 False
5.7.mysql_aurora.2.11.0 False
5.7.mysql_aurora.2.11.1 False
8.0.mysql_aurora.3.01.1 True
8.0.mysql_aurora.3.02.0 True
8.0.mysql_aurora.3.02.2 True


aws rds modify-db-cluster --db-cluster-identifier mynewdbcluster \
  --engine-version 8.0.mysql_aurora.3.02.0 --no-apply-immediately --allow-major-
version-upgrade
...
```

When a cluster is created without a specified maintenance window, Aurora picks a random day of the week. In this case, the `modify-db-cluster` command is submitted on a Monday. Thus, we change the maintenance window to be Tuesday morning. All times are represented in the UTC time zone. The `tue:10:00-tue:10:30` window corresponds to 2:00-2:30 AM Pacific time. The change in the maintenance window takes effect immediately.

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster --query '*[].
[PreferredMaintenanceWindow]'
[
    [
        "sat:08:20-sat:08:50"
    ]
]

aws rds modify-db-cluster --db-cluster-identifier mynewdbcluster --preferred-
maintenance-window tue:10:00-tue:10:30"
```

```
aws rds describe-db-clusters --db-cluster-identifier mynewdbcluster --query '*[].
[PreferredMaintenanceWindow]'
[
    [
        "tue:10:00-tue:10:30"
    ]
]
```

The following example shows how to get a report of the events generated by the upgrade. The `--duration` argument represents the number of minutes to retrieve the event information. This argument is needed, because by default `describe-events` only returns events from the last hour.

```
aws rds describe-events --source-type db-cluster --source-identifier mynewdbcluster --
duration 20160
{
    "Events": [
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "DB cluster created",
            "EventCategories": [
                "creation"
            ],
            "Date": "2022-11-17T01:24:11.093000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Performing online pre-upgrade checks.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T22:57:08.450000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Performing offline pre-upgrade checks.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T22:57:59.519000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-
mynewdbcluster-5-7-mysql-aurora-2-10-2-to-8-0-mysql-aurora-3-02-0-2022-11-18-22-55].",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:00:22.318000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Cloning volume.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:01:45.428000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Purging undo records for old row versions.
 Records remaining: 164",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:02:25.141000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Purging undo records for old row versions.
 Records remaining: 164",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:06:23.036000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Upgrade in progress: Upgrading database objects.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:06:48.208000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        },
        {
            "SourceIdentifier": "mynewdbcluster",
            "SourceType": "db-cluster",
            "Message": "Database cluster major version has been upgraded",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2022-11-18T23:10:28.999000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mynewdbcluster"
        }
    ]
}
```

Finding the reasons for upgrade failures

In the previous tutorial, the upgrade from Aurora MySQL version 2 to version 3 succeeded. But if the upgrade had failed, you would want to know why.

You can start by using the `describe-events` AWS CLI command to look at the DB cluster events. This example shows the events for `mydbcluster` for the last 10 hours.

```
aws rds describe-events \
    --source-type db-cluster \
    --source-identifier mydbcluster \
    --duration 600
```

In this case, we had an upgrade precheck failure.

```
{
    "Events": [
        {
            "SourceIdentifier": "mydbcluster",
            "SourceType": "db-cluster",
            "Message": "Database cluster engine version upgrade started.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2024-04-11T13:23:22.846000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster"
        },
        {
            "SourceIdentifier": "mydbcluster",
            "SourceType": "db-cluster",
            "Message": "Database cluster is in a state that cannot be upgraded: Upgrade
 prechecks failed. For more details, see the
             upgrade-prechecks.log file. For more information on troubleshooting the
 cause of the upgrade failure, see
             https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/
AuroraMySQL.Updates.MajorVersionUpgrade.html#AuroraMySQL.Upgrading.Troubleshooting.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2024-04-11T13:23:24.373000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster"
        }
    ]
}
```

To diagnose the exact cause of the problem, examine the database logs for the writer DB instance. When an upgrade to Aurora MySQL version 3 fails, the writer instance contains a log file with the name `upgrade-prechecks.log`. This example shows how to detect the presence of that log and then download it to a local file for examination.

```
aws rds describe-db-log-files --db-instance-identifier mydbcluster-instance \
    --query '*[].[LogFileName]' --output text

error/mysql-error-running.log
error/mysql-error-running.log.2024-04-11.20
error/mysql-error-running.log.2024-04-11.21
error/mysql-error.log
external/mysql-external.log
upgrade-prechecks.log

aws rds download-db-log-file-portion --db-instance-identifier mydbcluster-instance \
    --log-file-name upgrade-prechecks.log \
    --starting-token 0 \
    --output text >upgrade_prechecks.log
```

The `upgrade-prechecks.log` file is in JSON format. We download it using the `--output text` option to avoid encoding JSON output within another JSON wrapper. For Aurora MySQL version 3 upgrades, this log always includes certain informational and warning messages. It only includes error messages if the upgrade fails. If the upgrade succeeds, the log file isn't produced at all.

To summarize all of the errors and display the associated object and description fields, you can run the command `grep -A 2 '"level": "Error"'` on the contents of the `upgrade-prechecks.log` file. Doing so displays each error line and the two lines after it. These contain the name of the corresponding database object and guidance about how to correct the problem.

```
$ cat upgrade-prechecks.log | grep -A 2 '"level": "Error"'

"level": "Error",
"dbObject": "problematic_upgrade.dangling_fulltext_index",
"description": "Table `problematic_upgrade.dangling_fulltext_index` contains dangling
 FULLTEXT index. Kindly recreate the table before upgrade."
```

In this example, you can run the following SQL command on the offending table to try to fix the issue, or you can re-create the table without the dangling index.

```
OPTIMIZE TABLE problematic_upgrade.dangling_fulltext_index;
```

Then retry the upgrade.

Troubleshooting for Aurora MySQL in-place upgrade

Use the following tips to help troubleshoot problems with Aurora MySQL in-place upgrades. These tips don't apply to Aurora Serverless DB clusters.

Reason for in-place upgrade being canceled or slow	Effect	Solution to allow in-place upgrade to complete within maintenance window
Associated Aurora cross-Region replica isn't patched yet	Aurora cancels the upgrade.	Upgrade the Aurora cross-Region replica and try again.

Reason for in-place upgrade being canceled or slow	Effect	Solution to allow in-place upgrade to complete within maintenance window
Cluster has XA transactions in the prepared state	Aurora cancels the upgrade.	Commit or roll back all prepared XA transacti ons.
Cluster is processing any data definition language (DDL) statements	Aurora cancels the upgrade.	Consider waiting and performing the upgrade after all DDL statements are finished.
Cluster has uncommitted changes for many rows	Upgrade might take a long time.	The upgrade process rolls back the uncommitt ed changes. The indicator for this condition is the value of `TRX_ROWS_MODIFIED` in the `INFORMATION_SCHEMA.INNODB_TRX` table. Consider performing the upgrade only after all large transactions are committed or rolled back.

Reason for in-place upgrade being canceled or slow	Effect	Solution to allow in-place upgrade to complete within maintenance window
Cluster has high number of undo records	Upgrade might take a long time.	Even if the uncommitted transactions don't affect a large number of rows, they might involve a large volume of data. For example, you might be inserting large BLOBs. Aurora doesn't automatically detect or generate an event for this kind of transaction activity. The indicator for this condition is the history list length (HLL). The upgrade process rolls back the uncommitted changes. You can check the HLL in the output from the `SHOW ENGINE INNODB STATUS` SQL command, or directly by using the following SQL query: `SELECT count FROM information_schema.innodb_metrics WHERE name = 'trx_rseg_history_len';` You can also monitor the `RollbackSegmentHistoryListLength` metric in Amazon CloudWatch. Consider performing the upgrade only after the HLL is smaller.

Reason for in-place upgrade being canceled or slow	Effect	Solution to allow in-place upgrade to complete within maintenance window
Cluster is in the process of committing a large binary log transaction	Upgrade might take a long time.	The upgrade process waits until the binary log changes are applied. More transactions or DDL statements could start during this period, further slowing down the upgrade process. Schedule the upgrade process when the cluster isn't busy with generating binary log replication changes. Aurora doesn't automatically detect or generate an event for this condition.
Schema inconsistencies resulting from file removal or corruption	Aurora cancels the upgrade.	Change the default storage engine for temporary tables from MyISAM to InnoDB. Perform the following steps: 1. Modify the `default_tmp_storage_engine` DB parameter to `InnoDB`. 2. Reboot the DB cluster. 3. After rebooting, confirm that the `default_tmp_storage_engine` DB parameter is set to `InnoDB`. Use the following command: `show global variables like 'default_tmp_storage_engine';` 4. Make sure not to create any temporary tables that use the MyISAM storage engine. We recommend that you pause any database workload and not create any new database connections, because you're upgrading soon. 5. Try the in-place upgrade again.

Reason for in-place upgrade being canceled or slow	Effect	Solution to allow in-place upgrade to complete within maintenance window

Master user was deleted	Aurora cancels the upgrade.	⚠ **Important**
		Don't delete the master user.
		However, if for some reason you should happen to delete the master user, restore it using the following SQL commands:
		`CREATE USER 'master_username '@'%' IDENTIFIED BY ' master_user_password ' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK;`
		`GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD, PROCESS, REFERENCES, INDEX, ALTER, SHOW DATABASES, CREATE TEMPORARY TABLES, LOCK TABLES, EXECUTE, REPLICATION SLAVE, REPLICATION CLIENT, CREATE VIEW, SHOW VIEW, CREATE ROUTINE, ALTER ROUTINE, CREATE USER, EVENT, TRIGGER, LOAD FROM S3, SELECT INTO S3, INVOKE LAMBDA, INVOKE SAGEMAKER , INVOKE COMPREHEND ON *.* TO 'master_username '@'%' WITH GRANT OPTION;`

For more details on troubleshooting issues that cause upgrade prechecks to fail, see the following blogs:

- Amazon Aurora MySQL version 2 (with MySQL 5.7 compatibility) to version 3 (with MySQL 8.0 compatibility) upgrade checklist, Part 1
- Amazon Aurora MySQL version 2 (with MySQL 5.7 compatibility) to version 3 (with MySQL 8.0 compatibility) upgrade checklist, Part 2

You can use the following steps to perform your own checks for some of the conditions in the preceding table. That way, you can schedule the upgrade at a time when you know the database is in a state where the upgrade can complete successfully and quickly.

- You can check for open XA transactions by executing the `XA RECOVER` statement. You can then commit or roll back the XA transactions before starting the upgrade.
- You can check for DDL statements by executing a `SHOW PROCESSLIST` statement and looking for `CREATE`, `DROP`, `ALTER`, `RENAME`, and `TRUNCATE` statements in the output. Allow all DDL statements to finish before starting the upgrade.
- You can check the total number of uncommitted rows by querying the `INFORMATION_SCHEMA.INNODB_TRX` table. The table contains one row for each transaction. The `TRX_ROWS_MODIFIED` column contains the number of rows modified or inserted by the transaction.
- You can check the length of the InnoDB history list by executing the `SHOW ENGINE INNODB STATUS` SQL statement and looking for the `History list length` in the output. You can also check the value directly by running the following query:

  ```
  SELECT count FROM information_schema.innodb_metrics WHERE name =
   'trx_rseg_history_len';
  ```

  The length of the history list corresponds to the amount of undo information stored by the database to implement multi-version concurrency control (MVCC).

Post-upgrade cleanup for Aurora MySQL version 3

After you finish upgrading any Aurora MySQL version 2 clusters to Aurora MySQL version 3, you can perform these other cleanup actions:

- Create new MySQL 8.0–compatible versions of any custom parameter groups. Apply any necessary custom parameter values to the new parameter groups.
- Update any CloudWatch alarms, setup scripts, and so on to use the new names for any metrics whose names were affected by inclusive language changes. For a list of such metrics, see Inclusive language changes for Aurora MySQL version 3.
- Update any AWS CloudFormation templates to use the new names for any configuration parameters whose names were affected by inclusive language changes. For a list of such parameters, see Inclusive language changes for Aurora MySQL version 3.

Spatial indexes

After upgrading to Aurora MySQL version 3, check if you need to drop or recreate objects and indexes related to spatial indexes. Before MySQL 8.0, Aurora could optimize spatial queries using indexes that didn't contain a spatial resource identifier (SRID). Aurora MySQL version 3 only uses spatial indexes containing SRIDs. During an upgrade, Aurora automatically drops any spatial indexes without SRIDs and prints warning messages in the database log. If you observe such warning messages, create new spatial indexes with SRIDs after the upgrade. For more information about changes to spatial functions and data types in MySQL 8.0, see Changes in MySQL 8.0 in the *MySQL Reference Manual*.

Database engine updates and fixes for Amazon Aurora MySQL

You can find the following information in the *Release notes for Amazon Aurora MySQL-Compatible Edition*:

- Database engine updates for Amazon Aurora MySQL version 3
- Database engine updates for Amazon Aurora MySQL version 2
- Database engine updates for Amazon Aurora MySQL version 1 (Deprecated)
- MySQL bugs fixed by Aurora MySQL database engine updates
- Security vulnerabilities fixed in Amazon Aurora MySQL

Working with Amazon Aurora PostgreSQL

Amazon Aurora PostgreSQL is a fully managed, PostgreSQL–compatible, and ACID–compliant relational database engine that combines the speed, reliability, and manageability of Amazon Aurora with the simplicity and cost-effectiveness of open-source databases. Aurora PostgreSQL is a drop-in replacement for PostgreSQL and makes it simple and cost-effective to set up, operate, and scale your new and existing PostgreSQL deployments, thus freeing you to focus on your business and applications. To learn more about Aurora in general, see What is Amazon Aurora?.

In addition to the benefits of Aurora, Aurora PostgreSQL offers a convenient migration pathway from Amazon RDS into Aurora, with push-button migration tools that convert your existing RDS for PostgreSQL applications to Aurora PostgreSQL. Routine database tasks such as provisioning, patching, backup, recovery, failure detection, and repair are also easy to manage with Aurora PostgreSQL.

Aurora PostgreSQL can work with many industry standards. For example, you can use Aurora PostgreSQL databases to build HIPAA-compliant applications and to store healthcare related information, including protected health information (PHI), under a completed Business Associate Agreement (BAA) with AWS.

Aurora PostgreSQL is FedRAMP HIGH eligible. For details about AWS and compliance efforts, see AWS services in scope by compliance program.

Topics

- Best practices with Amazon Aurora PostgreSQL
- Replication with Amazon Aurora PostgreSQL
- Using Aurora PostgreSQL as a Knowledge Base for Amazon Bedrock
- Integrating Amazon Aurora PostgreSQL with other AWS services
- Monitoring query execution plans for Aurora PostgreSQL
- Managing query execution plans for Aurora PostgreSQL
- Working with extensions and foreign data wrappers
- Working with Trusted Language Extensions for PostgreSQL
- Amazon Aurora PostgreSQL reference
- Amazon Aurora PostgreSQL updates

Working with the database preview environment

The PostgreSQL community releases new major version of PostgreSQL annually. Similarly, Amazon Aurora makes PostgreSQL major versions available as Preview releases. This allows you to create DB cluster using the Preview version and test out its features in the Database Preview Environment.

Aurora PostgreSQL DB clusters in the Database Preview Environment are functionally similar to other Aurora PostgreSQL DB clusters. However, you can't use a Preview version for production.

Keep in mind the following important limitations:

- All DB instances and DB clusters are deleted 60 days after you create them, along with any backups and snapshots.
- You can only create a DB instance in a virtual private cloud (VPC) based on the Amazon VPC service.
- You can't copy a snapshot of a DB instance to a production environment.

The following options are supported by the Preview.

- You can create DB instances using r5, r6g, r6i, r7g, x2g, t3 and t4g instance types only. For more information about instance classes, see Aurora DB instance classes.
- You can use both single-AZ and multi-AZ deployments.
- You can use standard PostgreSQL dump and load functions to export databases from or import databases to the Database Preview Environment.

Supported DB instance class types

Amazon Aurora PostgreSQL supports the following DB instance classes in the preview region:

Memory Optimized Classes

- db.r5 – memory-optimized instance classes
- db.r6g – memory-optimized instance classes powered by AWS Graviton2 processors
- db.r6i – memory-optimized instance classes
- db.x2g – memory-optimized instance classes powered by AWS Graviton2 processors

> **Note**
>
> For more information on the list of instance classes, see DB instance class types.

Burstable classes

- db.t3.medium
- db.t3.large
- db.t4g.medium
- db.t4g.large

Unsupported features in the preview environment

The following features aren't available in the preview environment:

- Aurora Serverless v1 and v2
- Major version upgrades (MVU)
- No new minor versions will be released in preview region
- RDS for PostgreSQL to Aurora PostgreSQL inbound replication
- Amazon RDS Blue/Green deployment
- Cross-Region snapshot copy
- Aurora global database
- Database activity streams (DAS), RDS Proxy, and AWS DMS

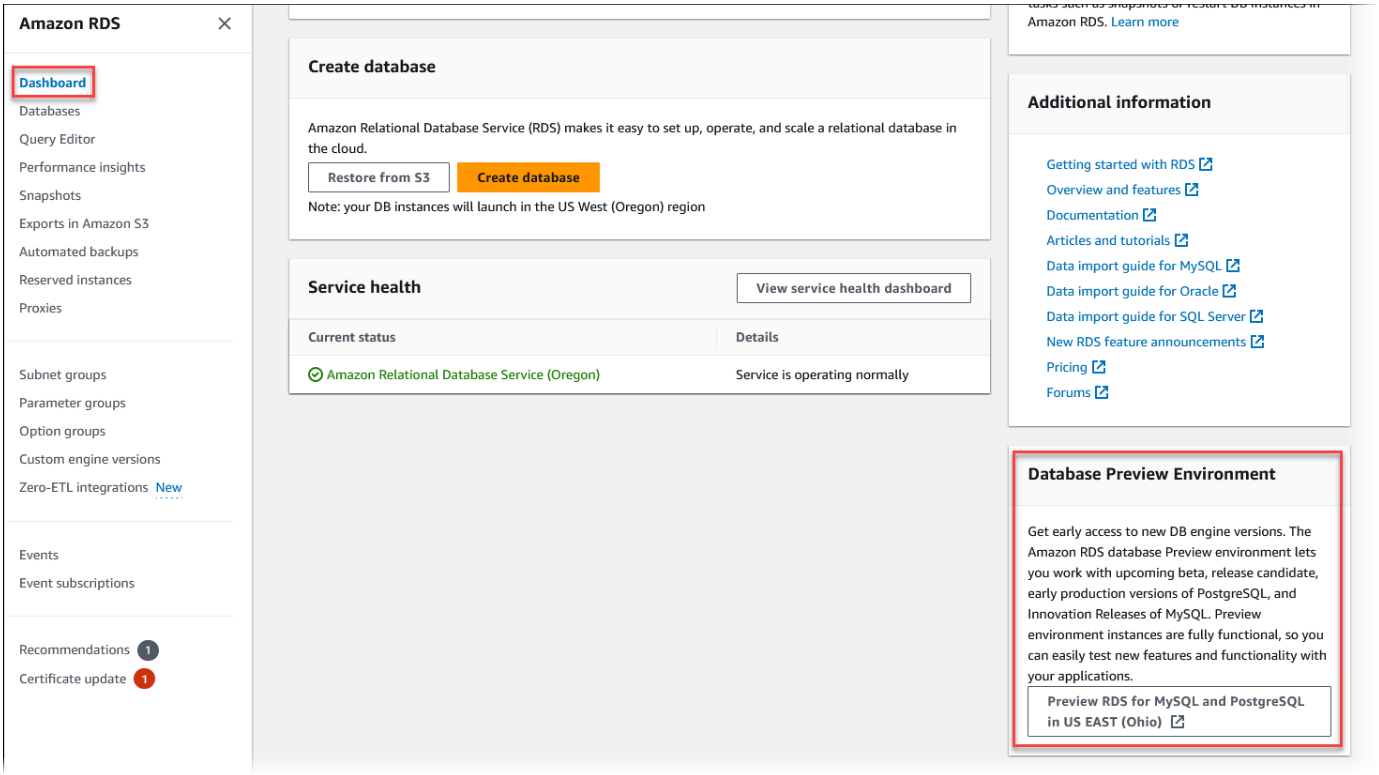
- Auto scaling read replicas
- AWS Bedrock
- RDS export
- Performance Insights
- Global write forwarding
- Optimized Reads
- Babelfish
- Custom endpoints
- Snapshot copy

Creating a new DB cluster in the preview environment

Use the following procedure to create a DB cluster in the preview environment.

To create a DB cluster in the preview environment

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Dashboard** from the navigation pane.
3. In the Dashboard page, locate the **Database Preview Environment** section on the Dashboard page, as shown in the following image.



Amazon RDS ✕

Dashboard

Databases

Query Editor

Performance insights

Snapshots

Exports in Amazon S3

Automated backups

Reserved instances

Proxies

Subnet groups

Parameter groups

Option groups

Custom engine versions

Zero-ETL integrations New

Events

Event subscriptions

Recommendations 1

Certificate update 1

Create database

Amazon Relational Database Service (RDS) makes it easy to set up, operate, and scale a relational database in the cloud.

Restore from S3 | Create database

Note: your DB instances will launch in the US West (Oregon) region

Service health | View service health dashboard

Current status	Details
✓ Amazon Relational Database Service (Oregon)	Service is operating normally

tasks such as snapshots or restart DB instances in Amazon RDS. Learn more

Additional information

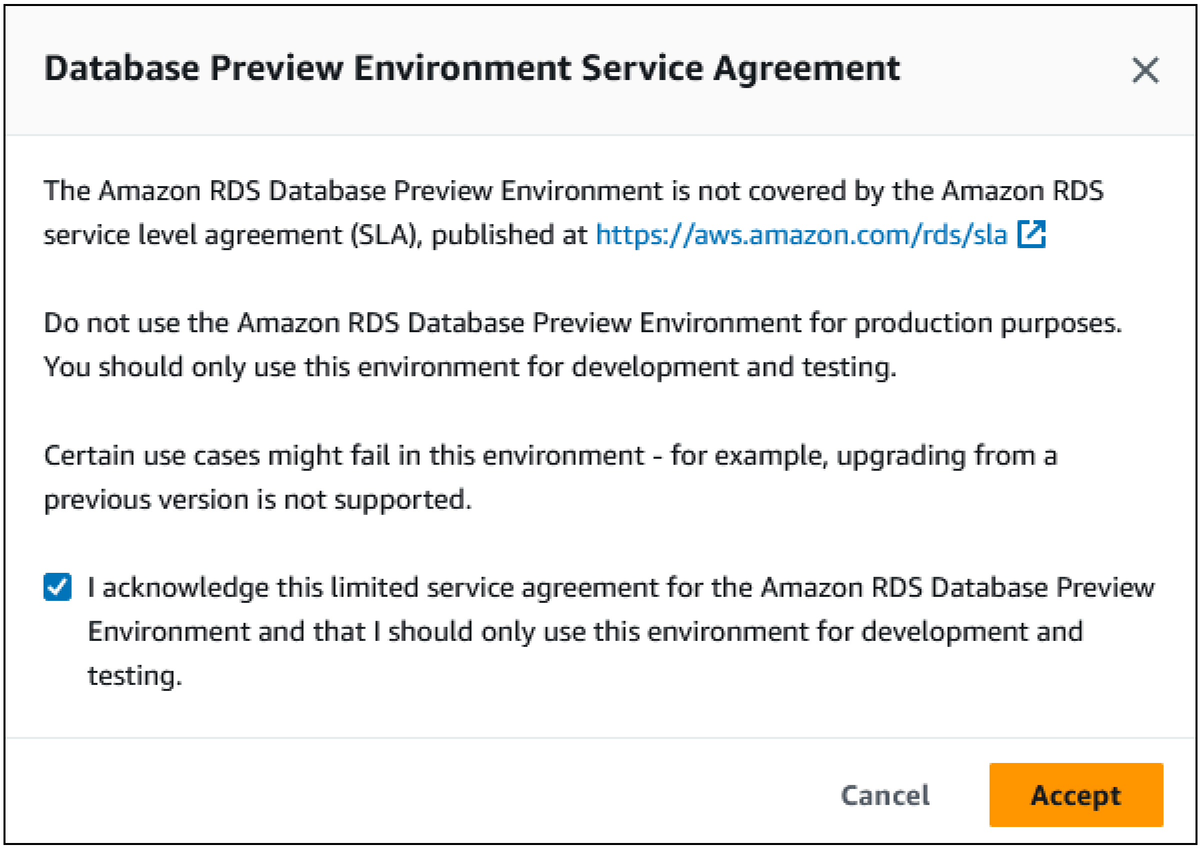
- Getting started with RDS
- Overview and features
- Documentation
- Articles and tutorials
- Data import guide for MySQL
- Data import guide for Oracle
- Data import guide for SQL Server
- New RDS feature announcements
- Pricing
- Forums

Database Preview Environment

Get early access to new DB engine versions. The Amazon RDS database Preview environment lets you work with upcoming beta, release candidate, early production versions of PostgreSQL, and Innovation Releases of MySQL. Preview environment instances are fully functional, so you can easily test new features and functionality with your applications.

Preview RDS for MySQL and PostgreSQL in US EAST (Ohio)

You can navigate directly to the Database preview environment. Before you can proceed, you must acknowledge and accept the limitations.



Database Preview Environment Service Agreement

The Amazon RDS Database Preview Environment is not covered by the Amazon RDS service level agreement (SLA), published at https://aws.amazon.com/rds/sla

Do not use the Amazon RDS Database Preview Environment for production purposes. You should only use this environment for development and testing.

Certain use cases might fail in this environment - for example, upgrading from a previous version is not supported.

☑ I acknowledge this limited service agreement for the Amazon RDS Database Preview Environment and that I should only use this environment for development and testing.

4. To create the Aurora PostgreSQL DB cluster, follow the same process as that for creating any Aurora DB cluster. For more information, see Creating an Amazon Aurora DB cluster.

To create an instance in the Database Preview Environment using the Aurora API or the AWS CLI, use the following endpoint.

```
rds-preview.us-east-2.amazonaws.com
```

PostgreSQL version 16 in the Database Preview environment

⚠ ***This is preview documentation for Aurora PostgreSQL version 16. It is subject to change.***

PostgreSQL version 16.0 is now available in the Amazon RDS Database Preview environment. PostgreSQL version 16 contains several improvements that are described in the following PostgreSQL documentation:

- PostgreSQL 16 Released

For information on the Database Preview Environment, see Working with the database preview environment. To access the Preview Environment from the console, select https://console.aws.amazon.com/rds-preview/.

> **Note**
>
> It is not recommend to use PostgreSQL version 16.0 in the Database Preview environment as Aurora PostgreSQL version 16.1 is now generally available. For more information, see Amazon Aurora PostgreSQL updates.

Security with Amazon Aurora PostgreSQL

For a general overview of Aurora security, see Security in Amazon Aurora. You can manage security for Amazon Aurora PostgreSQL at a few different levels:

- To control who can perform Amazon RDS management actions on Aurora PostgreSQL DB clusters and DB instances, use AWS Identity and Access Management (IAM). IAM handles the authentication of user identity before the user can access the service. It also handles authorization, that is, whether the user is allowed to do what they're trying to do. IAM database authentication is an additional authentication method that you can choose when you create your Aurora PostgreSQL DB cluster. For more information, see Identity and access management for Amazon Aurora.

  If you do use IAM with your Aurora PostgreSQL DB cluster, sign in to the AWS Management Console with your IAM credentials first, before opening the Amazon RDS console at https://console.aws.amazon.com/rds/.
- Make sure to create Aurora DB clusters in a virtual private cloud (VPC) based on the Amazon VPC service. To control which devices and Amazon EC2 instances can open connections to the endpoint and port of the DB instance for Aurora DB clusters in a VPC, use a VPC security group. You can make these endpoint and port connections by using Secure Sockets Layer (SSL). In addition, firewall rules at your company can control whether devices running at your company can open connections to a DB instance. For more information on VPCs, see Amazon VPC VPCs and Amazon Aurora.

  The supported VPC tenancy depends on the DB instance class used by your Aurora PostgreSQL DB clusters. With `default` VPC tenancy, the DB cluster runs on shared hardware. With `dedicated` VPC tenancy, the DB cluster runs on a dedicated hardware instance. The burstable performance DB instance classes support default VPC tenancy only. The burstable performance DB instance classes include the db.t3 and db.t4g DB instance classes. All other Aurora PostgreSQL DB instance classes support both default and dedicated VPC tenancy.

For more information about instance classes, see Aurora DB instance classes. For more information about `default` and `dedicated` VPC tenancy, see Dedicated instances in the *Amazon Elastic Compute Cloud User Guide*.

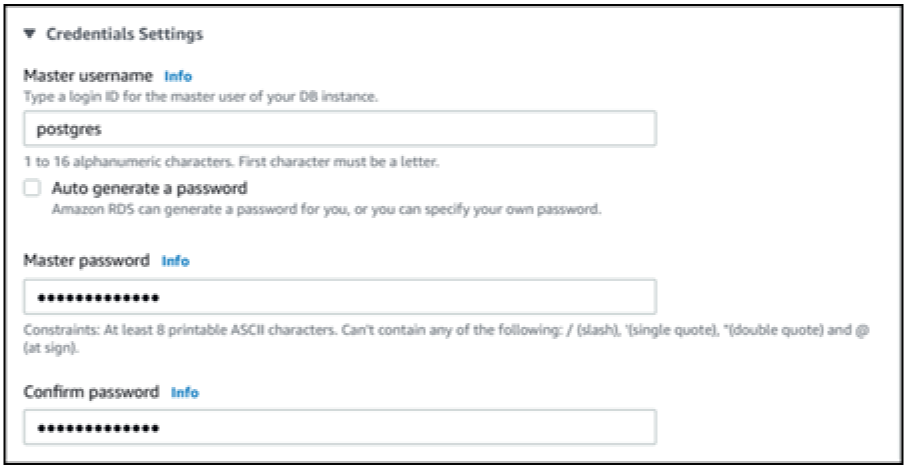
- To grant permissions to the PostgreSQL databases running on your Amazon Aurora DB cluster, you can take the same general approach as with stand-alone instances of PostgreSQL. Commands such as `CREATE ROLE`, `ALTER ROLE`, `GRANT`, and `REVOKE` work just as they do in on-premises databases, as does directly modifying databases, schemas, and tables.

  PostgreSQL manages privileges by using *roles*. The `rds_superuser` role is the most privileged role on an Aurora PostgreSQL DB cluster. This role is created automatically, and it's granted to the user that creates the DB cluster (the master user account, `postgres` by default). To learn more, see Understanding PostgreSQL roles and permissions.

All available Aurora PostgreSQL versions, including versions 10, 11, 12, 13, 14, and higher releases support the Salted Challenge Response Authentication Mechanism (SCRAM) for passwords as an alternative to message digest (MD5). We recommend that you use SCRAM because it's more secure than MD5. For more information, including how to migrate database user passwords from MD5 to SCRAM, see Using SCRAM for PostgreSQL password encryption.

Understanding PostgreSQL roles and permissions

When you create an Aurora PostgreSQL DB cluster using the AWS Management Console, an administrator account is created at the same time. By default, its name is `postgres`, as shown in the following screenshot:



▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

postgres

1 to 16 alphanumeric characters. First character must be a letter.

□ Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info

•••••••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm password Info

•••••••••••••

You can choose another name rather than accept the default (postgres). If you do, the name you choose must start with a letter and be between 1 and 16 alphanumeric characters. For simplicity's sake, we refer to this main user account by its default value (postgres) throughout this guide.

If you use the create-db-cluster AWS CLI rather than the AWS Management Console, you create the user name by passing it with the master-username parameter. For more information, see Step 2: Create an Aurora PostgreSQL DB cluster.

Whether you use the AWS Management Console, the AWS CLI, or the Amazon RDS API, and whether you use the default postgres name or choose a different name, this first database user account is a member of the rds_superuser group and has rds_superuser privileges.

Topics

- Understanding the rds_superuser role
- Controlling user access to the PostgreSQL database
- Delegating and controlling user password management
- Using SCRAM for PostgreSQL password encryption

Understanding the rds_superuser role

In PostgreSQL, a *role* can define a user, a group, or a set of specific permissions granted to a group or user for various objects in the database. PostgreSQL commands to CREATE USER and CREATE GROUP have been replaced by the more general, CREATE ROLE with specific properties to distinguish database users. A database user can be thought of as a role with the LOGIN privilege.

> **Note**
>
> The CREATE USER and CREATE GROUP commands can still be used. For more information, see Database Roles in the PostgreSQL documentation.

The postgres user is the most highly privileged database user on your Aurora PostgreSQL DB cluster. It has the characteristics defined by the following CREATE ROLE statement.

```
CREATE ROLE postgres WITH LOGIN NOSUPERUSER INHERIT CREATEDB CREATEROLE NOREPLICATION
 VALID UNTIL 'infinity'
```

The properties NOSUPERUSER, NOREPLICATION, INHERIT, and VALID UNTIL 'infinity' are the default options for CREATE ROLE, unless otherwise specified.

By default, postgres has privileges granted to the rds_superuser role, and permissions to create roles and databases. The rds_superuser role allows the postgres user to do the following:

- Add extensions that are available for use with Aurora PostgreSQL. For more information, see Working with extensions and foreign data wrappers.
- Create roles for users and grant privileges to users. For more information, see CREATE ROLE and GRANT in the PostgreSQL documentation.
- Create databases. For more information, see CREATE DATABASE in the PostgreSQL documentation.
- Grant rds_superuser privileges to user roles that don't have these privileges, and revoke privileges as needed. We recommend that you grant this role only to those users who perform superuser tasks. In other words, you can grant this role to database administrators (DBAs) or system administrators.
- Grant (and revoke) the rds_replication role to database users that don't have the rds_superuser role.
- Grant (and revoke) the rds_password role to database users that don't have the rds_superuser role.
- Obtain status information about all database connections by using the pg_stat_activity view. When needed, rds_superuser can stop any connections by using pg_terminate_backend or pg_cancel_backend.

In the CREATE ROLE postgres... statement, you can see that the postgres user role specifically disallows PostgreSQL superuser permissions. Aurora PostgreSQL is a managed service, so you can't access the host OS, and you can't connect using the PostgreSQL superuser account. Many of the tasks that require superuser access on a stand-alone PostgreSQL are managed automatically by Aurora.

For more information about granting privileges, see GRANT in the PostgreSQL documentation.

The rds_superuser role is one of several *predefined* roles in an Aurora PostgreSQL DB cluster.

> **Note**
>
> In PostgreSQL 13 and earlier releases, *predefined* roles are known as *default* roles.

In the following list, you find some of the other predefined roles that are created automatically for a new Aurora PostgreSQL DB cluster. Predefined roles and their privileges can't be changed. You can't drop, rename, or modify privileges for these predefined roles. Attempting to do so results in an error.

- **rds_password –** A role that can change passwords and set up password constraints for database users. The rds_superuser role is granted with this role by default, and can grant the role to database users. For more information, see Controlling user access to the PostgreSQL database.
  - For RDS for PostgreSQL versions older than 14, rds_password role can change passwords and set up password constraints for database users and users with rds_superuser role. From RDS for PostgreSQL version 14 and later, rds_password role can change passwords and set up password constraints only for database users. Only users with rds_superuser role can perform these actions on other users with rds_superuser role.
- **rdsadmin –** A role that's created to handle many of the management tasks that the administrator with superuser privileges would perform on a standalone PostgreSQL database. This role is used internally by Aurora PostgreSQL for many management tasks.

To see all predefined roles, you can connect to the primary instance of your Aurora PostgreSQL DB cluster and use the psql \du metacommand. The output looks as follows:

```
List of roles
  Role name    |       Attributes                    |       Member of
```

```
--------------+------------------------------------+------------------------------------
postgres      | Create role, Create DB            +| {rds_superuser}
              | Password valid until infinity      |
rds_superuser | Cannot login                       | {pg_monitor,pg_signal_backend,
              |                                   +|   rds_replication,rds_password}
...
```

In the output, you can see that `rds_superuser` isn't a database user role (it can't login), but it has the privileges of many other roles. You can also see that database user `postgres` is a member of the `rds_superuser` role. As mentioned previously, `postgres` is the default value in the Amazon RDS console's **Create database** page. If you chose another name, that name is shown in the list of roles instead.

> **Note**
>
> Aurora PostgreSQL versions 15.2 and 14.7 introduced restrictive behavior of the `rds_superuser` role. An Aurora PostgreSQL user needs to be granted the `CONNECT` privilege on the corresponding database to connect even if the user is granted the `rds_superuser` role. Prior to Aurora PostgreSQL versions 14.7 and 15.2, a user was able to connect to any database and system table if the user was granted the `rds_superuser` role. This restrictive behavior aligns with the AWS and Amazon Aurora commitments to the continuous improvement of security.
> Please update the respective logic in your applications if the above enhancement has an impact.

Controlling user access to the PostgreSQL database

New databases in PostgreSQL are always created with a default set of privileges in the database's `public` schema that allow all database users and roles to create objects. These privileges allow database users to connect to the database, for example, and create temporary tables while connected.

To better control user access to the databases instances that you create on your Aurora PostgreSQL DB cluster primary node , we recommend that you revoke these default `public` privileges. After doing so, you then grant specific privileges for database users on a more granular basis, as shown in the following procedure.

To set up roles and privileges for a new database instance

Suppose you're setting up a database on a newly created Aurora PostgreSQL DB cluster for use by several researchers, all of whom need read-write access to the database.

1. Use `psql` (or pgAdmin) to connect to the primary DB instance on your Aurora PostgreSQL DB cluster:

   ```
   psql --host=your-cluster-instance-1.666666666666.aws-region.rds.amazonaws.com --
   port=5432 --username=postgres --password
   ```

   When prompted, enter your password. The `psql` client connects and displays the default administrative connection database, `postgres=>`, as the prompt.
2. To prevent database users from creating objects in the `public` schema, do the following:

   ```
   postgres=> REVOKE CREATE ON SCHEMA public FROM PUBLIC;
   REVOKE
   ```
3. Next, you create a new database instance:

   ```
   postgres=> CREATE DATABASE lab_db;
   CREATE DATABASE
   ```
4. Revoke all privileges from the `PUBLIC` schema on this new database.

   ```
   postgres=> REVOKE ALL ON DATABASE lab_db FROM public;
   REVOKE
   ```
5. Create a role for database users.

   ```
   postgres=> CREATE ROLE lab_tech;
   CREATE ROLE
   ```
6. Give database users that have this role the ability to connect to the database.

   ```
   postgres=> GRANT CONNECT ON DATABASE lab_db TO lab_tech;
   GRANT
   ```
7. Grant all users with the `lab_tech` role all privileges on this database.

   ```
   postgres=> GRANT ALL PRIVILEGES ON DATABASE lab_db TO lab_tech;
   ```

```
GRANT
```

8. Create database users, as follows:

```
postgres=> CREATE ROLE lab_user1 LOGIN PASSWORD 'change_me';
CREATE ROLE
postgres=> CREATE ROLE lab_user2 LOGIN PASSWORD 'change_me';
CREATE ROLE
```

9. Grant these two users the privileges associated with the lab_tech role:

```
postgres=> GRANT lab_tech TO lab_user1;
GRANT ROLE
postgres=> GRANT lab_tech TO lab_user2;
GRANT ROLE
```

At this point, `lab_user1` and `lab_user2` can connect to the `lab_db` database. This example doesn't follow best practices for enterprise usage, which might include creating multiple database instances, different schemas, and granting limited permissions. For more complete information and additional scenarios, see Managing PostgreSQL Users and Roles.

For more information about privileges in PostgreSQL databases, see the GRANT command in the PostgreSQL documentation.

Delegating and controlling user password management

As a DBA, you might want to delegate the management of user passwords. Or, you might want to prevent database users from changing their passwords or reconfiguring password constraints, such as password lifetime. To ensure that only the database users that you choose can change password settings, you can turn on the restricted password management feature. When you activate this feature, only those database users that have been granted the `rds_password` role can manage passwords.

> **Note**
>
> To use restricted password management, your Aurora PostgreSQL DB cluster must be running Amazon Aurora PostgreSQL 10.6 or higher.

By default, this feature is `off`, as shown in the following:

```
postgres=> SHOW rds.restrict_password_commands;
  rds.restrict_password_commands
--------------------------------
 off
(1 row)
```

To turn on this feature, you use a custom parameter group and change the setting for `rds.restrict_password_commands` to 1. Be sure to reboot your Aurora PostgreSQL's primary DB instance so that the setting takes effect.

With this feature active, `rds_password` privileges are needed for the following SQL commands:

```
CREATE ROLE myrole WITH PASSWORD 'mypassword';
CREATE ROLE myrole WITH PASSWORD 'mypassword' VALID UNTIL '2023-01-01';
ALTER ROLE myrole WITH PASSWORD 'mypassword' VALID UNTIL '2023-01-01';
ALTER ROLE myrole WITH PASSWORD 'mypassword';
ALTER ROLE myrole VALID UNTIL '2023-01-01';
ALTER ROLE myrole RENAME TO myrole2;
```

Renaming a role (`ALTER ROLE myrole RENAME TO newname`) is also restricted if the password uses the MD5 hashing algorithm.

With this feature active, attempting any of these SQL commands without the `rds_password` role permissions generates the following error:

```
ERROR: must be a member of rds_password to alter passwords
```

We recommend that you grant the `rds_password` to only a few roles that you use solely for password management. If you grant `rds_password` privileges to database users that don't have `rds_superuser` privileges, you need to also grant them the `CREATEROLE` attribute.

Make sure that you verify password requirements such as expiration and needed complexity on the client side. If you use your own client-side utility for password related changes, the utility needs to be a member of `rds_password` and have `CREATE ROLE` privileges.

Using SCRAM for PostgreSQL password encryption

The *Salted Challenge Response Authentication Mechanism (SCRAM)* is an alternative to PostgreSQL's default message digest (MD5) algorithm for encrypting passwords. The SCRAM authentication mechanism is considered more secure than MD5. To learn more about these two different approaches to securing passwords, see Password Authentication in the PostgreSQL documentation.

We recommend that you use SCRAM rather than MD5 as the password encryption scheme for your Aurora PostgreSQL DB cluster. As of the Aurora PostgreSQL 14 release, SCRAM is supported in all available Aurora PostgreSQL versions, including versions 10, 11, 12, 13, and 14. It's a cryptographic challenge-response mechanism that uses the scram-sha-256 algorithm for password authentication and encryption.

You might need to update libraries for your client applications to support SCRAM. For example, JDBC versions before 42.2.0 don't support SCRAM. For more information, see PostgreSQL JDBC Driver in the PostgreSQL JDBC Driver documentation. For a list of other PostgreSQL drivers and SCRAM support, see List of drivers in the PostgreSQL documentation.

> **Note**
>
> Aurora PostgreSQL version 14 and higher support scram-sha-256 for password encryption by default for new DB clusters. That is, the default DB cluster parameter group (`default.aurora-postgresql14`) has its `password_encryption` value set to scram-sha-256.

Setting up Aurora PostgreSQL DB cluster to require SCRAM

For Aurora PostgreSQL 14.3 and higher versions, you can require the Aurora PostgreSQL DB cluster to accept only passwords that use the scram-sha-256 algorithm.

> **Important**
>
> For existing RDS Proxies with PostgreSQL databases, if you modify the database authentication to use `SCRAM` only, the proxy becomes unavailable for up to 60 seconds. To avoid the issue, do one of the following:
>
> - Ensure that the database allows both `SCRAM` and `MD5` authentication.
> - To use only `SCRAM` authentication, create a new proxy, migrate your application traffic to the new proxy, then delete the proxy previously associated with the database.

Before making changes to your system, be sure you understand the complete process, as follows:

- Get information about all roles and password encryption for all database users.
- Double-check the parameter settings for your Aurora PostgreSQL DB cluster for the parameters that control password encryption.
- If your Aurora PostgreSQL DB cluster uses a default parameter group, you need to create a custom DB cluster parameter group and apply it to your Aurora PostgreSQL DB cluster so that you can modify parameters when needed. If your Aurora PostgreSQL DB cluster uses a custom parameter group, you can modify the necessary parameters later in the process, as needed.
- Change the `password_encryption` parameter to `scram-sha-256`.
- Notify all database users that they need to update their passwords. Do the same for your `postgres` account. The new passwords are encrypted and stored using the scram-sha-256 algorithm.
- Verify that all passwords are encrypted using as the type of encryption.
- If all passwords use scram-sha-256, you can change the `rds.accepted_password_auth_method` parameter from `md5+scram` to `scram-sha-256`.

> ⚠ **Warning**
>
> After you change `rds.accepted_password_auth_method` to scram-sha-256 alone, any users (roles) with `md5`–encrypted passwords can't connect.

Getting ready to require SCRAM for your Aurora PostgreSQL DB cluster

Before making any changes to your Aurora PostgreSQL DB cluster, check all existing database user accounts. Also, check the type of encryption used for passwords. You can do these tasks by using the `rds_tools` extension. This extension is supported on Aurora PostgreSQL 13.1 and higher releases.

To get a list of database users (roles) and password encryption methods

1. Use `psql` to connect to the primary instance of your Aurora PostgreSQL DB cluster , as shown in the following.

   ```
   psql --host=cluster-name-instance-1.111122223333.aws-region.rds.amazonaws.com --
   port=5432 --username=postgres --password
   ```

2. Install the `rds_tools` extension.

```
postgres=> CREATE EXTENSION rds_tools;
CREATE EXTENSION
```

3. Get a listing of roles and encryption.

```
postgres=> SELECT * FROM
      rds_tools.role_password_encryption_type();
```

You see output similar to the following.

```
       rolname        | encryption_type
----------------------+-----------------
 pg_monitor           |
 pg_read_all_settings |
 pg_read_all_stats    |
 pg_stat_scan_tables  |
 pg_signal_backend    |
 lab_tester           | md5
 user_465             | md5
 postgres             | md5
(8 rows)
```

Creating a custom DB cluster parameter group

> **Note**
>
> If your Aurora PostgreSQL DB cluster already uses a custom parameter group, you don't need to create a new one.

For an overview of parameter groups for Aurora, see Creating a DB cluster parameter group.

The password encryption type used for passwords is set in one parameter, `password_encryption`. The encryption that the Aurora PostgreSQL DB cluster allows is set in another parameter, `rds.accepted_password_auth_method`. Changing either of these from the default values requires that you create a custom DB cluster parameter group and apply it to your cluster.

You can also use the AWS Management Console or the RDS API to create a custom DB cluster parameter group . For more information, see Creating a DB cluster parameter group.

You can now associate the custom parameter group with your DB instance.

To create a custom DB cluster parameter group

1. Use the `create-db-cluster-parameter-group` CLI command to create the custom parameter group for the cluster. The following example uses `aurora-postgresql13` as the source for this custom parameter group.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster-parameter-group --db-cluster-parameter-group-name 'docs-
   lab-scram-passwords' \
     --db-parameter-group-family aurora-postgresql13  --description 'Custom DB cluster
    parameter group for SCRAM'
   ```

   For Windows:

   ```
   aws rds create-db-cluster-parameter-group --db-cluster-parameter-group-name "docs-
   lab-scram-passwords" ^
     --db-parameter-group-family aurora-postgresql13  --description "Custom DB cluster
    parameter group for SCRAM"
   ```

   You can now associate the custom parameter group with your cluster.

2. Use the `modify-db-cluster` CLI command to apply this custom parameter group to your Aurora PostgreSQL DB cluster.

   For Linux, macOS, or Unix:

   ```
   aws rds modify-db-cluster --db-cluster-identifier 'your-instance-name' \
           --db-cluster-parameter-group-name "docs-lab-scram-passwords
   ```

   For Windows:

   ```
   aws rds modify-db-cluster --db-cluster-identifier "your-instance-name" ^
           --db-cluster-parameter-group-name "docs-lab-scram-passwords
   ```

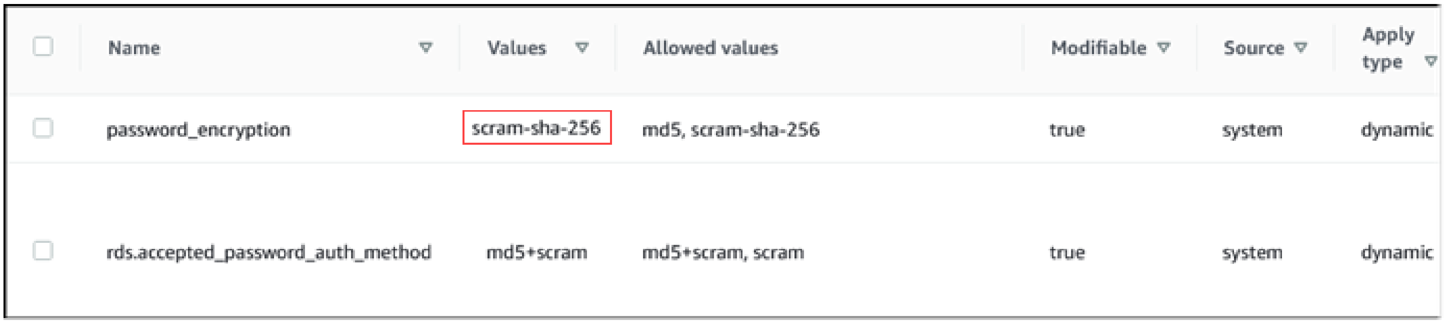
To resynchronize your Aurora PostgreSQL DB cluster with your custom DB cluster parameter group, reboot the primary and all other instances of the cluster.

Configuring password encryption to use SCRAM

The password encryption mechanism used by an Aurora PostgreSQL DB cluster is set in the DB cluster parameter group in the `password_encryption` parameter. Allowed values are unset, `md5`, or `scram-sha-256`. The default value depends on the Aurora PostgreSQL version, as follows:

- Aurora PostgreSQL 14 – Default is `scram-sha-256`
- Aurora PostgreSQL 13 – Default is `md5`

With a custom DB cluster parameter group attached to your Aurora PostgreSQL DB cluster, you can modify values for the password encryption parameter.



☐	Name	Values	Allowed values	Modifiable	Source	Apply type
☐	password_encryption	scram-sha-256	md5, scram-sha-256	true	system	dynamic
☐	rds.accepted_password_auth_method	md5+scram	md5+scram, scram	true	system	dynamic

To change password encryption setting to scram-sha-256

- Change the value of the password encryption to scram-sha-256, as shown following. The change can be applied immediately because the parameter is dynamic, so a restart isn't required for the change to take effect.

  For Linux, macOS, or Unix:

  ```
  aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name \
    'docs-lab-scram-passwords' --parameters
   'ParameterName=password_encryption,ParameterValue=scram-
  sha-256,ApplyMethod=immediate'
  ```

  For Windows:

```
aws rds modify-db-parameter-group --db-parameter-group-name ^
  "docs-lab-scram-passwords" --parameters
 "ParameterName=password_encryption,ParameterValue=scram-
sha-256,ApplyMethod=immediate"
```

Migrating passwords for user roles to SCRAM

You can migrate passwords for user roles to SCRAM as described following.

To migrate database user (role) passwords from MD5 to SCRAM

1. Log in as the administrator user (default user name, `postgres`) as shown following.

   ```
   psql --host=cluster-name-instance-1.111122223333.aws-region.rds.amazonaws.com --
   port=5432 --username=postgres --password
   ```

2. Check the setting of the `password_encryption` parameter on your RDS for PostgreSQL DB instance by using the following command.

   ```
   postgres=> SHOW password_encryption;
    password_encryption
   ---------------------
    md5
    (1 row)
   ```

3. Change the value of this parameter to scram-sha-256. This is a dynamic parameter, so you don't need to reboot the instance after making this change. Check the value again to make sure that it's now set to `scram-sha-256`, as follows.

   ```
   postgres=> SHOW password_encryption;
    password_encryption
   ---------------------
    scram-sha-256
    (1 row)
   ```

4. Notify all database users to change their passwords. Be sure to also change your own password for account `postgres` (the database user with `rds_superuser` privileges).

   ```
   labdb=> ALTER ROLE postgres WITH LOGIN PASSWORD 'change_me';
   ```

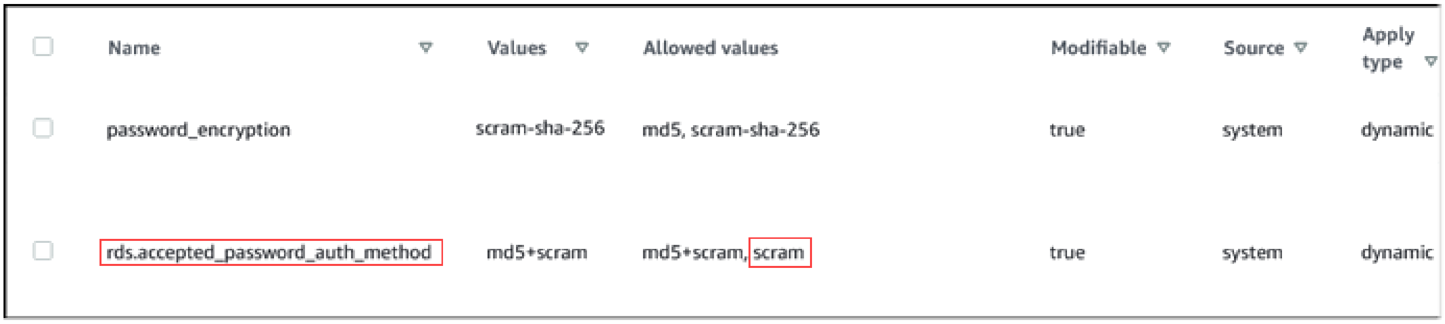
```
ALTER ROLE
```

5. Repeat the process for all databases on your Aurora PostgreSQL DB cluster.

Changing parameter to require SCRAM

This is the final step in the process. After you make the change in the following procedure, any user accounts (roles) that still use md5 encryption for passwords can't log in to the Aurora PostgreSQL DB cluster.

The rds.accepted_password_auth_method specifies the encryption method that the Aurora PostgreSQL DB cluster accepts for a user password during the login process. The default value is md5+scram, meaning that either method is accepted. In the following image, you can find the default setting for this parameter.



☐	Name	Values	Allowed values	Modifiable	Source	Apply type
☐	password_encryption	scram-sha-256	md5, scram-sha-256	true	system	dynamic
☐	rds.accepted_password_auth_method	md5+scram	md5+scram, scram	true	system	dynamic

The allowed values for this parameter are md5+scram or scram alone. Changing this parameter value to scram makes this a requirement.

To change the parameter value to require SCRAM authentication for passwords

1. Verify that all database user passwords for all databases on your Aurora PostgreSQL DB cluster use scram-sha-256 for password encryption. To do so, query rds_tools for the role (user) and encryption type, as follows.

```
postgres=> SELECT * FROM rds_tools.role_password_encryption_type();
  rolname              | encryption_type
  ----------------------+-----------------
  pg_monitor            |
  pg_read_all_settings  |
  pg_read_all_stats     |
  pg_stat_scan_tables   |
  pg_signal_backend     |
  lab_tester            | scram-sha-256
  user_465              | scram-sha-256
  postgres              | scram-sha-256
  ( rows)
```

2. Repeat the query across all DB instances in your Aurora PostgreSQL DB cluster.

   If all passwords use scram-sha-256, you can proceed.

3. Change the value of the accepted password authentication to scram-sha-256, as follows.

   For Linux, macOS, or Unix:

   ```
   aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name 'docs-
   lab-scram-passwords' \
     --parameters
    'ParameterName=rds.accepted_password_auth_method,ParameterValue=scram,ApplyMethod=immediat
   ```

   For Windows:

   ```
   aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name "docs-
   lab-scram-passwords" ^
     --parameters
    "ParameterName=rds.accepted_password_auth_method,ParameterValue=scram,ApplyMethod=immediat
   ```

Securing Aurora PostgreSQL data with SSL/TLS

Amazon RDS supports Secure Socket Layer (SSL) and Transport Layer Security (TLS) encryption for Aurora PostgreSQL DB clusters. Using SSL/TLS, you can encrypt a connection between your applications and your Aurora PostgreSQL DB clusters. You can also force all connections to your Aurora PostgreSQL DB cluster to use SSL/TLS. Amazon Aurora PostgreSQL supports Transport Layer Security (TLS) versions 1.1 and 1.2. We recommend using TLS 1.2 for encrypted connections. We have added support for TLSv1.3 from the following versions of Aurora PostgreSQL:

- 15.3 and all higher versions
- 14.8 and higher 14 versions
- 13.11 and higher 13 versions
- 12.15 and higher 12 versions
- 11.20 and higher 11 versions

For general information about SSL/TLS support and PostgreSQL databases, see SSL support in the PostgreSQL documentation. For information about using an SSL/TLS connection over JDBC, see Configuring the client in the PostgreSQL documentation.

Topics

- Requiring an SSL/TLS connection to an Aurora PostgreSQL DB cluster
- Determining the SSL/TLS connection status
- Configuring cipher suites for connections to Aurora PostgreSQL DB clusters

SSL/TLS support is available in all AWS Regions for Aurora PostgreSQL. Amazon RDS creates an SSL/TLS certificate for your Aurora PostgreSQL DB cluster when the DB cluster is created. If you enable SSL/TLS certificate verification, then the SSL/TLS certificate includes the DB cluster endpoint as the Common Name (CN) for the SSL/TLS certificate to guard against spoofing attacks.

To connect to an Aurora PostgreSQL DB cluster over SSL/TLS

1. Download the certificate.

   For information about downloading certificates, see Using SSL/TLS to encrypt a connection to a DB cluster.
2. Import the certificate into your operating system.
3. Connect to your Aurora PostgreSQL DB cluster over SSL/TLS.

   When you connect using SSL/TLS, your client can choose to verify the certificate chain or not. If your connection parameters specify `sslmode=verify-ca` or `sslmode=verify-full`, then your client requires the RDS CA certificates to be in their trust store or referenced in the connection URL. This requirement is to verify the certificate chain that signs your database certificate.

   When a client, such as psql or JDBC, is configured with SSL/TLS support, the client first tries to connect to the database with SSL/TLS by default. If the client can't connect with SSL/TLS, it reverts to connecting without SSL/TLS. By default, the `sslmode` option for JDBC and libpq-based clients is set to `prefer`.

   Use the `sslrootcert` parameter to reference the certificate, for example `sslrootcert=rds-ssl-ca-cert.pem`.

The following is an example of using psql to connect to an Aurora PostgreSQL DB cluster.

```
$ psql -h testpg.cdhmuqifdpib.us-east-1.rds.amazonaws.com -p 5432 \
    "dbname=testpg user=testuser sslrootcert=rds-ca-2015-root.pem sslmode=verify-full"
```

Requiring an SSL/TLS connection to an Aurora PostgreSQL DB cluster

You can require that connections to your Aurora PostgreSQL DB cluster use SSL/TLS by using the `rds.force_ssl` parameter. By default, the `rds.force_ssl` parameter is set to 0 (off). You can set the `rds.force_ssl` parameter to 1 (on) to require SSL/TLS for connections to your DB cluster. Updating the `rds.force_ssl` parameter also sets the PostgreSQL `ssl` parameter to 1 (on) and modifies your DB cluster's `pg_hba.conf` file to support the new SSL/TLS configuration.

You can set the `rds.force_ssl` parameter value by updating the DB cluster parameter group for your DB cluster. If the DB cluster parameter group isn't the default one, and the `ssl` parameter is already set to 1 when you set `rds.force_ssl` to 1, you don't need to reboot your DB cluster. Otherwise, you must reboot your DB cluster for the change to take effect. For more information on parameter groups, see Working with parameter groups.

When the `rds.force_ssl` parameter is set to 1 for a DB cluster, you see output similar to the following when you connect, indicating that SSL/TLS is now required:

```

$ psql postgres -h SOMEHOST.amazonaws.com -p 8192 -U someuser
psql (9.3.12, server 9.4.4)
WARNING: psql major version 9.3, server major version 9.4.
Some psql features might not work.
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

postgres=>

```

Determining the SSL/TLS connection status

The encrypted status of your connection is shown in the logon banner when you connect to the DB cluster.

```
Password for user master:
psql (9.3.12)
```

```
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

postgres=>
```

You can also load the `sslinfo` extension and then call the `ssl_is_used()` function to determine if SSL/TLS is being used. The function returns `t` if the connection is using SSL/TLS, otherwise it returns `f`.

```
postgres=> create extension sslinfo;
CREATE EXTENSION

postgres=> select ssl_is_used();
 ssl_is_used
---------
t
(1 row)
```

You can use the `select ssl_cipher()` command to determine the SSL/TLS cipher:

```
postgres=> select ssl_cipher();
ssl_cipher
--------------------
DHE-RSA-AES256-SHA
(1 row)
```

If you enable `set rds.force_ssl` and restart your DB cluster, non-SSL connections are refused with the following message:

```
$ export PGSSLMODE=disable
$ psql postgres -h SOMEHOST.amazonaws.com -p 8192 -U someuser
psql: FATAL: no pg_hba.conf entry for host "host.ip", user "someuser", database
 "postgres", SSL off
$
```

For information about the `sslmode` option, see [Database connection control functions]() in the PostgreSQL documentation.

Configuring cipher suites for connections to Aurora PostgreSQL DB clusters

By using configurable cipher suites, you can have more control over the security of your database connections. You can specify a list of cipher suites that you want to allow to secure client SSL/TLS connections to your database. With configurable cipher suites, you can control the connection encryption that your database server accepts. Doing this helps prevent the use of insecure or deprecated ciphers.

Configurable cipher suites is supported in Aurora PostgreSQL versions 11.8 and higher.

To specify the list of permissible ciphers for encrypting connections, modify the `ssl_ciphers` cluster parameter. Set the `ssl_ciphers` parameter to a string of comma-separated cipher values in a cluster parameter group using the AWS Management Console, the AWS CLI, or the RDS API. To set cluster parameters, see Modifying parameters in a DB cluster parameter group.

The following table shows the supported ciphers for the valid Aurora PostgreSQL engine versions.

Aurora PostgreSQL engine versions	Supported ciphers
9.6, 10.20 and lower, 11.15 and lower, 12.10 and lower, 13.6 and lower	• DHE-RSA-AES128-SHA • DHE-RSA-AES128-SHA256 • DHE-RSA-AES128-GCM-SHA256 • DHE-RSA-AES256-SHA • DHE-RSA-AES256-SHA256 • DHE-RSA-AES256-GCM-SHA384 • ECDHE-ECDSA-AES256-SHA • ECDHE-ECDSA-AES256-GCM-SHA384 • ECDHE-RSA-AES256-SHA384 • ECDHE-RSA-AES128-SHA • ECDHE-RSA-AES128-SHA256 • ECDHE-RSA-AES128-GCM-SHA256 • ECDHE-RSA-AES256-SHA

Aurora PostgreSQL engine versions	Supported ciphers
	• ECDHE-RSA-AES256-GCM-SHA384

Aurora PostgreSQL engine versions	Supported ciphers
10.21, 11.16, 12.11, 13.7, 14.3 and 14.4	• DHE-RSA-AES128-SHA • DHE-RSA-AES128-SHA256 • DHE-RSA-AES128-GCM-SHA256 • DHE-RSA-AES256-SHA • DHE-RSA-AES256-SHA256 • DHE-RSA-AES256-GCM-SHA384 • ECDHE-ECDSA-AES256-SHA • ECDHE-ECDSA-AES256-GCM-SHA384 • ECDHE-RSA-AES256-SHA384 • ECDHE-RSA-AES128-SHA • ECDHE-RSA-AES128-GCM-SHA256 • ECDHE-RSA-AES256-SHA • ECDHE-RSA-AES256-GCM-SHA384 • TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA • TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 • TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA • TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 • TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA • TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 • TLS_RSA_WITH_AES_256_GCM_SHA384

Aurora PostgreSQL engine versions	Supported ciphers
	• TLS_RSA_WITH_AES_256_CBC_SHA • TLS_RSA_WITH_AES_128_GCM_SHA256 • TLS_RSA_WITH_AES_128_CBC_SHA • TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256

Aurora PostgreSQL engine versions	Supported ciphers
10.22 and higher, 11.17 and higher, 12.12 and higher, 13.8 and higher, 14.5 and higher, and 15.2 and higher	• DHE-RSA-AES128-SHA • DHE-RSA-AES128-SHA256 • DHE-RSA-AES128-GCM-SHA256 • DHE-RSA-AES256-SHA • DHE-RSA-AES256-SHA256 • DHE-RSA-AES256-GCM-SHA384 • ECDHE-ECDSA-AES256-SHA • ECDHE-ECDSA-AES256-GCM-SHA384 • ECDHE-RSA-AES256-SHA384 • ECDHE-RSA-AES128-SHA • ECDHE-RSA-AES128-SHA256 • ECDHE-RSA-AES128-GCM-SHA256 • ECDHE-RSA-AES256-SHA • ECDHE-RSA-AES256-GCM-SHA384 • TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA • TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 • TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA • TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 • TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 • TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

Aurora PostgreSQL engine versions	Supported ciphers
	• TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 • TLS_RSA_WITH_AES_256_GCM_SHA384 • TLS_RSA_WITH_AES_256_CBC_SHA • TLS_RSA_WITH_AES_128_GCM_SHA256 • TLS_RSA_WITH_AES_128_CBC_SHA256 • TLS_RSA_WITH_AES_128_CBC_SHA • TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256

Aurora PostgreSQL engine versions	Supported ciphers
15.3, 14.8, 13.11, 12.15, and 11.20	• DHE-RSA-AES128-SHA • DHE-RSA-AES128-SHA256 • DHE-RSA-AES128-GCM-SHA256 • DHE-RSA-AES256-SHA • DHE-RSA-AES256-SHA256 • DHE-RSA-AES256-GCM-SHA384 • ECDHE-ECDSA-AES256-SHA • ECDHE-ECDSA-AES256-GCM-SHA384 • ECDHE-RSA-AES256-SHA384 • ECDHE-RSA-AES128-SHA • ECDHE-RSA-AES128-SHA256 • ECDHE-RSA-AES128-GCM-SHA256 • ECDHE-RSA-AES256-SHA • ECDHE-RSA-AES256-GCM-SHA384 • TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA • TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 • TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA • TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 • TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 • TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

Aurora PostgreSQL engine versions	Supported ciphers
	- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 - TLS_RSA_WITH_AES_256_GCM_SHA384 - TLS_RSA_WITH_AES_256_CBC_SHA - TLS_RSA_WITH_AES_128_GCM_SHA256 - TLS_RSA_WITH_AES_128_CBC_SHA256 - TLS_RSA_WITH_AES_128_CBC_SHA - TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 - TLS_AES_128_GCM_SHA256 - TLS_AES_256_GCM_SHA384

You can also use the describe-engine-default-cluster-parameters CLI command to determine which cipher suites are currently supported for a specific parameter group family. The following example shows how to get the allowed values for the `ssl_cipher` cluster parameter for Aurora PostgreSQL 11.

```
aws rds describe-engine-default-cluster-parameters --db-parameter-group-family aurora-
postgresql11

    ...some output truncated...
 {
  "ParameterName": "ssl_ciphers",
  "Description": "Sets the list of allowed TLS ciphers to be used on secure
 connections.",
  "Source": "engine-default",
  "ApplyType": "dynamic",
  "DataType": "list",
  "AllowedValues": "DHE-RSA-AES128-SHA,DHE-RSA-AES128-SHA256,DHE-RSA-AES128-GCM-
SHA256,DHE-RSA-AES256-SHA,DHE-RSA-AES256-SHA256,DHE-RSA-AES256-GCM-SHA384,
  ECDHE-RSA-AES128-SHA,ECDHE-RSA-AES128-SHA256,ECDHE-RSA-AES128-GCM-
SHA256,ECDHE-RSA-AES256-SHA,ECDHE-RSA-AES256-SHA384,ECDHE-RSA-AES256-GCM-
SHA384,TLS_RSA_WITH_AES_256_GCM_SHA384,

 TLS_RSA_WITH_AES_256_CBC_SHA,TLS_RSA_WITH_AES_128_GCM_SHA256,TLS_RSA_WITH_AES_128_CBC_SHA256,T

 TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_RSA_WITH_AE
  "IsModifiable": true,
  "MinimumEngineVersion": "11.8",
  "SupportedEngineModes": [
   "provisioned"
  ]
 },
    ...some output truncated...
```

The `ssl_ciphers` parameter defaults to all allowed cipher suites. For more information about ciphers, see the ssl_ciphers variable in the PostgreSQL documentation.

Updating applications to connect to Aurora PostgreSQL DB clusters using new SSL/TLS certificates

As of January 13, 2023, Amazon RDS has published new Certificate Authority (CA) certificates for connecting to your Aurora DB clusters using Secure Socket Layer or Transport Layer Security (SSL/TLS). Following, you can find information about updating your applications to use the new certificates.

This topic can help you to determine whether any client applications use SSL/TLS to connect to your DB clusters. If they do, you can further check whether those applications require certificate verification to connect.

Note

Some applications are configured to connect to Aurora PostgreSQL DB clusters only if they can successfully verify the certificate on the server.

For such applications, you must update your client application trust stores to include the new CA certificates.

After you update your CA certificates in the client application trust stores, you can rotate the certificates on your DB clusters. We strongly recommend testing these procedures in a development or staging environment before implementing them in your production environments.

For more information about certificate rotation, see Rotating your SSL/TLS certificate. For more information about downloading certificates, see Using SSL/TLS to encrypt a connection to a DB cluster. For information about using SSL/TLS with PostgreSQL DB clusters, see Securing Aurora PostgreSQL data with SSL/TLS.

Topics

Determining whether applications are connecting to Aurora PostgreSQL DB clusters using SSL

Check the DB cluster configuration for the value of the `rds.force_ssl` parameter. By default, the `rds.force_ssl` parameter is set to `0` (off). If the `rds.force_ssl` parameter is set to 1 (on), clients are required to use SSL/TLS for connections. For more information about parameter groups, see Working with parameter groups.

If `rds.force_ssl` isn't set to 1 (on), query the `pg_stat_ssl` view to check connections using SSL. For example, the following query returns only SSL connections and information about the clients using SSL.

```
select datname, usename, ssl, client_addr from pg_stat_ssl inner join pg_stat_activity
 on pg_stat_ssl.pid = pg_stat_activity.pid where ssl is true and usename<>'rdsadmin';
```

Only rows using SSL/TLS connections are displayed with information about the connection. The following is sample output.

```
 datname  | usename | ssl | client_addr
----------+---------+-----+-------------
 benchdb  | pgadmin | t   | 53.95.6.13
 postgres | pgadmin | t   | 53.95.6.13
```

```
(2 rows)
```

The preceding query displays only the current connections at the time of the query. The absence of results doesn't indicate that no applications are using SSL connections. Other SSL connections might be established at a different time.

Determining whether a client requires certificate verification in order to connect

When a client, such as psql or JDBC, is configured with SSL support, the client first tries to connect to the database with SSL by default. If the client can't connect with SSL, it reverts to connecting without SSL. The default `sslmode` mode used is different between libpq-based clients (such as psql) and JDBC. The libpq-based clients default to `prefer`, where JDBC clients default to `verify-full`. The certificate on the server is verified only when `sslrootcert` is provided with `sslmode` set to `verify-ca` or `verify-full`. An error is thrown if the certificate is invalid.

Use `PGSSLROOTCERT` to verify the certificate with the `PGSSLMODE` environment variable, with `PGSSLMODE` set to `verify-ca` or `verify-full`.

```
PGSSLMODE=verify-full PGSSLROOTCERT=/fullpath/ssl-cert.pem psql -h
 pgdbidentifier.cxxxxxxxx.us-east-2.rds.amazonaws.com -U primaryuser -d postgres
```

Use the `sslrootcert` argument to verify the certificate with `sslmode` in connection string format, with `sslmode` set to `verify-ca` or `verify-full`.

```
psql "host=pgdbidentifier.cxxxxxxxx.us-east-2.rds.amazonaws.com sslmode=verify-full
 sslrootcert=/full/path/ssl-cert.pem user=primaryuser dbname=postgres"
```

For example, in the preceding case, if you use an invalid root certificate, you see an error similar to the following on your client.

```
psql: SSL error: certificate verify failed
```

Updating your application trust store

For information about updating the trust store for PostgreSQL applications, see Secure TCP/IP connections with SSL in the PostgreSQL documentation.

Note

When you update the trust store, you can retain older certificates in addition to adding the new certificates.

Updating your application trust store for JDBC

You can update the trust store for applications that use JDBC for SSL/TLS connections.

For information about downloading the root certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

For sample scripts that import certificates, see Sample script for importing certificates into your trust store.

Using SSL/TLS connections for different types of applications

The following provides information about using SSL/TLS connections for different types of applications:

- **psql**

  The client is invoked from the command line by specifying options either as a connection string or as environment variables. For SSL/TLS connections, the relevant options are `sslmode` (environment variable `PGSSLMODE`), `sslrootcert` (environment variable `PGSSLROOTCERT`).

  For the complete list of options, see Parameter key words in the PostgreSQL documentation. For the complete list of environment variables, see Environment variables in the PostgreSQL documentation.
- **pgAdmin**

  This browser-based client is a more user-friendly interface for connecting to a PostgreSQL database.

  For information about configuring connections, see the pgAdmin documentation.
- **JDBC**

  JDBC enables database connections with Java applications.

For general information about connecting to a PostgreSQL database with JDBC, see Connecting to the database in the PostgreSQL documentation. For information about connecting with SSL/TLS, see Configuring the client in the PostgreSQL documentation.

- **Python**

  A popular Python library for connecting to PostgreSQL databases is `psycopg2`.

  For information about using `psycopg2`, see the psycopg2 documentation. For a short tutorial on how to connect to a PostgreSQL database, see Psycopg2 tutorial. You can find information about the options the connect command accepts in The psycopg2 module content.

> ⚠ **Important**
>
> After you have determined that your database connections use SSL/TLS and have updated your application trust store, you can update your database to use the rds-ca-rsa2048-g1 certificates. For instructions, see step 3 in Updating your CA certificate by modifying your DB instance .

Using Kerberos authentication with Aurora PostgreSQL

You can use Kerberos to authenticate users when they connect to your DB cluster running PostgreSQL. To do so, configure your DB cluster to use AWS Directory Service for Microsoft Active Directory for Kerberos authentication. AWS Directory Service for Microsoft Active Directory is also called AWS Managed Microsoft AD. It's a feature available with AWS Directory Service. To learn more, see What is AWS Directory Service? in the *AWS Directory Service Administration Guide*.

To start, create an AWS Managed Microsoft AD directory to store user credentials. Then, provide to your PostgreSQL DB cluster the Active Directory's domain and other information. When users authenticate with the PostgreSQL DB cluster, authentication requests are forwarded to the AWS Managed Microsoft AD directory.

Keeping all of your credentials in the same directory can save you time and effort. You have a centralized location for storing and managing credentials for multiple DB clusters. Using a directory can also improve your overall security profile.

In addition, you can access credentials from your own on-premises Microsoft Active Directory. To do so, create a trusting domain relationship so that the AWS Managed Microsoft AD directory trusts your on-premises Microsoft Active Directory. In this way, your users can access your PostgreSQL clusters with the same Windows single sign-on (SSO) experience as when they access workloads in your on-premises network.

A database can use Kerberos, AWS Identity and Access Management (IAM), or both Kerberos and IAM authentication. However, because Kerberos and IAM authentication provide different authentication methods, a specific database user can log in to a database using only one or the other authentication method but not both. For more information about IAM authentication, see IAM database authentication.

Topics

Region and version availability

Feature availability and support varies across specific versions of each database engine, and across AWS Regions. For more information on version and Region availability of Aurora PostgreSQL with Kerberos authentication, see Kerberos authentication with Aurora PostgreSQL.

Overview of Kerberos authentication for PostgreSQL DB clusters

To set up Kerberos authentication for a PostgreSQL DB cluster, take the following steps, described in more detail later:

1. Use AWS Managed Microsoft AD to create an AWS Managed Microsoft AD directory. You can use the AWS Management Console, the AWS CLI, or the AWS Directory Service API to create the directory. Make sure to open the relevant outbound ports on the directory security group so that the directory can communicate with the cluster.

2. Create a role that provides Amazon Aurora access to make calls to your AWS Managed Microsoft AD directory. To do so, create an AWS Identity and Access Management (IAM) role that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`.

   For the IAM role to allow access, the AWS Security Token Service (AWS STS) endpoint must be activated in the correct AWS Region for your AWS account. AWS STS endpoints are active by default in all AWS Regions, and you can use them without any further actions. For more information, see Activating and deactivating AWS STS in an AWS Region in the *IAM User Guide*.

3. Create and configure users in the AWS Managed Microsoft AD directory using the Microsoft Active Directory tools. For more information about creating users in your Active Directory, see Manage users and groups in AWS Managed Microsoft AD in the *AWS Directory Service Administration Guide*.

4. If you plan to locate the directory and the DB instance in different AWS accounts or virtual private clouds (VPCs), configure VPC peering. For more information, see What is VPC peering? in the *Amazon VPC Peering Guide*.

5. Create or modify a PostgreSQL DB cluster either from the console, CLI, or RDS API using one of the following methods:

   - Creating and connecting to an Aurora PostgreSQL DB cluster
   - Modifying an Amazon Aurora DB cluster
   - Restoring from a DB cluster snapshot
   - Restoring a DB cluster to a specified time

   You can locate the cluster in the same Amazon Virtual Private Cloud (VPC) as the directory or in a different AWS account or VPC. When you create or modify the PostgreSQL DB cluster, do the following:

   - Provide the domain identifier (`d-*` identifier) that was generated when you created your directory.
   - Provide the name of the IAM role that you created.
   - Ensure that the DB instance security group can receive inbound traffic from the directory security group.

6. Use the RDS master user credentials to connect to the PostgreSQL DB cluster. Create the user in PostgreSQL to be identified externally. Externally identified users can log in to the PostgreSQL DB cluster using Kerberos authentication.

Setting up Kerberos authentication for PostgreSQL DB clusters

You use AWS Directory Service for Microsoft Active Directory (AWS Managed Microsoft AD) to set up Kerberos authentication for a PostgreSQL DB cluster. To set up Kerberos authentication, take the following steps.

Topics

Step 1: Create a directory using AWS Managed Microsoft AD

AWS Directory Service creates a fully managed Active Directory in the AWS Cloud. When you create an AWS Managed Microsoft AD directory, AWS Directory Service creates two domain controllers and DNS servers for you. The directory servers are created in different subnets in a VPC. This redundancy helps make sure that your directory remains accessible even if a failure occurs.

When you create an AWS Managed Microsoft AD directory, AWS Directory Service performs the following tasks on your behalf:

- Sets up an Active Directory within your VPC.
- Creates a directory administrator account with the user name `Admin` and the specified password. You use this account to manage your directory.

> ⚠ **Important**
>
> Make sure to save this password. AWS Directory Service doesn't store this password, and it can't be retrieved or reset.

- Creates a security group for the directory controllers. The security group must permit communication with the PostgreSQL DB cluster.

When you launch AWS Directory Service for Microsoft Active Directory, AWS creates an Organizational Unit (OU) that contains all of your directory's objects. This OU, which has the NetBIOS name that you entered when you created your directory, is located in the domain root. The domain root is owned and managed by AWS.

The `Admin` account that was created with your AWS Managed Microsoft AD directory has permissions for the most common administrative activities for your OU:

- Create, update, or delete users
- Add resources to your domain such as file or print servers, and then assign permissions for those resources to users in your OU
- Create additional OUs and containers
- Delegate authority
- Restore deleted objects from the Active Directory Recycle Bin
- Run Active Directory and Domain Name Service (DNS) modules for Windows PowerShell on the Active Directory Web Service

The `Admin` account also has rights to perform the following domain-wide activities:

- Manage DNS configurations (add, remove, or update records, zones, and forwarders)
- View DNS event logs
- View security event logs

To create a directory with AWS Managed Microsoft AD

1. In the AWS Directory Service console navigation pane, choose **Directories**, and then choose **Set up directory**.
2. Choose **AWS Managed Microsoft AD**. AWS Managed Microsoft AD is the only option currently supported for use with Amazon Aurora.
3. Choose **Next**.
4. On the **Enter directory information** page, provide the following information:

Edition

Choose the edition that meets your requirements.

Directory DNS name

The fully qualified name for the directory, such as **`corp.example.com`**.

Directory NetBIOS name

An optional short name for the directory, such as `CORP`.

Directory description

An optional description for the directory.

Admin password

The password for the directory administrator. The directory creation process creates an administrator account with the user name `Admin` and this password.

The directory administrator password can't include the word "admin." The password is case-sensitive and must be 8–64 characters in length. It must also contain at least one character from three of the following four categories:

- Lowercase letters (a–z)
- Uppercase letters (A–Z)
- Numbers (0–9)
- Nonalphanumeric characters (~!@#$%^&*_-+=`|\(){}[]:;"'<>,.?/)

Confirm password

Retype the administrator password.

> ⚠ **Important**
>
> Make sure that you save this password. AWS Directory Service doesn't store this password, and it can't be retrieved or reset.

5. Choose **Next**.
6. On the **Choose VPC and subnets** page, provide the following information:

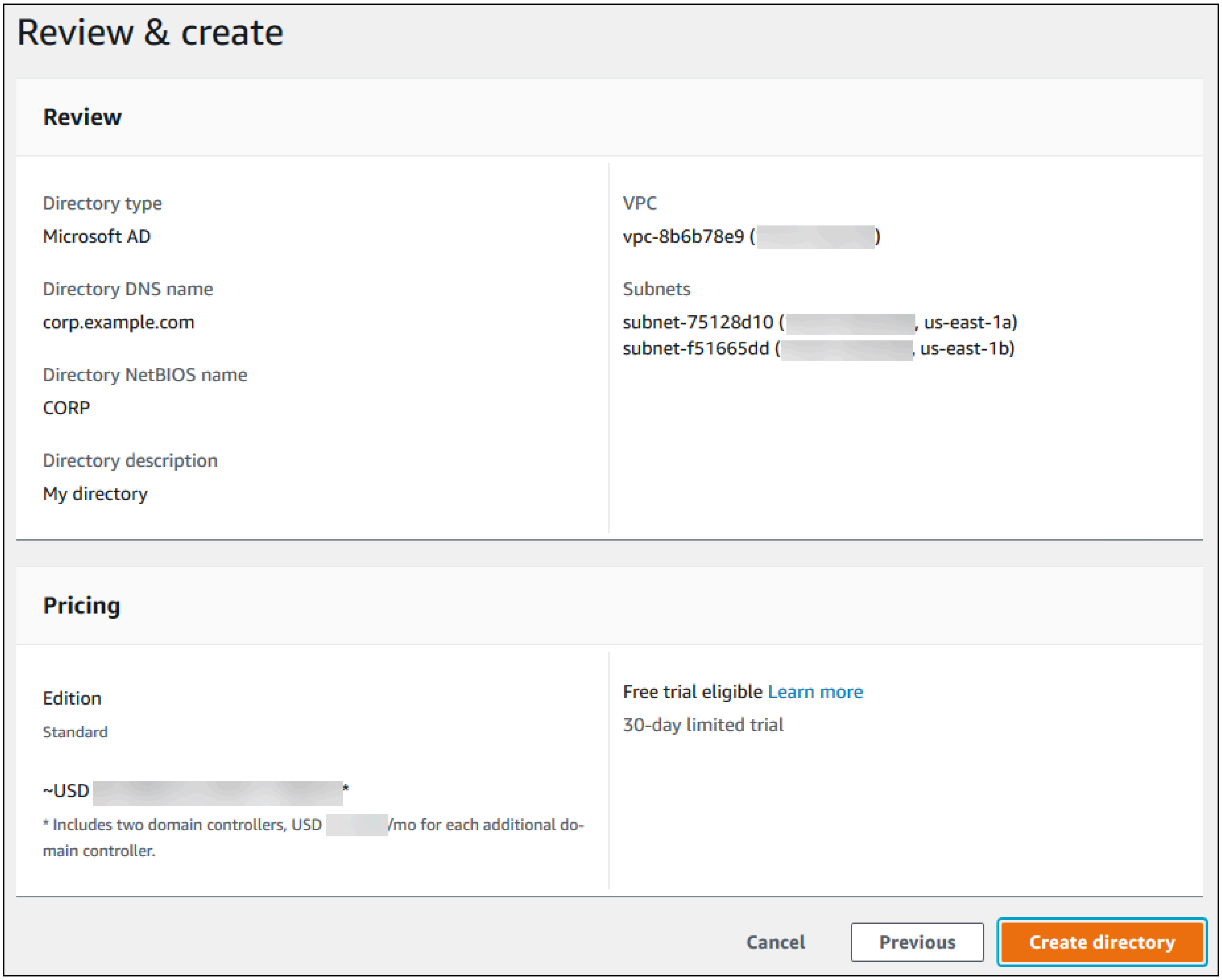
VPC

Choose the VPC for the directory. You can create the PostgreSQL DB cluster in this same VPC or in a different VPC.

Subnets

Choose the subnets for the directory servers. The two subnets must be in different Availability Zones.

7. Choose **Next**.
8. Review the directory information. If changes are needed, choose **Previous** and make the changes. When the information is correct, choose **Create directory**.



Review & create

Review

Directory type
Microsoft AD

Directory DNS name
corp.example.com

Directory NetBIOS name
CORP

Directory description
My directory

VPC
vpc-8b6b78e9 ([redacted])

Subnets
subnet-75128d10 ([redacted], us-east-1a)
subnet-f51665dd ([redacted], us-east-1b)

Pricing

Edition
Standard

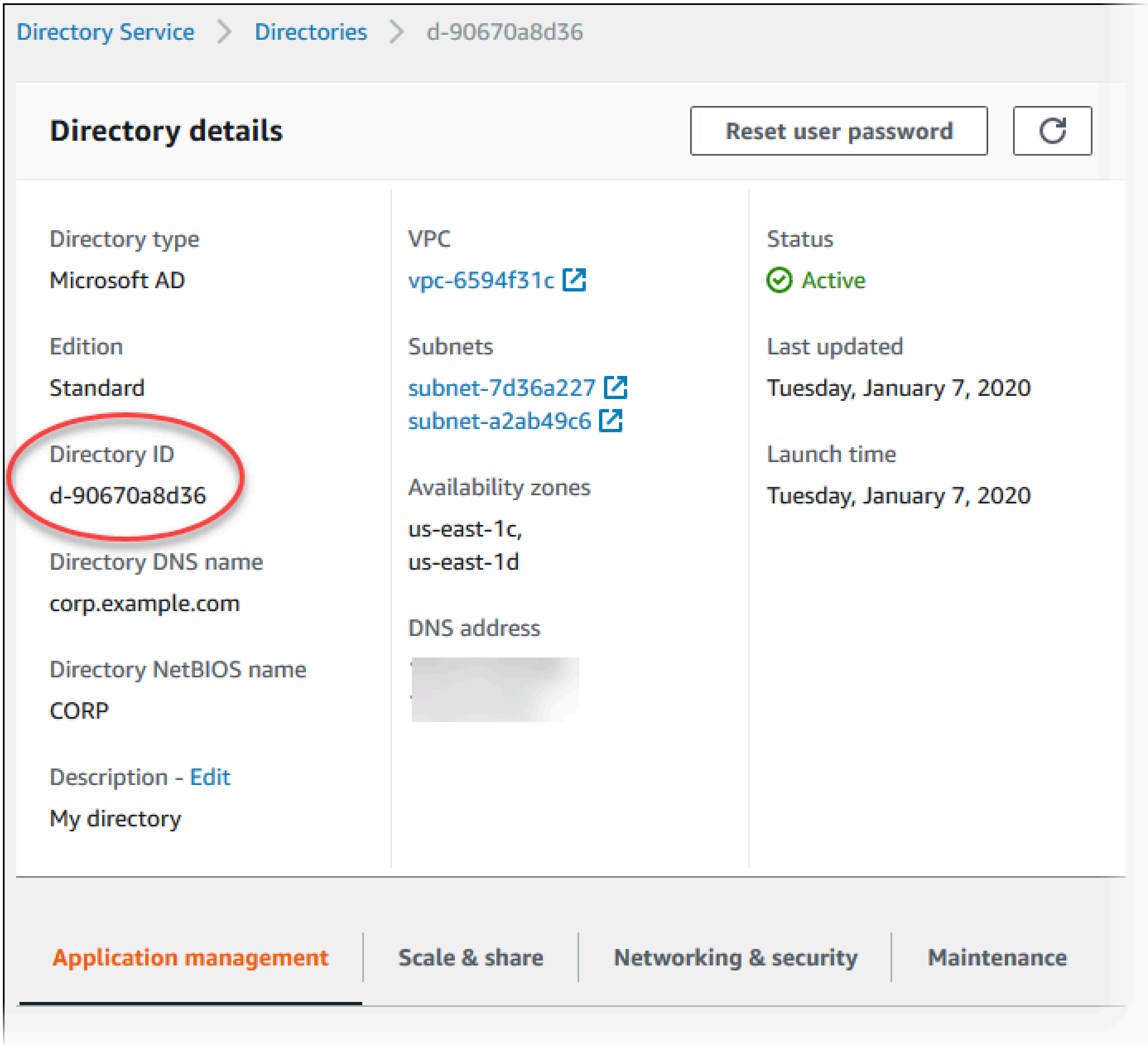
~USD [redacted]*

* Includes two domain controllers, USD [redacted]/mo for each additional domain controller.

Free trial eligible Learn more
30-day limited trial

It takes several minutes for the directory to be created. When it has been successfully created, the **Status** value changes to **Active**.

To see information about your directory, choose the directory ID in the directory listing. Make a note of the **Directory ID** value. You need this value when you create or modify your PostgreSQL DB instance.



Directory details

Directory type
Microsoft AD

Edition
Standard

Directory ID
d-90670a8d36

Directory DNS name
corp.example.com

Directory NetBIOS name
CORP

Description - Edit
My directory

VPC
vpc-6594f31c

Subnets
subnet-7d36a227
subnet-a2ab49c6

Availability zones
us-east-1c,
us-east-1d

DNS address
[redacted]

Status
Active

Last updated
Tuesday, January 7, 2020

Launch time
Tuesday, January 7, 2020

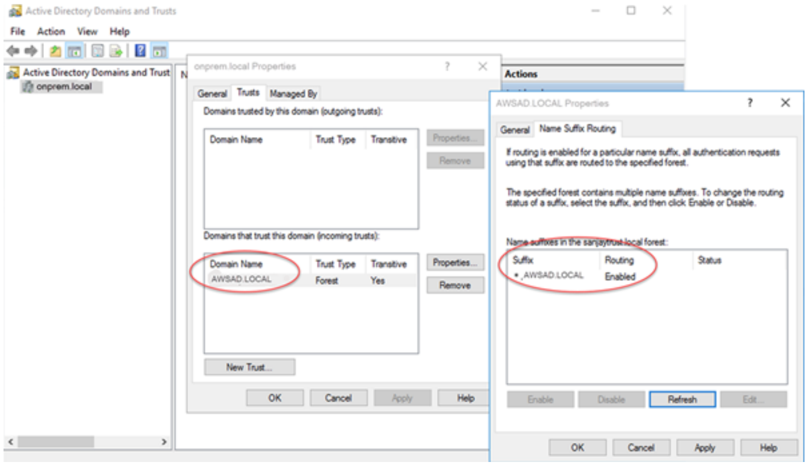
Step 2: (Optional) Create a trust relationship between your on-premises Active Directory and AWS Directory Service

If you don't plan to use your own on-premises Microsoft Active Directory, skip to Step 3: Create an IAM role for Amazon Aurora to access the AWS Directory Service.

To get Kerberos authentication using your on-premises Active Directory, you need to create a trusting domain relationship using a forest trust between your on-premises Microsoft Active Directory and the AWS Managed Microsoft AD directory (created in Step 1: Create a directory using AWS Managed Microsoft AD). The trust can be one-way, where the AWS Managed Microsoft AD directory trusts the on-premises Microsoft Active Directory. The trust can also be two-way, where both Active Directories trust each other. For more information about setting up trusts using AWS Directory Service, see When to create a trust relationship in the *AWS Directory Service Administration Guide*.

> **Note**
>
> If you use an on-premises Microsoft Active Directory:
>
> - Windows clients must connect using the domain name of the AWS Directory Service in the endpoint rather than rds.amazonaws.com. For more information, see Connecting to PostgreSQL with Kerberos authentication.
> - Windows clients can't connect using Aurora custom endpoints. To learn more, see Amazon Aurora connection management.
> - For global databases:
>   - Windows clients can connect using instance endpoints or cluster endpoints in the primary AWS Region of the global database only.
>   - Windows clients can't connect using cluster endpoints in secondary AWS Regions.

Make sure that your on-premises Microsoft Active Directory domain name includes a DNS suffix routing that corresponds to the newly created trust relationship. The following screenshot shows an example.



Domains trusted by this domain (outgoing trusts):

Domain Name	Trust Type	Transitive

Domains that trust this domain (incoming trusts):

Domain Name	Trust Type	Transitive
AWSAD.LOCAL	Forest	Yes

If routing is enabled for a particular name suffix, all authentication requests using that suffix are routed to the specified forest.

The specified forest contains multiple name suffixes. To change the routing status of a suffix, select the suffix, and then click Enable or Disable.

Name suffixes in the sanjaytrust.local forest:

Suffix	Routing	Status
*.AWSAD.LOCAL	Enabled	

Step 3: Create an IAM role for Amazon Aurora to access the AWS Directory Service

For Amazon Aurora to call AWS Directory Service for you, your AWS account needs an IAM role that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`. This role allows Amazon Aurora to make calls to AWS Directory Service. (Note that this IAM role to access the AWS Directory Service is different than the IAM role used for IAM database authentication.)

When you create a DB instance using the AWS Management Console and your console user account has the `iam:CreateRole` permission, the console creates the needed IAM role automatically. In this case, the role name is `rds-directoryservice-kerberos-access-role`. Otherwise, you must create the IAM role manually. When you create this IAM role, choose `Directory Service`, and attach the AWS managed policy `AmazonRDSDirectoryServiceAccess` to it.

For more information about creating IAM roles for a service, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Note**
>
> The IAM role used for Windows Authentication for RDS for Microsoft SQL Server can't be used for Amazon Aurora.

As an alternative to using the `AmazonRDSDirectoryServiceAccess` managed policy, you can create policies with the required permissions. In this case, the IAM role must have the following IAM trust policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "directoryservice.rds.amazonaws.com",
          "rds.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

The role must also have the following IAM role policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ds:DescribeDirectories",
        "ds:AuthorizeApplication",
        "ds:UnauthorizeApplication",
        "ds:GetAuthorizedApplicationDetails"
      ],
    "Effect": "Allow",
    "Resource": "*"
    }
  ]
}
```

Step 4: Create and configure users

You can create users by using the Active Directory Users and Computers tool. This is one of the Active Directory Domain Services and Active Directory Lightweight Directory Services tools. For more information, see Add Users and Computers to the Active Directory domain in the Microsoft documentation. In this case, users are individuals or other entities, such as their computers that are part of the domain and whose identities are being maintained in the directory.

To create users in an AWS Directory Service directory, you must be connected to a Windows-based Amazon EC2 instance that's a member of the AWS Directory Service directory. At the same time, you must be logged in as a user that has privileges to create users. For more information, see Create a user in the *AWS Directory Service Administration Guide*.

Step 5: Enable cross-VPC traffic between the directory and the DB instance

If you plan to locate the directory and the DB cluster in the same VPC, skip this step and move on to Step 6: Create or modify a PostgreSQL DB cluster.

If you plan to locate the directory and the DB instance in different VPCs, configure cross-VPC traffic using VPC peering or AWS Transit Gateway.

The following procedure enables traffic between VPCs using VPC peering. Follow the instructions in What is VPC peering? in the *Amazon Virtual Private Cloud Peering Guide*.

To enable cross-VPC traffic using VPC peering

1. Set up appropriate VPC routing rules to ensure that network traffic can flow both ways.
2. Ensure that the DB instance security group can receive inbound traffic from the directory security group.
3. Ensure that there is no network access control list (ACL) rule to block traffic.

If a different AWS account owns the directory, you must share the directory.

To share the directory between AWS accounts

1. Start sharing the directory with the AWS account that the DB instance will be created in by following the instructions in Tutorial: Sharing your AWS Managed Microsoft AD directory for seamless EC2 Domain-join in the *AWS Directory Service Administration Guide*.
2. Sign in to the AWS Directory Service console using the account for the DB instance, and ensure that the domain has the `SHARED` status before proceeding.
3. While signed into the AWS Directory Service console using the account for the DB instance, note the **Directory ID** value. You use this directory ID to join the DB instance to the domain.

Step 6: Create or modify a PostgreSQL DB cluster

Create or modify a PostgreSQL DB cluster for use with your directory. You can use the console, CLI, or RDS API to associate a DB cluster with a directory. You can do this in one of the following ways:

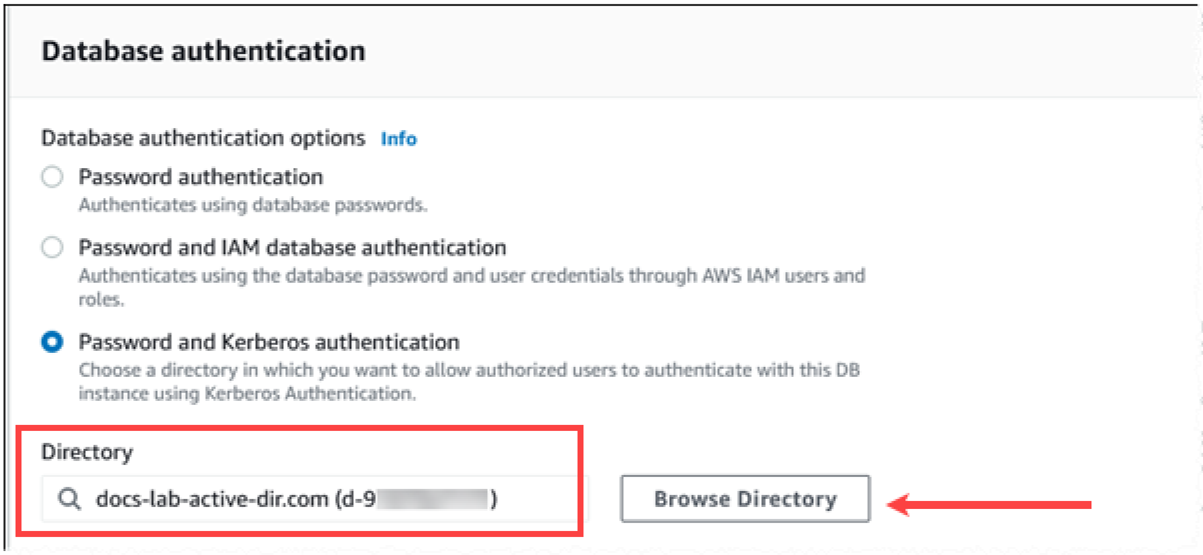
- Create a new PostgreSQL DB cluster using the console, the create-db-cluster CLI command, or the CreateDBCluster RDS API operation. For instructions, see Creating and connecting to an Aurora PostgreSQL DB cluster.
- Modify an existing PostgreSQL DB cluster using the console, the modify-db-cluster CLI command, or the ModifyDBCluster RDS API operation. For instructions, see Modifying an Amazon Aurora DB cluster.
- Restore a PostgreSQL DB cluster from a DB snapshot using the console, the restore-db-cluster-from-db-snapshot CLI command, or the RestoreDBClusterFromDBSnapshot RDS API operation. For instructions, see Restoring from a DB cluster snapshot.
- Restore a PostgreSQL DB cluster to a point-in-time using the console, the restore-db-instance-to-point-in-time CLI command, or the RestoreDBClusterToPointInTime RDS API operation. For instructions, see Restoring a DB cluster to a specified time.

Kerberos authentication is only supported for PostgreSQL DB clusters in a VPC. The DB cluster can be in the same VPC as the directory, or in a different VPC. The DB cluster must use a security group that allows ingress and egress within the directory's VPC so the DB cluster can communicate with the directory.

> **Note**
>
> Enabling Kerberos authentication isn't currently supported on Aurora PostgreSQL DB cluster during migration from RDS for PostgreSQL. You can enable Kerberos authentication only on a standalone Aurora PostgreSQL DB cluster.

Console

When you use the console to create, modify, or restore a DB cluster, choose **Kerberos authentication** in the **Database authentication** section. Then choose **Browse Directory**. Select the directory or choose **Create a new directory** to use the Directory Service.



Database authentication

Database authentication options Info

○ Password authentication
Authenticates using database passwords.

○ Password and IAM database authentication
Authenticates using the database password and user credentials through AWS IAM users and roles.

● Password and Kerberos authentication
Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

Directory
docs-lab-active-dir.com (d-9[redacted])

AWS CLI

When you use the AWS CLI, the following parameters are required for the DB cluster to be able to use the directory that you created:

- For the `--domain` parameter, use the domain identifier ("d-*" identifier) generated when you created the directory.
- For the `--domain-iam-role-name` parameter, use the role you created that uses the managed IAM policy `AmazonRDSDirectoryServiceAccess`.

For example, the following CLI command modifies a DB cluster to use a directory.

```
aws rds  modify-db-cluster --db-cluster-identifier mydbinstance --domain d-Directory-ID
 --domain-iam-role-name role-name
```

> **Important**
>
> If you modify a DB cluster to enable Kerberos authentication, reboot the DB cluster after making the change.

Step 7: Create PostgreSQL users for your Kerberos principals

At this point, your Aurora PostgreSQL DB cluster is joined to the AWS Managed Microsoft AD domain. The users that you created in the directory in Step 4: Create and configure users need to be set up as PostgreSQL database users and granted privileges to login to the database. You do that by signing in as the database user with rds_superuser privileges. For example, if you accepted the defaults when you created your Aurora PostgreSQL DB cluster, you use postgres, as shown in the following steps.

To create PostgreSQL database users for Kerberos principals

1. Use psql to connect to your Aurora PostgreSQL DB cluster's DB instance endpoint using psql. The following example uses the default postgres account for the rds_superuser role.

```
psql --host=cluster-instance-1.111122223333.aws-region.rds.amazonaws.com --
port=5432 --username=postgres --password
```

2. Create a database user name for each Kerberos principal (Active Directory username) that you want to have access to the database. Use the canonical username (identity) as defined in the Active Directory instance, that is, a lower-case alias (username in Active Directory) and the upper-case name of the Active Directory domain for that user name. The Active Directory user name is an externally authenticated user, so use quotes around the name as shown following.

```
postgres=> CREATE USER "username@CORP.EXAMPLE.COM" WITH LOGIN;
CREATE ROLE
```

3. Grant the rds_ad role to the database user.

```
postgres=> GRANT rds_ad TO "username@CORP.EXAMPLE.COM";
GRANT ROLE
```

After you finish creating all the PostgreSQL users for your Active Directory user identities, users can access the Aurora PostgreSQL DB cluster by using their Kerberos credentials.

It's required that the database users who authenticate using Kerberos are doing so from client machines that are members of the Active Directory domain.

Database users that have been granted the rds_ad role can't also have the rds_iam role. This also applies to nested memberships. For more information, see IAM database authentication.

Configuring your Aurora PostgreSQL DB cluster for case-insensitive user names

Aurora PostgreSQL versions 14.5, 13.8, 12.12, and 11.17 support the krb_caseins_users PostgreSQL parameter. This parameter supports case-insensitive Active Directory user names. By default, this parameter is set to false, so user names are interpreted case-sensitively by Aurora PostgreSQL. That's the default behavior in all older versions of Aurora PostgreSQL. However, you can set this parameter to `true` in your custom DB cluster parameter group and allow your Aurora PostgreSQL DB cluster to interpret user names, case-insensitively. Consider doing this as a convenience for your database users, who might sometimes mis-type the casing of their user name when authenticating using Active Directory.

To change the `krb_caseins_users` parameter, your Aurora PostgreSQL DB cluster must be using a custom DB cluster parameter group. For information about working with a custom DB cluster parameter group, see Working with parameter groups.

You can use the AWS CLI or the AWS Management Console to change the setting. For more information, see Modifying parameters in a DB cluster parameter group.

Step 8: Configure a PostgreSQL client

To configure a PostgreSQL client, take the following steps:

- Create a krb5.conf file (or equivalent) to point to the domain.
- Verify that traffic can flow between the client host and AWS Directory Service. Use a network utility such as Netcat for the following:
  - Verify traffic over DNS for port 53.
  - Verify traffic over TCP/UDP for port 53 and for Kerberos, which includes ports 88 and 464 for AWS Directory Service.
- Verify that traffic can flow between the client host and the DB instance over the database port. For example, use psql to connect and access the database.

The following is sample krb5.conf content for AWS Managed Microsoft AD.

```
[libdefaults]
 default_realm = EXAMPLE.COM
[realms]
 EXAMPLE.COM = {
  kdc = example.com
  admin_server = example.com
 }
[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
```

The following is sample krb5.conf content for an on-premises Microsoft Active Directory.

```
[libdefaults]
 default_realm = EXAMPLE.COM
[realms]
 EXAMPLE.COM = {
  kdc = example.com
  admin_server = example.com
 }
 ONPREM.COM = {
  kdc = onprem.com
  admin_server = onprem.com
 }
[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
 .onprem.com = ONPREM.COM
 onprem.com = ONPREM.COM
 .rds.amazonaws.com = EXAMPLE.COM
 .amazonaws.com.cn = EXAMPLE.COM
 .amazon.com = EXAMPLE.COM
```

Managing a DB cluster in a Domain

You can use the console, the CLI, or the RDS API to manage your DB cluster and its relationship with your Microsoft Active Directory. For example, you can associate an Active Directory to enable Kerberos authentication. You can also remove the association for an Active Directory to disable Kerberos authentication. You can also move a DB cluster to be externally authenticated by one Microsoft Active Directory to another.

For example, using the CLI, you can do the following:

- To reattempt enabling Kerberos authentication for a failed membership, use the modify-db-cluster CLI command. Specify the current membership's directory ID for the `--domain` option.
- To disable Kerberos authentication on a DB instance, use the modify-db-cluster CLI command. Specify none for the `--domain` option.
- To move a DB instance from one domain to another, use the modify-db-cluster CLI command. Specify the domain identifier of the new domain for the `--domain` option.

Understanding Domain membership

After you create or modify your DB cluster, the DB instances become members of the domain. You can view the status of the domain membership in the console or by running the describe-db-instances CLI command. The status of the DB instance can be one of the following:

- `kerberos-enabled` – The DB instance has Kerberos authentication enabled.
- `enabling-kerberos` – AWS is in the process of enabling Kerberos authentication on this DB instance.
- `pending-enable-kerberos` – Enabling Kerberos authentication is pending on this DB instance.
- `pending-maintenance-enable-kerberos` – AWS will attempt to enable Kerberos authentication on the DB instance during the next scheduled maintenance window.
- `pending-disable-kerberos` – Disabling Kerberos authentication is pending on this DB instance.
- `pending-maintenance-disable-kerberos` – AWS will attempt to disable Kerberos authentication on the DB instance during the next scheduled maintenance window.
- `enable-kerberos-failed` – A configuration problem prevented AWS from enabling Kerberos authentication on the DB instance. Correct the configuration problem before reissuing the command to modify the DB instance.
- `disabling-kerberos` – AWS is in the process of disabling Kerberos authentication on this DB instance.

A request to enable Kerberos authentication can fail because of a network connectivity issue or an incorrect IAM role. In some cases, the attempt to enable Kerberos authentication might fail when you create or modify a DB cluster. If so, make sure that you are using the correct IAM role, then modify the DB cluster to join the domain.

Connecting to PostgreSQL with Kerberos authentication

You can connect to PostgreSQL with Kerberos authentication with the pgAdmin interface or with a command-line interface such as psql. For more information about connecting, see Connecting to an Amazon Aurora PostgreSQL DB cluster. For information about obtaining the endpoint, port number, and other details needed for connection, see Viewing the endpoints for an Aurora cluster.

pgAdmin

To use pgAdmin to connect to PostgreSQL with Kerberos authentication, take the following steps:

1. Launch the pgAdmin application on your client computer.
2. On the **Dashboard** tab, choose **Add New Server**.
3. In the **Create - Server** dialog box, enter a name on the **General** tab to identify the server in pgAdmin.
4. On the **Connection** tab, enter the following information from your Aurora PostgreSQL database.
   - For **Host**, enter the endpoint for the Writer instance of your Aurora PostgreSQL DB cluster. An endpoint looks similar to the following:

     ```
     AUR-cluster-instance.111122223333.aws-region.rds.amazonaws.com
     ```

     To connect to an on-premises Microsoft Active Directory from a Windows client, you use the domain name of the AWS Managed Active Directory instead of `rds.amazonaws.com` in the host endpoint. For example, suppose that the domain name for the AWS Managed Active Directory is `corp.example.com`. Then for **Host**, the endpoint would be specified as follows:

     ```
     AUR-cluster-instance.111122223333.aws-region.corp.example.com
     ```
   - For **Port**, enter the assigned port.
   - For **Maintenance database**, enter the name of the initial database to which the client will connect.
   - For **Username**, enter the user name that you entered for Kerberos authentication in Step 7: Create PostgreSQL users for your Kerberos principals .
5. Choose **Save**.

Psql

To use psql to connect to PostgreSQL with Kerberos authentication, take the following steps:

1. At a command prompt, run the following command.

   ```
   kinit username
   ```

Replace *username* with the user name. At the prompt, enter the password stored in the Microsoft Active Directory for the user.

2. If the PostgreSQL DB cluster is using a publicly accessible VPC, put IP address for your DB cluster endpoint in your `/etc/hosts` file on the EC2 client. For example, the following commands obtain the IP address and then put it in the `/etc/hosts` file.

```
% dig +short PostgreSQL-endpoint.AWS-Region.rds.amazonaws.com
;; Truncated, retrying in TCP mode.
ec2-34-210-197-118.AWS-Region.compute.amazonaws.com.
34.210.197.118

% echo " 34.210.197.118  PostgreSQL-endpoint.AWS-Region.rds.amazonaws.com" >> /etc/
hosts
```

   If you're using an on-premises Microsoft Active Directory from a Windows client, then you need to connect using a specialized endpoint. Instead of using the Amazon domain `rds.amazonaws.com` in the host endpoint, use the domain name of the AWS Managed Active Directory.

   For example, suppose that the domain name for your AWS Managed Active Directory is `corp.example.com`. Then use the format *`PostgreSQL-endpoint.AWS-Region.corp.example.com`* for the endpoint and put it in the `/etc/hosts` file.

```
% echo " 34.210.197.118  PostgreSQL-endpoint.AWS-Region.corp.example.com" >> /etc/
hosts
```

3. Use the following psql command to log in to a PostgreSQL DB cluster that is integrated with Active Directory. Use a cluster or instance endpoint.

```
psql -U username@CORP.EXAMPLE.COM -p 5432 -h PostgreSQL-endpoint.AWS-
Region.rds.amazonaws.com postgres
```

   To log in to the PostgreSQL DB cluster from a Windows client using an on-premises Active Directory, use the following psql command with the domain name from the previous step (`corp.example.com`):

```
psql -U username@CORP.EXAMPLE.COM -p 5432 -h PostgreSQL-endpoint.AWS-
Region.corp.example.com postgres
```

Using AD security groups for Aurora PostgreSQL access control

From Aurora PostgreSQL 14.10 and 15.5 versions, Aurora PostgreSQL access control can be managed using AWS Directory Service for Microsoft Active Directory (AD) security groups. Earlier versions of Aurora PostgreSQL support Kerberos based authentication with AD only for individual users. Each AD user had to be explicitly provisioned to DB cluster to get access.

Instead of explicitly provisioning each AD user to DB cluster based on business needs, you can leverage AD security groups as explained below:

- AD users are members of various AD security groups in an Active Directory. These are not dictated by DB cluster administrator, but are based on business requirements, and are handled by an AD administrator.
- DB cluster administrators create DB roles in DB instances based on business requirements. These DB roles may have different permissions or privileges.
- DB cluster administrators configure a mapping from AD security groups to DB roles on a per DB cluster basis.
- DB users can access DB clusters using their AD credentials. Access is based on AD security group membership. AD users gain or lose access automatically based on their AD group memberships.

Prerequisites

Ensure that you have the following before setting up the extension for AD Security groups:

- Setup Kerberos authentication for PostgreSQL DB clusters. For more information, see Setting up Kerberos authentication for PostgreSQL DB clusters .

> **Note**
>
> For AD security groups, skip Step 7: Create PostgreSQL users for your Kerberos principals in this setup procedure.

- Managing a DB cluster in a Domain. For more information, see Managing a DB cluster in a Domain .

Setting up the pg_ad_mapping extension

Aurora PostgreSQL is now providing pg_ad_mapping extension to manage the mapping between AD security groups and DB roles in Aurora PostgreSQL cluster. For more information about the functions provided by pg_ad_mapping, see Using functions from the pg_ad_mapping extension.

To set up the pg_ad_mapping extension on your Aurora PostgreSQL DB cluster, you first add pg_ad_mapping to the shared libraries on the custom DB cluster parameter group for your Aurora PostgreSQL DB cluster. For information about creating a custom DB cluster parameter group, see Working with parameter groups. Next, you install the pg_ad_mapping extension. The procedures in this section show you how. You can use the AWS Management Console or the AWS CLI.

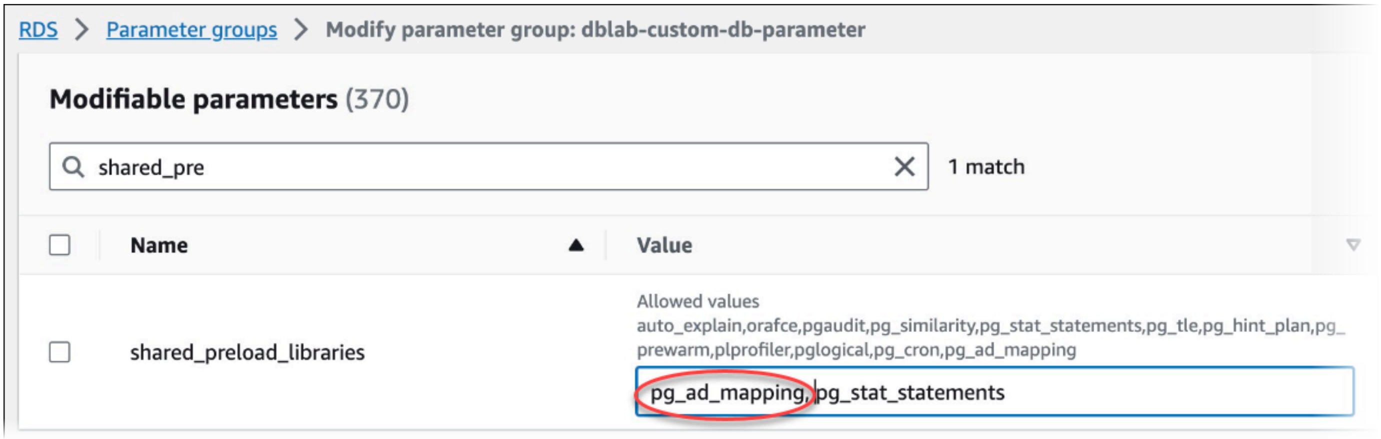
You must have permissions as the rds_superuser role to perform all these tasks.

The steps following assume that your Aurora PostgreSQL DB cluster is associated with a custom DB cluster parameter group.

Console

To set up the pg_ad_mapping extension

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose your Aurora PostgreSQL DB cluster's Writer instance.
3. Open the **Configuration** tab for your Aurora PostgreSQL DB cluster writer instance. Among the Instance details, find the **Parameter group** link.
4. Choose the link to open the custom parameters associated with your Aurora PostgreSQL DB cluster.
5. In the **Parameters** search field, type shared_pre to find the shared_preload_libraries parameter.
6. Choose **Edit parameters** to access the property values.
7. Add pg_ad_mapping to the list in the **Values** field. Use a comma to separate items in the list of values.



RDS > Parameter groups > Modify parameter group: dblab-custom-db-parameter

Modifiable parameters (370)

shared_pre

1 match

□	Name	Value
□	shared_preload_libraries	Allowed values auto_explain,orafce,pgaudit,pg_similarity,pg_stat_statements,pg_tle,pg_hint_plan,pg_prewarm,plprofiler,pglogical,pg_cron,pg_ad_mapping pg_ad_mapping,pg_stat_statements

8. Reboot the writer instance of your Aurora PostgreSQL DB cluster so that your change to the `shared_preload_libraries` parameter takes effect.

9. When the instance is available, verify that `pg_ad_mapping` has been initialized. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

```
SHOW shared_preload_libraries;
shared_preload_libraries
--------------------------
rdsutils,pg_ad_mapping
(1 row)
```

10. With `pg_ad_mapping` initialized, you can now create the extension. You need to create the extension after initializing the library to start using the functions provided by this extension.

```
CREATE EXTENSION pg_ad_mapping;
```

11. Close the `psql` session.

```
labdb=> \q
```

AWS CLI

To setup pg_ad_mapping

To setup pg_ad_mapping using the AWS CLI, you call the modify-db-parameter-group operation to add this parameter in your custom parameter group, as shown in the following procedure.

1. Use the following AWS CLI command to add pg_ad_mapping to the shared_preload_libraries parameter.

```
aws rds modify-db-parameter-group \
   --db-parameter-group-name custom-param-group-name \
   --parameters
 "ParameterName=shared_preload_libraries,ParameterValue=pg_ad_mapping,ApplyMethod=pending-
reboot" \
   --region aws-region
```

2. Use the following AWS CLI command to reboot the writer instance of your Aurora PostgreSQL DB cluster so that the pg_ad_mapping is initialized.

```
aws rds reboot-db-instance \
    --db-instance-identifier writer-instance \
    --region aws-region
```

3. When the instance is available, you can verify that pg_ad_mapping has been initialized. Use psql to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

```
SHOW shared_preload_libraries;
shared_preload_libraries
--------------------------
rdsutils,pg_ad_mapping
(1 row)
```

   With pg_ad_mapping initialized, you can now create the extension.

```
CREATE EXTENSION pg_ad_mapping;
```

4. Close the psql session so that you can use the AWS CLI.

```
labdb=> \q
```

Retrieving Active Directory Group SID in PowerShell

A security identifier (SID) is used to uniquely identify a security principal or security group. Whenever a security group or account is created in Active Directory a SID is assigned to it. To fetch the AD security group SID from the active directory, you can use the Get-ADGroup cmdlet from windows client machine which is joined with that Active Directory domain. The Identity parameter specifies the Active Directory group name to get the corresponding SID.

The following example returns the SID of AD group *adgroup1*.

```
C:\Users\Admin> Get-ADGroup -Identity adgroup1 | select SID

             SID
-----------------------------------------------
S-1-5-21-3168537779-1985441202-1799118680-1612
```

Mapping DB role with AD security group

You need to explicitly provision the AD security groups in the database as a PostgreSQL DB role. An AD user, who is part of at least one provisioned AD security group will get access to the database. You shouldn't grant `rds_ad role` to AD group security based DB role. Kerberos authentication for security group will get triggered by using the domain name suffix like *user1@example.com*. This DB role can't use Password or IAM authentication to gain access to database.

> **Note**
>
> AD users who have a corresponding DB role in the database with `rds_ad` role granted to them, can't login as part of the AD security group. They will get access through DB role as an individual user.

For example, accounts-group is a security group in AD where you would like to provision this security group in the Aurora PostgreSQL as accounts-role.

AD Security Group	PosgreSQL DB role
accounts-group	accounts-role

When mapping the DB role with the AD security group, you must ensure that DB role has the LOGIN attribute set and it has CONNECT privilege to the required login database.

```
postgres => alter role accounts-role login;

ALTER ROLE
postgres => grant connect on database accounts-db to accounts-role;
```

Admin can now proceed to create the mapping between AD security group and PostgreSQL DB role.

```
admin=>select pgadmap_set_mapping('accounts-group', 'accounts-role', <SID>, <Weight>);
```

For information on retrieving SID of AD security group, see Retrieving Active Directory Group SID in PowerShell.

There might be cases where an AD user belongs to multiple groups, in that case, AD user will inherit the privileges of the DB role, which was provisioned with the highest weight. If the two roles have the same weight, AD user will inherit the privileges of the DB role corresponding to the mapping that was added recently. The recommendation is to specify weights that reflect the relative permissions/privileges of individual DB roles. Higher the permissions or privileges of a DB role, higher the weight that should be associated with the mapping entry. This will avoid the ambiguity of two mappings having the same weight.

The following table shows a sample mapping from AD security groups to Aurora PostgreSQL DB roles.

AD Security Group	PosgreSQL DB role	Weight
accounts-group	accounts-role	7
sales-group	sales-role	10
dev-group	dev-role	7

In the following example, `user1` will inherit the privileges of sales-role since it has the higher weight while `user2` will inherit the privileges of `dev-role` as the mapping for this role was created after `accounts-role`, which share the same weight as `accounts-role`.

Username	Security Group membership
user1	accounts-group sales-group
user2	accounts-group dev-group

The psql commands to establish, list, and clear the mappings are shown below. Currently, it isn't possible to modify a single mapping entry. The existing entry needs to be deleted and the mapping recreated.

```
admin=>select pgadmap_set_mapping('accounts-group', 'accounts-role', 'S-1-5-67-890',
 7);
admin=>select pgadmap_set_mapping('sales-group', 'sales-role', 'S-1-2-34-560', 10);
admin=>select pgadmap_set_mapping('dev-group', 'dev-role', 'S-1-8-43-612', 7);

admin=>select * from pgadmap_read_mapping();

ad_sid       | pg_role        | weight | ad_grp
-------------+----------------+--------+---------------
S-1-5-67-890 | accounts-role | 7      | accounts-group
S-1-2-34-560 | sales-role    | 10     | sales-group
S-1-8-43-612 | dev-role      | 7      | dev-group
(3 rows)
```

AD user identity logging/auditing

Use the following command to determine the database role inherited by current or session user:

```
postgres=>select session_user, current_user;

session_user | current_user
-------------+--------------
dev-role     | dev-role

(1 row)
```

To determine the AD security principal identity, use the following command:

```
postgres=>select principal from pg_stat_gssapi where pid = pg_backend_pid();

 principal
-------------------------
 user1@example.com

(1 row)
```

Currently, AD user identity isn't visible in the audit logs. The `log_connections` parameter can be enabled to log DB session establishment. For more information, see log_connections. The output for this includes the AD user identity, as shown below. The backend PID associated with this output can then help attribute actions back to the actual AD user.

```
pgrole1@postgres:[615]:LOG: connection authorized: user=pgrole1
 database=postgres application_name=psql GSS (authenticated=yes, encrypted=yes,
 principal=Admin@EXAMPLE.COM)
```

Limitations

- Microsoft Entra ID known as Azure Active Directory isn't supported.

Using functions from the pg_ad_mapping extension

`pg_ad_mapping` extension provided support to the following functions:

pgadmap_set_mapping

This function establishes the mapping between AD security group and Database role with an associated weight.

Syntax

```
pgadmap_set_mapping(
ad_group,
db_role,
ad_group_sid,
weight)
```

Arguments

Parameter	Description
ad_group	Name of AD Group. Value cannot be null or empty string.
db_role	Database role to be mapped to the specified AD Group. Value cannot be null or empty string.
ad_group_sid	Security identifier that is used to uniquely identify the AD group. Value starts with 'S-1-' and cannot be null or empty string. For more information, see Retrieving Active Directory Group SID in PowerShell.
weight	Weight associated with the database role. The role with highest weight gets precedence when user is a member of multiple groups. Default value of weight is 1.

Return type

None

Usage notes

This function adds a new mapping from AD security group to database role. It can only be executed on the primary DB instance of the DB cluster by a user having rds_superuser privilege.

Examples

```
postgres=> select pgadmap_set_mapping('accounts-group','accounts-
role','S-1-2-33-12345-67890-12345-678',10);

pgadmap_set_mapping

(1 row)
```

pgadmap_read_mapping

This function lists the mappings between AD security group and DB role that were set using `pgadmap_set_mapping` function.

Syntax

```
pgadmap_read_mapping()
```

Arguments

None

Return type

Parameter	Description
ad_group_sid	Security identifier that is used to uniquely identify the AD group. Value starts with 'S-1-' and cannot be null or empty string. For more information, see Retrieving Active Directory Group SID in PowerShell.accounts-role@example.com
db_role	Database role to be mapped to the specified AD Group. Value cannot be null or empty string.
weight	Weight associated with the database role. The role with highest weight gets precedence when user is a member of multiple groups. Default value of weight is 1.
ad_group	Name of AD Group. Value cannot be null or empty string.

Usage notes

Call this function to list all the available mappings between AD security group and DB role.

Examples

```
postgres=> select * from pgadmap_read_mapping();

ad_sid                                  | pg_role       | weight | ad_grp
```

```
------------------------------------+---------------+--------+------------------
S-1-2-33-12345-67890-12345-678      | accounts-role | 10     | accounts-group
(1 row)

(1 row)
```

pgadmap_reset_mapping

This function resets one or all the mappings that were set using `pgadmap_set_mapping` function.

Syntax

```
pgadmap_reset_mapping(
ad_group_sid,
db_role,
weight)
```

Arguments

Parameter	Description
ad_group_sid	Security identifier that is used to uniquely identify the AD group.
db_role	Database role to be mapped to the specified AD Group.
weight	Weight associated with the database role.

If no arguments are provided, all AD group to DB role mappings are reset. Either all arguments need to be provided or none.

Return type

None

Usage notes

Call this function to delete a specific AD group to DB role mapping or to reset all mappings. This function can only be executed on the primary DB instance of the DB cluster by a user having `rds_superuser` privilege.

Examples

```
postgres=> select * from pgadmap_read_mapping();

   ad_sid                       | pg_role      | weight       | ad_grp
--------------------------------+--------------+--------------+-------------------
 S-1-2-33-12345-67890-12345-678 | accounts-role| 10           | accounts-group
 S-1-2-33-12345-67890-12345-666 | sales-role   | 10           | sales-group

(2 rows)
postgres=> select pgadmap_reset_mapping('S-1-2-33-12345-67890-12345-678', 'accounts-
role', 10);

pgadmap_reset_mapping
(1 row)

postgres=> select * from pgadmap_read_mapping();

   ad_sid                       | pg_role      | weight       | ad_grp
--------------------------------+--------------+--------------+---------------
 S-1-2-33-12345-67890-12345-666 | sales-role   | 10           | sales-group

(1 row)
postgres=> select pgadmap_reset_mapping();

pgadmap_reset_mapping
(1 row)

postgres=> select * from pgadmap_read_mapping();

   ad_sid                       | pg_role      | weight       | ad_grp
--------------------------------+--------------+--------------+--------------
 (0 rows)
```

Migrating data to Amazon Aurora with PostgreSQL compatibility

You have several options for migrating data from your existing database to an Amazon Aurora PostgreSQL-Compatible Edition DB cluster. Your migration options also depend on the database that you are migrating from and the size of the data that you are migrating. Following are your options:

Migrating an RDS for PostgreSQL DB instance using a snapshot

You can migrate data directly from an RDS for PostgreSQL DB snapshot to an Aurora PostgreSQL DB cluster.

Migrating an RDS for PostgreSQL DB instance using an Aurora read replica

You can also migrate from an RDS for PostgreSQL DB instance by creating an Aurora PostgreSQL read replica of an RDS for PostgreSQL DB instance. When the replica lag between the RDS for PostgreSQL DB instance and the Aurora PostgreSQL read replica is zero, you can stop replication. At this point, you can make the Aurora read replica a standalone Aurora PostgreSQL DB cluster for reading and writing.

Importing data from Amazon S3 into Aurora PostgreSQL

You can migrate data by importing it from Amazon S3 into a table belonging to an Aurora PostgreSQL DB cluster.

Migrating from a database that is not PostgreSQL-compatible

You can use AWS Database Migration Service (AWS DMS) to migrate data from a database that is not PostgreSQL-compatible. For more information on AWS DMS, see What is AWS Database Migration Service? in the *AWS Database Migration Service User Guide*.

> **Note**
>
> Enabling Kerberos authentication isn't currently supported on Aurora PostgreSQL DB cluster during migration from RDS for PostgreSQL. You can enable Kerberos authentication only on a standalone Aurora PostgreSQL DB cluster.

For a list of AWS Regions where Aurora is available, see Amazon Aurora in the *AWS General Reference*.

> **Important**
>
> If you plan to migrate an RDS for PostgreSQL DB instance to an Aurora PostgreSQL DB cluster in the near future, we strongly recommend that you turn off auto minor version

upgrades for the DB instance early in the migration planning phase. Migration to Aurora PostgreSQL might be delayed if the RDS for PostgreSQL version isn't yet supported by Aurora PostgreSQL.

For information about Aurora PostgreSQL versions, see Engine versions for Amazon Aurora PostgreSQL.

Migrating a snapshot of an RDS for PostgreSQL DB instance to an Aurora PostgreSQL DB cluster

To create an Aurora PostgreSQL DB cluster, you can migrate a DB snapshot of an RDS for PostgreSQL DB instance. The new Aurora PostgreSQL DB cluster is populated with the data from the original RDS for PostgreSQL DB instance. For information about creating a DB snapshot, see Creating a DB snapshot.

In some cases, the DB snapshot might not be in the AWS Region where you want to locate your data. If so, use the Amazon RDS console to copy the DB snapshot to that AWS Region. For information about copying a DB snapshot, see Copying a DB snapshot.

You can migrate RDS for PostgreSQL snapshots that are compatible with the Aurora PostgreSQL versions available in the given AWS Region. For example, you can migrate a snapshot from an RDS for PostgreSQL 11.1 DB instance to Aurora PostgreSQL version 11.4, 11.7, 11.8, or 11.9 in the US West (N. California) Region. You can migrate RDS for PostgreSQL 10.11 snapshot to Aurora PostgreSQL 10.11, 10.12, 10.13, and 10.14. In other words, the RDS for PostgreSQL snapshot must use the same or a lower minor version as the Aurora PostgreSQL.

You can also choose for your new Aurora PostgreSQL DB cluster to be encrypted at rest by using an AWS KMS key. This option is available only for unencrypted DB snapshots.

To migrate an RDS for PostgreSQL DB snapshot to an Aurora PostgreSQL DB cluster, you can use the AWS Management Console, the AWS CLI, or the RDS API. When you use the AWS Management Console, the console takes the actions necessary to create both the DB cluster and the primary instance.

Console

To migrate a PostgreSQL DB snapshot by using the RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Snapshots**.
3. On the **Snapshots** page, choose the RDS for PostgreSQL snapshot that you want to migrate into an Aurora PostgreSQL DB cluster.
4. Choose **Actions** then choose **Migrate snapshot**.
5. Set the following values on the **Migrate database** page:

   - **DB engine version**: Choose a DB engine version you want to use for the new migrated instance.
   - **DB instance identifier**: Enter a name for the DB cluster that is unique for your account in the AWS Region that you chose. This identifier is used in the endpoint addresses for the instances in your DB cluster. You might choose to add some intelligence to the name, such as including the AWS Region and DB engine that you chose, for example **`aurora-cluster1`**.

     The DB instance identifier has the following constraints:

     - It must contain 1–63 alphanumeric characters or hyphens.
     - Its first character must be a letter.
     - It can't end with a hyphen or contain two consecutive hyphens.
     - It must be unique for all DB instances per AWS account, per AWS Region.
   - **DB instance class**: Choose a DB instance class that has the required storage and capacity for your database, for example `db.r6g.large`. Aurora cluster volumes automatically grow as the amount of data in your database increases. So you only need to choose a DB instance class that meets your current storage requirements. For more information, see Overview of Amazon Aurora storage.
   - **Virtual private cloud (VPC)**: If you have an existing VPC, then you can use that VPC with your Aurora PostgreSQL DB cluster by choosing your VPC identifier, for example `vpc-a464d1c1`. For information about creating a VPC, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

     Otherwise, you can choose to have Amazon RDS create a VPC for you by choosing **Create new VPC**.

- **DB subnet group**: If you have an existing subnet group, then you can use that subnet group with your Aurora PostgreSQL DB cluster by choosing your subnet group identifier, for example `gs-subnet-group1`.
- **Public access**: Choose **No** to specify that instances in your DB cluster can only be accessed by resources inside of your VPC. Choose **Yes** to specify that instances in your DB cluster can be accessed by resources on the public network.

  > **Note**
  >
  > Your production DB cluster might not need to be in a public subnet, because only your application servers require access to your DB cluster. If your DB cluster doesn't need to be in a public subnet, set **Public access** to **No**.

- **VPC security group**: Choose a VPC security group to allow access to your database.
- **Availability Zone**: Choose the Availability Zone to host the primary instance for your Aurora PostgreSQL DB cluster. To have Amazon RDS choose an Availability Zone for you, choose **No preference**.
- **Database port**: Enter the default port to be used when connecting to instances in the Aurora PostgreSQL DB cluster. The default is 5432.

  > **Note**
  >
  > You might be behind a corporate firewall that doesn't allow access to default ports such as the PostgreSQL default port, 5432. In this case, provide a port value that your corporate firewall allows. Remember that port value later when you connect to the Aurora PostgreSQL DB cluster.

- **Enable Encryption**: Choose **Enable Encryption** for your new Aurora PostgreSQL DB cluster to be encrypted at rest. Also choose a KMS key as the **AWS KMS key** value.
- **Auto minor version upgrade**: Choose **Enable auto minor version upgrade** to enable your Aurora PostgreSQL DB cluster to receive minor PostgreSQL DB engine version upgrades automatically when they become available.

  The **Auto minor version upgrade** option only applies to upgrades to PostgreSQL minor engine versions for your Aurora PostgreSQL DB cluster. It doesn't apply to regular patches applied to maintain system stability.

6. Choose **Migrate** to migrate your DB snapshot.

7. Choose **Databases** to see the new DB cluster. Choose the new DB cluster to monitor the progress of the migration. When the migration completes, the Status for the cluster is **Available**. On the **Connectivity & security** tab, you can find the cluster endpoint to use for connecting to the primary writer instance of the DB cluster. For more information on connecting to an Aurora PostgreSQL DB cluster, see Connecting to an Amazon Aurora DB cluster.

AWS CLI

Using the AWS CLI to migrate an RDS for PostgreSQL DB snapshot to an Aurora PostgreSQL involves two separate AWS CLI commands. First, you use the `restore-db-cluster-from-snapshot` AWS CLI command create a new Aurora PostgreSQL DB cluster. You then use the `create-db-instance` command to create the primary DB instance in the new cluster to complete the migration. The following procedure creates an Aurora PostgreSQL DB cluster with primary DB instance that has the same configuration as the DB instance used to create the snapshot.

To migrate an RDS for PostgreSQL DB snapshot to an Aurora PostgreSQL DB cluster

1. Use the describe-db-snapshots command to obtain information about the DB snapshot you want to migrate. You can specify either the `--db-instance-identifier` parameter or the `--db-snapshot-identifier` in the command. If you don't specify one of these parameters, you get all snapshots.

   ```
   aws rds describe-db-snapshots --db-instance-identifier <your-db-instance-name>
   ```

2. The command returns all configuration details for any snapshots created from the DB instance specified. In the output, find the snapshot that you want to migrate and locate its Amazon Resource Name (ARN). To learn more about Amazon RDS ARNs, see Amazon Relational Database Service (Amazon RDS). An ARN looks similar to the output following.

   ```
   “DBSnapshotArn": "arn:aws:rds:aws-region:111122223333:snapshot:<snapshot_name>"
   ```

   Also in the output you can find configuration details for the RDS for PostgreSQL DB instance, such as the engine version, allocated storage, whether or not the DB instance is encrypted, and so on.

3. Use the restore-db-cluster-from-snapshot command to start the migration. Specify the following parameters:

- `--db-cluster-identifier` – The name that you want to give to the Aurora PostgreSQL DB cluster. This Aurora DB cluster is the target for your DB snapshot migration.
- `--snapshot-identifier` – The Amazon Resource Name (ARN) of the DB snapshot to migrate.
- `--engine` – Specify `aurora-postgresql` for the Aurora DB cluster engine.
- `--kms-key-id` – This optional parameter lets you create an encrypted Aurora PostgreSQL DB cluster from an unencrypted DB snapshot. It also lets you choose a different encryption key for the DB cluster than the key used for the DB snapshot.

> **Note**
>
> You can't create an unencrypted Aurora PostgreSQL DB cluster from an encrypted DB snapshot.

Without the `--kms-key-id` parameter specified as shown following, the restore-db-cluster-from-snapshot AWS CLI command creates an empty Aurora PostgreSQL DB cluster that's either encrypted using the same key as the DB snapshot or is unencrypted if the source DB snapshot isn't encrypted.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier cluster-name \
    --snapshot-identifier arn:aws:rds:aws-region:111122223333:snapshot:your-
snapshot-name \
    --engine aurora-postgresql
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-cluster-identifier new_cluster ^
    --snapshot-identifier arn:aws:rds:aws-region:111122223333:snapshot:your-
snapshot-name ^
    --engine aurora-postgresql
```

4. The command returns details about the Aurora PostgreSQL DB cluster that's being created for the migration. You can check the status of the Aurora PostgreSQL DB cluster by using the describe-db-clusters AWS CLI command.

   ```
   aws rds describe-db-clusters --db-cluster-identifier cluster-name
   ```

5. When the DB cluster becomes "available", you use create-db-instance command to populate the Aurora PostgreSQL DB cluster with the DB instance based on your Amazon RDS DB snapshot. Specify the following parameters:

   - `--db-cluster-identifier` – The name of the new Aurora PostgreSQL DB cluster that you created in the previous step.
   - `--db-instance-identifier` – The name you want to give to the DB instance. This instance becomes the primary node in your Aurora PostgreSQL DB cluster.
   - `----db-instance-class ` – Specify the DB instance class to use. Choose from among the DB instance classes supported by the Aurora PostgreSQL version to which you're migrating. For more information, see DB instance class types and Supported DB engines for DB instance classes.
   - `--engine` – Specify `aurora-postgresql` for the DB instance.

   You can also create the DB instance with a different configuration than the source DB snapshot, by passing in the appropriate options in the `create-db-instance` AWS CLI command. For more information, see the create-db-instance command.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-instance \
       --db-cluster-identifier cluster-name \
       --db-instance-identifier --db-instance-class db.instance.class \
       --engine aurora-postgresql
   ```

   For Windows:

   ```
   aws rds create-db-instance ^
       --db-cluster-identifier cluster-name ^
       --db-instance-identifier --db-instance-class db.instance.class ^
       --engine aurora-postgresql
   ```

When the migration process completes, the Aurora PostgreSQL cluster has a populated primary DB instance.

Migrating data from an RDS for PostgreSQL DB instance to an Aurora PostgreSQL DB cluster using an Aurora read replica

You can use an RDS for PostgreSQL DB instance as the basis for a new Aurora PostgreSQL DB cluster by using an Aurora read replica for the migration process. The Aurora read replica option is available only for migrating within the same AWS Region and account, and it's available only if the Region offers a compatible version of Aurora PostgreSQL for your RDS for PostgreSQL DB instance. By *compatible*, we mean that the Aurora PostgreSQL version is the same as the RDS for PostgreSQL version, or that it is a higher minor version in the same major version family.

For example, to use this technique to migrate an RDS for PostgreSQL 11.14 DB instance, the Region must offer Aurora PostgreSQL version 11.14 or a higher minor version in the PostgreSQL version 11 family.

Topics

Overview of migrating data by using an Aurora read replica

Migrating from an RDS for PostgreSQL DB instance to an Aurora PostgreSQL DB cluster is a multistep procedure. First, you create an Aurora read replica of your source RDS for PostgreSQL DB instance. That starts a replication process from your RDS for PostgreSQL DB instance to a special-purpose DB cluster known as a Replica *cluster*. The Replica cluster consists solely of an Aurora read replica (a reader instance).

Once the Replica cluster exists, you monitor the lag between it and the source RDS for PostgreSQL DB instance. When the replica lag is zero (0), you can promote the Replica cluster. Replication stops, the Replica cluster is promoted to a standalone Aurora DB cluster, and the reader is promoted to writer instance for the cluster. You can then add instances to the Aurora PostgreSQL DB cluster to size your Aurora PostgreSQL DB cluster for your use case. You can also delete the RDS for PostgreSQL DB instance if you have no further need of it.

> **Note**
>
> It can take several hours per terabyte of data for the migration to complete.

You can't create an Aurora read replica if your RDS for PostgreSQL DB instance already has an Aurora read replica or if it has a cross-Region read replica.

Preparing to migrate data by using an Aurora read replica

During the migration process using Aurora read replica, updates made to the source RDS for PostgreSQL DB instance are asynchronously replicated to the Aurora read replica of the Replica cluster. The process uses PostgreSQL's native streaming replication functionality which stores write-ahead logs (WAL) segments on the source instance. Before starting this migration process, make sure that your instance has sufficient storage capacity by checking values for the metrics listed in the table.

Metric	Description
`FreeStorageSpace`	The available storage space. Units: Bytes
`OldestReplicationSlotLag`	The size of the lag for WAL data in the replica that is lagging the most. Units: Megabytes
`RDSToAuroraPostgreSQLReplicaLag`	The amount of time in seconds that an Aurora PostgreSQL DB cluster lags behind the source RDS DB instance.
`TransactionLogsDiskUsage`	The disk space used by the transaction logs. Units: Megabytes

For more information about monitoring your RDS instance, see Monitoring in the *Amazon RDS User Guide*.

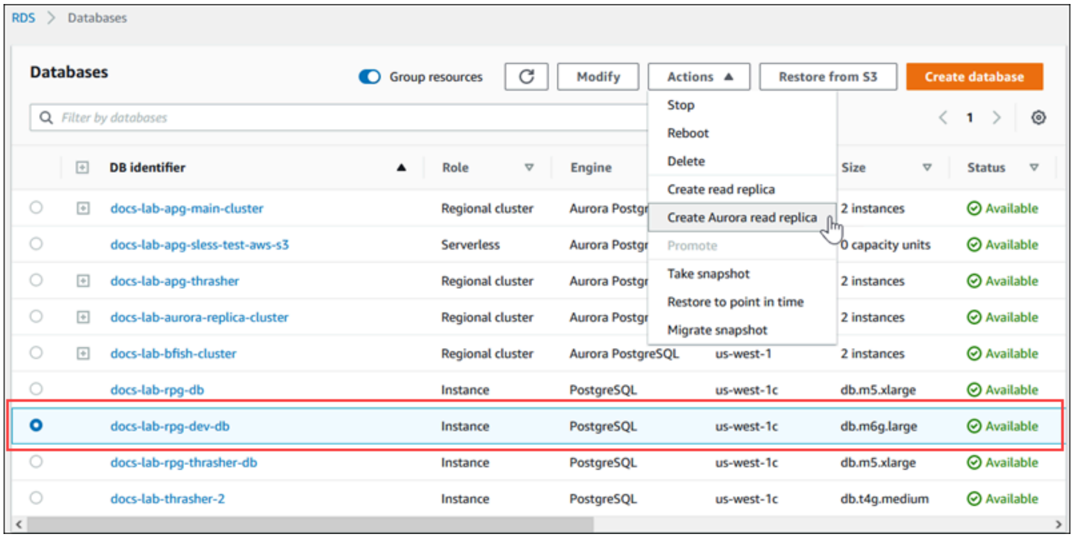
Creating an Aurora read replica

You can create an Aurora read replica for an RDS for PostgreSQL DB instance by using the AWS Management Console or the AWS CLI. The option to create an Aurora read replica using the AWS Management Console is available only if the AWS Region offers a compatible Aurora PostgreSQL version. That is, it's available only if there's an Aurora PostgreSQL version that is the same as the RDS for PostgreSQL version or a higher minor version in the same major version family.

Console

To create an Aurora read replica from a source PostgreSQL DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the RDS for PostgreSQL DB instance that you want to use as the source for your Aurora read replica. For **Actions**, choose **Create Aurora read replica**. If this choice doesn't display, it means that a compatible Aurora PostgreSQL version isn't available in the Region.



	DB identifier ▲	Role	Engine		Size	Status
○	docs-lab-apg-main-cluster	Regional cluster	Aurora Postgr		2 instances	Available
○	docs-lab-apg-sless-test-aws-s3	Serverless	Aurora Postgr		0 capacity units	Available
○	docs-lab-apg-thrasher	Regional cluster	Aurora Postgr		2 instances	Available
○	docs-lab-aurora-replica-cluster	Regional cluster	Aurora Postgr		2 instances	Available
○	docs-lab-bfish-cluster	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances	Available
○	docs-lab-rpg-db	Instance	PostgreSQL	us-west-1c	db.m5.xlarge	Available
●	docs-lab-rpg-dev-db	Instance	PostgreSQL	us-west-1c	db.m6g.large	Available
○	docs-lab-rpg-thrasher-db	Instance	PostgreSQL	us-west-1c	db.m5.xlarge	Available
○	docs-lab-thrasher-2	Instance	PostgreSQL	us-west-1c	db.t4g.medium	Available

4. On the Create Aurora read replica settings page, you configure the properties for the Aurora PostgreSQL DB cluster as shown in the following table. The Replica DB cluster is created from a snapshot of the source DB instance using the same 'master' user name and password as the source, so you can't change these at this time.

Option	Description
DB instance class	Choose a DB instance class that meets the processing and memory requirements primary instance in the DB cluster. For more information, see Aurora DB instance classes.
Multi-AZ deployment	Not available during the migration
DB instance identifier	Enter the name that you want to give to the DB instance. This identifier is used in the endpoint address for the primary instance of the new DB cluster. The DB instance identifier has the following constraints: • It must contain 1–63 alphanumeric characters or hyphens. • Its first character must be a letter. • It can't end with a hyphen or contain two consecutive hyphens. • It must be unique for all DB instances for each AWS account, for each AWS Region.
Virtual Private Cloud (VPC)	Choose the VPC to host the DB cluster. Choose **Create new VPC** to have Amazon RDS create a VPC for you. For more information, see DB cluster prerequisites.
DB subnet group	Choose the DB subnet group to use for the DB cluster. Choose **Create new DB Subnet Group** to have Amazon RDS create a DB subnet group for you. For more information, see DB cluster prerequisites.

Option	Description
Public accessibility	Choose **Yes** to give the DB cluster a public IP address; otherwise, choose **No**. The instances in your DB cluster can be a mix of both public and private DB instances. For more information about hiding instances from public access, see Hiding a DB cluster in a VPC from the internet.
Availability zone	Determine if you want to specify a particular Availabil ity Zone. For more information about Availability Zones, see Regions and Availability Zones.
VPC security groups	Choose one or more VPC security groups to secure network access to the DB cluster. Choose **Create new VPC security group** to have Amazon RDS create a VPC security group for you. For more information, see DB cluster prerequisites.
Database port	Specify the port for applications and utilities to use to access the database. Aurora PostgreSQL DB clusters default to the default PostgreSQL port, 5432. Firewalls at some companies block connections to this port. If your company firewall blocks the default port, choose another port for the new DB cluster.
DB parameter group	Choose a DB parameter group for the Aurora PostgreSQL DB cluster. Aurora has a default DB parameter group you can use, or you can create your own DB parameter group. For more informati on about DB parameter groups, see Working with parameter groups.

Option	Description
DB cluster parameter group	Choose a DB cluster parameter group for the Aurora PostgreSQL DB cluster. Aurora has a default DB cluster parameter group you can use, or you can create your own DB cluster parameter group. For more information about DB cluster parameter groups, see Working with parameter groups.
Encryption	Choose **Enable encryption** for your new Aurora DB cluster to be encrypted at rest. If you choose **Enable encryption**, also choose a KMS key as the **AWS KMS key** value.
Priority	Choose a failover priority for the DB cluster. If you don't choose a value, the default is **tier-1**. This priority determines the order in which Aurora Replicas are promoted when recovering from a primary instance failure. For more information, see Fault tolerance for an Aurora DB cluster.
Backup retention period	Choose the length of time, 1–35 days, for Aurora to retain backup copies of the database. Backup copies can be used for point-in-time restores (PITR) of your database down to the second.
Enhanced monitoring	Choose **Enable enhanced monitoring** to enable gathering metrics in real time for the operating system that your DB cluster runs on. For more information, see Monitoring OS metrics with Enhanced Monitoring.
Monitoring Role	Only available if you chose **Enable enhanced monitoring**. The AWS Identity and Access Management (IAM) role to use for Enhanced Monitoring. For more information, see Setting up and enabling Enhanced Monitoring.

Option	Description
Granularity	Only available if you chose **Enable enhanced monitoring**. Set the interval, in seconds, between when metrics are collected for your DB cluster.
Auto minor version upgrade	Choose **Yes** to enable your Aurora PostgreSQL DB cluster to receive minor PostgreSQL DB engine version upgrades automatically when they become available. The **Auto minor version upgrade** option only applies to upgrades to PostgreSQL minor engine versions for your Aurora PostgreSQL DB cluster. It doesn't apply to regular patches applied to maintain system stability.
Maintenance window	Choose the weekly time range during which system maintenance can occur.

5. Choose **Create read replica**.

AWS CLI

To create an Aurora read replica from a source RDS for PostgreSQL DB instance by using the AWS CLI, you first use the **create-db-cluster** CLI command to create an empty Aurora DB cluster. Once the DB cluster exists, you then use the **create-db-instance** command to create the primary instance for your DB cluster. The primary instance is the first instance that's created in an Aurora DB cluster. In this case, it's created initially as an Aurora read replica of your RDS for PostgreSQL DB instance. When the process concludes, your RDS for PostgreSQL DB instance has effectively been migrated to an Aurora PostgreSQL DB cluster.

You don't need to specify the main user account (typically, `postgres`), its password, or the database name. The Aurora read replica obtains these automatically from the source RDS for PostgreSQL DB instance that you identify when you invoke the AWS CLI commands.

You do need to specify the engine version to use for the Aurora PostgreSQL DB cluster and the DB instance. The version you specify should match the source RDS for PostgreSQL DB instance. If the source RDS for PostgreSQL DB instance is encrypted, you need to also specify encryption for the Aurora PostgreSQL DB cluster primary instance. Migrating an encrypted instance to an unencrypted Aurora DB cluster isn't supported.

The following examples create an Aurora PostgreSQL DB cluster named `my-new-aurora-cluster` that's going to use an unencrypted RDS DB source instance. You first create the Aurora PostgreSQL DB cluster by calling the [`create-db-cluster`]() CLI command. The example shows how to use the optional `--storage-encrypted` parameter to specify that the DB cluster should be encrypted. Because the source DB isn't encrypted, the `--kms-key-id` is used to specify the key to use. For more information about required and optional parameters, see the list following the example.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster  \
    --db-cluster-identifier my-new-aurora-cluster \
    --db-subnet-group-name my-db-subnet
    --vpc-security-group-ids sg-11111111
    --engine aurora-postgresql \
    --engine-version same-as-your-rds-instance-version \
    --replication-source-identifier arn:aws:rds:aws-region:111122223333:db/rpg-source-
db \
    --storage-encrypted \
    --kms-key-id arn:aws:kms:aws-
region:111122223333:key/11111111-2222-3333-444444444444
```

For Windows:

```
aws rds create-db-cluster  ^
    --db-cluster-identifier my-new-aurora-cluster ^
    --db-subnet-group-name my-db-subnet ^
    --vpc-security-group-ids sg-11111111 ^
    --engine aurora-postgresql ^
    --engine-version same-as-your-rds-instance-version ^
    --replication-source-identifier arn:aws:rds:aws-region:111122223333:db/rpg-source-
db ^
    --storage-encrypted ^
    --kms-key-id arn:aws:kms:aws-
region:111122223333:key/11111111-2222-3333-444444444444
```

In the following list you can find more information about some of the options shown in the example. Unless otherwise specified, these parameters are required.

- `--db-cluster-identifier` – You need to give your new Aurora PostgreSQL DB cluster a name.
- `--db-subnet-group-name` – Create your Aurora PostgreSQL DB cluster in the same DB subnet as the source DB instance.
- `--vpc-security-group-ids` – Specify the security group for your Aurora PostgreSQL DB cluster.
- `--engine-version` – Specify the version to use for the Aurora PostgreSQL DB cluster. This should be the same as the version used by your source RDS for PostgreSQL DB instance.
- `--replication-source-identifier` – Identify your RDS for PostgreSQL DB instance using its Amazon Resource Name (ARN). For more information about Amazon RDS ARNs, see Amazon Relational Database Service (Amazon RDS) in the *AWS General Reference*. of your DB cluster.
- `--storage-encrypted` – Optional. Use only when needed to specify encryption as follows:
  - Use this parameter when the source DB instance has encrypted storage. The call to `create-db-cluster` fails if you don't use this parameter with a source DB instance that has encrypted storage. If you want to use a different key for the Aurora PostgreSQL DB cluster than the key used by the source DB instance, you need to also specify the `--kms-key-id`.
  - Use if the source DB instance's storage is unencrypted but you want the Aurora PostgreSQL DB cluster to use encryption. If so, you also need to identify the encryption key to use with the `--kms-key-id` parameter.
- `--kms-key-id` – Optional. When used, you can specify the key to use for storage encryption (`--storage-encrypted`) by using the key's ARN, ID, alias ARN, or its alias name. This parameter is needed only for the following situations:
  - To choose a different key for the Aurora PostgreSQL DB cluster than that used by the source DB instance.
  - To create an encrypted cluster from an unencrypted source. In this case, you need to specify the key that Aurora PostgreSQL should use for encryption.

After creating the Aurora PostgreSQL DB cluster, you then create the primary instance by using the `create-db-instance` CLI command, as shown in the following:

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
    --db-cluster-identifier my-new-aurora-cluster \
    --db-instance-class db.x2g.16xlarge \
    --db-instance-identifier rpg-for-migration \
    --engine aurora-postgresql
```

For Windows:

```
aws rds create-db-instance ^
    --db-cluster-identifier my-new-aurora-cluster ^
    --db-instance-class db.x2g.16xlarge ^
    --db-instance-identifier rpg-for-migration ^
    --engine aurora-postgresql
```

In the following list, you can find more information about some of the options shown in the example.

- `--db-cluster-identifier` – Specify the name of the Aurora PostgreSQL DB cluster that you created with the `create-db-instance` command in the previous steps.
- `--db-instance-class` – The name of the DB instance class to use for your primary instance, such as `db.r4.xlarge`, `db.t4g.medium`, `db.x2g.16xlarge`, and so on. For a list of available DB instance classes, see DB instance class types.
- `--db-instance-identifier` – Specify the name to give your primary instance.
- `--engine aurora-postgresql` – Specify `aurora-postgresql` for the engine.

RDS API

To create an Aurora read replica from a source RDS for PostgreSQL DB instance, first use the RDS API operation `CreateDBCluster` to create a new Aurora DB cluster for the Aurora read replica that gets created from your source RDS for PostgreSQL DB instance. When the Aurora PostgreSQL DB cluster is available, you then use the `CreateDBInstance` to create the primary instance for the Aurora DB cluster.

You don't need to specify the main user account (typically, `postgres`), its password, or the database name. The Aurora read replica obtains these automatically from the source RDS for PostgreSQL DB instance specified with `ReplicationSourceIdentifier`.

You do need to specify the engine version to use for the Aurora PostgreSQL DB cluster and the DB instance. The version you specify should match the source RDS for PostgreSQL DB instance. If the source RDS for PostgreSQL DB instance is encrypted, you need to also specify encryption for the Aurora PostgreSQL DB cluster primary instance. Migrating an encrypted instance to an unencrypted Aurora DB cluster isn't supported.

To create the Aurora DB cluster for the Aurora read replica, use the RDS API operation `CreateDBCluster` with the following parameters:

- `DBClusterIdentifier` – The name of the DB cluster to create.
- `DBSubnetGroupName` – The name of the DB subnet group to associate with this DB cluster.
- `Engine=aurora-postgresql` – The name of the engine to use.
- `ReplicationSourceIdentifier` – The Amazon Resource Name (ARN) for the source PostgreSQL DB instance. For more information about Amazon RDS ARNs, see Amazon Relational Database Service (Amazon RDS) in the *Amazon Web Services General Reference.* If `ReplicationSourceIdentifier` identifies an encrypted source, Amazon RDS uses your default KMS key unless you specify a different key using the `KmsKeyId` option.
- `VpcSecurityGroupIds` – The list of Amazon EC2 VPC security groups to associate with this DB cluster.
- `StorageEncrypted` – Indicates that the DB cluster is encrypted. When you use this parameter without also specifying the `ReplicationSourceIdentifier`, Amazon RDS uses your default KMS key.
- `KmsKeyId` – The key for an encrypted cluster. When used, you can specify the key to use for storage encryption by using the key's ARN, ID, alias ARN, or its alias name.

For more information, see `CreateDBCluster` in the *Amazon RDS API Reference*.

Once the Aurora DB cluster is available, you can then create a primary instance for it by using the RDS API operation `CreateDBInstance` with the following parameters:

- `DBClusterIdentifier` – The name of your DB cluster.
- `DBInstanceClass` – The name of the DB instance class to use for your primary instance.
- `DBInstanceIdentifier` – The name of your primary instance.
- `Engine=aurora-postgresql` – The name of the engine to use.

For more information, see `CreateDBInstance` in the *Amazon RDS API Reference*.

Promoting an Aurora read replica

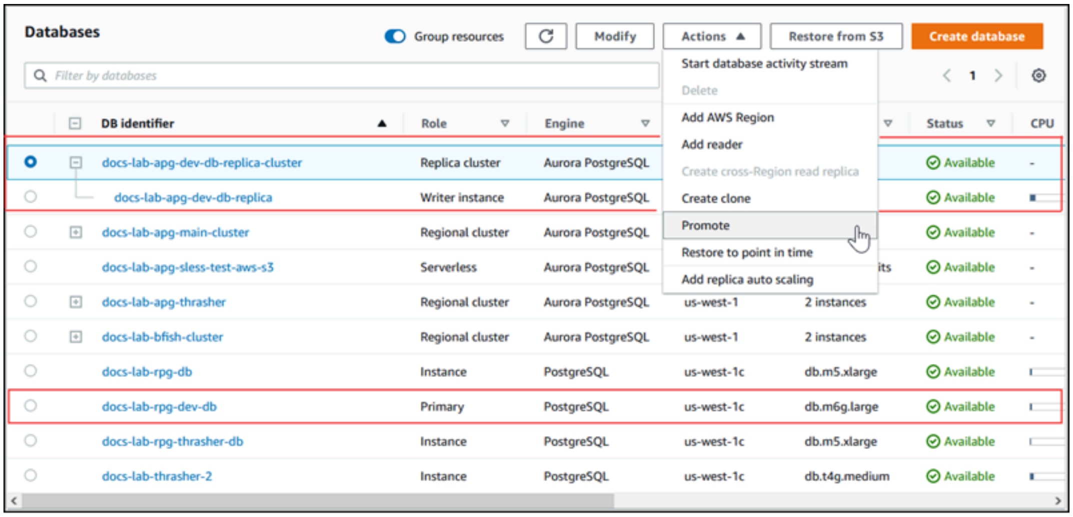
The migration to Aurora PostgreSQL isn't complete until you promote the Replica cluster, so don't delete the RDS for PostgreSQL source DB instance just yet.

Before promoting the Replica cluster, make sure that the RDS for PostgreSQL DB instance doesn't have any in-process transactions or other activity writing to the database. When the replica lag on the Aurora read replica reaches zero (0), you can promote the Replica cluster. For more information about monitoring replica lag, see Monitoring Aurora PostgreSQL replication and Instance-level metrics for Amazon Aurora.

Console

To promote an Aurora read replica to an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Replica cluster.



DB identifier	Role	Engine			Status	CPU
docs-lab-apg-dev-db-replica-cluster	Replica cluster	Aurora PostgreSQL			Available	-
docs-lab-apg-dev-db-replica	Writer instance	Aurora PostgreSQL			Available	
docs-lab-apg-main-cluster	Regional cluster	Aurora PostgreSQL			Available	-
docs-lab-apg-sless-test-aws-s3	Serverless	Aurora PostgreSQL		its	Available	-
docs-lab-apg-thrasher	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances	Available	-
docs-lab-bfish-cluster	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances	Available	-
docs-lab-rpg-db	Instance	PostgreSQL	us-west-1c	db.m5.xlarge	Available	
docs-lab-rpg-dev-db	Primary	PostgreSQL	us-west-1c	db.m6g.large	Available	
docs-lab-rpg-thrasher-db	Instance	PostgreSQL	us-west-1c	db.m5.xlarge	Available	
docs-lab-thrasher-2	Instance	PostgreSQL	us-west-1c	db.t4g.medium	Available	

4. For **Actions**, choose **Promote**. This may take a few minutes and can cause downtime.

When the process completes, the Aurora Replica cluster is a Regional Aurora PostgreSQL DB cluster, with a Writer instance containing the data from the RDS for PostgreSQL DB instance.

AWS CLI

To promote an Aurora read replica to a stand-alone DB cluster, use the **promote-read-replica-db-cluster** AWS CLI command.

Example

For Linux, macOS, or Unix:

```
aws rds promote-read-replica-db-cluster \
    --db-cluster-identifier myreadreplicacluster
```

For Windows:

```
aws rds promote-read-replica-db-cluster ^
    --db-cluster-identifier myreadreplicacluster
```

RDS API

To promote an Aurora read replica to a stand-alone DB cluster, use the RDS API operation PromoteReadReplicaDBCluster.

After you promote the Replica cluster, you can confirm that the promotion has completed by checking the event log, as follows.

To confirm that the Aurora Replica cluster was promoted

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Events**.
3. On the **Events** page, find the name of your cluster in the **Source** list. Each event has a source, type, time, and message. You can see all events that have occurred in your AWS Region for your account. A successful promotion generates the following message.

   ```
   Promoted Read Replica cluster to a stand-alone database cluster.
   ```

After promotion is complete, the source RDS for PostgreSQL DB instance and the Aurora PostgreSQL DB cluster are unlinked. You can direct your client applications to the endpoint for the

Aurora read replica. For more information on the Aurora endpoints, see Amazon Aurora connection management. At this point, you can safely delete the DB instance.

Improving query performance for Aurora PostgreSQL with Aurora Optimized Reads

You can achieve faster query processing for Aurora PostgreSQL with Aurora Optimized Reads. An Aurora PostgreSQL DB instance that uses Aurora Optimized Reads delivers up to 8x improved query latency and up to 30% cost savings for applications with large datasets, that exceed the memory capacity of a DB instance.

Topics

Overview of Aurora Optimized Reads in PostgreSQL

Aurora Optimized Reads is available by default when you create a DB cluster that uses Graviton-based R6gd and Intel-based R6id instances with non-volatile memory express (NVMe) storage. It is available from the following PostgreSQL versions:

- 16.1 and all higher versions
- 15.4 and higher versions
- 14.9 and higher versions

Aurora Optimized Reads supports two capabilities: tiered cache and temporary objects.

Optimized Reads-enabled tiered cache - Using tiered cache, you can extend your DB instance caching capacity by up to 5x the instance memory. This automatically maintains the cache to contain the most recent, transactionally consistent data, freeing applications from the overhead of managing the data currency of external result-set based caching solutions. It offers up to 8x better latency for queries that were previously fetching data from Aurora storage.

In Aurora, the value for `shared_buffers` in the default parameter group is usually set to around 75% of the available memory. However, for the r6gd and r6id instance types, Aurora will reduce the `shared_buffers` space by 4.5% to host the metadata for the Optimized Reads cache.

Optimized Reads-enabled temporary objects - Using temporary objects, you can achieve faster query processing by placing the temporary files that are generated by PostgreSQL on the local NVMe storage. This reduces the traffic to Elastic Block Storage (EBS) over the network. It offers up to 2x better latency and throughput for advanced queries that sort, join, or merge large volumes of data that do not fit within the memory capacity available on a DB instance.

On an Aurora I/O-Optimized cluster, Optimized Reads makes use of both tiered cache and temporary objects on NVMe storage. With Optimized Reads-enabled tiered cache capability, Aurora allocates 2x the instance memory for temporary objects, approximately 10% of the storage for internal operations and the remaining storage as tiered cache. On an Aurora Standard cluster, Optimized Reads makes use of only temporary objects.

Engine	Cluster storage configuration	Optimized Reads-enabled temporary objects	Optimized Reads-enabled tiered cache	Versions supported
Aurora PostgreSQL-Compatible Edition	Standard	Yes	No	Aurora PostgreSQL version 16.1 and all higher versions, 15.4 and higher, version 14.9 and higher
	I/O-Optimized	Yes	Yes	

Note

A switch between IO-Optimized and Standard clusters on a NVMe-based DB instance class causes an immediate database engine restart.

In Aurora PostgreSQL, use the `temp_tablespaces` parameter to configure the table space where the temporary objects are stored.

To check whether the temporary objects are configured, use the following command:

```
postgres=> show temp_tablespaces;
temp_tablespaces
---------------------
aurora_temp_tablespace
(1 row)
```

The `aurora_temp_tablespace` is a tablespace configured by Aurora that points to the NVMe local storage. You can't modify this parameter or switch back to Amazon EBS storage.

To check whether optimized reads cache is turned on, use the following command:

```
postgres=> show shared_preload_libraries;
                 shared_preload_libraries
--------------------------------------------------------
rdsutils,pg_stat_statements,aurora_optimized_reads_cache
```

Using Aurora Optimized Reads

When you provision an Aurora PostgreSQL DB instance with one of the NVMe-based DB instances, the DB instance automatically uses Aurora Optimized Reads.

To turn on Aurora Optimized Reads, do one of the following:

- Create an Aurora PostgreSQL DB cluster using one of the NVMe-based DB instance classes. For more information, see Creating an Amazon Aurora DB cluster.
- Modify an existing Aurora PostgreSQL DB cluster to use one of the NVMe-based DB instance classes. For more information, see Modifying an Amazon Aurora DB cluster.

Aurora Optimized Reads is available in all AWS Regions where one or more of the DB instance classes with local NVMe SSD storage are supported. For more information, see Aurora DB instance classes.

To switch back to a non-optimized reads Aurora instance, modify the DB instance class of your Aurora instance to the similar instance class without NVMe ephemeral storage for your database workloads. For example, if the current DB instance class is db.r6gd.4xlarge, choose db.r6g.4xlarge to switch back. For more information, see Modifying an Aurora DB instance.

Use cases for Aurora Optimized Reads

Optimized Reads-enabled tiered cache

The following are some use cases that can benefit from Optimized Reads with tiered cache:

- Internet scale applications such as payments processing, billing, e-commerce with strict performance SLAs.
- Real-time reporting dashboards that run hundreds of point queries for metrics/data collection.
- Generative AI applications with the pgvector extension to search exact or nearest neighbors across millions of vector embeddings.

Optimized Reads-enabled temporary objects

The following are some use cases that can benefit from Optimized Reads with temporary objects:

- Analytical queries that include Common Table Expressions (CTEs), derived tables, and grouping operations.
- Read replicas that handle the unoptimized queries for an application.
- On-demand or dynamic reporting queries with complex operations such as GROUP BY and ORDER BY that can't always use appropriate indexes.
- `CREATE INDEX` or `REINDEX` operations for sorting.
- Other workloads that use internal temporary tables.

Monitoring DB instances that use Aurora Optimized Reads

You can monitor your queries that use Optimized Reads-enabled tiered cache with the EXPLAIN command as shown in the following example:

```
Postgres=> EXPLAIN (ANALYZE, BUFFERS) SELECT c FROM sbtest15 WHERE id=100000000


QUERY PLAN
--------------------------------------------------------------------------------------
```

```
 Index Scan using sbtest15_pkey on sbtest15  (cost=0.57..8.59 rows=1 width=121) (actual
 time=0.287..0.288 rows=1 loops=1)
   Index Cond: (id = 100000000)
   Buffers: shared hit=3 read=2 aurora_orcache_hit=2
   I/O Timings: shared/local read=0.264
 Planning:
   Buffers: shared hit=33 read=6 aurora_orcache_hit=6
   I/O Timings: shared/local read=0.607
 Planning Time: 0.929 ms
 Execution Time: 0.303 ms
(9 rows)
Time: 2.028 ms
```

Note

aurora_orcache_hit and aurora_storage_read fields in the Buffers section of the explain plan are shown only when Optimized Reads is turned on and their values are greater than zero. The read field is the total of the aurora_orcache_hit and aurora_storage_read fields.

You can monitor DB instances that use Aurora Optimized Reads using the following CloudWatch metrics:

- AuroraOptimizedReadsCacheHitRatio
- FreeEphemeralStorage
- ReadIOPSEphemeralStorage
- ReadLatencyEphemeralStorage
- ReadThroughputEphemeralStorage
- WriteIOPSEphemeralStorage
- WriteLatencyEphemeralStorage
- WriteThroughputEphemeralStorage

These metrics provide data about available instance store storage, IOPS, and throughput. For more information about these metrics, see Instance-level metrics for Amazon Aurora.

You can also use the `pg_proctab` extension to monitor NVMe storage.

```
postgres=>select * from pg_diskusage();

major | minor |       devname       | reads_completed | reads_merged | sectors_read |
 readtime | writes_completed | writes_merged | sectors_written | writetime | current_io
 | iotime   | totaliotime
------+-------+---------------------+-----------------+--------------+--------------
+----------+------------------+---------------+-----------------+-----------
+------------+---------+------------
      |       | rdstemp             |           23264 |            0 |       191450 |
  11670 |          1750892 |             0 |         24540576 |    819350 |           0 |
 3847580 |      831020
      |       | rdsephemeralstorage |           23271 |            0 |       193098 |
   2620 |           114961 |             0 |         13845120 |    130770 |           0 |
  215010 |      133410
(2 rows)
```

Best practices for Aurora Optimized Reads

Use the following best practices for Aurora Optimized Reads:

- Monitor the storage space available on the instance store with the CloudWatch metric `FreeEphemeralStorage`. If the instance store is reaching its limit because of the workload on the DB instance, tune the concurrency and queries which heavily use temporary objects or modify it to use a larger DB instance class.
- Monitor the CloudWatch metric for the Optimized Reads cache hit rate. Operations like VACUUM modify large numbers of blocks very quickly. This can cause a temporary drop in the hit ratio. The `pg_prewarm` extension can be used to load data into the buffer cache that allows Aurora to proactively write some of those blocks to the Optimized Reads cache.
- You can enable cluster cache management (CCM) to warm up the buffer cache and tiered cache on a tier-0 reader, which will be used as a failover target. When CCM is enabled, the buffer cache is periodically scanned to write pages eligible for eviction in tiered cache. For more information on CCM, see Fast recovery after failover with cluster cache management for Aurora PostgreSQL.

Using Babelfish for Aurora PostgreSQL

Babelfish for Aurora PostgreSQL extends your Aurora PostgreSQL DB cluster with the ability to accept database connections from SQL Server clients. With Babelfish, applications that were originally built for SQL Server can work directly with Aurora PostgreSQL with few code changes compared to a traditional migration and without changing database drivers. For more information about migrating, see Migrating a SQL Server database to Babelfish for Aurora PostgreSQL.

Babelfish provides an additional endpoint for an Aurora PostgreSQL database cluster that allows it to understand the SQL Server wire-level protocol and commonly used SQL Server statements. Client applications that use the Tabular Data Stream (TDS) wire protocol can connect natively to the TDS listener port on Aurora PostgreSQL. To learn more about TDS, see [MS-TDS]: Tabular Data Stream Protocol on the Microsoft website.

Note

Babelfish for Aurora PostgreSQL supports TDS versions 7.1 through 7.4.

Babelfish also provides access to data using the PostgreSQL connection. By default, both SQL dialects supported by Babelfish are available through their native wire protocols at the following ports:

- SQL Server dialect (T-SQL), clients connect to port 1433.
- PostgreSQL dialect (PL/pgSQL), clients connect to port 5432.

Babelfish runs the Transact-SQL (T-SQL) language with some differences. For more information, see Differences between Babelfish for Aurora PostgreSQL and SQL Server.

In the following sections, you can find information about setting up and using a Babelfish for Aurora PostgreSQL DB cluster.

Topics

- Database authentication with Babelfish for Aurora PostgreSQL
- Connecting to a Babelfish DB cluster
- Working with Babelfish
- Troubleshooting Babelfish
- Turning off Babelfish
- Babelfish version updates
- Babelfish for Aurora PostgreSQL reference

Babelfish limitations

The following limitations currently apply to Babelfish for Aurora PostgreSQL:

- Babelfish currently doesn't support the following Aurora features:
  - Amazon RDS Blue/Green Deployments
  - AWS Identity and Access Management
  - Database Activity Streams (DAS)
  - PostgreSQL logical replication
  - RDS Data API with Aurora PostgreSQL Serverless v2 and provisioned
  - RDS Proxy with RDS for SQL Server
  - Salted challenge response authentication mechanism (SCRAM)
  - Query editor
- Babelfish currently doesn't support Kerberos based authentication for Active Directory groups.
- Babelfish doesn't provide the following client driver API support:
  - API requests with the connection attributes related to Microsoft Distributed Transaction Coordinator (MSDTC) aren't supported. These include XA calls by the SQLServerXAResource class in the SQL server JDBC driver.
  - Babelfish supports connection pooling with drivers that use the latest versions of the TDS protocol. With older drivers, API requests with the connection attributes and methods related to connection pooling aren't supported.
- Babelfish currently doesn't support the following Aurora PostgreSQL extensions:
  - `bloom`
  - `btree_gin`
  - `btree_gist`
  - `citext`
  - `cube`
  - `hstore`
  - `hypopg`
  - Logical replication using `pglogical`
  - `ltree`
  - `pgcrypto`

  - Query plan management using `apg_plan_mgmt`

  To learn more about PostgreSQL extensions, see Working with extensions and foreign data wrappers.

- The open source jTDS driver that is designed as an alternative to the Microsoft JDBC driver is not supported.

Understanding Babelfish architecture and configuration

You manage the Aurora PostgreSQL-Compatible Edition DB cluster running Babelfish much as you would any Aurora DB cluster. That is, you benefit from the scalability, high-availability with failover support, and built-in replication provided by an Aurora DB cluster. To learn more about these capabilities, see Managing performance and scaling for Aurora DB clusters, High availability for Amazon Aurora, and Replication with Amazon Aurora. You also have access to many other AWS tools and utilities, including the following:

- Amazon CloudWatch is a monitoring and observability service that provides you with data and actionable insights. For more information, see Monitoring Amazon Aurora metrics with Amazon CloudWatch.
- Performance Insights is a database performance tuning and monitoring feature that helps you quickly assess the load on your database. To learn more, see Monitoring DB load with Performance Insights on Amazon Aurora.
- Aurora global databases span multiple AWS Regions, enabling low latency global reads and providing fast recovery from the rare outage that might affect an entire AWS Region. For more information, see Using Amazon Aurora global databases.
- Automatic software patching keeps your database up-to-date with the latest security and feature patches when they become available.
- Amazon RDS events notify you by email or SMS message of important database events, such as an automated failover. For more information, see Monitoring Amazon Aurora events.

Following, you can learn about Babelfish architecture and how the SQL Server databases that you migrate are handled by Babelfish. When you create your Babelfish DB cluster, you need to make some decisions up front about single database or multiple databases, collations, and other details.

Topics

- DB cluster parameter group settings for Babelfish
- Collations supported by Babelfish
- Managing Babelfish error handling with escape hatches

Babelfish architecture

When you create an Aurora PostgreSQL cluster with Babelfish turned on, Aurora provisions the cluster with a PostgreSQL database named `babelfish_db`. This database is where all migrated SQL Server objects and structures reside.

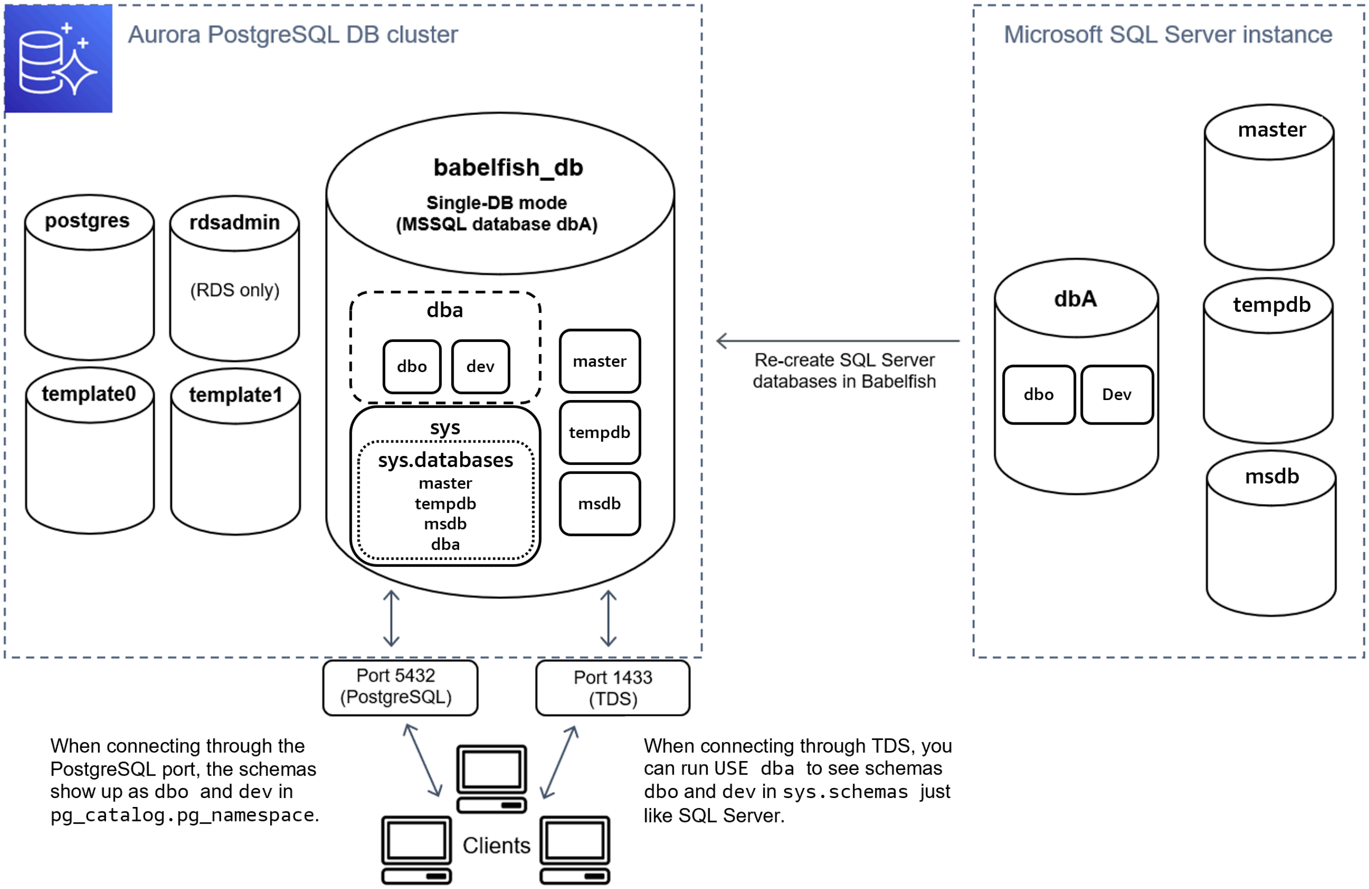
> **Note**
>
> In an Aurora PostgreSQL cluster, the `babelfish_db` database name is reserved for Babelfish. Creating your own "babelfish_db" database on a Babelfish DB cluster prevents Aurora from successfully provisioning Babelfish.

When you connect to the TDS port, the session is placed in the `babelfish_db` database. From T-SQL, the structure looks similar to being connected to a SQL Server instance. You can see the `master`, `msdb`, and `tempdb` databases, and the `sys.databases` catalog. You can create additional user databases and switch between databases with the USE statement. When you create a SQL Server user database, it's flattened into the `babelfish_db` PostgreSQL database. Your database retains cross-database syntax and semantics equal to or similar to those provided by SQL Server.

Using Babelfish with a single database or multiple databases

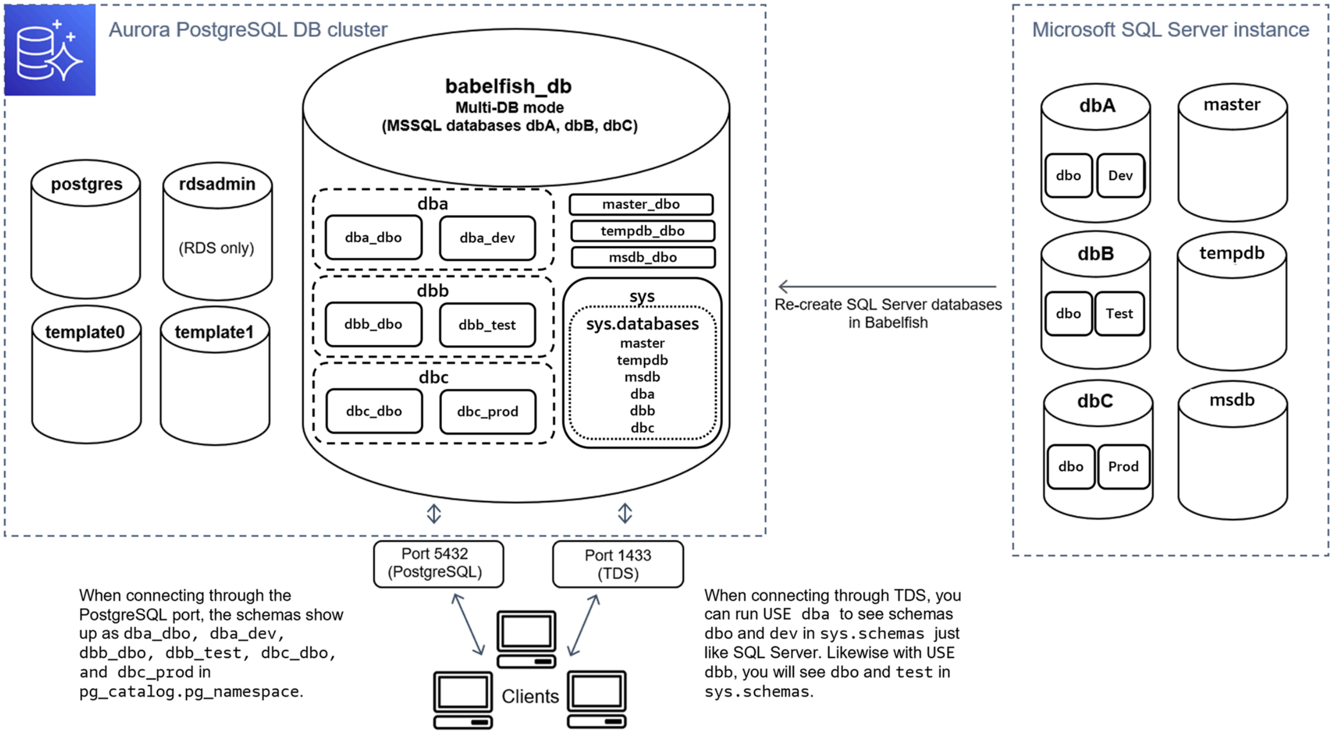
When you create an Aurora PostgreSQL cluster to use with Babelfish, you choose between using a single SQL Server database on its own or multiple SQL Server databases together. Your choice affects how the names of SQL Server schemas inside the `babelfish_db` database appear from Aurora PostgreSQL. The migration mode is stored in the `migration_mode` parameter. You must not change this parameter after creating your cluster as you could lose access to all your previously created SQL objects.

In single-db mode, the schema names of the SQL Server database remain the same in the `babelfish_db` database of the PostgreSQL. If you choose to migrate only a single database, the schema names of the migrated user database can be referenced in PostgreSQL with the same names used in SQL Server. For example, the `dbo` and `smith` schemas reside inside the `dbA` database.


Aurora PostgreSQL DB cluster
postgres
rdsadmin
(RDS only)
template0
template1
babelfish_db
Single-DB mode
(MSSQL database dbA)
dba
dbo
dev
master
tempdb
msdb
sys
sys.databases
master
tempdb
msdb
dba
Microsoft SQL Server instance
master
dbA
dbo
Dev
tempdb
msdb
Re-create SQL Server databases in Babelfish
Port 5432 (PostgreSQL)
Port 1433 (TDS)
Clients
When connecting through the PostgreSQL port, the schemas show up as dbo and dev in pg_catalog.pg_namespace.
When connecting through TDS, you can run USE dba to see schemas dbo and dev in sys.schemas just like SQL Server.

When connecting through TDS, you can run `USE  dba` to see schemas `dbo` and `dev` from T-SQL, as you would in SQL Server. The unchanged schema names are also visible from PostgreSQL.

In multiple-database mode, the schema names of user databases become `dbname_schemaname` when accessed from PostgreSQL. The schema names remain the same when accessed from T-SQL.



When connecting through the PostgreSQL port, the schemas show up as dba_dbo, dba_dev, dbb_dbo, dbb_test, dbc_dbo, and dbc_prod in pg_catalog.pg_namespace.

When connecting through TDS, you can run USE dba to see schemas dbo and dev in sys.schemas just like SQL Server. Likewise with USE dbb, you will see dbo and test in sys.schemas.

As shown in the image, multiple-database mode and single-database mode are the same as SQL Server when connecting through the TDS port and using T-SQL. For example, USE dbA lists schemas dbo and dev just as it does in SQL Server. The mapped schema names, such as dba_dbo and dba_dev, are visible from PostgreSQL.

Each database still contains your schemas. The name of each database is prepended to the name of the SQL Server schema, using an underscore as a delimiter, for example:

- dba contains dba_dbo and dba_dev.
- dbb contains dbb_dbo and dbb_test.
- dbc contains dbc_dbo and dbc_prod.

Inside the babelfish_db database, the T-SQL user still needs to run USE dbname to change database context, so the look and feel remains similar to SQL Server.

Choosing a migration mode

Each migration mode has advantages and disadvantages. Choose your migration mode based on the number of user databases you have, and your migration plans. After you create a cluster for use with Babelfish, you must not change the migration mode as you might lose access to all your previously created SQL objects. When choosing a migration mode, consider the requirements of your user databases and clients.

When you create a cluster for use with Babelfish, Aurora PostgreSQL creates the system databases, `master` and `tempdb`. If you created or modified objects in the system databases (`master` or `tempdb`), make sure to recreate those objects in your new cluster. Unlike SQL Server, Babelfish doesn't reinitialize `tempdb` after a cluster reboot.

Use single database migration mode in the following cases:

- If you are migrating a single SQL Server database. In single database mode, migrated schema names when accessed from PostgreSQL are identical to those in original SQL Server schema names. This reduces code changes to existing SQL queries if you want to optimize them to run with a PostgreSQL connection.
- If your end goal is a complete migration to native Aurora PostgreSQL. Before migrating, consolidate your schemas into a single schema (dbo) and then migrate into a single cluster to lessen required changes.

Use multiple database migration mode in the following cases:

- If you want the default SQL Server experience with multiple user databases in the same instance.
- If multiple user databases need to be migrated together.

DB cluster parameter group settings for Babelfish

When you create an Aurora PostgreSQL DB cluster and choose **Turn on Babelfish**, a DB cluster parameter group is created for you automatically if you choose **Create new**. This DB cluster parameter group is based on the Aurora PostgreSQL DB cluster parameter group for the Aurora PostgreSQL version chosen for the install, for example, Aurora PostgreSQL version 14. It's named using the following general pattern:

```
custom-aurora-postgresql14-babelfish-compat-3
```

You can change the following settings during the cluster creation process but some of these can't be changed once they're stored in the custom parameter group, so choose carefully:

- Single database or Multiple databases
- Default collation locale
- Collation name
- DB parameter group

To use an existing Aurora PostgreSQL DB cluster version 13 or higher parameter group, edit the group and set the `babelfish_status` parameter to on. Specify any Babelfish options before creating your Aurora PostgreSQL cluster. To learn more, see Working with parameter groups.

The following parameters control Babelfish preferences. Unless otherwise stated in the Description, parameters are modifiable. The default value is included in the description. To see the allowable values for any parameter, do as follows:

> **Note**
>
> When you associate a new DB parameter group with a DB instance, the modified static and dynamic parameters are applied only after the DB instance is rebooted. However, if you modify dynamic parameters in the DB parameter group after you associate it with the DB instance, these changes are applied immediately without a reboot.

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Parameter groups** from the navigation menu.

3. Choose the `default.aurora-postgresql14` DB cluster parameter group from the list.
4. Enter the name of a parameter in the search field. For example, enter `babelfishpg_tsql.default_locale` in the search field to display this parameter and its default value and allowable settings.

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tds.tds_default_numeric_scale	Sets the default scale of numeric type to be sent in the TDS column metadata if the engine doesn't specify one. (Default: 8) (Allowable: 0–38)	dynamic	true
babelfishpg_tds.tds_default_numeric_precision	An integer that sets the default precision of numeric type to be sent in the TDS column metadata if the engine doesn't specify one. (Default: 38) (Allowable: 1–38)	dynamic	true
babelfishpg_tds.tds_default_packet_size	An integer that sets the default packet size for connecting SQL Server clients. (Default: 4096) (Allowable: 512–32767)	dynamic	true
babelfishpg_tds.tds_default_protocol_version	An integer that sets a default TDS protocol version for connecting clients. (Default: DEFAULT) (Allowabl	dynamic	true

Parameter	Description	Apply Type	Is Modifiable
	e: TDSv7.0, TDSv7.1, TDSv7.1.1, TDSv7.2, TDSv7.3A, TDSv7.3B, TDSv7.4, DEFAULT)		
babelfishpg_tds.default_server_name	A string that identifies the default name of the Babelfish server. (Default: Microsoft SQL Server) (Allowable: null)	dynamic	true
babelfishpg_tds.tds_debug_log_level	An integer that sets the logging level in TDS; 0 turns off logging. (Default: 1) (Allowable: 0, 1, 2, 3)	dynamic	true
babelfishpg_tds.listen_addresses	A string that sets the host name or IP address or addresses to listen for TDS on. This parameter can't be modified after the Babelfish DB cluster is created. (Default: *) (Allowable: null)	–	false
babelfishpg_tds.port	An integer that specifies the TCP port used for requests in SQL Server syntax. (Default: 1433) (Allowable: 1–65535)	static	true

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tds.tds_ssl_encrypt	A boolean that turns encryption on (0) or off (1) for data traversing the TDS listener port. For detailed information about using SSL for client connections, see Babelfish SSL settings and client connections. (Default: 0) (Allowable: 0, 1)	dynamic	true
babelfishpg_tds.tds_ssl_max_protocol_version	A string that specifies the highest SSL/TLS protocol version to use for the TDS session. (Default: 'TLSv1.2') (Allowable: 'TLSv1', 'TLSv1.1', 'TLSv1.2')	dynamic	true
babelfishpg_tds.tds_ssl_min_protocol_version	A string that specifies the minimum SSL/TLS protocol version to use for the TDS session. (Default: 'TLSv1.2' from Aurora PostgreSQL version 16, 'TLSv1' for versions older than Aurora PostgreSQL version 16) (Allowable: 'TLSv1', 'TLSv1.1', 'TLSv1.2')	dynamic	true

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tds.unix_socket_directories	A string that identifies the TDS server Unix socket directory. This parameter can't be modified after the Babelfish DB cluster is created. (Default: /tmp) (Allowable: null)	–	false
babelfishpg_tds.unix_socket_group	A string that identifies the TDS server Unix socket group. This parameter can't be modified after the Babelfish DB cluster is created. (Default: rdsdb) (Allowable: null)	–	false

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tsql.default_locale	A string that specifies the default locale used for Babelfish collations. The default locale is the locale only and doesn't include any qualifiers. Set this parameter when you provision a Babelfish DB cluster. After the DB cluster is provisioned, changes to this parameter are ignored. (Default: en_US) (Allowable: See tables)	static	true

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tsql.migration_mode	A non-modifiable list that specifies support for single- or multiple user databases. Set this parameter when you provision a Babelfish DB cluster. After the DB cluster is provisioned, you can't modify this parameter's value. (Default: multi-db from Aurora PostgreSQL version 16, single-db for versions older than Aurora PostgreSQL version 16) (Allowable: single-db, multi-db,null)	static	true

Parameter	Description	Apply Type	Is Modifiable
babelfishpg_tsql.server_collation_name	A string that specifies the name of the collation used for server-level actions. Set this parameter when you provision a Babelfish DB cluster. After the DB cluster is provisioned, don't modify the value of this parameter. (Default: bbf_unicode_general_ci_as) (Allowable: See tables)	static	true
babelfishpg_tsql.version	A string that sets the output of @@VERSION variable. Don't modify this value for Aurora PostgreSQL DB clusters. (Default: null) (Allowable: default)	dynamic	true

Parameter	Description	Apply Type	Is Modifiable
rds.babelfish_status	A string that sets the state of Babelfish functionality. When this parameter is set to `datatypes only` , Babelfish is turned off but SQL Server data types are still available . (Default: off) (Allowable: on, off, datatypesonly)	static	true
unix_socket_permissions	An integer that sets the TDS server Unix socket permissions. This parameter can't be modified after the Babelfish DB cluster is created. (Default: 0700) (Allowable: 0–511)	–	false

Babelfish SSL settings and client connections

When a client connects to the TDS port (default `1433`), Babelfish compares the Secure Sockets Layer (SSL) setting sent during the client handshake to the Babelfish SSL parameter setting (`tds_ssl_encrypt`). Babelfish then determines if a connection is allowed. If a connection is allowed, encryption behavior is either enforced or not, depending on your parameter settings and the support for encryption offered by the client.

The table following shows how Babelfish behaves for each combination.

Client SSL setting	Babelfish SSL setting	Connection allowed?	Value returned to client
ENCRYPT_OFF	tds_ssl_encrypt=0	Allowed, the login packet is encrypted	ENCRYPT_OFF
ENCRYPT_OFF	tds_ssl_encrypt=1	Allowed, the entire connection is encrypted	ENCRYPT_REQ
ENCRYPT_ON	tds_ssl_encrypt=0	Allowed, the entire connection is encrypted	ENCRYPT_ON
ENCRYPT_ON	tds_ssl_encrypt=1	Allowed, the entire connection is encrypted	ENCRYPT_ON
ENCRYPT_NOT_SUP	tds_ssl_encrypt=0	Yes	ENCRYPT_NOT_SUP

Client SSL setting	Babelfish SSL setting	Connection allowed?	Value returned to client
ENCRYPT_NOT_SUP	tds_ssl_encrypt=1	No, connection closed	ENCRYPT_REQ
ENCRYPT_REQ	tds_ssl_encrypt=0	Allowed, the entire connection is encrypted	ENCRYPT_ON
ENCRYPT_REQ	tds_ssl_encrypt=1	Allowed, the entire connection is encrypted	ENCRYPT_ON
ENCRYPT_CLIENT_CERT	tds_ssl_encrypt=0	No, connection closed	Unsupported
ENCRYPT_CLIENT_CERT	tds_ssl_encrypt=1	No, connection closed	Unsupported

Collations supported by Babelfish

When you create an Aurora PostgreSQL DB cluster with Babelfish, you choose a collation for your data. A *collation* specifies the sort order and bit patterns that produce the text or characters in a given written human language. A collation includes rules comparing data for a given set of bit patterns. Collation is related to localization. Different locales affect character mapping, sort order, and the like. Collation attributes are reflected in the names of various collations. For information about attributes, see the Babelfish collation attributes table.

Babelfish maps SQL Server collations to comparable collations provided by Babelfish. Babelfish predefines Unicode collations with culturally sensitive string comparisons and sort orders. Babelfish also provides a way to translate the collations in your SQL Server DB to the closest-matching Babelfish collation. Locale-specific collations are provided for different languages and regions.

Some collations specify a code page that corresponds to a client-side encoding. Babelfish automatically translates from the server encoding to the client encoding depending on the collation of each output column.

Babelfish supports the collations listed in the Babelfish supported collations table. Babelfish maps SQL Server collations to comparable collations provided by Babelfish.

Babelfish uses version 153.80 of the International Components for Unicode (ICU) collation library. For more information about ICU collations, see Collation in the ICU documentation. To learn more about PostgreSQL and collation, see Collation Support in the the PostgreSQL documentation.

Topics

DB cluster parameters that control collation and locale

The following parameters affect collation behavior.

babelfishpg_tsql.default_locale

This parameter specifies the default locale used by the collation. This parameter is used in combination with attributes listed in the Babelfish collation attributes table to customize collations for a specific language and region. The default value for this parameter is `en-US`.

The default locale applies to all Babelfish collation names that start with "BBF" and to all SQL Server collations that are mapped to Babelfish collations. Changing the setting for this parameter on an existing Babelfish DB cluster doesn't affect the locale of existing collations. For the list of collations, see the Babelfish supported collations table.

babelfishpg_tsql.server_collation_name

This parameter specifies the default collation for the server (Aurora PostgreSQL DB cluster instance) and the database. The default value is `sql_latin1_general_cp1_ci_as`. The `server_collation_name` has to be a `CI_AS` collation because in T-SQL, the server collation determines how identifiers are compared.

When you create your Babelfish DB cluster, you choose the **Collation name** from the selectable list. These include the collations listed in the Babelfish supported collations table. Don't modify the `server_collation_name` after the Babelfish database is created.

The settings you choose when you create your Babelfish for Aurora PostgreSQL DB cluster are stored in the DB cluster parameter group associated with the cluster for these parameters and set its collation behavior.

Deterministic and nondeterministic collations and Babelfish

Babelfish supports deterministic and nondeterministic collations:

- A *deterministic collation* evaluates characters that have identical byte sequences as equal. That means that `x` and `X` aren't equal in a deterministic collation. Deterministic collations can be case-sensitive (CS) and accent-sensitive (AS).
- A *nondeterministic collation* doesn't need an identical match. A nondeterministic collation evaluates `x` and `X` as equal. Nondeterministic collations are case-insensitive (CI) and accent-insensitive (AI).

In the table following, you can find some behavior differences between Babelfish and PostgreSQL when using nondeterministic collations.

Babelfish	PostgreSQL
Supports the LIKE clause for CI_AS collations.	Doesn't support the LIKE clause on nondeterministic collations.
Doesn't support the `LIKE` clause on AI collations.	
Don't support pattern-matching operations on nondeterministic collations.	

For a list of other limitations and behavior differences for Babelfish compared to SQL Server and PostgreSQL, see Collation limitations and behavior differences.

Babelfish and SQL Server follow a naming convention for collations that describe the collation attributes, as shown in the table following.

Attribute	Description
AI	Accent-insensitive.
AS	Accent-sensitive.
BIN2	BIN2 requests data to be sorted in code point order. Unicode code point order is the same character order for UTF-8, UTF-16, and UCS-2 encodings. Code point order is a fast deterministic collation.
CI	Case-insensitive.
CS	Case-sensitive.
PREF	To sort uppercase letters before lowercase letters, use a PREF collation. If comparison is case-insensitive, the uppercase version of a letter sorts before the lowercase version, if there is no other distinction. The ICU library supports uppercase preference with `colCaseFirst=upper`, but not for CI_AS collations. PREF can be applied only to `CS_AS` deterministic collations.

Collations supported by Babelfish

Use the following collations as a server collation or an object collation.

Collation ID	Notes
bbf_unicode_general_ci_as	Supports case-insensitive comparison and the LIKE operator.
bbf_unicode_cp1_ci_as	Nondeterministic collation also known as CP1252.
bbf_unicode_CP1250_ci_as	Nondeterministic collation used to represent texts in Central European and Eastern European languages that use Latin scri pt.
bbf_unicode_CP1251_ci_as	Nondeterministic collation for languages that use the Cyrillic script.
bbf_unicode_cp1253_ci_as	Nondeterministic collation used to represent modern Greek.
bbf_unicode_cp1254_ci_as	Nondeterministic collation that supports Turkish.
bbf_unicode_cp1255_ci_as	Nondeterministic collation that supports Hebrew.
bbf_unicode_cp1256_ci_as	Nondeterministic collation used to write languages that use Arabic script.
bbf_unicode_cp1257_ci_as	Nondeterministic collation used to support Estonian, Latvian, and Lithuanian languages.
bbf_unicode_cp1258_ci_as	Nondeterministic collation used to write Vietnamese characte rs.

Collation ID	Notes
bbf_unicode_cp874_ci_as	Nondeterministic collation used to write Thai characters.
sql_latin1_general_cp1250_ci_as	Nondeterministic single-byte character encoding used to represent Latin characters.
sql_latin1_general_cp1251_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1253_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1254_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1255_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1256_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1257_ci_as	Nondeterministic collation that supports Latin characters.
sql_latin1_general_cp1258_ci_as	Nondeterministic collation that supports Latin characters.
chinese_prc_ci_as	Nondeterministic collation that supports Chinese (PRC).

Collation ID	Notes
cyrillic_general_ci_as	Nondeterministic collation that supports Cyrillic.
finnish_swedish_ci_as	Nondeterministic collation that supports Finnish.
french_ci_as	Nondeterministic collation that supports French.
japanese_ci_as	Nondeterministic collation that supports Japanese. Supported in Babelfish 2.1.0 and higher releases.
korean_wansung_ci_as	Nondeterministic collation that supports Korean (with dictionar y sort).
latin1_general_ci_as	Nondeterministic collation that supports Latin characters.
modern_spanish_ci_as	Nondeterministic collation that supports Modern Spanish.
polish_ci_as	Nondeterministic collation that supports Polish.
thai_ci_as	Nondeterministic collation that supports Thai.
traditional_spanish_ci_as	Nondeterministic collation that supports Spanish (traditional sort).
turkish_ci_as	Nondeterministic collation that supports Turkish.
ukrainian_ci_as	Nondeterministic collation that supports Ukrainian.
vietnamese_ci_as	Nondeterministic collation that supports Vietnamese.

You can use the following collations as object collations.

Dialect	Deterministic options	Nondeterministic options
Arabic	Arabic_CS_AS	Arabic_CI_AS, Arabic_CI_AI
Chinese	Chinese_CS_AS	Chinese_CI_AS, Chinese_CI_AI
Cyrillic_General	Cyrillic_General_CS_AS	Cyrillic_General_CI_AS, Cyrillic_General_CI_AI
Estonian	Estonian_CS_AS	Estonian_CI_AS, Estonian_CI_AI
Finnish_Swedish	Finnish_Swedish_CS_AS	Finnish_Swedish_CI_AS, Finnish_Swedish_CI_AI
French	French_CS_AS	French_CI_AS, French_CI_AI
Greek	Greek_CS_AS	Greek_CI_AS, Greek_CI_AI
Hebrew	Hebrew_CS_AS	Hebrew_CI_AS, Hebrew_CI_AI
Japanese (Babelfish 2.1.0 and higher)	Japanese_CS_AS	Japanese_CI_AI, Japanese_CI_AS
Korean_Wamsung	Korean_Wamsung_CS_AS	Korean_Wamsung_CI_AS, Korean_Wamsung_CI_AI
Modern_Spanish	Modern_Spanish_CS_AS	Modern_Spanish_CI_AS, Modern_Spanish_CI_AI

Dialect	Deterministic options	Nondeterministic options
Mongolian	Mongolian_CS_AS	Mongolian_CI_AS, Mongolian_CI_AI
Polish	Polish_CS_AS	Polish_CI_AS, Polish_CI_AI
Thai	Thai_CS_AS	Thai_CI_AS, Thai_CI_AI
Traditional_Spanish	Traditional_Spanish_CS_AS	Traditional_Spanish_CI_AS, Traditional_Spanish_CI_AI
Turkish	Turkish_CS_AS	Turkish_CI_AS, Turkish_CI_AI
Ukranian	Ukranian_CS_AS	Ukranian_CI_AS, Ukranian_CI_AI
Vietnamese	Vietnamese_CS_AS	Vietnamese_CI_AS, Vietnamese_CI_AI

Default Collation in Babelfish

Earlier, the default collation of the collatable datatypes was `pg_catalog.default`. The datatypes and the objects that depends on these datatypes follows cases-sensitive collation. This condition potentially impacts the T-SQL objects of the data set with case-insensitive collation. Starting with Babelfish 2.3.0, the default collation for the collatable data types (except TEXT and NTEXT) is the same as the collation in the `babelfishpg_tsql.server_collation_name` parameter. When you upgrade to Babelfish 2.3.0, the default collation is picked automatically at the time of DB cluster creation, which doesn't create any visible impact.

Managing collations

The ICU library provides collation version tracking to ensure that indexes that depend on collations can be reindexed when a new version of ICU becomes available. To see if your current database has collations that need refreshing, you can use the following query after connecting using `psql` or or `pgAdmin`:

```
SELECT pg_describe_object(refclassid, refobjid,
    refobjsubid) AS "Collation",
    pg_describe_object(classid, objid, objsubid) AS "Object"
    FROM pg_depend d JOIN pg_collation c ON refclassid = 'pg_collation'::regclass
    AND refobjid = c.oid WHERE c.collversion <> pg_collation_actual_version(c.oid)
    ORDER BY 1, 2;
```

This query returns output such as the following:

```
 Collation | Object
-----------+--------
(0 rows)
```

In this example, no collations need to be updated.

To get a listing of the predefined collations in your Babelfish database, you can use `psql` or `pgAdmin` with the following query:

```
SELECT * FROM pg_collation;
```

Predefined collations are stored in the `sys.fn_helpcollations` table. You can use the following command to display information about a collation (such as its lcid, style, and collate flags). To get a listing of all collations by using `sqlcmd`, connect to the T-SQL port (1433, by default) and run the following query:

```
1> :setvar SQLCMDMAXVARTYPEWIDTH 40
2> :setvar SQLCMDMAXFIXEDTYPEWIDTH 40
3> SELECT * FROM fn_helpcollations()
4> GO
name                                     description
---------------------------------------- ----------------------------------------
arabic_cs_as                             Arabic, case-sensitive, accent-sensitive
arabic_ci_ai                             Arabic, case-insensitive, accent-insensi
arabic_ci_as                             Arabic, case-insensitive, accent-sensiti
bbf_unicode_bin2                         Unicode-General, case-sensitive, accent-
bbf_unicode_cp1250_ci_ai                 Default locale, code page 1250, case-ins
bbf_unicode_cp1250_ci_as                 Default locale, code page 1250, case-ins
bbf_unicode_cp1250_cs_ai                 Default locale, code page 1250, case-sen
bbf_unicode_cp1250_cs_as                 Default locale, code page 1250, case-sen
bbf_unicode_pref_cp1250_cs_as            Default locale, code page 1250, case-sen
bbf_unicode_cp1251_ci_ai                 Default locale, code page 1251, case-ins
bbf_unicode_cp1251_ci_as                 Default locale, code page 1251, case-ins
bbf_unicode_cp1254_ci_ai                 Default locale, code page 1254, case-ins
```

```
...
(124 rows affected)
```

Lines 1 and 2 shown in the example narrow the output for documentation readability purposes only.

```
1> SELECT SERVERPROPERTY('COLLATION')
2> GO
serverproperty
---------------------------------------------------------------------------
sql_latin1_general_cp1_ci_as

(1 rows affected)
1>
```

Collation limitations and behavior differences

Babelfish uses the ICU library for collation support. PostgreSQL is built with a specific version of ICU and can match at most one version of a collation. Variations across versions are unavoidable, as are minor variations across time as languages evolve. In the following list you can find known limitations and behavior variations of Babelfish collations:

- **Indexes and collation type dependency** – An index on a user-defined type that depends on the International Components for Unicode (ICU) collation library (the library used by Babelfish) isn't invalidated when the library version changes.
- **COLLATIONPROPERTY function** – Collation properties are implemented only for the supported Babelfish BBF collations. For more information, see the Babelfish supported collations table.
- **Unicode sorting rule differences** – SQL collations for SQL Server sort Unicode-encoded data (`nchar` and `nvarchar`) differently than data that's not Unicode-encoded (`char` and `varchar`). Babelfish databases are always UTF-8 encoded and always apply Unicode sorting rules consistently, regardless of data type, so the sort order for `char` or `varchar` is the same as it is for `nchar` or `nvarchar`.
- **Secondary-equal collations and sorting behavior** – The default ICU Unicode secondary-equal (`CI_AS`) collation sorts punctuation marks and other nonalphanumeric characters before numeric characters, and numeric characters before alphabetic characters. However, the order of punctuation and other special characters is different.
- **Tertiary collations, workaround for ORDER BY** – SQL collations, such as `SQL_Latin1_General_Pref_CP1_CI_AS`, support the `TERTIARY_WEIGHTS` function and the

ability to sort strings that compare equally in a CI_AS collation to be sorted uppercase first: ABC, ABc, AbC, Abc, aBC, aBc, abC, and finally abc. Thus, the DENSE_RANK OVER (ORDER BY column) analytic function assesses these strings as having the same rank but orders them uppercase first within a partition.

You can get a similar result with Babelfish by adding a COLLATE clause to the ORDER BY clause that specifies a tertiary CS_AS collation that specifies @colCaseFirst=upper. However, the colCaseFirst modifier applies only to strings that are tertiary-equal (rather than secondary-equal such as with CI_AS collation). Thus, you can't emulate tertiary SQL collations using a single ICU collation.

As a workaround, we recommend that you modify applications that use the SQL_Latin1_General_Pref_CP1_CI_AS collation to use the BBF_SQL_Latin1_General_CP1_CI_AS collation first. Then add COLLATE BBF_SQL_Latin1_General_Pref_CP1_CS_AS to any ORDER BY clause for this column.

- **Character expansion** – A character expansion treats a single character as equal to a sequence of characters at the primary level. SQL Server's default CI_AS collation supports character expansion. ICU collations support character expansion for accent-insensitive collations only.

  When character expansion is required, then use a AI collation for comparisons. However, such collations aren't currently supported by the LIKE operator.

- **char and varchar encoding** – When SQL collations are used for char or varchar data types, the sort order for characters preceding ASCII 127 is determined by the specific code page for that SQL collation. For SQL collations, strings declared as char or varchar might sort differently than strings declared as nchar or nvarchar.

  PostgreSQL encodes all strings with the database encoding, so all characters are converted to UTF-8 and sorted using Unicode rules.

  Because SQL collations sort nchar and nvarchar data types using Unicode rules, Babelfish encodes all strings on the server using UTF-8. Babelfish sorts nchar and nvarchar strings the same way it sorts char and varchar strings, using Unicode rules.

- **Supplementary character** – The SQL Server functions NCHAR, UNICODE, and LEN support characters for code-points outside the Unicode Basic Multilingual Plane (BMP). In contrast, non-SC collations use surrogate pair characters to handle supplementary characters. For Unicode data types, SQL Server can represent up to 65,535 characters using UCS-2, or the full Unicode range (1,114,114 characters) if supplementary characters are used.

- **Kana-sensitive (KS) collations –** A Kana-sensitive (KS) collation is one that treats `Hiragana` and `Katakana` Japanese Kana characters differently. ICU supports the Japanese collation standard `JIS X 4061`. The now deprecated `colhiraganaQ [on | off]` locale modifier might provide the same functionality as KS collations. However, KS collations of the same name as SQL Server aren't currently supported by Babelfish.
- **Width-sensitive (WS) collations –** When a single-byte character (half-width) and the same character represented as a double-byte character (full-width) are treated differently, the collation is called *width-sensitive (WS)*. WS collations with the same name as SQL Server aren't currently supported by Babelfish.
- **Variation-selector sensitive (VSS) collations –** Variation-selector sensitive (VSS) collations distinguish between ideographic variation selectors in Japanese collations `Japanese_Bushu_Kakusu_140` and `Japanese_XJIS_140`. A variation sequence is made up of a base character plus an additional variation selector. If you don't select the `_VSS` option, the variation selector isn't considered in the comparison.

  VSS collations aren't currently supported by Babelfish.
- **BIN and BIN2 collations –** A BIN2 collation sorts characters according to code point order. The byte-by-byte binary order of UTF-8 preserves Unicode code point order, so this is also likely to be the best-performing collation. If Unicode code point order works for an application, consider using a BIN2 collation. However, using a BIN2 collation can result in data being displayed on the client in an order that is culturally unexpected. New mappings to lowercase characters are added to Unicode as time progresses, so the `LOWER` function might perform differently on different versions of ICU. This is a special case of the more general collation versioning problem rather than as something specific to the BIN2 collation.

  Babelfish provides the `BBF_Latin1_General_BIN2` collation with the Babelfish distribution to collate in Unicode code point order. In a BIN collation only the first character is sorted as a wchar. Remaining characters are sorted byte-by-byte, effectively in code point order according to its encoding. This approach doesn't follow Unicode collation rules and isn't supported by Babelfish.
- **Non-deterministic collations and CHARINDEX limitation –** For Babelfish releases older than version 2.1.0, you can't use CHARINDEX with non-deterministic collations. By default, Babelfish uses a case-insensitive (non-deterministic) collation. Using CHARINDEX for older versions of Babelfish raises the following runtime error:

  ```
  nondeterministic collations are not supported for substring searches
  ```

Note

This limitation and workaround apply to Babelfish version 1.x only (Aurora PostgreSQL 13.x versions). Babelfish 2.1.0 and higher releases don't have this issue.

You can work around this issue in one of the following ways:

- Explicitly convert the expression to a case-sensitive collation and case-fold both arguments by applying LOWER or UPPER. For example, `SELECT charindex('x', a) FROM t1` would become the following:

```
SELECT charindex(LOWER('x'), LOWER(a COLLATE sql_latin1_general_cp1_cs_as)) FROM t1
```

- Create a SQL function f_charindex, and replace CHARINDEX calls with calls to the following function:

```
CREATE function f_charindex(@s1 varchar(max), @s2 varchar(max)) RETURNS int
AS
BEGIN
declare @i int = 1
WHILE len(@s2) >= len(@s1)
BEGIN
  if LOWER(@s1) = LOWER(substring(@s2,1,len(@s1))) return @i
  set @i += 1
  set @s2 = substring(@s2,2,999999999)
END
return 0
END
go
```

Managing Babelfish error handling with escape hatches

Babelfish mimics SQL behavior for control flow and transaction state whenever possible. When Babelfish encounters an error, it returns an error code similar to the SQL Server error code. If Babelfish can't map the error to a SQL Server code, it returns a fixed error code (`33557097`) and takes specific actions based on the type of error, as follows:

- For compile time errors, Babelfish rolls back the transaction.
- For runtime errors, Babelfish ends the batch and rolls back the transaction.
- For protocol error between client and server, the transaction isn't rolled back.

If an error code can't be mapped to an equivalent code and the code for a similar error is available, the error code is mapped to the alternative code. For example, the behaviors that cause SQL Server codes `8143` and `8144` are both mapped to `8143`.

Errors that can't be mapped don't respect a `TRY... CATCH` construct.

You can use `@@ERROR` to return a SQL Server error code, or the `@@PGERROR` function to return a PostgreSQL error code. You can also use the `fn_mapped_system_error_list` function to return a list of mapped error codes. For information about PostgreSQL error codes, see the PostgreSQL website.

Modifying Babelfish escape hatch settings

To handle statements that might fail, Babelfish defines certain options called escape hatches. An *escape hatch* is an option that specifies Babelfish behavior when it encounters an unsupported feature or syntax.

You can use the `sp_babelfish_configure` stored procedure to control the settings of an escape hatch. Use the script to set the escape hatch to `ignore` or `strict`. If it's set to `strict`, Babelfish returns an error that you need to correct before continuing.

To apply changes to the current session and on the cluster level, include the `server` keyword.

The usage is as follows:

- To list all escape hatches and their status, plus usage information, run `sp_babelfish_configure`.

- To list the named escape hatches and their values, for the current session or cluster-wide, run the command `sp_babelfish_configure 'hatch_name'` where *hatch_name* is the identifier of one or more escape hatches. *hatch_name* can use SQL wildcards, such as '%'.
- To set one or more escape hatches to the value specified, run `sp_babelfish_configure ['hatch_name' [, 'strict'|'ignore' [, 'server']]`. To make the settings permanent on a cluster-wide level, include the `server` keyword, such as shown in the following:

  ```
  EXECUTE sp_babelfish_configure 'escape_hatch_unique_constraint', 'ignore', 'server'
  ```

  To set them for the current session only, don't use `server`.
- To reset all escape hatches to their default values, run `sp_babelfish_configure 'default'` (Babelfish 1.2.0 and higher).

The string identifying the hatch (or hatches) can include SQL wildcards. For example, the following sets all syntax escape hatches to `ignore` for the Aurora PostgreSQL cluster.

```
EXECUTE sp_babelfish_configure '%', 'ignore', 'server'
```

In the following table you can find descriptions and default values for the Babelfish predefined escape hatches.

Escape hatch	Description	Default
`escape_hatch_checkpoint`	Allows the use of CHECKPOINT statement in the procedural code, but the CHECKPOINT statement is currently not implemented.	ignore
`escape_hatch_constraint_name_for_default`	Controls Babelfish behavior related to default constraint names.	ignore
`escape_hatch_database_misc_options`	Controls Babelfish behavior related to the following options on CREATE or ALTER DATABASE:	ignore

Escape hatch	Description	Default
	CONTAINMENT, DB_CHAINING, TRUSTWORTHY, PERSISTENT_LOG_BUFFER.	
`escape_hatch_for_replication`	Controls Babelfish behavior related to the [NOT] FOR REPLICATION clause when creating or altering a table.	strict
`escape_hatch_fulltext`	Controls Babelfish behavior related to FULLTEXT features, such a DEFAULT_FULLTEXT_LANGUAGE in CREATE/ALTER DATABASE, CREATE FULLTEXT INDEX, or sp_fulltext_database.	ignore
`escape_hatch_ignore_dup_key`	Controls Babelfish behavior related to CREATE/ALTER TABLE and CREATE INDEX. When IGNORE_DUP_KEY=ON, raises an error when set to `strict` (the default) or ignores the error when set to `ignore` (Babelfish version 1.2.0 and higher).	strict
`escape_hatch_index_clustering`	Controls Babelfish behavior related to the CLUSTERED or NONCLUSTERED keywords for indexes and PRIMARY KEY or UNIQUE constraints. When CLUSTERED is ignored, the index or constraint is still created as if NONCLUSTERED was specified.	ignore

Escape hatch	Description	Default
`escape_hatch_index_columnstore`	Controls Babelfish behavior related to the COLUMNSTORE clause. If you specify `ignore`, Babelfish creates a regular B-tree index.	strict
`escape_hatch_join_hints`	Controls the behavior of keywords in a JOIN operator: LOOP, HASH, MERGE, REMOTE, REDUCE, REDISTRIBUTE, REPLICATE.	ignore
`escape_hatch_language_non_english`	Controls Babelfish behavior related to languages other than English for onscreen messages. Babelfish currently supports only `us_english` for onscreen messages. SET LANGUAGE might use a variable containing the language name, so the actual language being set can only be detected at run time.	strict
`escape_hatch_login_hashed_password`	When ignored, suppresses the error for the HASHED keyword for `CREATE LOGIN` and `ALTER LOGIN`.	strict

Escape hatch	Description	Default
`escape_hatch_login_misc_options`	When ignored, suppresses the error for other keywords besides `HASHED`, `MUST_CHANGE` , `OLD_PASSWORD` , and `UNLOCK` for `CREATE LOGIN` and `ALTER LOGIN`.	strict
`escape_hatch_login_old_password`	When ignored, suppresses the error for the `OLD_PASSWORD` keyword for `CREATE LOGIN` and `ALTER LOGIN`.	strict
`escape_hatch_login_password_must_change`	When ignored, suppresses the error for the `MUST_CHANGE` keyword for `CREATE LOGIN` and `ALTER LOGIN`.	strict
`escape_hatch_login_password_unlock`	When ignored, suppresses the error for the `UNLOCK` keyword for `CREATE LOGIN` and `ALTER LOGIN`.	strict
`escape_hatch_nocheck_add_constraint`	Controls Babelfish behavior related to the WITH CHECK or NOCHECK clause for constraints.	strict
`escape_hatch_nocheck_existing_constraint`	Controls Babelfish behavior related to FOREIGN KEY or CHECK constraints.	strict

Escape hatch	Description	Default
`escape_hatch_query_hints`	Controls Babelfish behavior related to query hints. When this option is set to ignore, the server ignores hints that use the OPTION (...) clause to specify query processing aspects. Examples include S ELECT FROM ... OPTION(MERGE JOIN HASH, MAXRECURSION 10)).	ignore
`escape_hatch_rowversion`	Controls the behavior of the ROWVERSION and TIMESTAMP datatypes. For usage information, see Using Babelfish features with limited implementation.	strict
`escape_hatch_schemabinding_function`	Controls Babelfish behavior related to the WITH SCHEMABIN DING clause. By default, the WITH SCHEMABINDING clause is ignored when specified with the CREATE or ALTER FUNCTION command.	ignore
`escape_hatch_schemabinding_procedure`	Controls Babelfish behavior related to the WITH SCHEMABIN DING clause. By default, the WITH SCHEMABINDING clause is ignored when specified with the CREATE or ALTER PROCEDURE command.	ignore

Escape hatch	Description	Default
`escape_hatch_rowguidcol_column`	Controls Babelfish behavior related to the ROWGUIDCOL clause when creating or altering a table.	strict
`escape_hatch_schemabinding_trigger`	Controls Babelfish behavior related to the WITH SCHEMABINDING clause. By default, the WITH SCHEMABINDING clause is ignored when specified with the CREATE or ALTER TRIGGER command.	ignore
`escape_hatch_schemabinding_view`	Controls Babelfish behavior related to the WITH SCHEMABINDING clause. By default, the WITH SCHEMABINDING clause is ignored when specified with the CREATE or ALTER VIEW command.	ignore
`escape_hatch_session_settings`	Controls Babelfish behavior toward unsupported session-level SET statements.	ignore
`escape_hatch_showplan_all`	Controls Babelfish behavior related to SET SHOWPLAN_ALL and SET STATISTICS PROFILE. When set to ignore, they behave like SET BABELFISH_SHOWPLAN_ALL and SET BABELFISH_STATISTICS PROFILE; when set to strict, they are silently ignored.	strict

Escape hatch	Description	Default
`escape_hatch_storage_on_partition`	Controls Babelfish behavior related to the `ON partition_scheme column` clause when defining partitioning. Babelfish currently doesn't implement partitioning.	strict
`escape_hatch_storage_options`	Escape hatch on any storage option used in CREATE, ALTER DATABASE, TABLE, INDEX. This includes clauses (LOG) ON, TEXTIMAGE_ON, FILESTREAM_ON that define storage locations (partitions, file groups) for tables, indexes, and constraints, and also for a database. This escape hatch setting applies to all of these clauses (including ON [PRIMARY] and ON "DEFAULT"). The exception is when a partition is specified for a table or index with ON partition_scheme (column).	ignore
`escape_hatch_table_hints`	Controls the behavior of table hints specified using the WITH (...) clause.	ignore

Escape hatch	Description	Default
`escape_hatch_unique_constraint`	When set to strict, an obscure semantic difference between SQL Server and PostgreSQL in handling NULL values on indexed columns can raise errors. The semantic difference only emerges in unrealistic use cases, so you can set this escape hatch to 'ignore' to avoid seeing the error.	strict

Creating a Babelfish for Aurora PostgreSQL DB cluster

Babelfish for Aurora PostgreSQL is supported on Aurora PostgreSQL version 13.4 and higher.

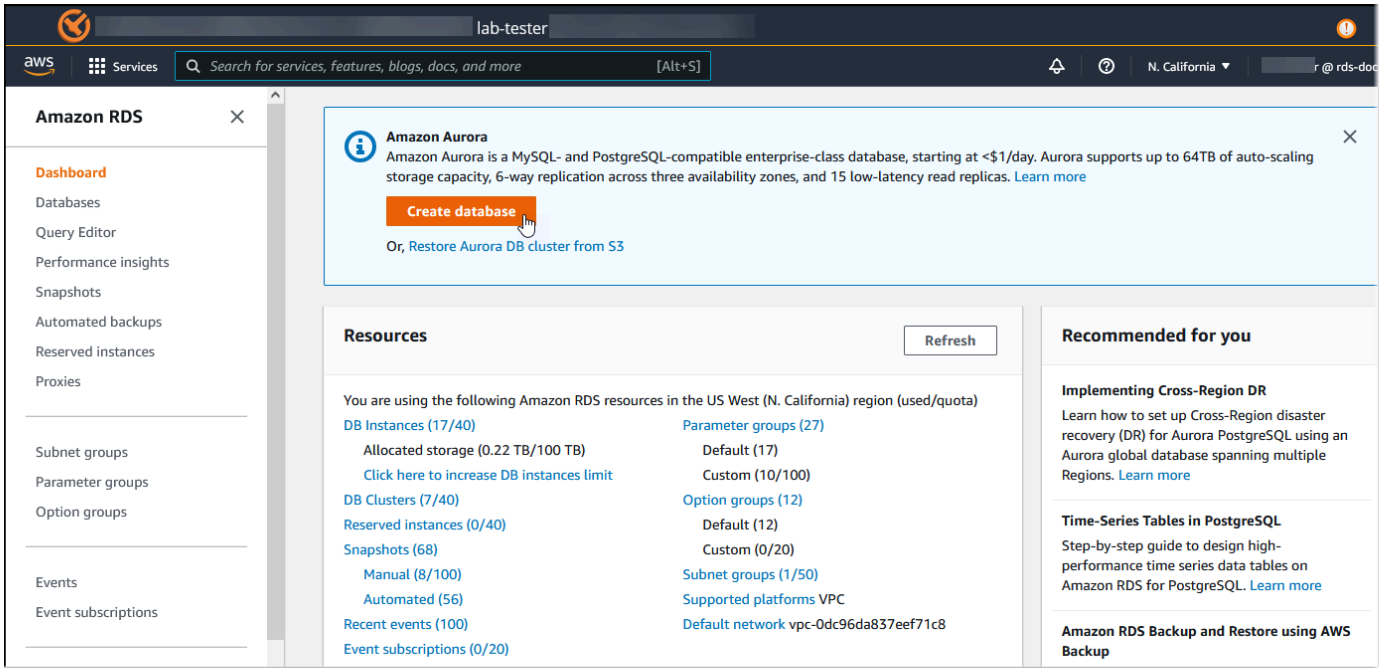
You can use the AWS Management Console or the AWS CLI to create an Aurora PostgreSQL cluster with Babelfish.

> **Note**
>
> In an Aurora PostgreSQL cluster, the `babelfish_db` database name is reserved for Babelfish. Creating your own "babelfish_db" database on a Babelfish for Aurora PostgreSQL prevents Aurora from successfully provisioning Babelfish.

Console

To create a cluster with Babelfish running with the AWS Management Console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/, and choose **Create database**.



Amazon Aurora

Amazon Aurora is a MySQL- and PostgreSQL-compatible enterprise-class database, starting at <$1/day. Aurora supports up to 64TB of auto-scaling storage capacity, 6-way replication across three availability zones, and 15 low-latency read replicas. Learn more

Or, Restore Aurora DB cluster from S3

Resources

You are using the following Amazon RDS resources in the US West (N. California) region (used/quota)

DB Instances (17/40)
    Allocated storage (0.22 TB/100 TB)
    Click here to increase DB instances limit
DB Clusters (7/40)
Reserved instances (0/40)
Snapshots (68)
    Manual (8/100)
    Automated (56)
Recent events (100)
Event subscriptions (0/20)

Parameter groups (27)
    Default (17)
    Custom (10/100)
Option groups (12)
    Default (12)
    Custom (0/20)
Subnet groups (1/50)
Supported platforms VPC
Default network vpc-0dc96da837eef71c8

Recommended for you

Implementing Cross-Region DR
Learn how to set up Cross-Region disaster recovery (DR) for Aurora PostgreSQL using an Aurora global database spanning multiple Regions. Learn more

Time-Series Tables in PostgreSQL
Step-by-step guide to design high-performance time series data tables on Amazon RDS for PostgreSQL. Learn more

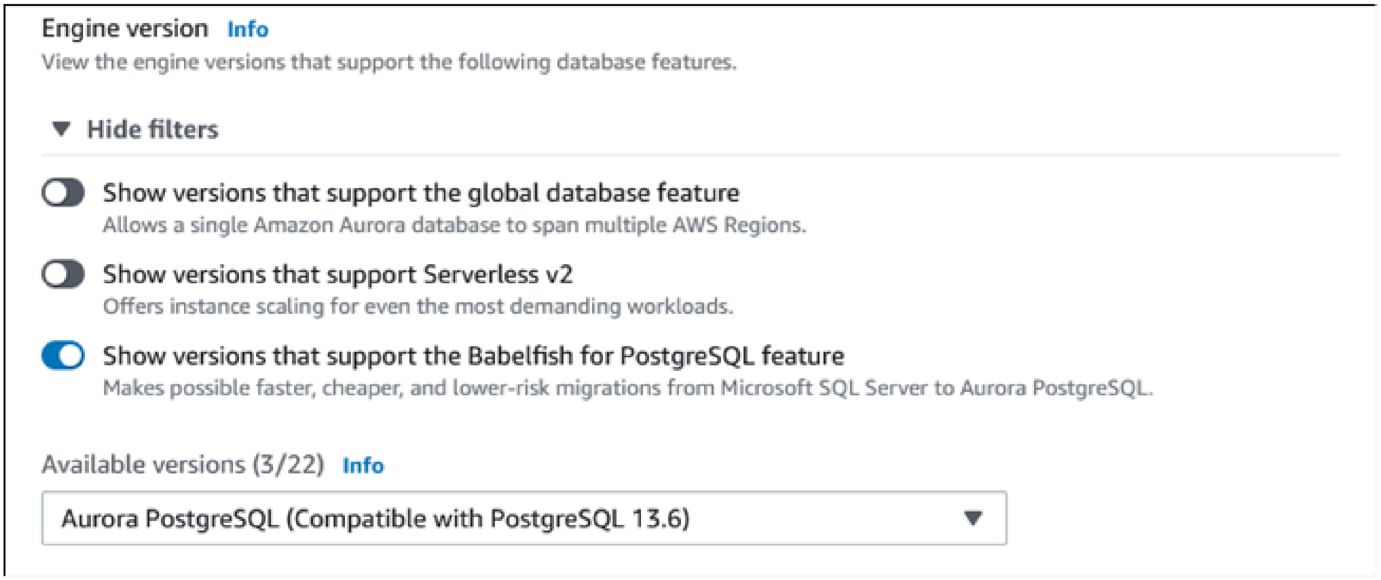
Amazon RDS Backup and Restore using AWS Backup

2. For **Choose a database creation method**, do one of the following:

   - To specify detailed engine options, choose **Standard create**.

- To use preconfigured options that support best practices for an Aurora cluster, choose **Easy create**.

3. For **Engine type**, choose **Aurora (PostgreSQL Compatible)**.
4. Choose **Show filters**, and then choose **Show versions that support the Babelfish for PostgreSQL feature** to list the engine types that support Babelfish. Babelfish is currently supported on Aurora PostgreSQL 13.4 and higher versions.
5. For **Available versions**, choose an Aurora PostgreSQL version. To get the latest Babelfish features, choose the highest Aurora PostgreSQL major version.



Engine version Info

View the engine versions that support the following database features.

▼ Hide filters

○ Show versions that support the global database feature

Allows a single Amazon Aurora database to span multiple AWS Regions.

○ Show versions that support Serverless v2

Offers instance scaling for even the most demanding workloads.

● Show versions that support the Babelfish for PostgreSQL feature

Makes possible faster, cheaper, and lower-risk migrations from Microsoft SQL Server to Aurora PostgreSQL.

Available versions (3/22) Info

Aurora PostgreSQL (Compatible with PostgreSQL 13.6) ▼

6. For **Templates**, choose the template that matches your use case.
7. For **DB cluster identifier**, enter a name that you can easily find later in the DB cluster list.
8. For **Master username**, enter an administrator user name. The default value for Aurora PostgreSQL is `postgres`. You can accept the default or choose a different name. For example, to follow the naming convention used on your SQL Server databases, you can enter `sa` (system administrator) for the Master username.

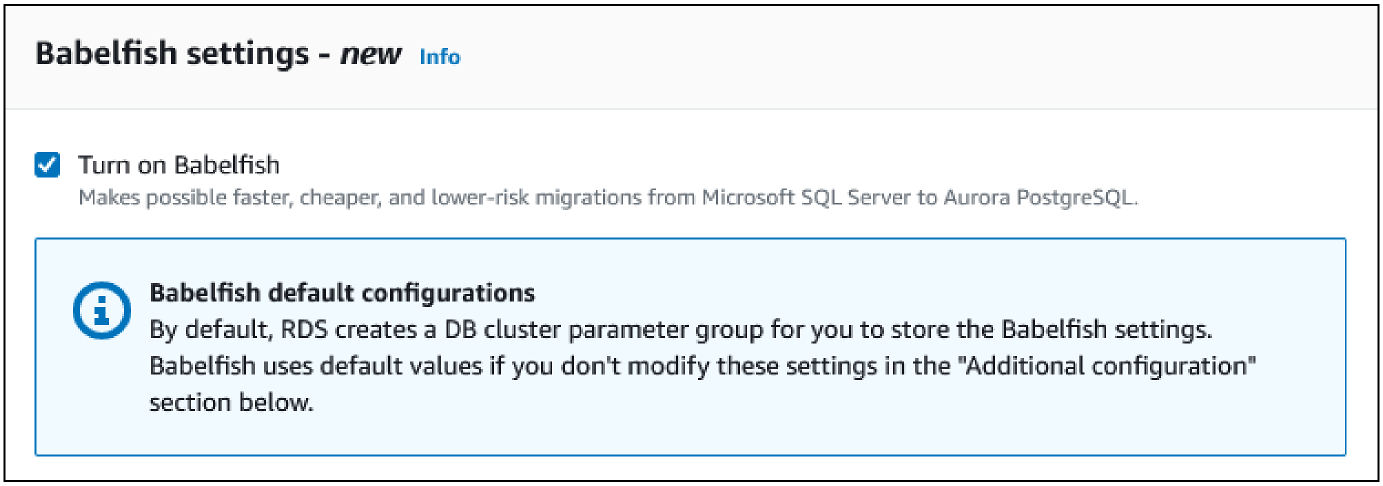
   If you don't create a user named `sa` at this time, you can create one later with your choice of client. After creating the user, use the `ALTER SERVER ROLE` command to add it to the `sysadmin` group (role) for the cluster.

   > **Warning**
   >
   > Master username must always use lowercase characters failing which the DB cluster can't connect to Babelfish via the TDS port.

9. For **Master password**, create a strong password and confirm the password.

10. For the options that follow, until the **Babelfish settings** section, specify your DB cluster settings. For information about each setting, see Settings for Aurora DB clusters.

11. To make Babelfish functionality available, select the **Turn on Babelfish** box.

    Babelfish settings - *new* Info

    ☑ Turn on Babelfish
    Makes possible faster, cheaper, and lower-risk migrations from Microsoft SQL Server to Aurora PostgreSQL.

    Babelfish default configurations
    By default, RDS creates a DB cluster parameter group for you to store the Babelfish settings. Babelfish uses default values if you don't modify these settings in the "Additional configuration" section below.

12. For **DB cluster parameter group**, do one of the following:

    - Choose **Create new** to create a new parameter group with Babelfish turned on.
    - Choose **Choose existing** to use an existing parameter group. If you use an existing group, make sure to modify the group before creating the cluster and add values for Babelfish parameters. For information about Babelfish parameters, see DB cluster parameter group settings for Babelfish.

      If you use an existing group, provide the group name in the box that follows.

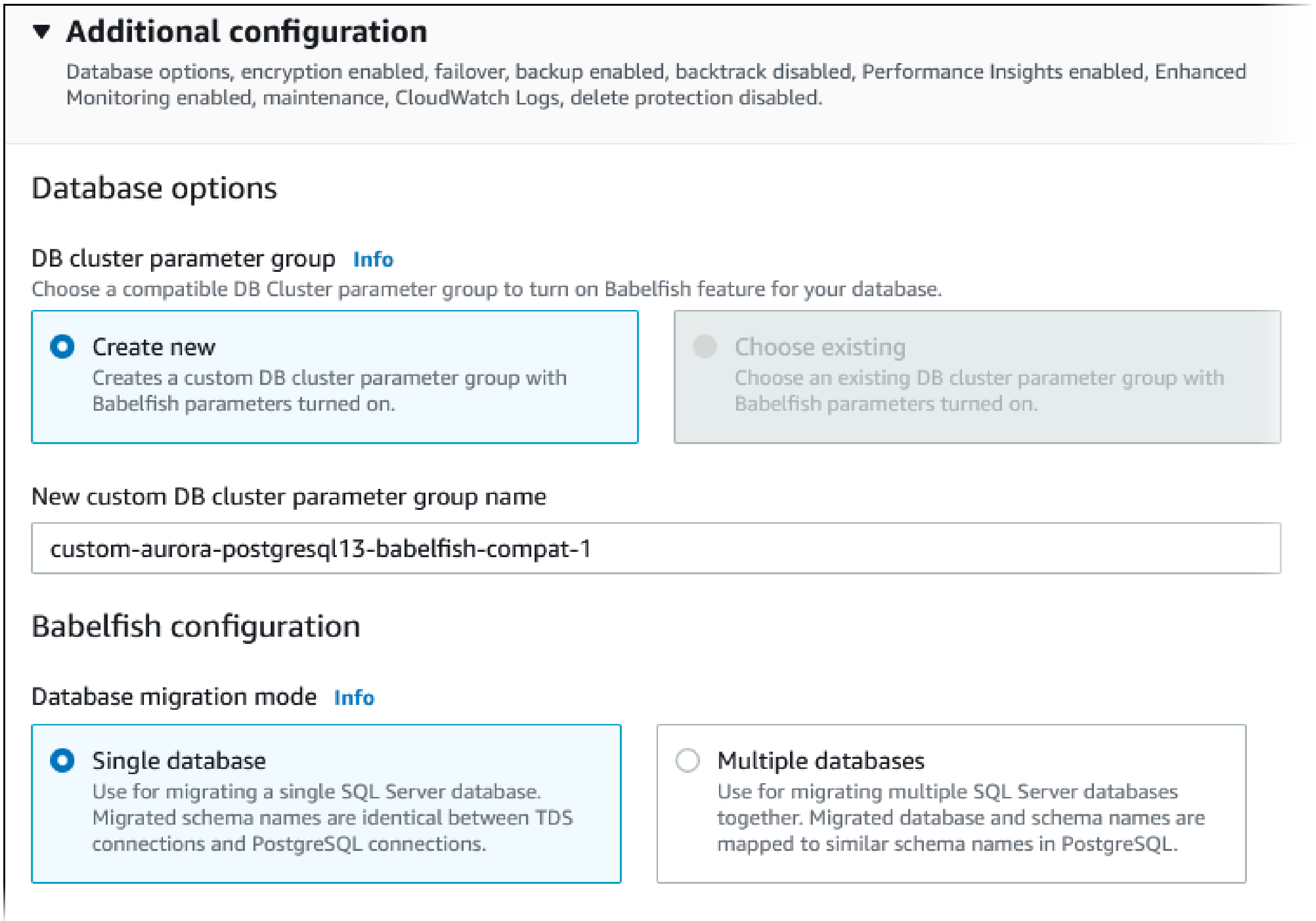
13. For **Database migration mode**, choose one of the following:

    - **Single database** to migrate a single SQL Server database.

      In some cases, you might migrate multiple user databases together, with your end goal a complete migration to native Aurora PostgreSQL without Babelfish. If the final applications require consolidated schemas (a single dbo schema), make sure to first consolidate your SQL Server databases into a single SQL server database. Then migrate to Babelfish using **Single database** mode.
    - **Multiple databases** to migrate multiple SQL Server databases (originating from a single SQL Server installation). Multiple database mode doesn't consolidate multiple databases that don't originate from a single SQL Server installation. For information about migrating multiple databases, see Using Babelfish with a single database or multiple databases.

Note

From Aurora PostgreSQL 16 version, **Multiple databases** is chosen by default as the Database migration mode.



▼ Additional configuration

Database options, encryption enabled, failover, backup enabled, backtrack disabled, Performance Insights enabled, Enhanced Monitoring enabled, maintenance, CloudWatch Logs, delete protection disabled.

Database options

DB cluster parameter group Info

Choose a compatible DB Cluster parameter group to turn on Babelfish feature for your database.

● Create new
Creates a custom DB cluster parameter group with Babelfish parameters turned on.

○ Choose existing
Choose an existing DB cluster parameter group with Babelfish parameters turned on.

New custom DB cluster parameter group name

custom-aurora-postgresql13-babelfish-compat-1

Babelfish configuration

Database migration mode Info

● Single database
Use for migrating a single SQL Server database. Migrated schema names are identical between TDS connections and PostgreSQL connections.

○ Multiple databases
Use for migrating multiple SQL Server databases together. Migrated database and schema names are mapped to similar schema names in PostgreSQL.

14. For **Default collation locale**, enter your server locale. The default is `en-US`. For detailed information about collations, see Collations supported by Babelfish.

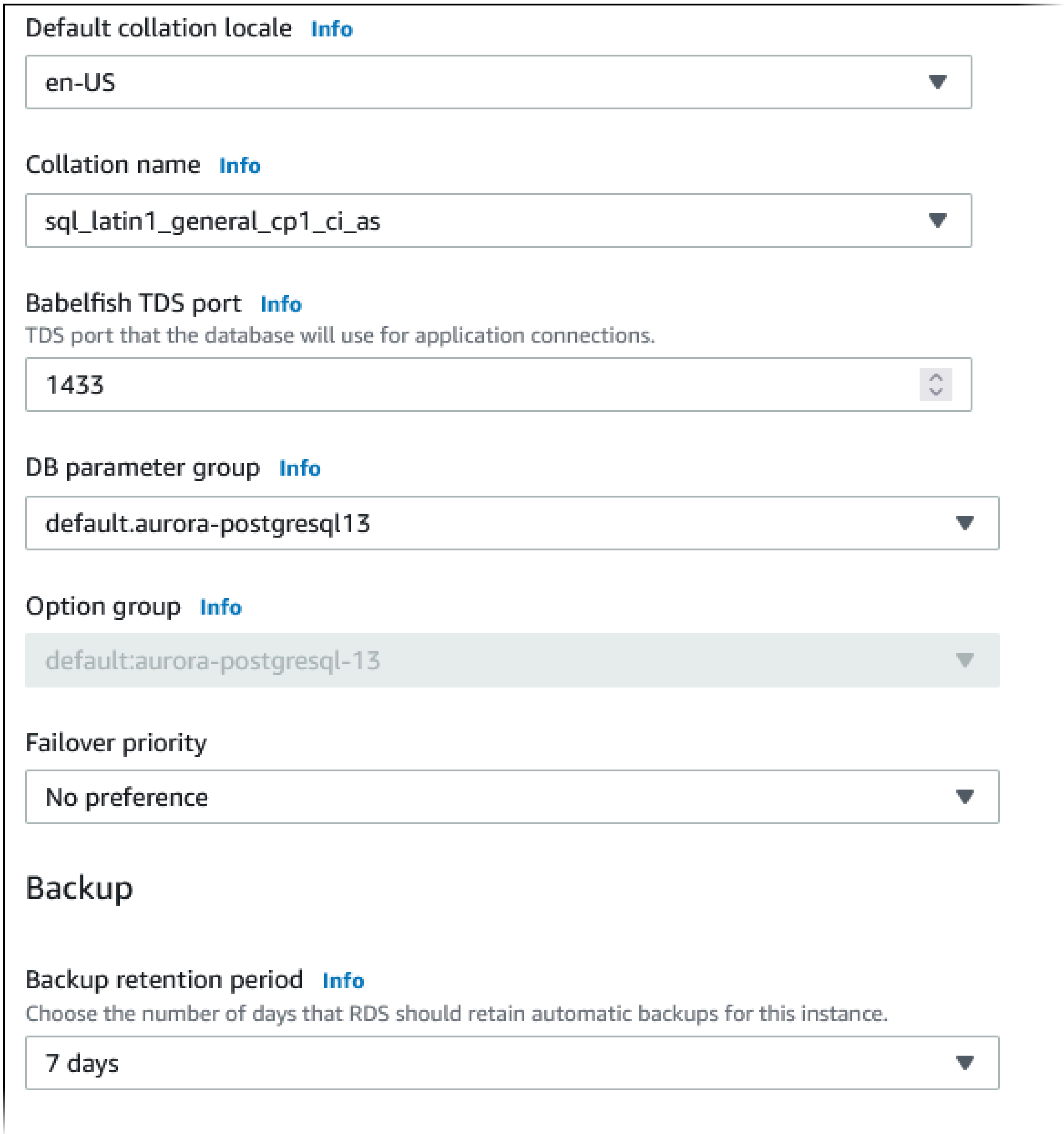
15. For **Collation name** field, enter your default collation. The default is `sql_latin1_general_cp1_ci_as`. For detailed information, see Collations supported by Babelfish.

16. For **Babelfish TDS port**, enter the default port `1433`. Currently, Babelfish only supports port `1433` for your DB cluster.

17. For **DB parameter group**, choose a parameter group or have Aurora create a new group for you with default settings.

18. For **Failover priority**, choose a failover priority for the instance. If you don't choose a value, the default is `tier-1`. This priority determines the order in which replicas are promoted when recovering from a primary instance failure. For more information, see [Fault tolerance for an Aurora DB cluster](#).

19. For **Backup retention period**, choose the length of time (1–35 days) that Aurora retains backup copies of the database. You can use backup copies for point-in-time restores (PITR) of your database down to the second. The default retention period is seven days.



Default collation locale Info

en-US

Collation name Info

sql_latin1_general_cp1_ci_as

Babelfish TDS port Info

TDS port that the database will use for application connections.

1433

DB parameter group Info

default.aurora-postgresql13

Option group Info

default:aurora-postgresql-13

Failover priority

No preference

Backup

Backup retention period Info

Choose the number of days that RDS should retain automatic backups for this instance.

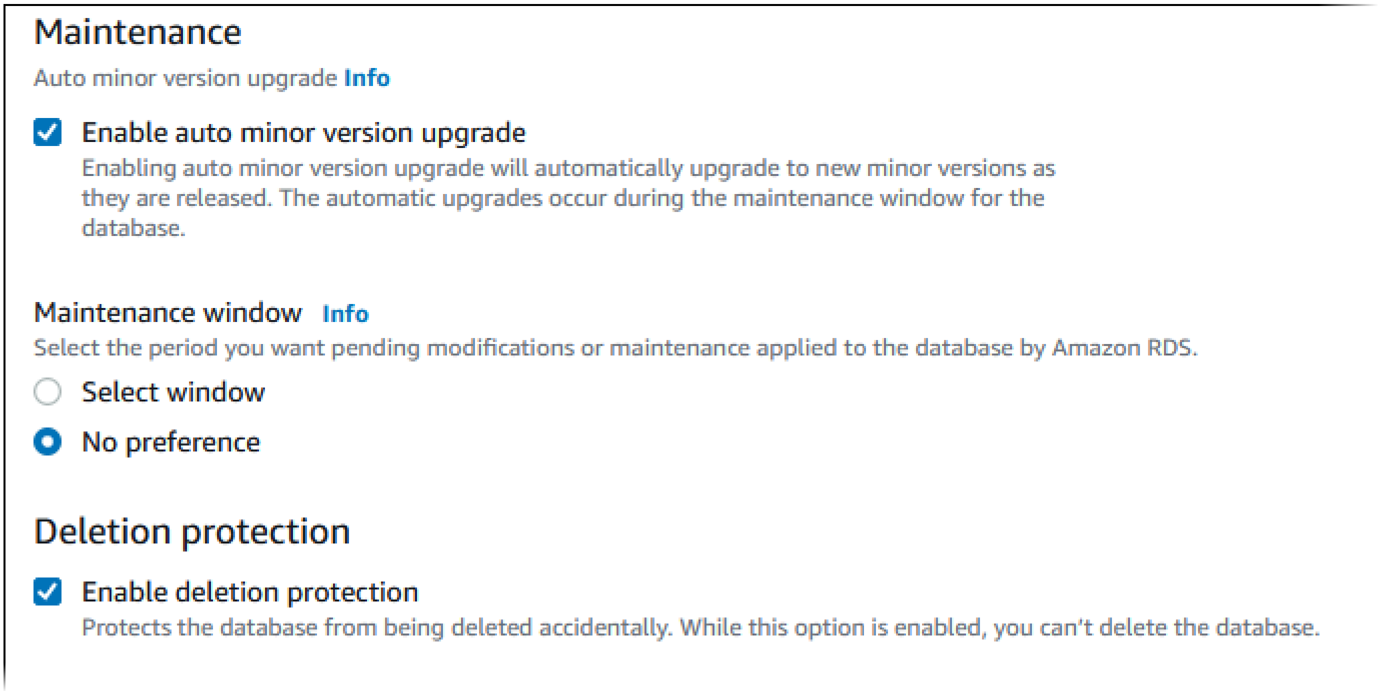
7 days

20. Choose **Copy tags to snapshots** to copy any DB instance tags to a DB snapshot when you create a snapshot.
21. Choose **Enable encryption** to turn on encryption at rest (Aurora storage encryption) for this DB cluster.
22. Choose **Enable Performance Insights** to turn on Amazon RDS Performance Insights.
23. Choose **Enable Enhanced monitoring** to start gathering metrics in real time for the operating system that your DB cluster runs on.
24. Choose **PostgreSQL log** to publish the log files to Amazon CloudWatch Logs.
25. Choose **Enable auto minor version upgrade** to automatically update your Aurora DB cluster when a minor version upgrade is available.
26. For **Maintenance window**, do the following:

    - To choose a time for Amazon RDS to make modifications or perform maintenance, choose **Select window**.
    - To perform Amazon RDS maintenance at an unscheduled time, choose **No preference**.

27. Select the **Enable deletion protection** box to protect your database from being deleted by accident.

    If you turn on this feature, you can't directly delete the database. Instead, you need to modify the database cluster and turn off this feature before deleting the database.



Maintenance

Auto minor version upgrade Info

☑ Enable auto minor version upgrade
Enabling auto minor version upgrade will automatically upgrade to new minor versions as they are released. The automatic upgrades occur during the maintenance window for the database.

Maintenance window Info
Select the period you want pending modifications or maintenance applied to the database by Amazon RDS.

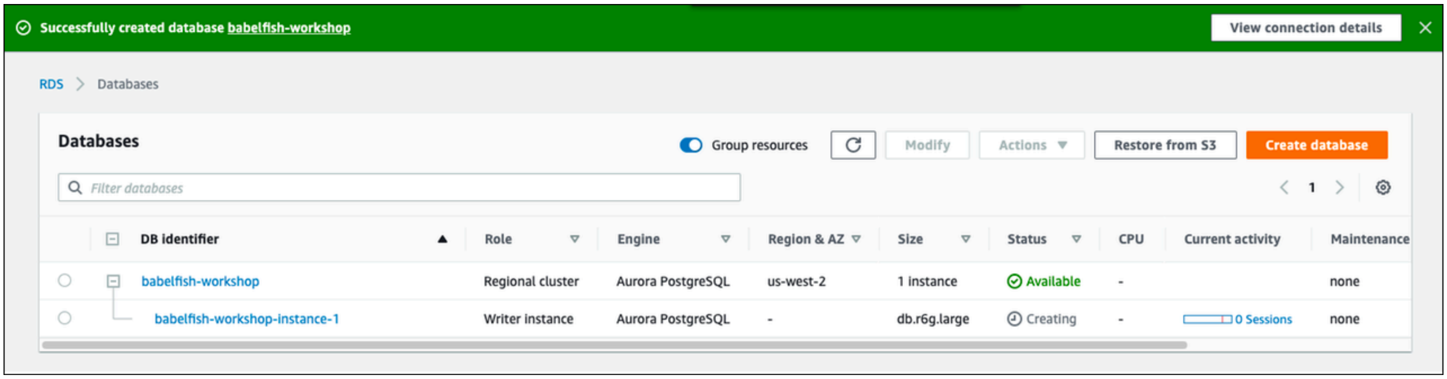
○ Select window
● No preference

Deletion protection

☑ Enable deletion protection
Protects the database from being deleted accidentally. While this option is enabled, you can't delete the database.

28. Choose **Create database**.

You can find your new database set up for Babelfish in the **Databases** listing. The **Status** column displays **Available** when the deployment is complete.



Successfully created database babelfish-workshop

	DB identifier	Role	Engine	Region & AZ	Size	Status	CPU	Current activity	Maintenance
○	babelfish-workshop	Regional cluster	Aurora PostgreSQL	us-west-2	1 instance	Available	-		none
○	babelfish-workshop-instance-1	Writer instance	Aurora PostgreSQL	-	db.r6g.large	Creating	-	0 Sessions	none

AWS CLI

When you create an Babelfish for Aurora PostgreSQL; using the AWS CLI, you need to pass the command the name of the DB cluster parameter group to use for the cluster. For more information, see DB cluster prerequisites.

Before you can use the AWS CLI to create an Aurora PostgreSQL cluster with Babelfish, do the following:

- Choose your endpoint URL from the list of services at Amazon Aurora endpoints and quotas.
- Create a parameter group for the cluster. For more information about parameter groups, see Working with parameter groups.
- Modify the parameter group, adding the parameter that turns on Babelfish.

To create an Aurora PostgreSQL DB cluster with Babelfish using the AWS CLI

The examples that follow use the default Master username, `postgres`. Replace as needed with the username that you created for your DB cluster, such as `sa` or whatever username you chose if you didn't accept the default.

1. Create a parameter group.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster-parameter-group \
   ```

```
--endpoint-url endpoint-url \
--db-cluster-parameter-group-name parameter-group \
--db-parameter-group-family aurora-postgresql14 \
--description "description"
```

For Windows:

```
aws rds create-db-cluster-parameter-group ^
--endpoint-url endpoint-URL ^
--db-cluster-parameter-group-name parameter-group ^
--db-parameter-group-family aurora-postgresql14 ^
--description "description"
```

2. Modify your parameter group to turn on Babelfish.

   For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
--endpoint-url endpoint-url \
--db-cluster-parameter-group-name parameter-group \
--parameters
 "ParameterName=rds.babelfish_status,ParameterValue=on,ApplyMethod=pending-reboot"
```

   For Windows:

```
aws rds modify-db-cluster-parameter-group ^
--endpoint-url endpoint-url ^
--db-cluster-parameter-group-name paramater-group ^
--parameters
 "ParameterName=rds.babelfish_status,ParameterValue=on,ApplyMethod=pending-reboot"
```

3. Identify your DB subnet group and the virtual private cloud (VPC) security group ID for your new DB cluster, and then call the create-db-cluster command.

   For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
--db-cluster-identifier  cluster-name\
--master-username postgres \
--manage-master-user-password \
--engine aurora-postgresql \
--engine-version 14.3            \
--vpc-security-group-ids security-group \
--db-subnet-group-name subnet-group-name \
--db-cluster-parameter-group-name parameter-group
```

For Windows:

```
aws rds create-db-cluster ^
--db-cluster-identifier cluster-name ^
--master-username postgres ^
--manage-master-user-password ^
--engine aurora-postgresql ^
--engine-version 14.3 ^
--vpc-security-group-ids security-group ^
--db-subnet-group-name subnet-group ^
--db-cluster-parameter-group-name parameter-group
```

This example specifies the `--manage-master-user-password` option to generate the master user password and manage it in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

4. Explicitly create the primary instance for your DB cluster. Use the name of the cluster that you created in step 3 for the `--db-cluster-identifier` argument when you call the create-db-instance command, as shown following.

   For Linux, macOS, or Unix:

```
aws rds create-db-instance \
--db-instance-identifier instance-name \
--db-instance-class db.r6g \
--db-subnet-group-name subnet-group \
--db-cluster-identifier cluster-name \
--engine aurora-postgresql
```

   For Windows:

```
aws rds create-db-instance ^
--db-instance-identifier instance-name ^
--db-instance-class db.r6g ^
--db-subnet-group-name subnet-group ^
--db-cluster-identifier cluster-name ^
```

```
--engine aurora-postgresql
```

Migrating a SQL Server database to Babelfish for Aurora PostgreSQL

You can use Babelfish for Aurora PostgreSQL to migrate an SQL Server database to an Amazon Aurora PostgreSQL DB cluster. Before migrating, review Using Babelfish with a single database or multiple databases.

Topics

Overview of the migration process

The following summary lists the steps required to successfully migrate your SQL Server application and make it work with Babelfish. For information about the tools you can use for the export and import processes and for more detail, see Import/export tools for migrating from SQL Server to Babelfish. To load the data, we recommend using AWS DMS with an Aurora PostgreSQL DB cluster as the target endpoint.

1. Create a new Aurora PostgreSQL DB cluster with Babelfish turned on. To learn how, see Creating a Babelfish for Aurora PostgreSQL DB cluster.

   To import the various SQL artifacts exported from your SQL Server database, connect to the Babelfish cluster using a SQL Server tool such as sqlcmd. For more information, see Using a SQL Server client to connect to your DB cluster.
2. On the SQL Server database that you want to migrate, export the data definition language (DDL). The DDL is SQL code that describes database objects that contain user data (such as tables, indexes, and views) and user-written database code (such as stored procedures, user-defined functions, and triggers).

   For more information, see Using SQL Server Management Studio (SSMS) to migrate to Babelfish.
3. Run an assessment tool to evaluate the scope of any changes that you might need to make so that Babelfish can effectively support the application running on SQL Server. For more information, see Evaluating and handling differences between SQL Server and Babelfish.
4. Review the AWS DMS target endpoint limitations and update the DDL script as necessary. For more information, see Limitations to using a PostgreSQL target endpoint with Babelfish tables in Using for Aurora PostgreSQL as a target.

5. On your new Babelfish DB cluster, run the DDL within your specified T-SQL database to create only the schemas, user-defined data types, and tables with their primary key constraints.
6. Use AWS DMS to migrate your data from SQL Server to Babelfish tables. For continuous replication using SQL Server Change Data Capture or SQL Replication, use Aurora PostgreSQL instead of Babelfish as the endpoint. To do so, see the Using Babelfish for Aurora PostgreSQL as a target for AWS Database Migration Service.
7. When the data load completes, create all the remaining T-SQL objects that support the application on your Babelfish cluster.
8. Reconfigure your client application to connect to the Babelfish endpoint instead of your SQL Server database. For more information, see Connecting to a Babelfish DB cluster.
9. Modify your application as needed and retest. For more information, see Differences between Babelfish for Aurora PostgreSQL and SQL Server.

You still need to assess your client-side SQL queries. The schemas generated from your SQL Server instance convert only the server-side SQL code. We recommend that you take the following steps:

- Capture client-side queries by using the SQL Server Profiler with the TSQL_Replay predefined template. This template captures T-SQL statement information that you can then replay for iterative tuning and testing. You can start the profiler within SQL Server Management Studio from the **Tools** menu. Choose **SQL Server Profiler** to open the profiler and choose the TSQL_Replay template.

  To use for your Babelfish migration, start a trace and then run your application using your functional tests. The profiler captures the T-SQL statements. When you finish testing, stop the trace. Save the result to an XML file with your client-side queries (File > Save as > Trace XML File for Replay).

  For more information, see SQL Server Profiler in the Microsoft documentation. For more information about the TSQL_Replay template, see SQL Server Profiler Templates.
- For applications with complex client-side SQL queries, we recommend that you use Babelfish Compass to analyze these queries for Babelfish compatibility. If the analysis indicates that the client-side SQL statements contain unsupported SQL features, review the SQL aspects in the client application and modify as needed.
- You can also capture the SQL queries as extended events (.xel format). To do so, use the SSMS XEvent Profiler. After generating the .xel file, extract the SQL statements into .xml files that

Compass can then process. For more information, see Use the SSMS XEvent Profler in the Microsoft documentation.

When you're satisfied with all testing, analysis, and any modifications needed for your migrated application, you can start using your Babelfish database for production. To do so, stop the original database and redirect live client applications to use the Babelfish TDS port.

> **Note**
>
> AWS DMS now supports replicating data from Babelfish. For more information, see AWS DMS now supports Babelfish for Aurora PostgreSQL as a source.

Evaluating and handling differences between SQL Server and Babelfish

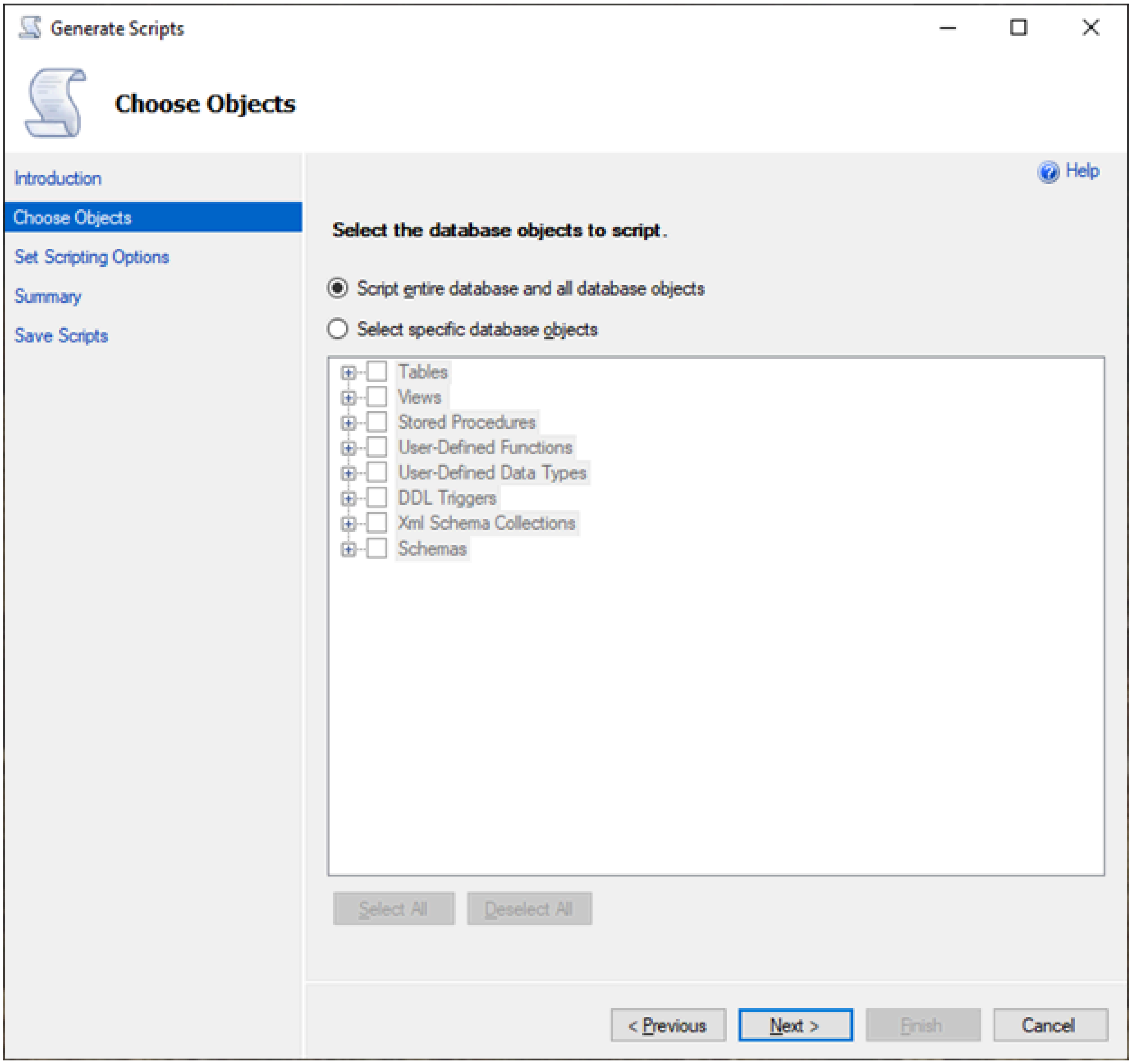
For best results, we recommend that you evaluate the generated DDL/DML and the client query code before actually migrating your SQL Server database application to Babelfish. Depending on the version of Babelfish and the specific features of SQL Server that your application implements, you might need to refactor your application or use alternatives for functionality that aren't fully supported yet in Babelfish.

- To assess your SQL Server application code, use Babelfish Compass on the generated DDL to determine how much T-SQL code is supported by Babelfish. Identify T-SQL code that might need modifications before running on Babelfish. For more information about this tool, see Babelfish Compass tool on GitHub.

  > **Note**
  >
  > Babelfish Compass is an open-source tool. Report any issues with Babelfish Compass through GitHub instead of through AWS Support.

You can use Generate Script Wizard with SQL Server Management Studio (SSMS) to generate the SQL file that is assessed by Babelfish Compass or AWS Schema Conversion Tool CLI. We recommend the following steps to streamline the assessment.

1. On the **Choose Objects** page, choose **Script entire database and all database objects**.

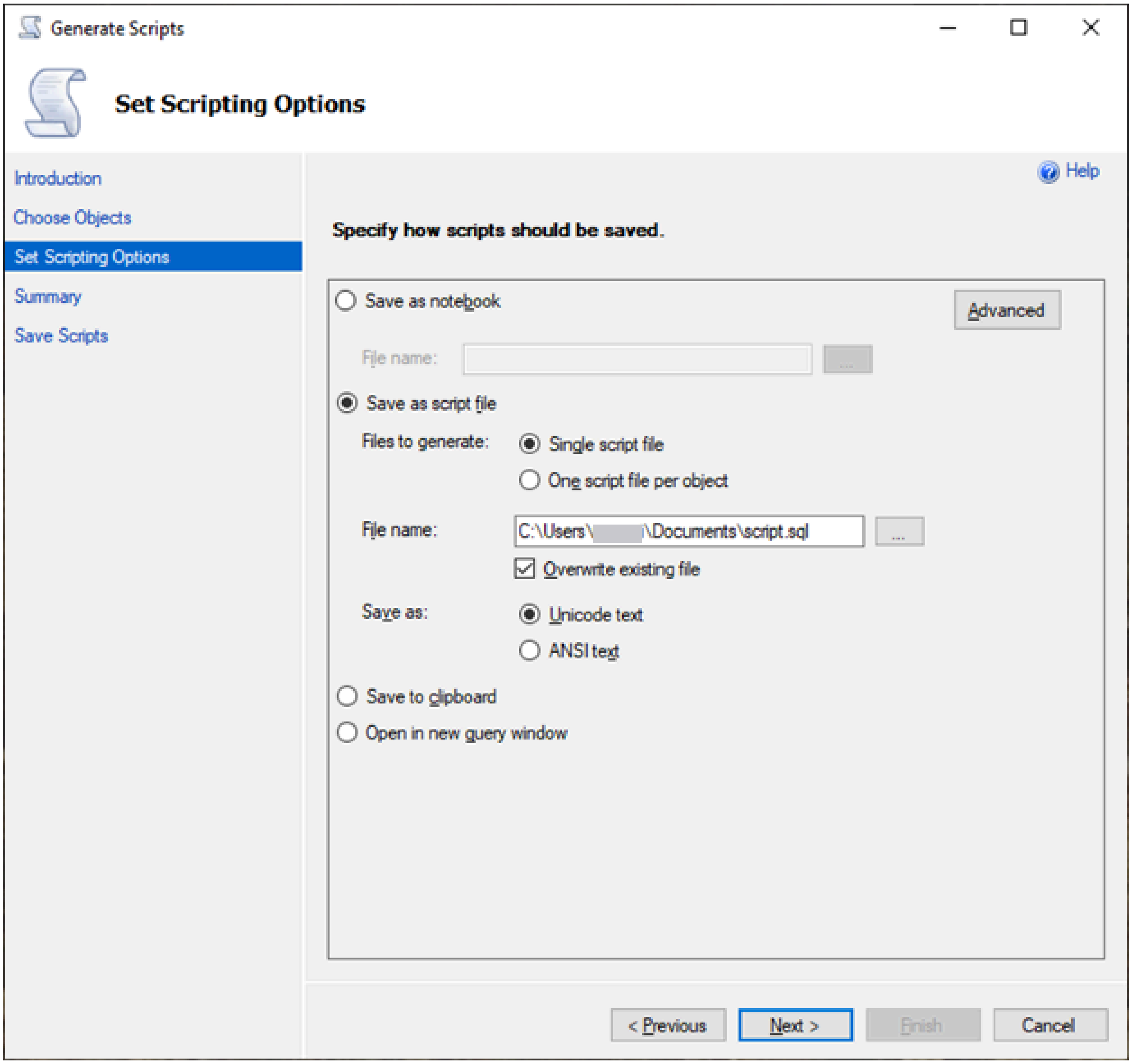


Select the database objects to script.

● Script entire database and all database objects

○ Select specific database objects

2. For the **Set Scripting Options**, choose **Save as script file** as a **Single script file**.



Set Scripting Options

Specify how scripts should be saved.

○ Save as notebook

File name:

● Save as script file

Files to generate: ● Single script file ○ One script file per object

File name: C:\Users\[redacted]\Documents\script.sql

☑ Overwrite existing file

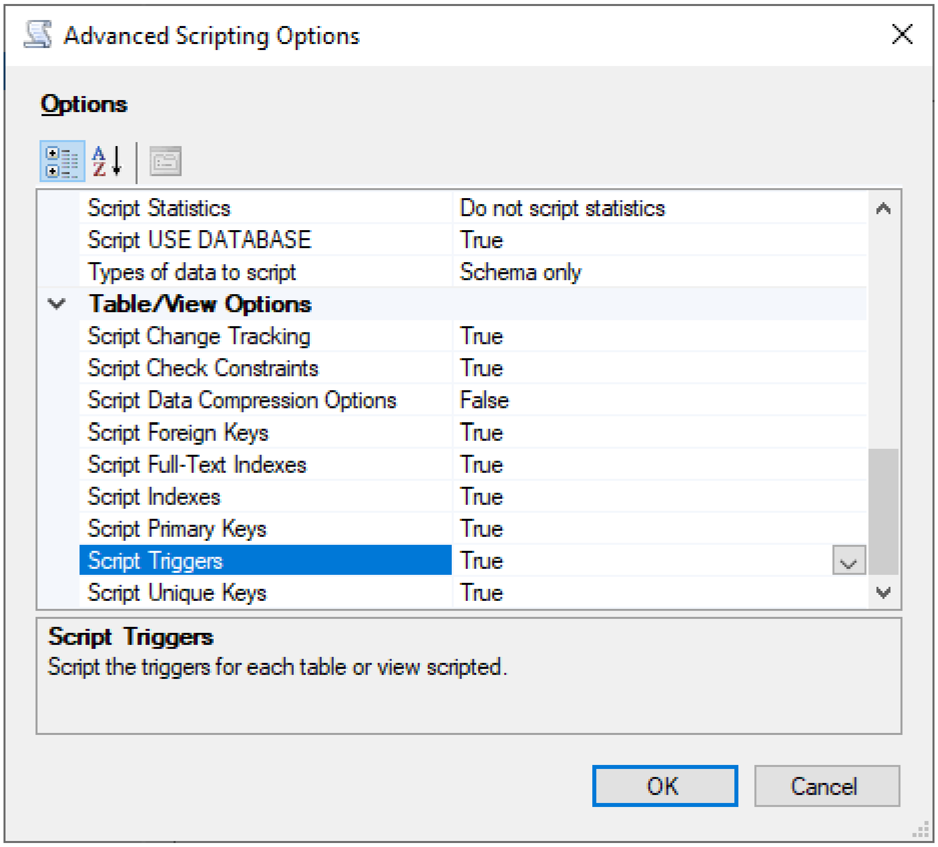
Save as: ● Unicode text ○ ANSI text

○ Save to clipboard

○ Open in new query window

3. Choose **Advanced** to change the default scripting options to identify features that normally are set to false for a full assessment:

    - Script Change Tracking to True
    - Script Full-Text Indexes to True
    - Script Triggers to True
    - Script Logins to True
    - Script Owner to True
    - Script Object-Level Permissions to True
    - Script Collations to True



Script Statistics	Do not script statistics
Script USE DATABASE	True
Types of data to script	Schema only
Table/View Options	
Script Change Tracking	True
Script Check Constraints	True
Script Data Compression Options	False
Script Foreign Keys	True
Script Full-Text Indexes	True
Script Indexes	True
Script Primary Keys	True
Script Triggers	True
Script Unique Keys	True

Script Triggers
Script the triggers for each table or view scripted.

4. Perform the remaining steps in the wizard to generate the file.

Import/export tools for migrating from SQL Server to Babelfish

We recommend that you use AWS DMS as the primary tool for migrating from SQL Server to Babelfish. However, Babelfish supports several other ways to migrate data using SQL Server tools that includes the following.

- SQL Server Integration Services (SSIS) for all versions of Babelfish. For more information, see Migrate from SQL Server to Aurora PostgreSQL using SSIS and Babelfish.
- Use the SSMS Import/Export Wizard for Babelfish versions 2.1.0 and later. This tool is available through the SSMS, but it's also available as a standalone tool. For more information, see Welcome to SQL Server Import and Export Wizard in the Microsoft documentation.
- The Microsoft bulk data copy program (bcp) utility lets you copy data from a Microsoft SQL Server instance to a data file in the format you specify. For more information, see bcp Utility in the Microsoft documentation. Babelfish now supports the data migration using the BCP client and the bcp utility now supports `-E` flag (for identity columns) and -b flag (for batching inserts). Certain bcp options aren't supported, including `-C`, `-T`, `-G`, `-K`, `-R`, `-V`, and `-h`.

Using SQL Server Management Studio (SSMS) to migrate to Babelfish

We recommend generating separate files for each of the specific object types. You can use the Generate Scripts wizard in SSMS for each set of DDL statements first, and then modify the objects as a group to fix any issues found during the assessment.

Perform these steps to migrate the data using AWS DMS or other data migration methods. Run these create script types first for a better and faster approach to load the data on the Babelfish tables in Aurora PostgreSQL.

1. Run `CREATE SCHEMA` statements.
2. Run `CREATE TYPE` statements to create user-defined data types.
3. Run basic `CREATE TABLE` statements with the primary keys or unique constraints.

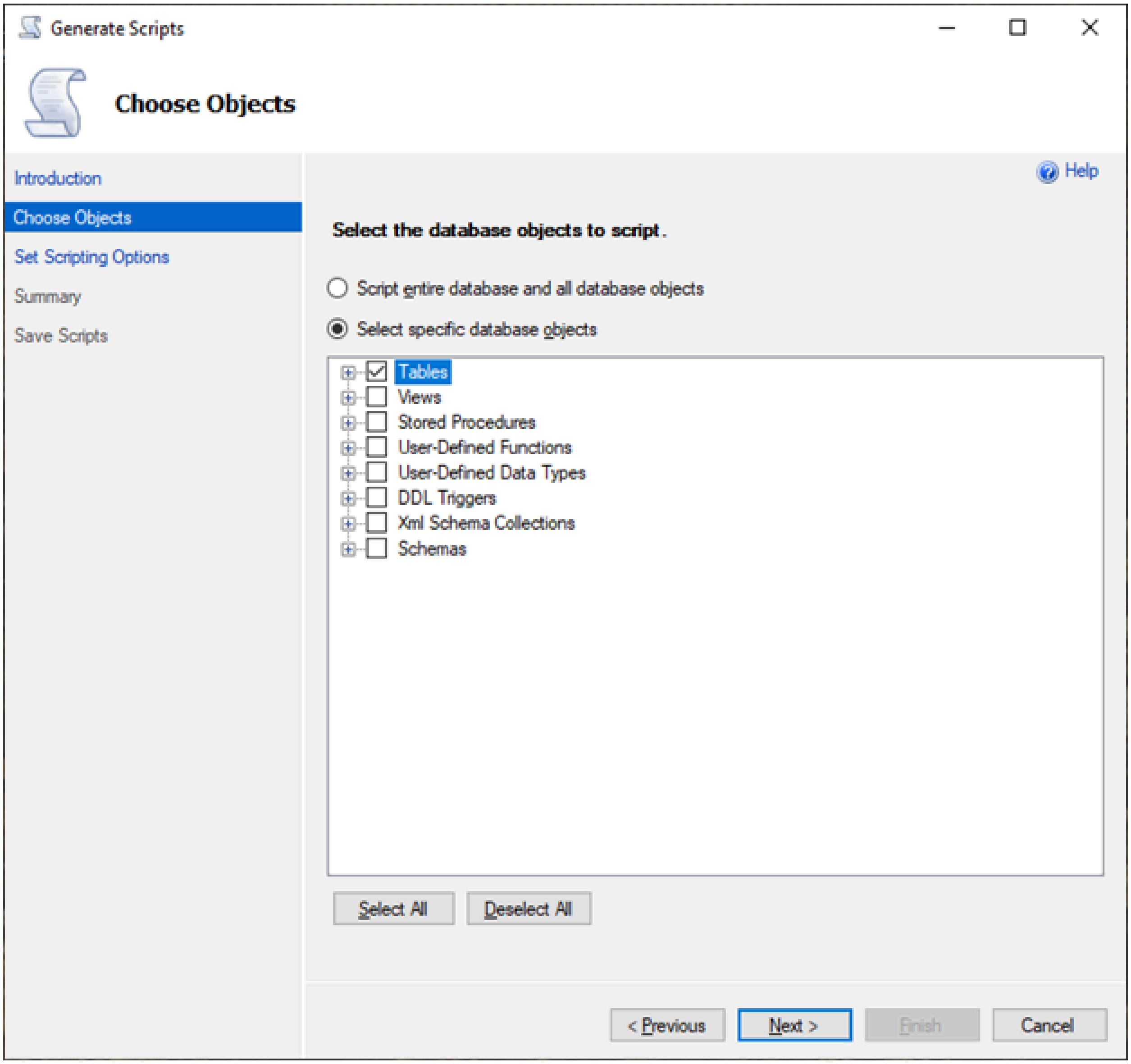
Perform the data load using the recommended import/export tool. Run the modified scripts for the following steps to add the remaining database objects. You need the create table statements to run these scripts for the constraints, triggers, and indexes. After the scripts generate, delete the create table statements.

1. Run `ALTER TABLE` statements for the check constraints, foreign key constraints, default constraints.
2. Run `CREATE TRIGGER` statements.
3. Run `CREATE INDEX` statements.
4. Run `CREATE VIEW` statements.
5. Run `CREATE STORED PROCEDURE` statements.

To generate scripts for each object type

Use the following steps to create the basic create table statements using the Generate Scripts wizard in SSMS. Follow the same steps to generate scripts for the different object types.

1. Connect to your existing SQL Server instance.
2. Open the context (right-click) menu for a database name.
3. Choose **Tasks**, and then choose **Generate Scripts...**.
4. On the **Choose Objects** pane, choose **Select specific database objects**. Choose **Tables**, select all tables. Choose **Next** to continue.



Generate Scripts

Choose Objects

Select the database objects to script.

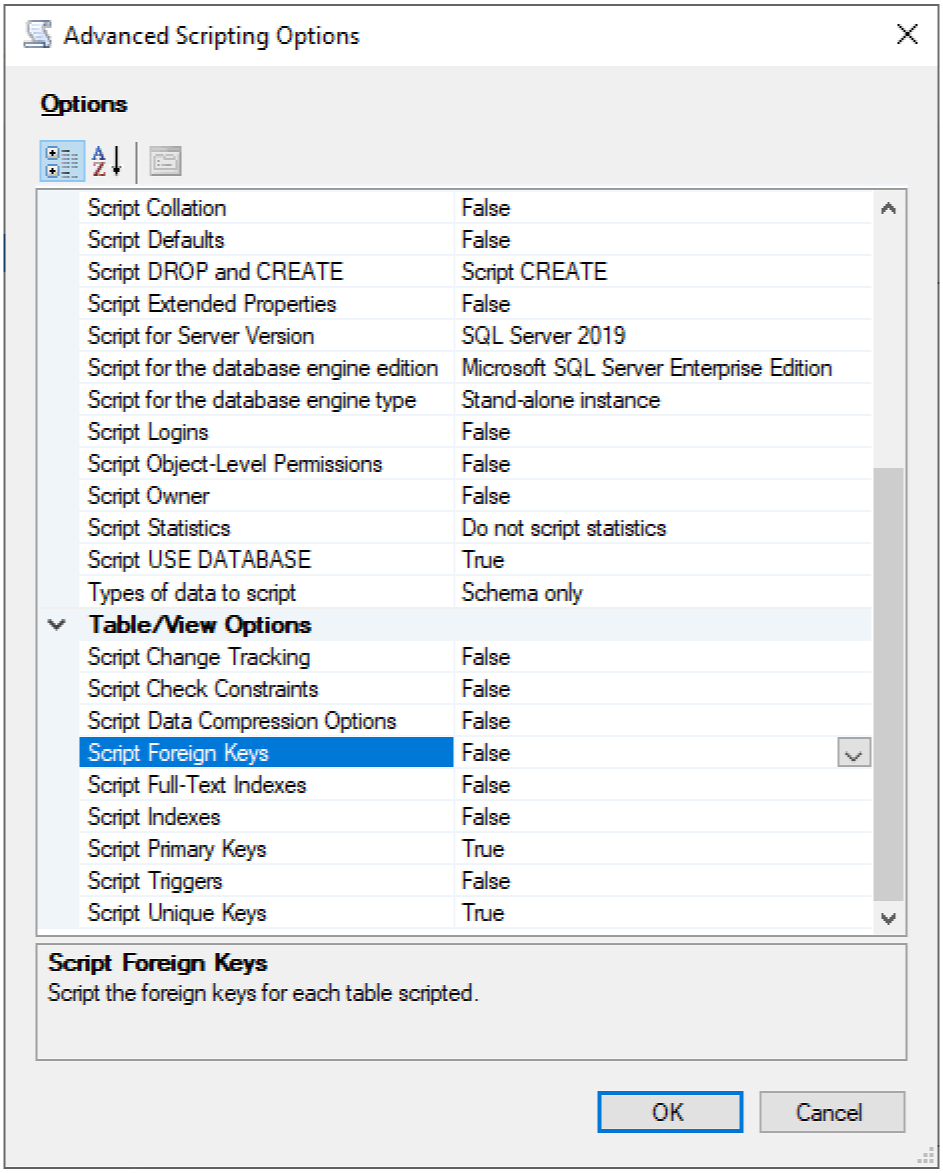
○ Script entire database and all database objects

● Select specific database objects

- ☑ Tables
- □ Views
- □ Stored Procedures
- □ User-Defined Functions
- □ User-Defined Data Types
- □ DDL Triggers
- □ Xml Schema Collections
- □ Schemas

5. On the **Set Scripting Options** page, choose **Advanced** to open the **Options** settings. To generate the basic create table statements, change the following default values:

   - Script Defaults to False.
   - Script Extended Properties to False. Babelfish does not support extended properties.
   - Script Check Constraints to False. Script Foreign Keys to False.

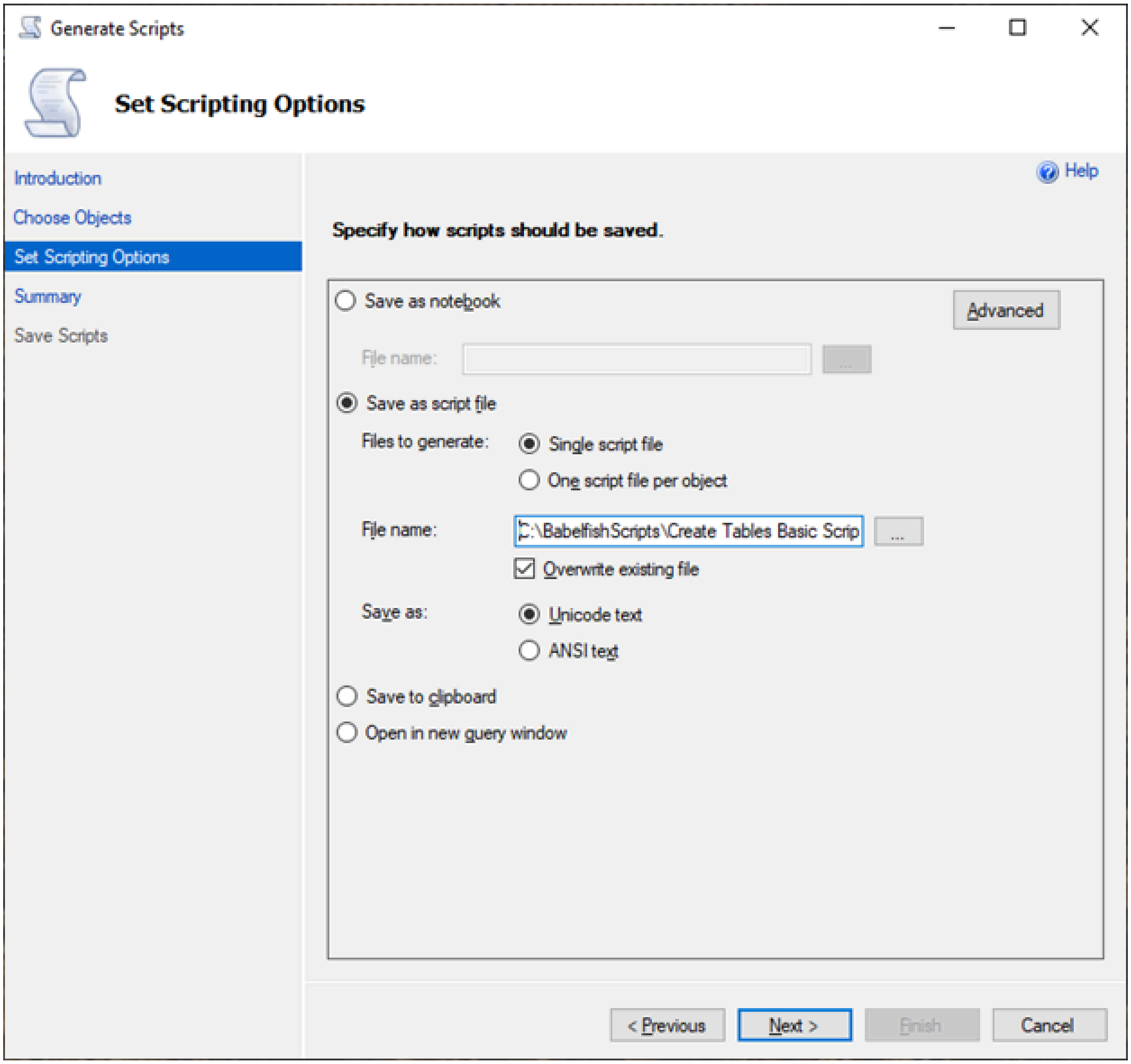


Script Collation	False
Script Defaults	False
Script DROP and CREATE	Script CREATE
Script Extended Properties	False
Script for Server Version	SQL Server 2019
Script for the database engine edition	Microsoft SQL Server Enterprise Edition
Script for the database engine type	Stand-alone instance
Script Logins	False
Script Object-Level Permissions	False
Script Owner	False
Script Statistics	Do not script statistics
Script USE DATABASE	True
Types of data to script	Schema only
Table/View Options	
Script Change Tracking	False
Script Check Constraints	False
Script Data Compression Options	False
Script Foreign Keys	False
Script Full-Text Indexes	False
Script Indexes	False
Script Primary Keys	True
Script Triggers	False
Script Unique Keys	True

Script Foreign Keys
Script the foreign keys for each table scripted.

6. Choose **OK**.

7. On the **Set Scripting Options** page, choose **Save as script file** and then choose the **Single script file** option. Enter your **File name**.



Specify how scripts should be saved.

○ Save as notebook

File name:

● Save as script file

Files to generate: ● Single script file

○ One script file per object

File name: C:\BabelfishScripts\Create Tables Basic Scrip

☑ Overwrite existing file

Save as: ● Unicode text

○ ANSI text

○ Save to clipboard

○ Open in new query window

8. Choose **Next** to view the **Summary wizard** page.

9. Choose **Next** to start the script generation.

   You can continue to generate scripts for the other object types in the wizard. Instead of choosing **Finish** after the file is saved, choose the **Previous** button three times to go back to the **Choose Objects** page. Then repeat the steps in the wizard to generate scripts for the other object types.

Database authentication with Babelfish for Aurora PostgreSQL

Babelfish for Aurora PostgreSQL supports two ways to authenticate database users. Password authentication is available by default for all Babelfish DB clusters. You can also add Kerberos authentication for the same DB cluster.

Topics

- Password authentication with Babelfish
- Kerberos authentication with Babelfish

Password authentication with Babelfish

Babelfish for Aurora PostgreSQL supports password authentication. Passwords are stored in encrypted form on disk. For more information about authentication on an Aurora PostgreSQL cluster, see Security with Amazon Aurora PostgreSQL.

You might be prompted for credentials each time you connect to Babelfish. Any user migrated to or created on Aurora PostgreSQL can use the same credentials on both the SQL Server port and the PostgreSQL port. Babelfish doesn't enforce password policies, but we recommend that you do the following:

- Require a complex password that's at least eight (8) characters long.
- Enforce a password expiration policy.

To review a complete list of database users, use the command `SELECT * FROM pg_user;`.

Kerberos authentication with Babelfish

Babelfish for Aurora PostgreSQL 15.2 version supports authentication to your DB cluster using Kerberos. This method allows you to use Microsoft Windows Authentication to authenticate users when they connect to your Babelfish database. To do so, you must first configure your DB cluster to use AWS Directory Service for Microsoft Active Directory for Kerberos authentication. For more information, see What is AWS Directory Service? in the *AWS Directory Service Administration Guide*.

Setting up Kerberos Authentication

Babelfish for Aurora PostgreSQL DB cluster can connect using two different ports, but Kerberos authentication setup is a one-time process. Therefore, you must first set up Kerberos authentication for your DB cluster. For more information, see Setting up Kerberos authentication. After completing the setup, ensure that you can connect with a PostgreSQL client using Kerberos. For more information, see Connecting with Kerberos Authentication.

Login and user provisioning in Babelfish

Windows logins created from the Tabular Data Stream (TDS) port can be used either with the TDS port or the PostgreSQL port. First, the login that can use Kerberos for authentication must be provisioned from the TDS port before it is used by the T-SQL users and applications to connect to a Babelfish database. When creating Windows logins, administrators can provide the login using either the DNS domain name or the NetBIOS domain name. Typically, NetBIOS domain is the subdomain of the DNS domain name. For example, if the DNS domain name is `CORP.EXAMPLE.COM`, then the NetBIOS domain can be `CORP`. If the NetBIOS domain name format is provided for a login, a mapping must exist to the DNS domain name.

Managing NetBIOS domain name to DNS domain name mapping

To manage mappings between the NetBIOS domain name and DNS domain name, Babelfish provides system stored procedures to add, remove, and truncate mappings. Only a user with a `sysadmin` role can run these procedures.

To create mapping between NetBIOS and DNS domain name, use the Babelfish provided system stored procedure `babelfish_add_domain_mapping_entry`. Both arguments must have a valid value and are not NULL.

Example

```
EXEC babelfish_add_domain_mapping_entry 'netbios_domain_name',
 'fully_qualified_domain_name'
```

The following example shows how to create the mapping between the NetBIOS name CORP and DNS domain name CORP.EXAMPLE.COM.

Example

```
EXEC babelfish_add_domain_mapping_entry 'corp', 'corp.example.com'
```

To delete an existing mapping entry, use the system stored procedure babelfish_remove_domain_mapping_entry.

Example

```
EXEC babelfish_remove_domain_mapping_entry 'netbios_domain_name'
```

The following example shows how to remove the mapping for the NetBIOS name CORP.

Example

```
EXEC babelfish_remove_domain_mapping_entry 'corp'
```

To remove all existing mapping entries, use the system stored procedure babelfish_truncate_domain_mapping_table:

Example

```
EXEC babelfish_truncate_domain_mapping_table
```

To view all mappings between NetBIOS and DNS domain name, use the following query.

Example

```
SELECT netbios_domain_name, fq_domain_name FROM babelfish_domain_mapping;
```

Managing Logins

Create logins

Connect to the DB through the TDS endpoint using a login that has the correct permissions. If there is no database user created for the login, then the login is mapped to guest user. If the guest user is not enabled, then the login attempt fails.

Create a Windows login using the following query. The `FROM WINDOWS` option allows authentication using Active Directory.

```
CREATE LOGIN login_name FROM WINDOWS [WITH DEFAULT_DATABASE=database]
```

Example

The following example shows creating a login for the Active Directory user [corp\test1] with a default database of db1.

```
CREATE LOGIN [corp\test1] FROM WINDOWS WITH DEFAULT_DATABASE=db1
```

This example assumes that there is a mapping between the NetBIOS domain CORP and the DNS domain name CORP.EXAMPLE.COM. If there is no mapping, then you must provide the DNS domain name [CORP.EXAMPLE.COM\test1].

> **Note**
>
> Logins based on Active Directory users, are limited to names of fewer than 21 characters.

Drop login

To drop a login, use the same syntax as for any login, as shown in the following example:

```
DROP LOGIN [DNS domain name\login]
```

Alter login

To alter a login, use the same syntax as for any login, as in the following example:

```
ALTER LOGIN [DNS domain name\login] { ENABLE|DISABLE|WITH DEFAULT_DATABASE=[master] }
```

The ALTER LOGIN command supports limited options for Windows logins, including the following:

- DISABLE – To disable a login. You can't use a disabled login for authentication.

- ENABLE – To enable a disabled login.
- DEFAULT_DATABASE – To change the default database of a login.

> **Note**
>
> All password management is performed through AWS Directory Service, so the ALTER LOGIN command doesn't allow database administrators to change or set passwords for Windows logins.

Connecting to Babelfish for Aurora PostgreSQL with Kerberos authentication

Typically, the database users who authenticate using Kerberos are doing so from their client machines. These machines are members of the Active Directory domain. They use Windows Authentication from their client applications to access the Babelfish for Aurora PostgreSQL server on the TDS port.

Connecting to Babelfish for Aurora PostgreSQL on the PostgreSQL port with Kerberos authentication

You can use logins created from the TDS port with either the TDS port or the PostgreSQL port. However, PostgreSQL uses case-sensitive comparisons by default for usernames. For Aurora PostgreSQL to interpret Kerberos usernames as case-insensitive, you must set the `krb_caseins_users` parameter as `true` in the custom Babelfish cluster parameter group. This parameter is set to `false` by default. For more information, see Configuring for case-insensitive user names. In addition, you must specify the login username in the format <login@DNS domain name> from the PostgreSQL client applications. You can't use <DNS domain name\login> format.

Frequently occurring errors

You can configure forest trust relationships between your on-premises Microsoft Active Directory and the AWS Managed Microsoft AD. For more information, see Create a trust relationship. Then, you must connect using a specialized domain specific endpoint instead of using the Amazon domain `rds.amazonaws.com` in the host endpoint. If you don't use the correct domain specific endpoint, you might encounter the following error:

```
Error: “Authentication method "NTLMSSP" not supported (Microsoft SQL Server, Error:
 514)"
```

This error occurs when the TDS client can't cache the service ticket for the supplied endpoint URL. For more information, see Connecting with Kerberos.

Connecting to a Babelfish DB cluster

To connect to Babelfish, you connect to the endpoint of the Aurora PostgreSQL cluster running Babelfish. Your client can use one of the following client drivers compliant with TDS version 7.1 through 7.4:

- Open Database Connectivity (ODBC)
- OLE DB Driver/MSOLEDBSQL
- Java Database Connectivity (JDBC) version 8.2.2 (mssql-jdbc-8.2.2) and higher
- Microsoft SqlClient Data Provider for SQL Server
- .NET Data Provider for SQL Server
- SQL Server Native Client 11.0 (deprecated)
- OLE DB Provider/SQLOLEDB (deprecated)

With Babelfish, you run the following:

- SQL Server tools, applications, and syntax on the TDS port, by default port 1433.
- PostgreSQL tools, applications, and syntax on the PostgreSQL port, by default port 5432.

To learn more about connecting to Aurora PostgreSQL in general, see Connecting to an Amazon Aurora PostgreSQL DB cluster.

Note

Third-party developer tools using the SQL Server OLEDB provider to access metadata aren't supported. We recommend you to use SQL Server JDBC, ODBC, or SQL Native client connections for these tools.

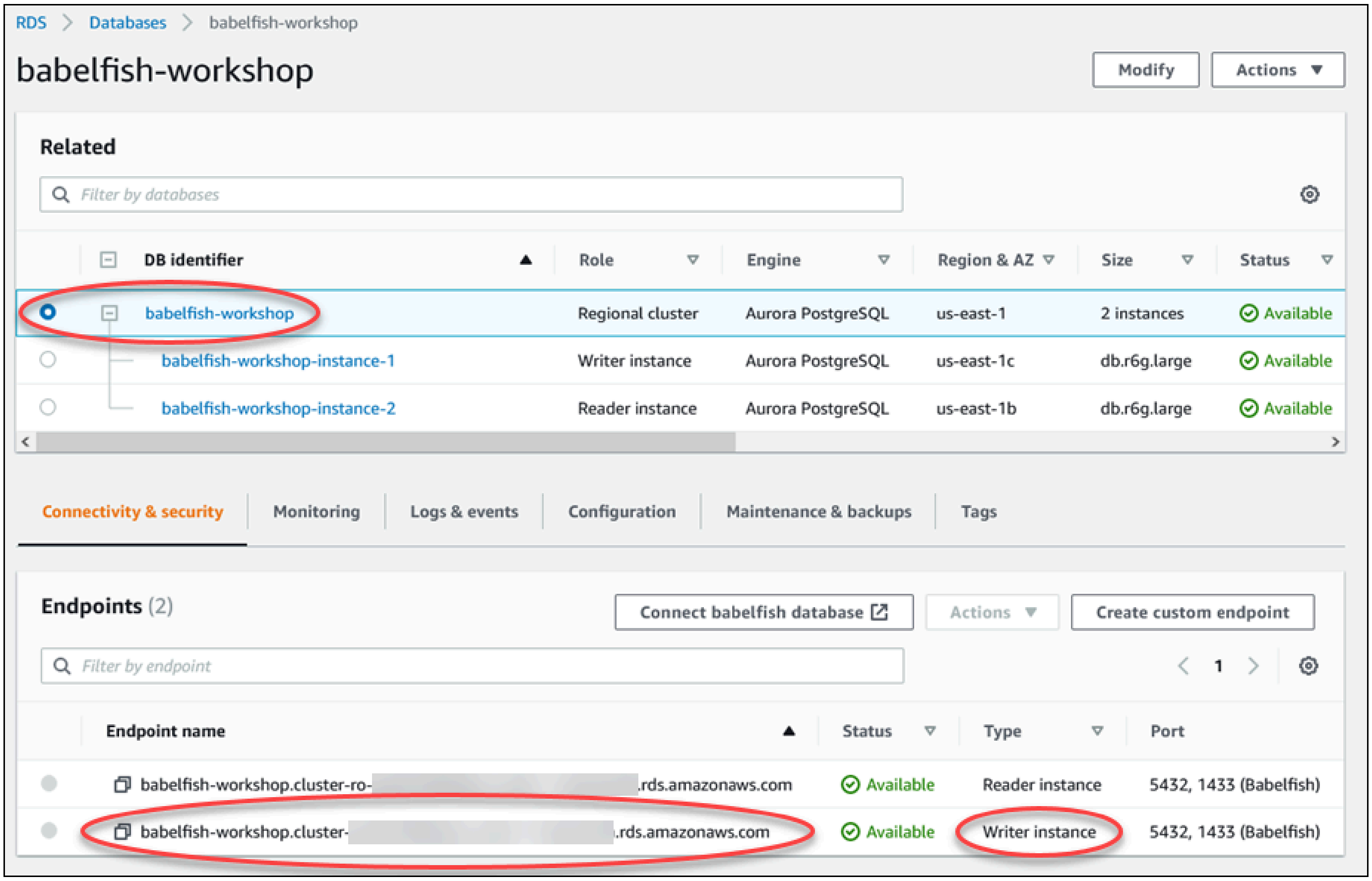
Topics

Finding the writer endpoint and port number

To connect to your Babelfish DB cluster, you use the endpoint associated with the DB cluster's writer (primary) instance. The instance must have a status of **Available**. It can take up to 20 minutes for the instances to be available after creating the Babelfish for Aurora PostgreSQL DB cluster.

To find your database endpoint

1. Open the console for Babelfish.
2. Choose **Databases** from the navigation pane.
3. Choose your Babelfish for Aurora PostgreSQL DB cluster from those listed to see its details.
4. On the **Connectivity & security** tab, note the available cluster **Endpoints** values. Use the cluster endpoint for the writer instance in your connection strings for any applications that perform database write or read operations.



	DB identifier	Role	Engine	Region & AZ	Size	Status
●	babelfish-workshop	Regional cluster	Aurora PostgreSQL	us-east-1	2 instances	✓ Available
○	babelfish-workshop-instance-1	Writer instance	Aurora PostgreSQL	us-east-1c	db.r6g.large	✓ Available
○	babelfish-workshop-instance-2	Reader instance	Aurora PostgreSQL	us-east-1b	db.r6g.large	✓ Available

	Endpoint name	Status	Type	Port
○	babelfish-workshop.cluster-ro-[redacted].rds.amazonaws.com	✓ Available	Reader instance	5432, 1433 (Babelfish)
○	babelfish-workshop.cluster-[redacted].rds.amazonaws.com	✓ Available	Writer instance	5432, 1433 (Babelfish)

For more information about Aurora DB cluster details, see Creating an Amazon Aurora DB cluster.

Creating C# or JDBC client connections to Babelfish

In the following you can find some examples of using C# and JDBC classes to connect to an Babelfish for Aurora PostgreSQL.

Example : Using C# code to connect to a DB cluster

```
string dataSource = 'babelfishServer_11_24';

//Create connection
connectionString = @"Data Source=" + dataSource
    +";Initial Catalog=your-DB-name"
    +";User ID=user-id;Password=password";

SqlConnection cnn = new SqlConnection(connectionString);
cnn.Open();
```

Example : Using generic JDBC API classes and interfaces to connect to a DB cluster

```
String dbServer =
   "database-babelfish.cluster-123abc456def.us-east-1-rds.amazonaws.com";
String connectionUrl = "jdbc:sqlserver://" + dbServer + ":1433;" +
    "databaseName=your-DB-name;user=user-id;password=password";

// Load the SQL Server JDBC driver and establish the connection.
System.out.print("Connecting Babelfish Server ... ");
Connection cnn = DriverManager.getConnection(connectionUrl);
```

Example : Using SQL Server-specific JDBC classes and interfaces to connect to a DB cluster

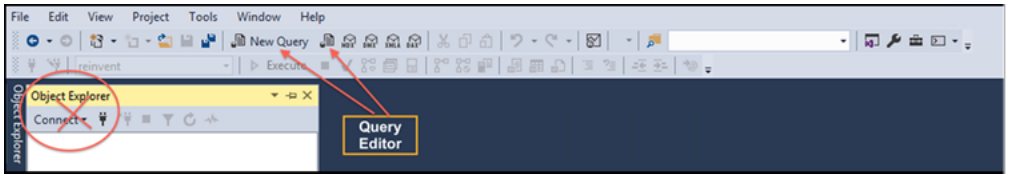
```
// Create datasource.
SQLServerDataSource ds = new SQLServerDataSource();
ds.setUser("user-id");
ds.setPassword("password");
String babelfishServer =
   "database-babelfish.cluster-123abc456def.us-east-1-rds.amazonaws.com";

ds.setServerName(babelfishServer);
ds.setPortNumber(1433);
ds.setDatabaseName("your-DB-name");

Connection con = ds.getConnection();
```

Using a SQL Server client to connect to your DB cluster

You can use a SQL Server client to connect with Babelfish on the TDS port. As of Babelfish 2.1.0 and higher releases, you can use the SSMS Object Explorer or the SSMS Query Editor to connect to your Babelfish cluster.

Limitations

- In Babelfish 2.1.0 and older versions, using PARSE to check SQL syntax doesn't work as it should. Rather than checking the syntax without running the query, the PARSE command runs the query but doesn't display any results. Using the SMSS <Ctrl><F5> key combination to check syntax has the same anomalous behavior, that is, Babelfish unexpectedly runs the query without providing any output.
- Babelfish doesn't support MARS (Multiple Active Result Sets). Be sure that any client applications that you use to connect to Babelfish aren't set to use MARS.
- For Babelfish 1.3.0 and older versions, only the Query Editor is supported for SSMS. To use SSMS with Babelfish, be sure to open the Query Editor connection dialog in SSMS, and not the Object Explorer. If the Object Explorer dialog does open, cancel the dialog and re-open the Query Editor. In the following image, you can find the menu options to choose when connecting to Babelfish 1.3.0 or older versions.



For more information about interoperability and behavioral differences between SQL Server and Babelfish, see Differences between Babelfish for Aurora PostgreSQL and SQL Server.

Using sqlcmd to connect to the DB cluster

You can connect to and interact with an Aurora PostgreSQL DB cluster that supports Babelfish by only using version 19.1 and earlier SQL Server `sqlcmd` command line client. SSMS version 19.2 isn't supported to connect to a Babelfish cluster. Use the following command to connect.

```
sqlcmd -S endpoint,port -U login-id -P password -d your-DB-name
```

The options are as follows:

- -S is the endpoint and (optional) TDS port of the DB cluster.
- -U is the login name of the user.
- -P is the password associated with the user.
- -d is the name of your Babelfish database.

After connecting, you can use many of the same commands that you use with SQL Server. For some examples, see Getting information from the Babelfish system catalog.

Using SSMS to connect to the DB cluster

You can connect to an Aurora PostgreSQL DB cluster running Babelfish by using Microsoft SQL Server Management Studio (SSMS). SSMS includes a variety of tools, including the SQL Server Import amd Export Wizard discussed in Migrating a SQL Server database to Babelfish for Aurora PostgreSQL. For more information about SSMS, see Download SQL Server Management Studio (SSMS) in the Microsoft documentation.

To connect to your Babelfish database with SSMS

1. Start SSMS.
2. Open the **Connect to Server** dialog box. To continue with the connection, do one of the following:
   - Choose **New Query**.
   - If the Query Editor is open, choose **Query**, **Connection**, **Connect**.
3. Provide the following information for your database:

   a. For **Server type**, choose **Database Engine**.

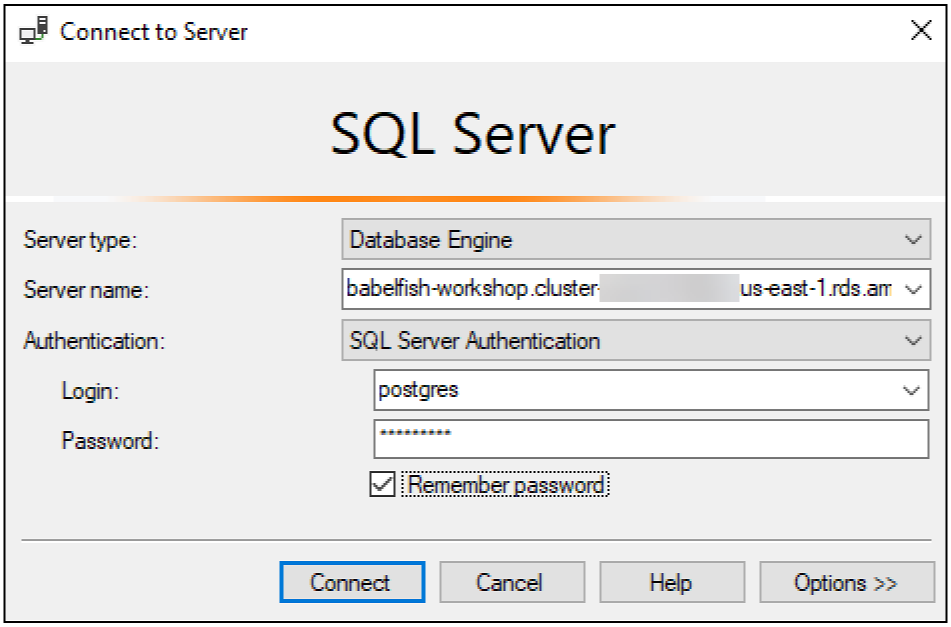
   b. For **Server name**, enter the DNS name. For example, your server name should look similar to the following.

   ```
   cluster-name.cluster-555555555555.aws-region.rds.amazonaws.com,1433
   ```

   c. For **Authentication**, choose **SQL Server Authentication**.

   d. For **Login**, enter the user name that you chose when you created your database.

   e. For **Password**, enter the password that you chose when you created your database.



Server type: Database Engine

Server name: babelfish-workshop.cluster-[redacted]us-east-1.rds.am

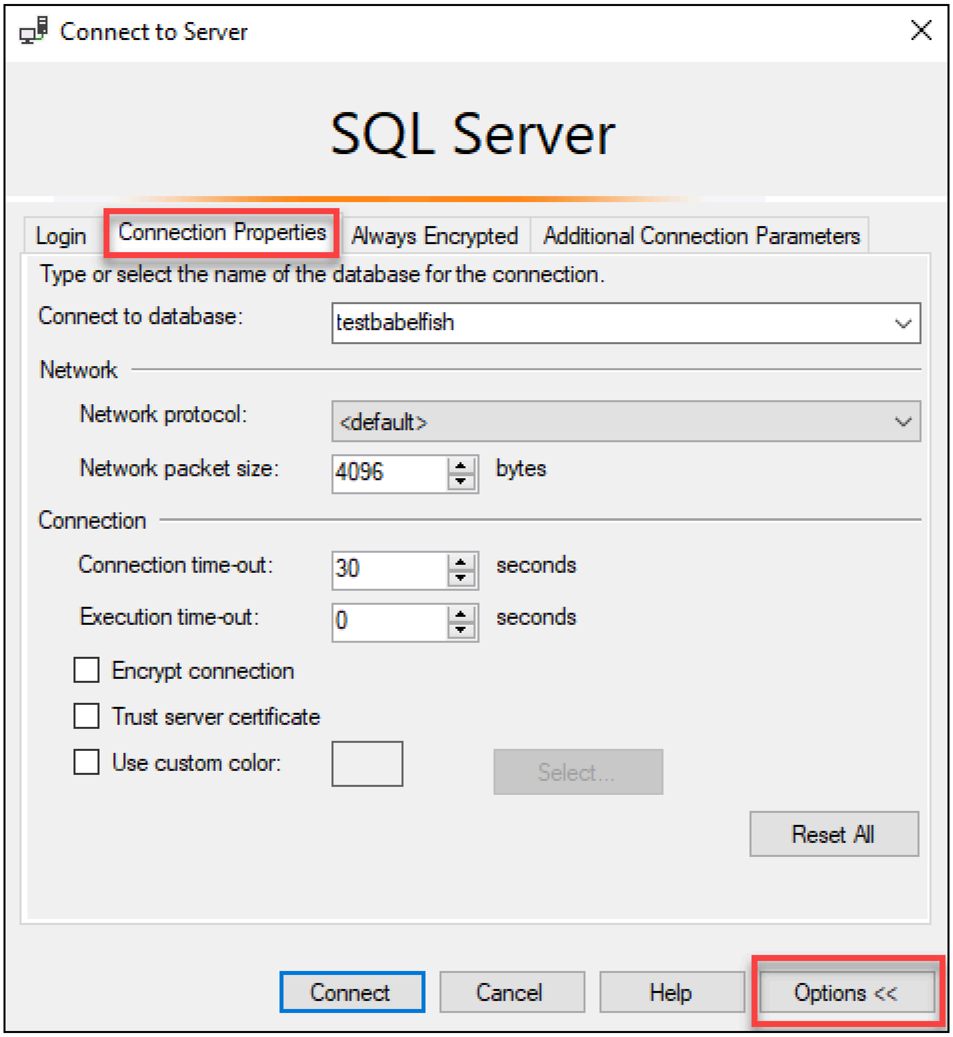
Authentication: SQL Server Authentication

Login: postgres

Password: **********

☑ Remember password

4. (Optional) Choose **Options**, and then choose the **Connection Properties** tab.



Type or select the name of the database for the connection.

Connect to database: testbabelfish

Network protocol: <default>

Network packet size: 4096 bytes

Connection time-out: 30 seconds

Execution time-out: 0 seconds

□ Encrypt connection

□ Trust server certificate

□ Use custom color:

5. (Optional) For **Connect to database**, specify the name of the migrated SQL Server database to connect to, and choose **Connect**.

   If a message appears indicating that SSMS can't apply connection strings, choose **OK**.

   If you are having trouble connecting to Babelfish, see Connection failure.

   For more information about SQL Server connection issues, see Troubleshooting connections to your SQL Server DB instance in the *Amazon RDS User Guide*.

Using a PostgreSQL client to connect to your DB cluster

You can use a PostgreSQL client to connect to Babelfish on the PostgreSQL port.

Using psql to connect to the DB cluster

You can download the PostgreSQL client from the PostgreSQL website. Follow the instructions specific to your operating system version to install psql.

You can query an Aurora PostgreSQL DB cluster that supports Babelfish with the `psql` command line client. When connecting, use the PostgreSQL port (by default, port 5432). Typically, you don't need to specify the port number unless you changed it from the default. Use the following command to connect to Babelfish from the `psql` client:

```
psql -h bfish-db.cluster-123456789012.aws-region.rds.amazonaws.com
-p 5432 -U postgres -d babelfish_db
```

The parameters are as follows:

- `-h` – The host name of the DB cluster (cluster endpoint) that you want to access.
- `-p` – The PostgreSQL port number used to connect to your DB instance.
- `-d` – The database that you want to connect to. The default is `babelfish_db`.
- `-U` – The database user account that you want to access. (The example shows the default master username.)

When you run a SQL command on the psql client, you end the command with a semicolon. For example, the following SQL command queries the pg_tables system view to return information about each table in the database.

```
SELECT * FROM pg_tables;
```

The psql client also has a set of built-in metacommands. A *metacommand* is a shortcut that adjusts formatting or provides a shortcut that returns meta-data in an easy-to-use format. For example, the following metacommand returns similar information to the previous SQL command:

```
\d
```

Metacommands don't need to be terminated with a semicolon (;).

To exit the psql client, enter \q.

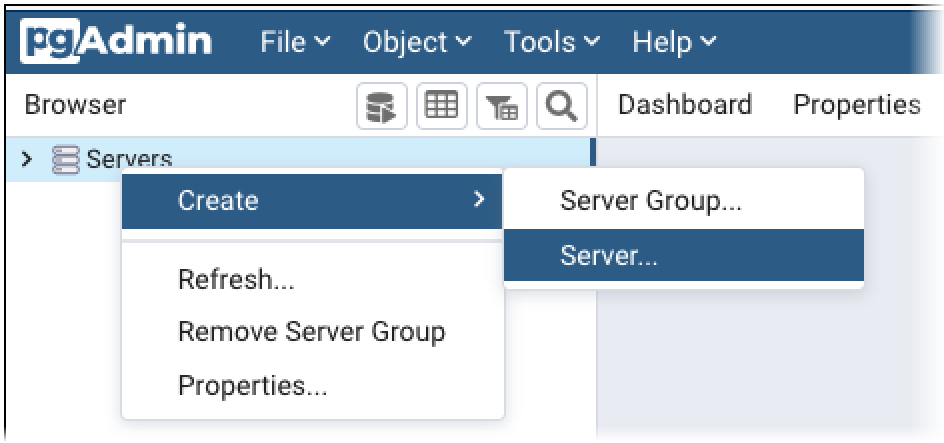
For more information about using the psql client to query an Aurora PostgreSQL cluster, see the PostgreSQL documentation.

Using pgAdmin to connect to the DB cluster

You can use the pgAdmin client to access your data in native PostgreSQL dialect.

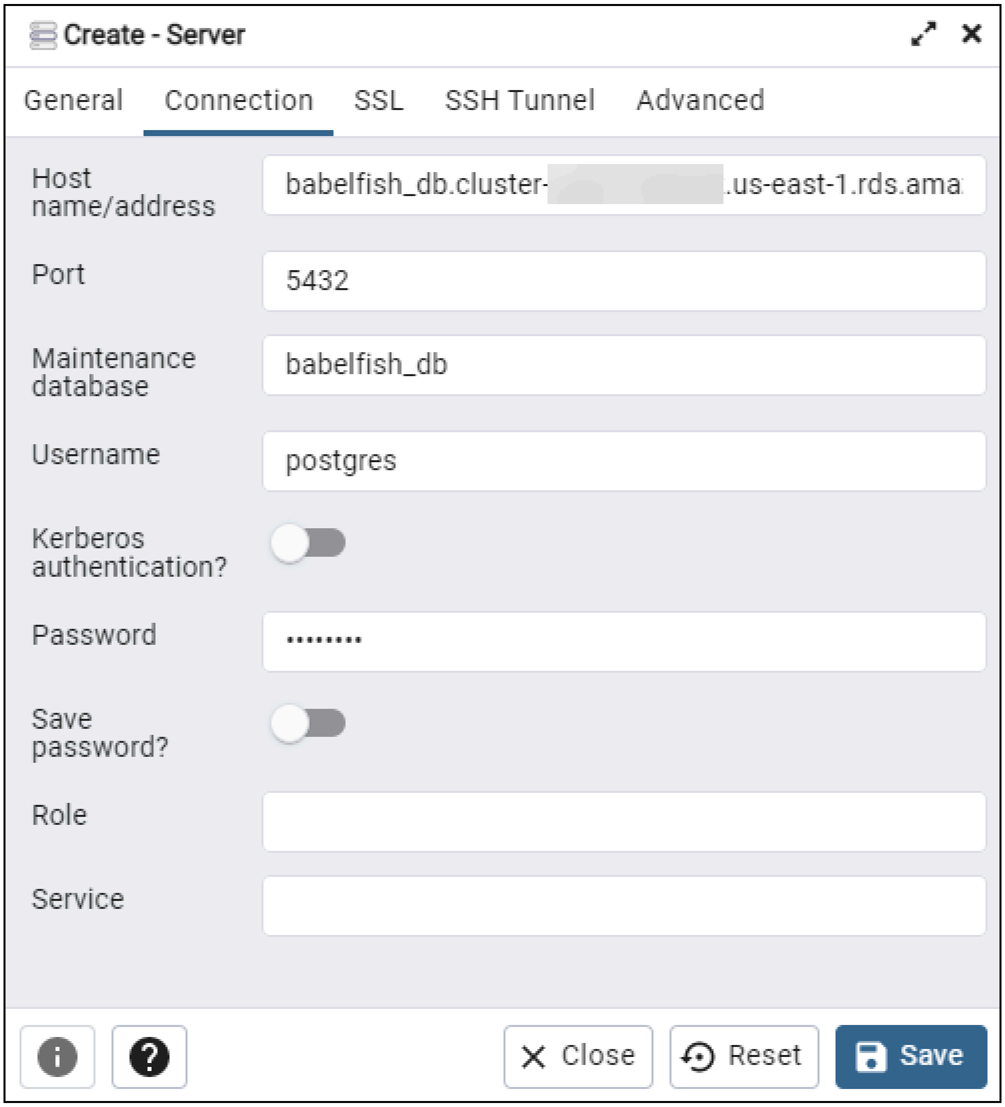
To connect to the cluster with the pgAdmin client

1. Download and install the pgAdmin client from the pgAdmin website.
2. Open the client and authenticate with pgAdmin.
3. Open the context (right-click) menu for **Servers**, and then choose **Create**, **Server**.



4. Enter information in the **Create - Server** dialog box.

   On the **Connection** tab, add the Aurora PostgreSQL cluster address for **Host** and the PostgreSQL port number (by default, 5432) for **Port**. Provide authentication details, and choose **Save**.



Host name/address: babelfish_db.cluster-[redacted].us-east-1.rds.ama

Port: 5432

Maintenance database: babelfish_db

Username: postgres

Kerberos authentication?: ○

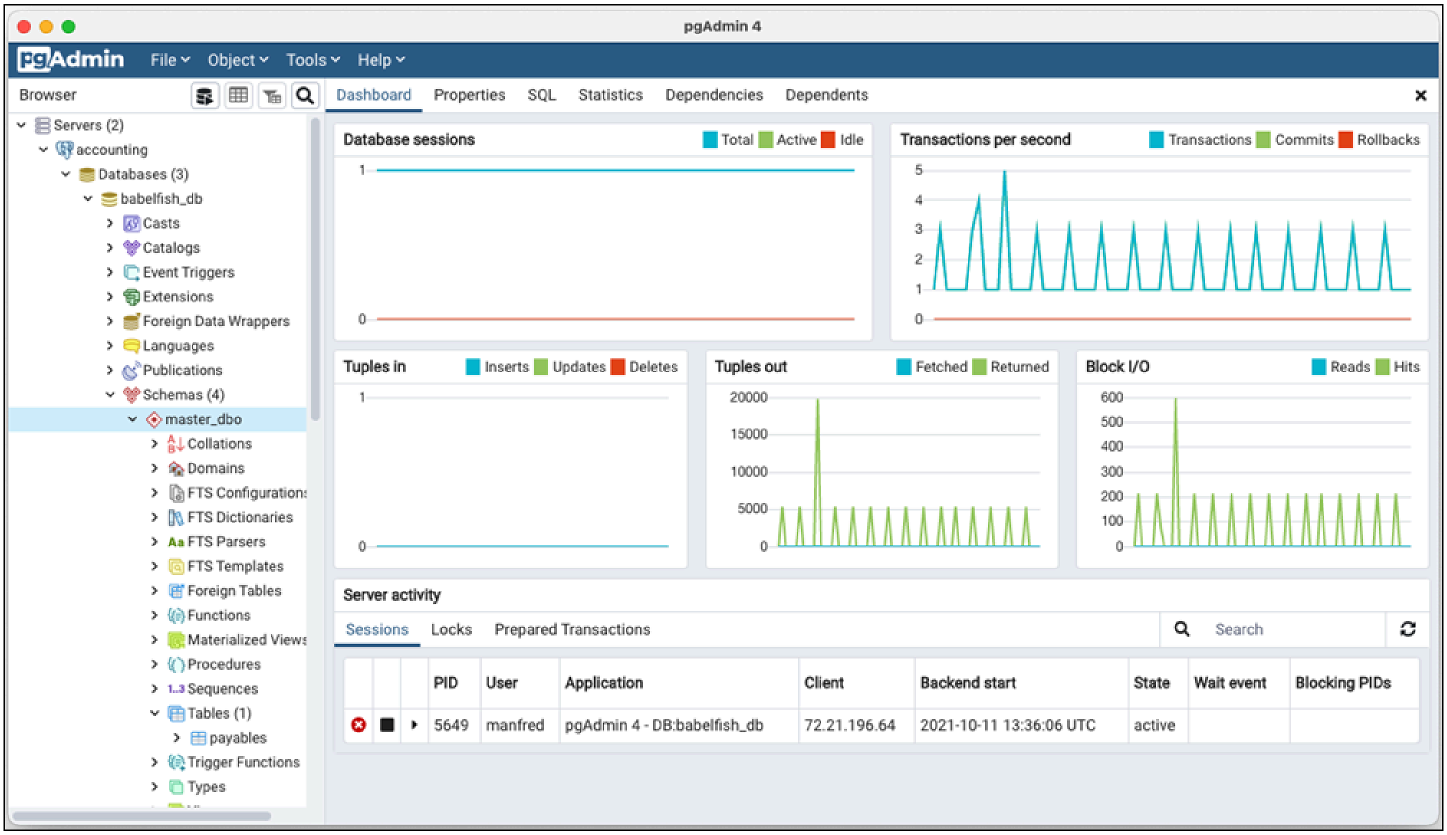
Password: ••••••••

Save password?: ○

Role:

Service:

After connecting, you can use pgAdmin functionality to monitor and manage your Aurora PostgreSQL cluster on the PostgreSQL port.



			PID	User	Application	Client	Backend start	State	Wait event	Blocking PIDs
☒	■	▸	5649	manfred	pgAdmin 4 - DB:babelfish_db	72.21.196.64	2021-10-11 13:36:06 UTC	active		

To learn more, see the pgAdmin web page.

Working with Babelfish

Following, you can find usage information for Babelfish, including some of the differences between working with Babelfish and SQL Server, and between Babelfish and PostgreSQL databases.

Topics

Getting information from the Babelfish system catalog

You can obtain information about the database objects that are stored in your Babelfish cluster by querying many of the same system views as used in SQL Server. Each new release of Babelfish adds support for more system views. For a list of available views currently available, see the SQL Server system catalog views table.

These system views provide information from the system catalog (`sys.schemas`). In the case of Babelfish, these views contain both SQL Server and PostgreSQL system schemas. To query Babelfish for system catalog information, you can use the TDS port or the PostgreSQL port, as shown in the following examples.

- **Query the T-SQL port using `sqlcmd` or another SQL Server client**.

  ```
  1> SELECT * FROM sys.schemas
  2> GO
  ```

  This query returns SQL Server and Aurora PostgreSQL system schemas, as shown in the following.

  ```
  name
  ---------------------------------------------------------
  ```

```
demographic_dbo
public
sys
master_dbo
tempdb_dbo
...
```

- **Query the PostgreSQL port using `psql` or `pgAdmin`**. This example uses the `psql` list schemas metacommand (\dn):

  ```
  babelfish_db=> \dn
  ```

  The query returns the same result set as that returned by `sqlcmd` on the T-SQL port.

  ```
          List of schemas
               Name
  ------------------------------

   demographic_dbo

   public
   sys
   master_dbo
   tempdb_dbo
  ...
  ```

SQL Server system catalogs available in Babelfish

In the following table you can find the SQL Server views currently implemented in Babelfish. For more information about the system catalogs in SQL Server, see [System Catalog Views (Transact-SQL)]() in Microsoft documentation.

View name	Description or Babelfish limitation (if any)
`sys.all_columns`	All columns in all tables and views
`sys.all_objects`	All objects in all schemas
`sys.all_sql_modules`	The union of `sys.sql_modules` and `sys.system_sql_modules`

View name	Description or Babelfish limitation (if any)
`sys.all_views`	All views in all schemas
`sys.columns`	All columns in user-defined tables and views
`sys.configurations`	Babelfish support limited to a single read-only configuration.
`sys.data_spaces`	Contains a row for each data space. This can be a filegroup, partition scheme, or FILESTREAM data filegroup.
`sys.database_files`	A per-database view that contains one row for each file of a database as stored in the database itself.
`sys.database_mirroring`	For information, see sys.database_mirroring in Microsoft Transact-SQL documentation.
`sys.database_principals`	For information, see sys.database_principals in Microsoft Transact-SQL documentation.
`sys.database_role_members`	For information, see sys.database_role_members in Microsoft Transact-SQL documentation.
`sys.databases`	All databases in all schemas
`sys.dm_exec_connections`	For information, see sys.dm_exec_connections in Microsoft Transact-SQL documentation.
`sys.dm_exec_sessions`	For information, see sys.dm_exec_sessions in Microsoft Transact-SQL documentation.
`sys.dm_hadr_database_replica_states`	For information, see sys.dm_hadr_database_replica_states in Microsoft Transact-SQL documentation.

View name	Description or Babelfish limitation (if any)
`sys.dm_os_host_info`	For information, see sys.dm_os_host_info in Microsoft Transact-SQL documentation.
`sys.endpoints`	For information, see sys.endpoints in Microsoft Transact-SQL documentation.
`sys.indexes`	For information, see sys.indexes in Microsoft Transact-SQL documentation.
`sys.languages`	For information, see sys.languages in Microsoft Transact-SQL documentation.
`sys.schemas`	All schemas
`sys.server_principals`	All logins and roles
`sys.sql_modules`	For information, see sys.sql_modules in Microsoft Transact-SQL documentation.
`sys.sysconfigures`	Babelfish support limited to a single read-only configuration.
`sys.syscurconfigs`	Babelfish support limited to a single read-only configuration.
`sys.sysprocesses`	For information, see sys.sysprocesses in Microsoft Transact-SQL documentation.
`sys.system_sql_modules`	For information, see sys.system_sql_modules in Microsoft Transact-SQL documentation.
`sys.table_types`	For information, see sys.table_types in Microsoft Transact-SQL documentation.
`sys.tables`	All tables in a schema

View name	Description or Babelfish limitation (if any)
`sys.xml_schema_collections`	For information, see sys.xml_schema_collections in Microsoft Transact-SQL documentation.

PostgreSQL implements system catalogs that are similar to the SQL Server object catalog views. For a complete list of system catalogs, see System Catalogs in the PostgreSQL documentation.

DDL exports supported by Babelfish

From Babelfish 2.4.0 and 3.1.0 versions, Babelfish supports DDL exports using various tools. For example, you can use this functionality from SQL Server Management Studio (SSMS) to generate the data definition scripts for various objects in a Babelfish for Aurora PostgreSQL database. You can then use the generated DDL commands in this script to create the same objects in another Babelfish for Aurora PostgreSQL or SQL Server database.

Babelfish supports DDL exports for the following objects in the specified versions.

List of objects	2.4.0	3.1.0
User tables	Yes	Yes
Primary keys	Yes	Yes
Foreign keys	Yes	Yes
Unique constraints	Yes	Yes
Indexes	Yes	Yes
Check constraints	Yes	Yes
Views	Yes	Yes
Stored procedures	Yes	Yes
User-defined functions	Yes	Yes
Table-valued functions	Yes	Yes

List of objects	2.4.0	3.1.0
Triggers	Yes	Yes
User Defined Datatypes	No	No
User Defined Table Types	No	No
Users	No	No
Logins	No	No
Sequences	No	No
Roles	No	No

Limitations with the exported DDLs

- **Use escape hatches before recreating the objects with the exported DDLs –** Babelfish doesn't support all the commands in the exported DDL script. Use escape hatches to avoid errors caused when recreating the objects from the DDL commands in Babelfish. For more information on escape hatches, see Managing Babelfish error handling with escape hatches
- **Objects containing CHECK constraints with explicit COLLATE clauses –** The scripts with these objects generated from a SQL Server database have different but equivalent collations as in the Babelfish database. For example, a few collations, such as sql_latin1_general_cp1_cs_as, sql_latin1_general_cp1251_cs_as, and latin1_general_cs_as are generated as latin1_general_cs_as, which is the closest Windows collation.

Differences between Babelfish for Aurora PostgreSQL and SQL Server

Babelfish is an evolving Aurora PostgreSQL feature, with new functionality added in each release since the initial offering in Aurora PostgreSQL 13.4. It's designed to provide T-SQL semantics on top of PostgreSQL through the T-SQL dialect using the TDS port. Each new version of Babelfish adds features and functions that better align with T-SQL functionality and behavior, as shown in the Supported functionality in Babelfish by version table. For best results when working with Babelfish, we recommend that you understand the differences that currently exist between the T-SQL supported by SQL Server and Babelfish for the latest version. To learn more, see T-SQL differences in Babelfish.

In addition to the differences between T-SQL supported by Babelfish and SQL Server, you might also need to consider interoperability issues between Babelfish and PostgreSQL in the context of the Aurora PostgreSQL DB cluster. As mentioned previously, Babelfish supports T-SQL semantics on top of PostgreSQL through the T-SQL dialect using the TDS port. At the same time, the Babelfish database can also be accessed through the standard PostgreSQL port with PostgreSQL SQL statements. If you're considering using both PostgreSQL and Babelfish functionality in a production deployment, you need to be aware of the potential interoperability issues between schema names, identifiers, permissions, transactional semantics, multiple result sets, default collations, and so on. In simple terms, when PostgreSQL statements or PostgreSQL access occur in the context of Babelfish, interference between PostgreSQL and Babelfish can occur and can potentially affecting syntax, semantics, and compatibility when new versions of Babelfish are released. For complete information and guidance about all the considerations, see the Guidance on Babelfish Interoperability in the Babelfish for PostgreSQL documentation.

Note

Before using both PostgreSQL native functionality and Babelfish functionality in the same application context, we strongly recommend that you consider the issues discussed in the Guidance on Babelfish Interoperability in the Babelfish for PostgreSQL documentation. These interoperability issues (Aurora PostgreSQL and Babelfish) are relevant only if you plan to use the PostgreSQL database instance in the same application context as Babelfish.

Topics

Babelfish dump and restore

Starting with version 4.0.0 and 3.4.0, Babelfish users can now utilize the dump and restore utilities to backup and restore their databases. For more information, see Babelfish dump and restore. This feature is built on top of PostgreSQL dump and restore utilities. For more information, see pg_dump and see pg_restore. In order to effectively use this feature in Babelfish, you need to use PostgreSQL-based tools that are specifically adapted for Babelfish. The backup and restore feature for Babelfish differs significantly from that of SQL Server. For more information on these differences, see Dump and restore functionality differences : Babelfish and SQL Server. Babelfish for Aurora PostgreSQL provides additional capabilities for backing up and restoring Amazon Aurora PostgreSQL DB clusters. For more information, see Backing up and restoring an Amazon Aurora DB cluster.

T-SQL differences in Babelfish

Following, you can find a table of T-SQL functionality as supported in the current release of Babelfish with some notes about differences in the behavior from that of SQL Server.

For more information about support in various versions, see Supported functionality in Babelfish by version. For information about features that currently aren't supported, see Unsupported functionality in Babelfish.

Babelfish is available with Aurora PostgreSQL-Compatible Edition. For more information about Babelfish releases, see the *Release Notes for Aurora PostgreSQL*.

Functionality or syntax	Description of behavior or difference
\ (line continuation character)	The line continuation character (a backslash prior to a newline) for character and hexadecimal strings isn't currently supported . For character strings, the backslash-newline is interpreted as characters in the string. For hexadecimal strings, backslash-newline results in a syntax error.
@@version	The format of the value returned by `@@version` is slightly different from the value returned by SQL Server. Your code might not work correctly if it depends on the formatting of `@@version` .
Aggregate functions	Aggregate functions are partially supported (AVG, COUNT, COUNT_BIG, GROUPING, MAX, MIN, STRING_AGG, and SUM are supported). For a list of unsupported aggregate functions, see Functions that aren't supported.
ALTER TABLE	Supports adding or dropping a single column or constraint only.

Functionality or syntax	Description of behavior or difference
ALTER TABLE..ALTER COLUMN	NULL and NOT NULL can't currently be specified. To change the nullability of a column, use the postgreSQL statement ALTER TABLE..{SET|DROP} NOT NULL.
Blank column names with no column alias	The `sqlcmd` and `psql` utilities handle columns with blank names differently: - SQL Server `sqlcmd` returns a blank column name. - PostgreSQL `psql` returns a generated column name.
CHECKSUM function	Babelfish and SQL Server use different hashing algorithm s for the CHECKSUM function. As a result, the hash values generated by CHECKSUM function in Babelfish might be different from those generated by CHECKSUM function in SQL Server.
Column default	When creating a column default, the constraint name is ignored. To drop a column default, use the following syntax: `ALTER TABLE...ALTER COLUMN..DROP DEFAULT...`
Constraints	PostgreSQL doesn't support turning on and turning off individual constraints. The statement is ignored and a warning is raised.
Constraints created with DESC (descending) columns	Constraints are created with ASC (ascending) columns.
Constraints with IGNORE_DUP_KEY	Constraints are created without this property.

Functionality or syntax	Description of behavior or difference
CREATE, ALTER, DROP SERVER ROLE	ALTER SERVER ROLE is supported only for `sysadmin`. All other syntax is unsupported. The T-SQL user in Babelfish has an experience that is similar to SQL Server for the concepts of a login (server principal), a database, and a database user (database principal).
CREATE, ALTER LOGIN clauses are supported with limited syntax	The CREATE LOGIN... PASSWORD clause, ...DEFAULT_DATABASE clause, and ...DEFAULT_LANGUAGE clause are supported. The ALTER LOGIN... PASSWORD clause is supported, but ALTER LOGIN... OLD_PASSWORD clause isn't supported. Only a login that is a sysadmin member can modify a password.
CREATE DATABASE case-sensitive collation	Case-sensitive collations aren't supported with the CREATE DATABASE statement.
CREATE DATABASE keywords and clauses	Options except COLLATE and CONTAINMENT=NONE aren't supported. The COLLATE clause is accepted and is always set to the value of `babelfishpg_tsql.server_collation_name` .
CREATE SCHEMA... supporting clauses	You can use the CREATE SCHEMA command to create an empty schema. Use additional commands to create schema objects.
Database ID values are different on Babelfish	The master and tempdb databases won't be database IDs 1 and 2.
FORMAT date type function is supported with the following limitations	Single character meridian isn't supported. "yyy" format in SQL server returns 4 digits for year above 1000, but only 3 digits for others. "g" and "R" formats aren't supported "vi-VN" locale translation is slightly different.

Functionality or syntax	Description of behavior or difference
Identifiers exceeding 63 characters	PostgreSQL supports a maximum of 63 characters for identifiers. Babelfish converts identifiers longer than 63 characters to a name that includes a hash of the original name. For example, a table created as "AB(ABC1234567890123456789012345678901234567890123456789012345678901234567890" might be converted to "ABC123456789012345678901234567890123456789012345678901234567890".
IDENTITY columns support	`IDENTITY` columns are supported for data types `tinyint`, `smallint`, `int`, `bigint`. `numeric`, and `decimal`. SQL Server supports precision to 38 places for data types `numeric` and `decimal` in IDENTITY columns. PostgreSQL supports precision to 19 places for data types `numeric` and `decimal` in IDENTITY columns.
Indexes with IGNORE_DUP_KEY	Syntax that creates an index that includes IGNORE_DUP_KEY creates an index as if this property is omitted.
Indexes with more than 32 columns	An index can't include more than 32 columns. Included index columns count toward the maximum in PostgreSQL but not in SQL Server.
Indexes (clustered)	Clustered indexes are created as if NONCLUSTERED was specified.
Index clauses	The following clauses are ignored: FILLFACTOR, ALLOW_PAGE_LOCKS, ALLOW_ROW_LOCKS, PAD_INDEX, STATISTICS_NORECOMPUTE, OPTIMIZE_FOR_SEQUENTIAL_KEY, SORT_IN_TEMPDB, DROP_EXISTING, ONLINE, COMPRESSION_DELAY, MAXDOP, and DATA_COMPRESSION
JSON support	Order of the name-value pairs isn't guaranteed. But the array type remains unaffected.

Functionality or syntax	Description of behavior or difference
LOGIN objects	All options for LOGIN objects are not supported except for PASSWORD, DEFAULT_DATABASE, DEFAULT_LANGUAGE, ENABLE, DISABLE.
NEWSEQUENTIALID function	Implemented as NEWID; sequential behavior isn't guaranteed. When calling `NEWSEQUENTIALID` , PostgreSQL generates a new GUID value.
OUTPUT clause is supported with the following limitations	OUTPUT and OUTPUT INTO aren't supported in the same DML query. References to non-target table of UPDATE or DELETE operations in an OUTPUT clause aren't supported. OUTPUT... DELETED *, INSERTED * aren't supported in the same query.
Procedure or function parameter limit	Babelfish supports a maximum of 100 parameters for a procedure or function.
ROWGUIDCOL	This clause is currently ignored. Queries referencing `$GUIDGOL` cause a syntax error.
SEQUENCE object support	SEQUENCE objects are supported for the data types tinyint, smallint, int, bigint, numeric, and decimal. Aurora PostgreSQL supports precision to 19 places for data types numeric and decimal in a SEQUENCE.
Server-level roles	The `sysadmin` server-level role is supported. Other server-level roles (other than `sysadmin`) aren't supported.
Database-level roles other than `db_owner`	The `db_owner` database-level roles and user-defined database-level roles are supported. Other database-level roles (other than db_owner) aren't supported.
SQL keyword `SPARSE`	The keyword SPARSE is accepted and ignored.
SQL keyword clause `ON filegroup`	This clause is currently ignored.

Functionality or syntax	Description of behavior or difference
SQL keywords `CLUSTERED` and `NONCLUSTERED` for indexes and constraints	Babelfish accepts and ignores the `CLUSTERED` and `NONCLUSTERED` keywords.
`sysdatabases.cmptlevel`	`sysdatabases.cmptlevel` is always set to 120.
tempdb isn't reinitialized at restart	Permanent objects (like tables and procedures) created in tempdb aren't removed when the database is restarted.
TEXTIMAGE_ON filegroup	Babelfish ignores the `TEXTIMAGE_ON` *`filegroup`* clause.
Time precision	Babelfish supports 6-digit precision for fractional seconds. No adverse effects are anticipated with this behavior.
Transaction isolation levels	READUNCOMMITTED is treated the same as READCOMMITTED.
Virtual computed columns (non-persistent)	Virtual computed columns are created as persistent.
Without SCHEMABINDING clause	This clause isn't supported in functions, procedures, triggers, or views. The object is created, but as if WITH SCHEMABINDING was specified.

Transaction Isolation Levels in Babelfish

Babelfish supports Transaction Isolation Levels READ UNCOMMITTED, READ COMMITTED and SNAPSHOT. Starting from Babelfish 3.4 version additional Isolation Levels REPEATABLE READ and SERIALIZABLE are supported. All the Isolation Levels in Babelfish are supported with the behavior of corresponding Isolation Levels in PostgreSQL. SQL Server and Babelfish use different underlying mechanisms for implementing Transaction Isolation Levels (blocking for concurrent access, locks held by transactions, error handling etc). And, there are some subtle differences in how concurrent access may work out for different workloads. For more information on this PostgreSQL behavior, see Transaction Isolation.

Topics

Overview of the Transaction Isolation Levels

The original SQL Server Transaction Isolation Levels are based on pessimistic locking where only one copy of data exists and queries must lock resources such as rows before accessing them. Later, a variation of the Read Committed Isolation Level was introduced. This enables the use of row versions to provide better concurrency between readers and writers using non-blocking access. In addition, a new Isolation Level called Snapshot is available. It also uses row versions to provide better concurrency than REPEATABLE READ Isolation Level by avoiding shared locks on read data that are held till the end of the transaction.

Unlike SQL Server, all Transaction Isolation Levels in Babelfish are based on optimistic Locking (MVCC). Each transaction sees a snapshot of the data either at the beginning of the statement (READ COMMITTED) or at the beginning of the transaction (REPEATABLE READ, SERIALIZABLE), regardless of the current state of the underlying data. Therefore, the execution behavior of concurrent transactions in Babelfish might differ from SQL Server.

For example, consider a transaction with Isolation Level SERIALIZABLE that is initially blocked in SQL Server but succeeds later. It may end up failing in Babelfish due to a serialization conflict with a concurrent transaction that reads or updates the same rows. There could also be cases where executing multiple concurrent transactions yields a different final result in Babelfish as compared to SQL Server. Applications that use Isolation Levels, should be thoroughly tested for concurrency scenarios.

Isolation Levels in SQL Server	Babelfish Isolation Level	PostgreSQL Isolation Level	Comments
READ UNCOMMITTED	READ UNCOMMITTED	READ UNCOMMITTED	Read Uncommitted is same as Read Committed in Babelfish/PostgreSQL
READ COMMITTED	READ COMMITTED	READ COMMITTED	SQL Server Read Committed is pessimistic locking based, Babelfish Read Committed is snapshot (MVCC) based.
READ COMMITTED SNAPSHOT	READ COMMITTED	READ COMMITTED	Both are snapshot (MVCC) based but not exactly same.
SNAPSHOT	SNAPSHOT	REPEATABLE READ	Exactly same.
REPEATABLE READ	REPEATABLE READ	REPEATABLE READ	SQL Server Repeatable Read is pessimistic locking based, Babelfish Repeatable Read is snapshot (MVCC) based.
SERIALIZABLE	SERIALIZABLE	SERIALIZABLE	SQL Server Serializable is pessimistic isolation, Babelfish Serializable is

Isolation Levels in SQL Server	Babelfish Isolation Level	PostgreSQL Isolation Level	Comments
			snapshot (MVCC) based.

Note

The table hints are not currently supported and their behavior is controlled by using the Babelfish predefined escape hatch `escape_hatch_table_hints`.

Setting up the Transaction Isolation Levels

Use the following command to set Transaction Isolation Level:

Example

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
 SNAPSHOT | SERIALIZABLE }
```

Enabling or disabling Transaction Isolation Levels

Transaction Isolation Levels REPEATABLE READ and SERIALIZABLE are disabled by default in Babelfish and you have to explicitly enable them by setting the `babelfishpg_tsql.isolation_level_serializable` or `babelfishpg_tsql.isolation_level_repeatable_read` escape hatch to `pg_isolation` using `sp_babelfish_configure`. For more information, see Managing Babelfish error handling with escape hatches.

Below are examples for enabling or disabling the use of REPEATABLE READ and SERIALIZABLE in the current session by setting their respective escape hatches. Optionally include `server` parameter to set the escape hatch for the current session as well as for all subsequent new sessions.

To enable the use of SET TRANSACTION ISOLATION LEVEL REPEATABLE READ in current session only.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_repeatable_read', 'pg_isolation'
```

To enable the use of SET TRANSACTION ISOLATION LEVEL REPEATABLE READ in current session and all consequent new sessions.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_repeatable_read', 'pg_isolation',
 'server'
```

To disable the use of SET TRANSACTION ISOLATION LEVEL REPEATABLE READ in current session and consequent new sessions.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_repeatable_read', 'off', 'server'
```

To enable the use of SET TRANSACTION ISOLATION LEVEL SERIALIZABLE in current session only.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_serializable', 'pg_isolation'
```

To enable the use of SET TRANSACTION ISOLATION LEVEL SERIALIZABLE in current session and all consequent new sessions.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_serializable', 'pg_isolation', 'server'
```

To disable the use of SET TRANSACTION ISOLATION LEVEL SERIALIZABLE in current session and consequent new sessions.

Example

```
EXECUTE sp_babelfish_configure 'isolation_level_serializable', 'off', 'server'
```

Differences between Babelfish and SQL Server Isolation Levels

Below are a few examples on the nuances in how SQL Server and Babelfish implement the ANSI Isolation Levels.

> **Note**
>
> - Isolation Level Repeatable Read and Snapshot are the same in Babelfish.
> - Isolation Level Read Uncommitted and Read Committed are the same in Babelfish.

The following example shows how to create the base table for all the examples mentioned below:

```
CREATE TABLE employee (
    id sys.INT NOT NULL PRIMARY KEY,
    name sys.VARCHAR(255)NOT NULL,
    age sys.INT NOT NULL
);
INSERT INTO employee (id, name, age) VALUES (1, 'A', 10);
INSERT INTO employee (id, name, age) VALUES (2, 'B', 20);
INSERT INTO employee (id, name, age) VALUES (3, 'C', 30);
```

Topics

BABELFISH READ UNCOMMITTED VS SQL SERVER READ UNCOMMITTED ISOLATION LEVEL

DIRTY READS IN SQL SERVER

Transaction 1	Transaction 2	SQL Server Read Uncommitted	Babelfish Read Uncommitted
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;	SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;		
	UPDATE employee SET age=0;	Update successful.	Update successful.
	INSERT INTO employee VALUES (4, 'D', 40);	Insert successful.	Insert successful.
SELECT * FROM employee;		Transaction 1 can see uncommitted changes from Transaction 2.	Same as Read Committed in Babelfish. Uncommitted changes from Transaction 2 are not visible to Transaction 1.
	COMMIT		
SELECT * FROM employee;		Sees the changes committed by Transaction 2.	Sees the changes committed by Transaction 2.

BABELFISH READ COMMITTED VS SQL SERVER READ COMMITTED ISOLATION LEVEL

READ - WRITE BLOCKING

Transaction 1	Transaction 2	SQL Server Read Committed	Babelfish Read Committed
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;		
SELECT * FROM employee;			
	UPDATE employee SET age=100 WHERE id = 1;	Update successful.	Update successful.
UPDATE employee SET age = 0 WHERE age IN (SELECT MAX(age) FROM employee);		Step blocked until Transaction 2 commits.	Transaction 2 changes is not visible yet. Updates row with id=3.
	COMMIT	Transaction 2 commits successfully. Transaction 1 is now unblocked and sees the update from Transaction 2.	Transaction 2 commits successfully.
SELECT * FROM employee;		Transaction 1 updates row with id = 1.	Transaction 1 updates row with id = 3.

BABELFISH READ COMMITTED VS SQL SERVER READ COMMITTED SNAPSHOT ISOLATION LEVEL

BLOCKING BEHAVIOUR ON NEW INSERTED ROWS

Transaction 1	Transaction 2	SQL Server Read Committed Snapshot	Babelfish Read Committed
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;		
INSERT INTO employee VALUES (4, 'D', 40);			
	UPDATE employee SET age = 99;	Step is blocked until transaction 1 commits. Inserted row is locked by transaction 1.	Updated three rows. The newly inserted row is not visible yet.
COMMIT		Commit successful. Transaction 2 is now unblocked.	Commit successful.
	SELECT * FROM employee;	All 4 rows have age=99.	Row with id = 4 has age value 40 since it was not visible to transaction 2 during update query. Other rows are updated to age=99.

BABELFISH REPEATABLE READ VS SQL SERVER REPEATABLE READ ISOLATION LEVEL

READ / WRITE BLOCKING BEHAVIOR

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		
SELECT * FROM employee;			
UPDATE employee SET name='A_TXN1' WHERE id=1;			
	SELECT * FROM employee WHERE id != 1;		
	SELECT * FROM employee;	Transaction 2 is blocked until Transaction 1 commits.	Transaction 2 proceeds normally.
COMMIT			
	SELECT * FROM employee;	Update from Transaction 1 is visible.	Update from Transaction 1 is not visible.
COMMIT			
	SELECT * FROM employee;	sees the update from Transaction 1.	sees the update from Transaction 1.

WRITE / WRITE BLOCKING BEHAVIOR

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		
UPDATE employee SET name='A_TXN1' WHERE id=1;			
	UPDATE employee SET name='A_TXN2' WHERE id=1;	Transaction 2 blocked.	Transaction 2 blocked.
COMMIT		Commit successful and transaction 2 has been unblocked.	Commit successfu l and transaction 2 fails with error could not serialize access due to concurrent update.
	COMMIT	Commit successful.	Transaction 2 has already been aborted.
	SELECT * FROM employee;	Row with id=1 has name='A_TX2'.	Row with id=1 has name='A_TX1'.

PHANTOM READ

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
BEGIN TRANSACTION	BEGIN TRANSACTION		

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		
SELECT * FROM employee;			
	INSERT INTO employee VALUES (4, 'NewRowName', 20);	Transaction 2 proceeds without any blocking.	Transaction 2 proceeds without any blocking.
	SELECT * FROM employee;	Newly inserted row is visible.	Newly inserted row is visible.
	COMMIT		
SELECT * FROM employee;		New row inserted by transaction 2 is visible.	New row inserted by transaction 2 is not visible.
COMMIT			
SELECT * FROM employee;		Newly inserted row is visible.	Newly inserted row is visible.

DIFFERENT FINAL RESULTS

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		

Transaction 1	Transaction 2	SQL Server Repeatable Read	Babelfish Repeatable Read
UPDATE employee SET age = 100 WHERE age IN (SELECT MIN(age) FROM employee);		Transaction 1 updates row with id 1.	Transaction 1 updates row with id 1.
	UPDATE employee SET age = 0 WHERE age IN (SELECT MAX(age) FROM employee);	Transaction 2 is blocked since the SELECT statement tries to read rows locked by UPDATE query in transaction 1.	Transaction 2 proceeds without any blocking since read is never blocked, SELECT statement executes and finally row with id = 3 is updated since transaction 1 changes are not visible yet.
	SELECT * FROM employee;	This step is executed after transaction 1 has committed. Row with id = 1 is updated by transaction 2 in previous step and is visible here.	Row with id = 3 is updated by Transacti on 2.
COMMIT		Transaction 2 is now unblocked.	Commit successful.
	COMMIT		
SELECT * FROM employee;		Both transaction execute update on row with id = 1.	Different rows are updated by transacti on 1 and 2.

BABELFISH SERIALIZABLE VS SQL SERVER SERIALIZABLE ISOLATION LEVEL

RANGE LOCKS IN SQL SERVER

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;	SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;		
SELECT * FROM employee;			
	INSERT INTO employee VALUES (4, 'D', 35);	Transaction 2 is blocked until Transaction 1 commits.	Transaction 2 proceeds without any blocking.
	SELECT * FROM employee;		
COMMIT		Transaction 1 commits successfully. Transaction 2 is now unblocked.	Transaction 1 commits successfully.
	COMMIT		
SELECT * FROM employee;		Newly inserted row is visible.	Newly inserted row is visible.

DIFFERENT FINAL RESULTS

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;	SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;		
	INSERT INTO employee VALUES (4, 'D', 40);		
UPDATE employee SET age =99 WHERE id = 4;		Transaction 1 is blocked until Transaction 2 commits.	Transaction 1 proceeds without any blocking.
	COMMIT	Transaction 2 commits successfully. Transaction 1 is now unblocked.	Transaction 2 commits successfully.
COMMIT			
SELECT * FROM employee;		Newly inserted row is visible with age value = 99.	Newly inserted row is visible with age value = 40.

INSERT INTO TABLE WITH UNIQUE CONSTRAINT

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
BEGIN TRANSACTION	BEGIN TRANSACTION		

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;	SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;		
	INSERT INTO employee VALUES (4, 'D', 40);		
INSERT INTO employee VALUES ((SELECT MAX(id)+1 FROM employee), 'E', 50);		Transaction 1 is blocked until Transaction 2 commits.	Transaction 1 is blocked until Transaction 2 commits.
	COMMIT	Transaction 2 commits successfully. Transaction 1 is now unblocked.	Transaction 2 commits successfully. Transaction 1 aborted with error duplicate key value violates unique constraint.
COMMIT		Transaction 1 commits successfully.	Transaction 1 commits fails with could not serialize access due to read/ write dependencies among transactions.
SELECT * FROM employee;		row (5, 'E', 50) is inserted.	Only 4 rows exists.

In Babelfish, concurrent transactions running with Isolation Level serializable will fail with serialization anomaly error if the execution of these transaction is inconsistent with all possible serial (one at a time) executions of those transactions.

SERIALIZATION ANOMALY

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;	SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;		
SELECT * FROM employee;			
UPDATE employee SET age=5 WHERE age=10;			
	SELECT * FROM employee;	Transaction 2 is blocked until Transaction 1 commits.	Transaction 2 proceeds without any blocking.
	UPDATE employee SET age=35 WHERE age=30;		
COMMIT		Transaction 1 commits successfully.	Transaction 1 is committed first and is able to commit successfully.
	COMMIT	Transaction 2 commits successfully.	Transaction 2 commit fails with serialization error, the whole transaction has been rolled back. Retry transaction 2.

Transaction 1	Transaction 2	SQL Server Serializable	Babelfish Serializable
SELECT * FROM employee;		Changes from both transactions are visible.	Transaction 2 was rolled back. Only transaction 1 changes are seen.

In Babelfish, serialization anomaly is only possible if all the concurrent transactions are executing at Isolation Level SERIALIZABLE. For example lets take the above example but set transaction 2 to Isolation Level REPEATABLE READ instead.

Transaction 1	Transaction 2	SQL Server Isolation Levels	Babelfish Isolation Levels
BEGIN TRANSACTION	BEGIN TRANSACTION		
SET TRANSACTION ISOLATION LEVEL SERILAIZABLE;	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		
SELECT * FROM employee;			
UPDATE employee SET age=5 WHERE age=10;			
	SELECT * FROM employee;	Transaction 2 is blocked until transaction 1 commits.	Transaction 2 proceeds without any blocking.
	UPDATE employee SET age=35 WHERE age=30;		

Transaction 1	Transaction 2	SQL Server Isolation Levels	Babelfish Isolation Levels
COMMIT		Transaction 1 commits successfully.	Transaction 1 commits successfully.
	COMMIT	Transaction 2 commits successfully.	Transaction 2 commits successfully.
SELECT * FROM employee;		Changes from both transactions are visible.	Changes from both transactions are visible.

Using Babelfish features with limited implementation

Each new version of Babelfish adds support for features that better align with T-SQL functionality and behavior. Still, there are some unsupported features and differences in the current implementation. In the following, you can find information about functional differences between Babelfish and T-SQL, with some workarounds or usage notes.

As of version 1.2.0 of Babelfish, the following features currently have limited implementations:

- **SQL Server catalogs (system views)** – The catalogs `sys.sysconfigures`, `sys.syscurconfigs`, and `sys.configurations` support a single read-only configuration only. The `sp_configure` isn't currently supported. For more information about the other SQL Server views implemented by Babelfish, see Getting information from the Babelfish system catalog.
- **GRANT permissions** – GRANT…TO PUBLIC is supported, but GRANT..TO PUBLIC WITH GRANT OPTION is not currently supported.
- **SQL Server *ownership chain* and permission mechanism limitation** – In Babelfish, the SQL Server ownership chain works for views but not for stored procedures. This means that procedures must be granted explicit access to other objects owned by the same owner as the calling procedures. In SQL Server, granting the caller EXECUTE permissions on the procedure is sufficient to call other objects owned by same owner. In Babelfish, caller must also be granted permissions on the objects accessed by the procedure.
- **Resolution of unqualified (without schema name) object references** – When a SQL object (procedure, view, function or trigger) references an object without qualifying it with a schema

name, SQL Server resolves the object's schema name by using the schema name of the SQL object in which the reference occurs. Currently, Babelfish resolves this differently, by using the default schema of the database user executing the procedure.

- **Default schema changes, sessions, and connections –** If users change their default schema with `ALTER USER...WITH DEFAULT SCHEMA`, the change takes effect immediately in that session. However, for other currently connected sessions belonging to the same user, the timing differs, as follows:
  - For SQL Server: – The change takes effect across all other connections for this user immediately.
  - For Babelfish: – The change takes effect for this user for new connections only.
- **ROWVERSION and TIMESTAMP datatypes implementation and escape hatch setting** – The ROWVERSION and TIMESTAMP datatypes are now supported in Babelfish. To use ROWVERSION or TIMESTAMP in Babelfish, you must change the setting for the escape hatch `babelfishpg_tsql.escape_hatch_rowversion` from its default (strict) to `ignore`. The Babelfish implementation of the ROWVERSION and TIMESTAMP datatypes is mostly semantically identical to SQL Server, with the following exceptions:
  - The built-in @@DBTS function behaves similarly to SQL Server, but with small differences. Rather than returning the last-used value for `SELECT @@DBTS`, Babelfish generates a new timestamp, due to the underlying PostgreSQL database engine and its multi-version concurrency control (MVCC) implementation.
  - In SQL Server, every inserted or updated row gets a unique ROWVERSION/TIMESTAMP value. In Babelfish, every inserted row updated by the same statement is assigned the same ROWVERSION/TIMESTAMP value.

    For example, when an UPDATE statement or INSERT-SELECT statement affects multiple rows, in SQL Server, the affected rows all have different values in their ROWVERSION/TIMESTAMP column. In Babelfish (PostgreSQL), the rows have the same value.
  - In SQL Server, when you create a new table with SELECT-INTO, you can cast an explicit value (such as NULL) to a to-be-created ROWVERSION/TIMESTAMP column. When you do the same thing in Babelfish, an actual ROWVERSION/TIMESTAMP value is assigned to each row in the new table for you, by Babelfish.

  These minor differences in ROWVERSION/TIMESTAMP datatypes shouldn't have an adverse impact on applications running on Babelfish.

Schema creation, ownership, and permissions – Permissions to create and access objects in a schema owned by a non-DBO user (using CREATE SCHEMA *schema name* AUTHORIZATION *user name*) differ for SQL Server and Babelfish non-DBO users, as shown in the following table:

Database user (non-DBO) who owns the schema can do the following:	SQL Server	Babelfish
Create objects in the schema without additional grants by the DBO?	No	Yes
Access objects created by DBO in the schema without additional grants?	Yes	No

Improving Babelfish query performance

You can achieve faster query processing in Babelfish using query hints and the PostgreSQL optimizer.

Topics

- Using explain plan to improve Babelfish query performance
- Using T-SQL query hints to improve Babelfish query performance

You can also improve the query performance using `sp_babelfish_volatility` procedure. For more information, see sp_babelfish_volatility.

Using explain plan to improve Babelfish query performance

Starting with version 2.1.0, Babelfish includes two functions that transparently use the PostgreSQL optimizer to generate estimated and actual query plans for T-SQL queries on the TDS port. These functions are similar to using SET STATISTICS PROFILE or SET SHOWPLAN_ALL with SQL Server databases to identify and improve slow running queries.

> **Note**
>
> Getting query plans from functions, control flows, and cursors isn't currently supported.

In the table you can find a comparison of query plan explain functions across SQL Server, Babelfish, and PostgreSQL.

SQL Server	Babelfish	PostgreSQL
SHOWPLAN_ALL	BABELFISH_SHOWPLAN_ALL	EXPLAIN
STATISTICS PROFILE	BABELFISH_STATISTICS PROFILE	EXPLAIN ANALYZE
Uses the SQL Server optimizer	Uses the PostgreSQL optimizer	Uses the PostgreSQL optimizer
SQL Server input and output format	SQL Server input and PostgreSQL output format	PostgreSQL input and output format
Set for the session	Set for the session	Apply to a specific statement
Supports the following: • SELECT • INSERT • UPDATE • DELETE • CURSOR • CREATE • EXECUTE • EXEC and functions, including control flow (CASE, WHILE-BREAK-CONTINUE, WAITFOR, BEGIN-END, IF-ELSE, and so on)	Supports the following: • SELECT • INSERT • UPDATE • DELETE • CREATE • EXECUTE • EXEC • RAISEERROR • THROW • PRINT • USE	Supports the following: • SELECT • INSERT • UPDATE • DELETE • CURSOR • CREATE • EXECUTE

Use the Babelfish functions as follows:

- SET BABELFISH_SHOWPLAN_ALL [ON|OFF] – Set to ON to generate an estimated query execution plan. This function implements the behavior of the PostgreSQL `EXPLAIN` command. Use this command to obtain the explain plan for given query.
- SET BABELFISH_STATISTICS PROFILE [ON|OFF] – Set to ON for actual query execution plans. This function implements the behavior of PostgreSQL's `EXPLAIN ANALYZE` command.

For more information about PostgreSQL `EXPLAIN` and `EXPLAIN ANALYZE` see EXPLAIN in the PostgreSQL documentation.

Note

Starting with version 2.2.0, you can set the `escape_hatch_showplan_all` parameter to *ignore* in order to avoid the use of *BABELFISH_* prefix in the SQL Server syntax for `SHOWPLAN_ALL` and `STATISTICS PROFILE` SET commands.

For example, the following command sequence turns on query planning and then returns an estimated query execution plan for the SELECT statement without running the query. This example uses the SQL Server sample `northwind` database using the `sqlcmd` command-line tool to query the TDS port:

```
1> SET BABELFISH_SHOWPLAN_ALL ON
2> GO
1> SELECT t.territoryid, e.employeeid FROM
2> dbo.employeeterritories e, dbo.territories t
3> WHERE e.territoryid=e.territoryid ORDER BY t.territoryid;
4> GO

QUERY PLAN


------------------------------------------------------------------------------------


Query Text: SELECT t.territoryid, e.employeeid FROM
dbo.employeeterritories e, dbo.territories t
WHERE e.territoryid=e.territoryid ORDER BY t.territoryid
Sort  (cost=6231.74..6399.22 rows=66992 width=10)
  Sort Key: t.territoryid NULLS FIRST
  ->  Nested Loop  (cost=0.00..861.76 rows=66992 width=10)
```

```
        ->  Seq Scan on employeeterritories e  (cost=0.00..22.70 rows=1264 width=4)
              Filter: ((territoryid)::"varchar" IS NOT NULL)
        ->  Materialize  (cost=0.00..1.79 rows=53 width=6)
              ->  Seq Scan on territories t  (cost=0.00..1.53 rows=53 width=6)
```

When you finish reviewing and adjusting your query, you turn off the function as shown following:

```
1> SET BABELFISH_SHOWPLAN_ALL OFF
```

With BABELFISH_STATISTICS PROFILE set to ON, each executed query returns its regular result set followed by an additional result set that shows actual query execution plans. Babelfish generates the query plan that provides the fastest result set when it invokes the SELECT statement.

```
1> SET BABELFISH_STATISTICS PROFILE ON
1>
2> GO
1> SELECT e.employeeid, t.territoryid FROM
2> dbo.employeeterritories e, dbo.territories t
3> WHERE t.territoryid=e.territoryid ORDER BY t.territoryid;
4> GO
```

The result set and the query plan are returned (this example shows only the query plan).

```
QUERY PLAN



---------------------------------------------------------------------------
Query Text: SELECT e.employeeid, t.territoryid FROM
dbo.employeeterritories e, dbo.territories t
WHERE t.territoryid=e.territoryid ORDER BY t.territoryid
Sort  (cost=42.44..43.28 rows=337 width=10)
  Sort Key: t.territoryid NULLS FIRST



  ->  Hash Join  (cost=2.19..28.29 rows=337 width=10)
       Hash Cond: ((e.territoryid)::"varchar" = (t.territoryid)::"varchar")
        ->  Seq Scan on employeeterritories e  (cost=0.00..22.70 rows=1270 width=36)
        ->  Hash  (cost=1.53..1.53 rows=53 width=6)
             ->  Seq Scan on territories t  (cost=0.00..1.53 rows=53 width=6)
```

To learn more about how to analyze your queries and the results returned by the PostgreSQL optimizer, see explain.depesz.com. For more information about PostgreSQL EXPLAIN and EXPLAIN ANALYZE, see EXPLAIN in the PostgreSQL documentation.

Parameters that control Babelfish explain options

You can use the parameters shown in the following table to control the type of information that's displayed by your query plan.

Parameter	Description
babelfishpg_tsql.explain_buffers	A boolean that turns on (and off) buffer usage information for the optimizer. (Default: off) (Allowable: off, on)
babelfishpg_tsql.explain_costs	A boolean that turns on (and off) estimated startup and total cost information for the optimizer. (Default: on) (Allowable: off, on)
babelfishpg_tsql.explain_format	Specifies the output format for the `EXPLAIN` plan. (Default: text) (Allowable: text, xml, json, yaml)
babelfishpg_tsql.explain_settings	A boolean that turns on (or off) the inclusion of information about configuration parameter s in the EXPLAIN plan output. (Default: off) (Allowable: off, on)
babelfishpg_tsql.explain_summary	A boolean that turns on (or off) summary information such as total time after the query plan. (Default: on) (Allowable: off, on)
babelfishpg_tsql.explain_timing	A boolean that turns on (or off) actual startup time and time spent in each node in the output. (Default: on) (Allowable: off, on)
babelfishpg_tsql.explain_verbose	A boolean that turns on (or off) the most detailed version of an explain plan. (Default: off) (Allowable: off, on)

Parameter	Description
babelfishpg_tsql.explain_wal	A boolean that turns on (or off) generation of WAL record information as part of an explain plan. (Default: off) (Allowable: off, on)

You can check the values of any Babelfish-related parameters on your system by using either PostgreSQL client or SQL Server client. Run the following command to get your current parameter values:

```
1> execute sp_babelfish_configure '%explain%';
2> GO
```

In the following output, you can see that all settings on this particular Babelfish DB cluster are at their default values. Not all output is shown in this example.

```

             name                      setting                     short_desc
---------------------------------- --------
 --------------------------------------------------------
babelfishpg_tsql.explain_buffers   off      Include information on buffer usage
babelfishpg_tsql.explain_costs     on       Include information on estimated startup
 and total cost
babelfishpg_tsql.explain_format    text     Specify the output format, which can be
 TEXT, XML, JSON, or YAML
babelfishpg_tsql.explain_settings  off      Include information on configuration
 parameters
babelfishpg_tsql.explain_summary   on       Include summary information (e.g.,totaled
 timing information) after the query plan
babelfishpg_tsql.explain_timing    on       Include actual startup time and time spent
 in each node in the output
babelfishpg_tsql.explain_verbose   off      Display additional information regarding
 the plan
babelfishpg_tsql.explain_wal       off      Include information on WAL record
 generation

(8 rows affected)
```

You can change the setting for these parameters using `sp_babelfish_configure`, as shown in the following example.

```
1> execute sp_babelfish_configure 'explain_verbose', 'on';
2> GO
```

If you want make the setting permanent on a cluster-wide level, include the keyword *server*, as shown in the following example.

```
1> execute sp_babelfish_configure 'explain_verbose', 'on', 'server';
2> GO
```

Using T-SQL query hints to improve Babelfish query performance

Starting with version 2.3.0, Babelfish supports the use of query hints using `pg_hint_plan`. In Aurora PostgreSQL, `pg_hint_plan` is installed by default. For more information about the PostgreSQL extension `pg_hint_plan`, see https://github.com/ossc-db/pg_hint_plan. For details about the version of this extension supported by Aurora PostgreSQL, see Extension versions for Amazon Aurora PostgreSQL in *Release Notes for Aurora PostgreSQL*.

The query optimizer is well-designed to find the optimal execution plan for a SQL statement. When selecting a plan, the query optimizer considers both the engine’s cost model, and column and table statistics. However, the suggested plan might not meet the needs of your datasets. Thus, query hints addresses the performance issues to improve execution plans. A `query hint` is syntax added to the SQL standard that instructs the database engine about how to execute the query. For example, a hint may instruct the engine to follow a sequential scan and override any plan that the query optimizer had selected.

Turning on T-SQL query hints in Babelfish

Currently, Babelfish ignores all T-SQL hints by default. To apply T-SQL hints, run the command `sp_babelfish_configure` with the enable_pg_hint value as ON.

```
EXECUTE sp_babelfish_configure 'enable_pg_hint', 'on' [, 'server']
```

You can make the settings permanent on a cluster-wide level by including the *server* keyword. To configure the setting for the current session only, don't use server.

After `enable_pg_hint` is ON, Babelfish applies the following T-SQL hints.

- INDEX hints
- JOIN hints

- FORCE ORDER hint
- MAXDOP hint

For example, the following command sequence turns on `pg_hint_plan`.

```
1> CREATE TABLE t1 (a1 INT PRIMARY KEY, b1 INT);
2> CREATE TABLE t2 (a2 INT PRIMARY KEY, b2 INT);
3> GO
1> EXECUTE sp_babelfish_configure 'enable_pg_hint', 'on';
2> GO
1> SET BABELFISH_SHOWPLAN_ALL ON;
2> GO
1> SELECT * FROM t1 JOIN t2 ON t1.a1 = t2.a2; --NO HINTS (HASH JOIN)
2> GO
```

No hint is applied to the SELECT statement. The query plan with no hint is returned.

```
QUERY PLAN



---------------------------------------------------------------------------
Query Text: SELECT * FROM t1 JOIN t2 ON t1.a1 = t2.a2
Hash Join (cost=60.85..99.39 rows=2260 width=16)
 Hash Cond: (t1.a1 = t2.a2)
 -> Seq Scan on t1 (cost=0.00..32.60 rows=2260 width=8)
 -> Hash (cost=32.60..32.60 rows=2260 width=8)
 -> Seq Scan on t2 (cost=0.00..32.60 rows=2260 width=8)
```

```

1> SELECT * FROM t1 INNER MERGE JOIN t2 ON t1.a1 = t2.a2;
2> GO
```

The query hint is applied to the SELECT statement. The following output shows that the query plan with merge join is returned.

```
QUERY PLAN
```

```
---------------------------------------------------------------------------
Query Text: SELECT/*+ MergeJoin(t1 t2) Leading(t1 t2)*/ * FROM t1 INNER JOIN t2 ON
 t1.a1 = t2.a2
Merge Join (cost=0.31..190.01 rows=2260 width=16)
 Merge Cond: (t1.a1 = t2.a2)
 -> Index Scan using t1_pkey on t1 (cost=0.15..78.06 rows=2260 width=8)
 -> Index Scan using t2_pkey on t2 (cost=0.15..78.06 rows=2260 width=8)
```

```
1> SET BABELFISH_SHOWPLAN_ALL OFF;
2> GO
```

Limitations

While using the query hints, consider the following limitations:

- If a query plan is cached before `enable_pg_hint` is turned on, hints won't be applied in the same session. It will be applied in the new session .
- If schema names are explicitly given, then hints can't be applied. You can use table aliases as a workaround.
- A query hint can't be applied to views and sub-queries.
- Hints don't work for UPDATE/DELETE statements with JOINs.
- An index hint for a non-existing index or table is ignored.
- The FORCE ORDER hint doesn't work for HASH JOINs and non-ANSI JOINs.

Using Aurora PostgreSQL extensions with Babelfish

Aurora PostgreSQL provides extensions for working with other AWS services. These are optional extensions that support various use cases, such as using Amazon S3 with your DB cluster for importing or exporting data.

- To import data from an Amazon S3 bucket to your Babelfish DB cluster, you set up the `aws_s3` Aurora PostgreSQL extension. This extension also lets you export data from your Aurora PostgreSQL DB cluster to an Amazon S3 bucket.
- AWS Lambda is a compute service that lets you run code without provisioning or managing servers. You can use Lambda functions to do things like process event notifications from your DB instance. To learn more about Lambda, see [What is AWS Lambda?](#) in the *AWS Lambda Developer*

Guide. To invoke Lambda functions from your Babelfish DB cluster, you set up the `aws_lambda` Aurora PostgreSQL extension.

To set up these extensions for your Babelfish cluster, you first need to grant permission to the internal Babelfish user to load the extensions. After granting permission, you can then load Aurora PostgreSQL extensions.

Enabling Aurora PostgreSQL extensions in your Babelfish DB cluster

Before you can load the `aws_s3` or the `aws_lambda` extensions, you grant the needed privileges to your Babelfish DB cluster.

The procedure following uses the `psql` PostgreSQL command line tool to connect to the DB cluster. For more information, see Using psql to connect to the DB cluster. You can also use pgAdmin. For details, see Using pgAdmin to connect to the DB cluster.

This procedure loads both `aws_s3` and `aws_lambda`, one after the other. You don't need to load both if you want to use only one of these extensions. The `aws_commons` extension is required by each, and it's loaded by default as shown in the output.

To set up your Babelfish DB cluster with privileges for the Aurora PostgreSQL extensions

1. Connect to your Babelfish DB cluster. Use the name for the "master" user (-U) that you specified when you created the Babelfish DB cluster. The default (`postgres`) is shown in the examples.

   For Linux, macOS, or Unix:

   ```
   psql -h your-Babelfish.cluster.444455556666-us-east-1.rds.amazonaws.com \
   -U postgres \
   -d babelfish_db \
   -p 5432
   ```

   For Windows:

   ```
   psql -h your-Babelfish.cluster.444455556666-us-east-1.rds.amazonaws.com ^
   -U postgres ^
   -d babelfish_db ^
   -p 5432
   ```

The command responds with a prompt to enter the password for the user name (-U).

```
Password:
```

Enter the password for the user name (-U) for the DB cluster. When you successfully connect, you see output similar to the following.

```
psql (13.4)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
 compression: off)
Type "help" for help.

postgres=>
```

2. Grant privileges to the internal Babelfish user to create and load extensions.

```
babelfish_db=> GRANT rds_superuser TO master_dbo;
GRANT ROLE
```

3. Create and load the `aws_s3` extension. The `aws_commons` extension is required, and it's installed automatically when the `aws_s3` is installed.

```
babelfish_db=> create extension aws_s3 cascade;
NOTICE:  installing required extension "aws_commons"
CREATE EXTENSION
```

4. Create and load the `aws_lambda` extension.

```
babelfish_db=> create extension aws_lambda cascade;
CREATE EXTENSION
babelfish_db=>
```

Using Babelfish with Amazon S3

If you don't already have an Amazon S3 bucket to use with your Babelfish DB cluster, you can create one. For any Amazon S3 bucket that you want to use, you provide access.

Before trying to import or export data using an Amazon S3 bucket, complete the following one-time steps.

To set up access for your Babelfish DB instance to your Amazon S3 bucket

1. Create an Amazon S3 bucket for your Babelfish instance, if needed. To do so, follow the instructions in Create a bucket in the *Amazon Simple Storage Service User Guide*.
2. Upload files to your Amazon S3 bucket. To do so, follow the steps in Add an object to a bucket in the *Amazon Simple Storage Service User Guide.*
3. Set up permissions as needed:

   - To import data from Amazon S3, the Babelfish DB cluster needs permission to access the bucket. We recommend using an AWS Identity and Access Management (IAM) role and attaching an IAM policy to that role for your cluster. To do so, follow the steps in Using an IAM role to access an Amazon S3 bucket.
   - To export data from your Babelfish DB cluster, your cluster must be granted access to the Amazon S3 bucket. As with importing, we recommend using an IAM role and policy. To do so, follow the steps in Setting up access to an Amazon S3 bucket.

You can now use Amazon S3 with the `aws_s3` extension with your Babelfish DB cluster.

To import data from Amazon S3 to Babelfish and to export Babelfish data to Amazon S3

1. Use the `aws_s3` extension with your Babelfish DB cluster.

   When you do, make sure to reference the tables as they exist in the context of PostgreSQL. That is, if you want to import into a Babelfish table named `[database].[schema].[tableA]`, refer to that table as `database_schema_tableA` in the `aws_s3` function:

   - For an example of using an `aws_s3` function to import data, see Importing data from Amazon S3 to your Aurora PostgreSQL DB cluster.
   - For examples of using `aws_s3` functions to export data, see Exporting query data using the aws_s3.query_export_to_s3 function.
2. Make sure to reference Babelfish tables using PostgreSQL naming when using the `aws_s3` extension and Amazon S3, as shown in the following table.

Babelfish table	Aurora PostgreSQL table
database.*schema*.*table*	*database*_*schema*_*table*

To learn more about using Amazon S3 with Aurora PostgreSQL, see Importing data from Amazon S3 into an Aurora PostgreSQL DB cluster and Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3.

Using Babelfish with AWS Lambda

After the `aws_lambda` extension is loaded in your Babelfish DB cluster but before you can invoke Lambda functions, you give Lambda access to your DB cluster by following this procedure.

To set up access for your Babelfish DB cluster to work with Lambda

This procedure uses the AWS CLI to create the IAM policy and role, and associate these with the Babelfish DB cluster.

1. Create an IAM policy that allows access to Lambda from your Babelfish DB cluster.

```
aws iam create-policy  --policy-name rds-lambda-policy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Sid": "AllowAccessToExampleFunction",
        "Effect": "Allow",
        "Action": "lambda:InvokeFunction",
        "Resource": "arn:aws:lambda:aws-region:444455556666:function:my-function"
        }
    ]
}'
```

2. Create an IAM role that the policy can assume at runtime.

```
aws iam create-role  --role-name rds-lambda-role --assume-role-policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "rds.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}'
```

3. Attach the policy to the role.

```
aws iam attach-role-policy \
    --policy-arn arn:aws:iam::444455556666:policy/rds-lambda-policy \
    --role-name rds-lambda-role --region aws-region
```

4. Attach the role to your Babelfish DB cluster

```
aws rds add-role-to-db-cluster \
       --db-cluster-identifier my-cluster-name \
       --feature-name Lambda \
       --role-arn  arn:aws:iam::444455556666:role/rds-lambda-role   \
       --region aws-region
```

After you complete these tasks, you can invoke your Lambda functions. For more information and examples of setting up AWS Lambda for Aurora PostgreSQL DB cluster with AWS Lambda, see Step 2: Configure IAM for your Aurora PostgreSQL DB cluster and AWS Lambda.

To invoke a Lambda function from your Babelfish DB cluster

AWS Lambda supports functions written in Java, Node.js, Python, Ruby, and other languages. If the function returns text when invoked, you can invoke it from your Babelfish DB cluster. The following example is a placeholder python function that returns a greeting.

```
lambda_function.py
import json
def lambda_handler(event, context):
    #TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
```

Currently, Babelfish doesn't support JSON. If your function returns JSON, you use a wrapper to handle the JSON. For example, say that the `lambda_function.py` shown preceding is stored in Lambda as `my-function`.

1. Connect to your Babelfish DB cluster using the `psql` client (or the pgAdmin client). For more information, see Using psql to connect to the DB cluster.

2. Create the wrapper. This example uses PostgreSQL's procedural language for SQL, PL/pgSQL. To learn more, see PL/pgSQL–SQL Procedural Language.

```
create or replace function master_dbo.lambda_wrapper()
returns text
language plpgsql
as
$$
declare
   r_status_code integer;
   r_payload text;
begin
   SELECT payload INTO r_payload
     FROM aws_lambda.invoke(  aws_commons.create_lambda_function_arn('my-function',
 'us-east-1')
                            ,'{"body": "Hello from Postgres!"}'::json );
   return r_payload ;
end;
$$;
```

The function can now be run from the Babelfish TDS port (1433) or from the PostgreSQL port (5433).

a. To invoke (call) this function from your PostgreSQL port:

```
SELECT * from aws_lambda.invoke(aws_commons.create_lambda_function_arn('my-
function', 'us-east-1'), '{"body": "Hello from Postgres!"}'::json );
```

The output is similar to the following:

```
status_code |                        payload                        |
 executed_version | log_result
-------------+-------------------------------------------------------
+------------------+-----------
         200 | {"statusCode": 200, "body": "\"Hello from Lambda!\""} | $LATEST
         |
(1 row)
```

b. To invoke (call) this function from the TDS port, connect to the port using the SQL Server `sqlcmd` command line client. For details, see Using a SQL Server client to connect to your DB cluster. When connected, run the following:

```
1> select lambda_wrapper();
2> go
```

The command returns output similar to the following:

```
{"statusCode": 200, "body": "\"Hello from Lambda!\""}
```

To learn more about using Lambda with Aurora PostgreSQL, see Invoking an AWS Lambda function from an Aurora PostgreSQL DB cluster. For more information about working with Lambda functions, see Getting started with Lambda in the *AWS Lambda Developer Guide.*

Using pg_stat_statements in Babelfish

Babelfish for Aurora PostgreSQL supports `pg_stat_statements` extension from 3.3.0. To learn more, see pg_stat_statements.

For details about the version of this extension supported by Aurora PostgreSQL, see Extension versions.

Creating pg_stat_statements extension

To turn on `pg_stat_statements`, you must turn on the Query identifier calculation. This is done automatically if `compute_query_id` is set to on or `auto` in the parameter group. The default value of `compute_query_id` parameter is `auto`. You also need to create this extension to turn on this feature. Use the following command to install the extension from T-SQL endpoint:

```
1>EXEC sp_execute_postgresql 'CREATE EXTENSION pg_stat_statements WITH SCHEMA sys';
```

You can access the query statistics using the following query:

```
postgres=>select * from pg_stat_statements;
```

Note

During installation, if you don't provide the schema name for the extension then by default it will create it in public schema. To access it, you must use square brackets with schema qualifier as shown below:

```
postgres=>select * from [public].pg_stat_statements;
```

You can also create the extension from PSQL endpoint.

Authorizing the extension

By default, you can see the statistics for queries performed within your T-SQL database without the need of any authorization.

To access query statistics created by others, you need to have `pg_read_all_stats` PostgreSQL role. Follow the steps mentioned below to construct GRANT pg_read_all_stats command.

1. In T-SQL, use the following query that returns the internal PG role name.

   ```
   SELECT rolname FROM pg_roles WHERE oid = USER_ID();
   ```

2. Connect to Babelfish for Aurora PostgreSQL database with rds_superuser privilege and use the following command:

   ```
   GRANT pg_read_all_stats TO <rolname_from_above_query>
   ```

Example

From T-SQL endpoint:

```
1>SELECT rolname FROM pg_roles WHERE oid = USER_ID();
2>go
```

```


rolname
-------
```

```
master_dbo
(1 rows affected)
```

From PSQL endpoint:

```
babelfish_db=# grant pg_read_all_stats to master_dbo;
```

```
GRANT ROLE
```

You can access the query statistics using the pg_stat_statements view:

```
1>create table t1(cola int);
2>go
1>insert into t1 values (1),(2),(3);
2>go
```

```
(3 rows affected)
```

```
1>select userid, dbid, queryid, query from pg_stat_statements;
2>go
```

```
userid dbid queryid             query
------ ---- -------             -----
37503 34582 6487973085327558478 select * from t1
37503 34582 6284378402749466286 SET QUOTED_IDENTIFIER OFF
37503 34582 2864302298511657420 insert into t1 values ($1),($2),($3)
10    34582 NULL                <insufficient privilege>
37503 34582 5615368793313871642 SET TEXTSIZE 4096
37503 34582 639400815330803392  create table t1(cola int)
(6 rows affected)
```

Resetting query statistics

You can use `pg_stat_statements_reset()` to reset the statistics gathered so far by pg_stat_statements. To learn more, see pg_stat_statements. It is currently supported through

PSQL endpoint only. Connect to Babelfish for Aurora PostgreSQL with `rds_superuser` privilege, use the following command:

```
SELECT pg_stat_statements_reset();
```

Limitations

- Currently, `pg_stat_statements()` is not supported through T-SQL endpoint. `pg_stat_statements` view is the recommended way to gather the statistics.
- Some of the queries might be re-written by the T-SQL parser implemented by Aurora PostgreSQL engine, `pg_stat_statements` view will show the re-written query and not the original query.

  Example

  ```
  select next value for [dbo].[newCounter];
  ```

  The above query is re-written as the following in the pg_stat_statements view.

  ```
  select nextval($1);
  ```
- Based on the execution flow of the statements, some of the queries might not be tracked by pg_stat_statements and will not be visible in the view. This includes the following statements: `use dbname`, `goto`, `print`, `raise error`, `set`, `throw`, `declare cursor`.
- For CREATE LOGIN and ALTER LOGIN statements, query and queryid will not be shown. It will show insufficient privileges.
- `pg_stat_statements` view always contains the below two entries, as these are executed internally by `sqlcmd` client.
  - SET QUOTED_IDENTIFIER OFF
  - SET TEXTSIZE 4096

Using pgvector in Babelfish

pgvector, an open-source extension, lets you search for similar data directly within your Postgres database. Babelfish now supports this extension starting with versions 15.6 and 16.2. For more information, pgvector Open source Documentation.

Prerequisites

To enable pgvector functionality, install the extension in sys schema using one of the following methods:

- Run the following command in sqlcmd client:

```
exec sys.sp_execute_postgresql 'CREATE EXTENSION vector WITH SCHEMA sys';
```

- Connect to `babelfish_db` and run the following command in psql client:

```
CREATE EXTENSION vector WITH SCHEMA sys;
```

> **Note**
>
> After installing the pgvector extension, the vector data type will only be available in new database connections you establish. Existing connections won't recognize the new data type.

Supported Functionality

Babelfish extends the T-SQL functionality to support the following:

- **Storing**

  Babelfish now supports vector datatype compatible syntax, enhancing its T-SQL compatibility. To learn more about storing data with pgvector, see Storing.

- **Querying**

  Babelfish expands T-SQL expression support to include vector similarity operators. However, for all other queries, standard T-SQL syntax is still required.

  > **Note**
  >
  > T-SQL doesn't support Array type, and the database drivers do not have any interface to handle them. As a workaround, Babelfish uses text strings (varchar/nvarchar) to store vector data. For example, when you request a vector value [1,2,3], Babelfish will return a

string '[1,2,3]' as the response. You can parse and split this string at application level as per your needs.

To learn more about querying data with pgvector, see Querying.

- **Indexing**

  T-SQL `Create Index` now supports `USING INDEX_METHOD` syntax. You can now define similarity search operator to be used on a specific column when creating an index.

  The grammar is also extended to support Vector similarity operations on the required column (Check column_name_list_with_order_for_vector grammar).

```
CREATE [UNIQUE] [clustered] [COLUMNSTORE] INDEX <index_name> ON <table_name> [USING
 vector_index_method] (<column_name_list_with_order_for_vector>)
Where column_name_list_with_order_for_vector is:
    <column_name> [ASC | DESC] [VECTOR_COSINE_OPS | VECTOR_IP_OPS | VECTOR_L2_OPS]
 (COMMA simple_column_name [ASC | DESC] [VECTOR_COSINE_OPS | VECTOR_IP_OPS |
 VECTOR_L2_OPS])
```

  To learn more about indexing data with pgvector, see Indexing.

- **Performance**

  - Use `SET BABELFISH_STATISTICS PROFILE ON` to debug Query Plans from T-SQL endpoint.
  - Increase `max_parallel_workers_get_gather` using the `set_config` function supported in T-SQL.
  - Use `IVFFlat` for approximate searches. For more information, see IVFFlat.

  To improve performance with pgvector, see Performance.

Limitations

- Babelfish doesn't support Full Text Search for Hybrid Search. For more information, see Hybrid Search.
- Babelfish doesn't currently support re-indexing functionality. However, you can still use PostgreSQL endpoint to re-index. For more information, see Vacuuming.

Using Amazon Aurora machine learning with Babelfish

You can extend the capabilities of your Babelfish for Aurora PostgreSQL DB cluster by integrating it with Amazon Aurora machine learning. This seamless integration grants you access to a range of powerful services like Amazon Comprehend or Amazon SageMaker or Amazon Bedrock, each tailored to address distinct machine learning needs.

As a Babelfish user, you can use existing knowledge of T-SQL syntax and semantics when working with Aurora machine learning. Follow the instructions provided in the AWS documentation for Aurora PostgreSQL. For more information, see Using Amazon Aurora machine learning with Aurora PostgreSQL.

Prerequisites

- Before trying to set up your Babelfish for Aurora PostgreSQL DB cluster to use Aurora machine learning, you must understand the related requirements and prerequisites. For more information, see Requirements for using Aurora machine learning with Aurora PostgreSQL.
- Make sure you install the `aws_ml` extension either using Postgres endpoint or the `sp_execute_postgresql` store procedure.

```
exec sys.sp_execute_postgresql 'Create Extension aws_ml'
```

> **Note**
>
> Currently Babelfish doesn't support cascade operations with `sp_execute_postgresql` in Babelfish. Since `aws_ml` relies on `aws_commons`, you'll need to install it separately using Postgres endpoint.

```
create extension aws_common;
```

Handling T-SQL syntax and semantics with `aws_ml` functions

The following examples explains how T-SQL syntax and semantics are applied to the Amazon ML services:

Example : aws_bedrock.invoke_model – A simple query using Amazon Bedrock functions

```
aws_bedrock.invoke_model(
   model_id      varchar,
   content_type  text,
   accept_type   text,
   model_input   text)
Returns Varchar(MAX)
```

The following example shows how to invoke a Anthropic Claude 2 model for Bedrock using invoke_model.

```

SELECT aws_bedrock.invoke_model (
    'anthropic.claude-v2', -- model_id
    'application/json', -- content_type
    'application/json', -- accept_type
    '{"prompt": "\n\nHuman:
    You are a helpful assistant that answers questions directly
    and only using the information provided in the context below.
    \nDescribe the answerin detail.\n\nContext: %s \n\nQuestion:
    %s \n\nAssistant:","max_tokens_to_sample":4096,"temperature"
    :0.5,"top_k":250,"top_p":0.5,"stop_sequences":[]}' -- model_input
);

```

Example : aws_comprehend.detect_sentiment – A simple query using Amazon Comprehend functions

```

aws_comprehend.detect_sentiment(
   input_text varchar,
   language_code varchar,
   max_rows_per_batch int)
Returns table (sentiment varchar, confidence real)

```

The following example shows how to invoke the Amazon Comprehend service.

```

select sentiment from aws_comprehend.detect_sentiment('This is great', 'en');
```

Example : aws_sagemaker.invoke_endpoint – A simple query using Amazon SageMaker functions

```
aws_sagemaker.invoke_endpoint(
  endpoint_name varchar,
  max_rows_per_batch int,
  VARIADIC model_input "any") -- Babelfish inherits PG's variadic parameter type
Rerurns Varchar(MAX)
```

Since model_input is marked as VARIADIC and of type “any”, users can pass a list of any length and any datatype to the function which will act as the input the input to the model. The following example shows how to invoke the Amazon SageMaker service.

```
SELECT CAST (aws_sagemaker.invoke_endpoint(
    'sagemaker_model_endpoint_name',
    NULL,
    arg1, arg2 -- model inputs are separate arguments )
AS INT) -- cast the output to INT
```

For more detailed information on using Aurora machine learning with Aurora PostgreSQL, see Using Amazon Aurora machine learning with Aurora PostgreSQL.

Limitations

- While Babelfish doesn't allow creating arrays, it can still handle data that represents arrays. When you use functions like `aws_bedrock.invoke_model_get_embeddings` that return arrays, the results is delivered as a string containing the array elements.

Babelfish supports linked servers

Babelfish for Aurora PostgreSQL supports linked servers by using the PostgreSQL `tds_fdw` extension in version 3.1.0. To work with linked servers, you must install the `tds_fdw` extension. For more information about the `tds_fdw` extension, see Working with the supported foreign data wrappers for Amazon Aurora PostgreSQL.

Installing the `tds_fdw` extension

You can install `tds_fdw` extension using the following methods.

Using CREATE EXTENSION from PostgreSQL endpoint

1. Connect to your PostgreSQL DB instance on the Babelfish database in the PostgreSQL port. Use an account that has the rds_superuser role.

   ```
   psql --host=your-DB-instance.aws-region.rds.amazonaws.com --port=5432 --
   username=test --dbname=babelfish_db --password
   ```

2. Install the `tds_fdw` extension. This is a one-time installation process. You don't need to reinstall when the DB cluster restarts.

   ```

   babelfish_db=> CREATE EXTENSION tds_fdw;
   CREATE EXTENSION
   ```

Calling `sp_execute_postgresql` stored procedure from TDS endpoint

Babelfish supports installing `tds_fdw` extension by calling `sp_execute_postgresql` procedure from version 3.3.0. You can execute PostgreSQL statements from T-SQL endpoint without exiting the T-SQL port. For more information, see Babelfish for Aurora PostgreSQL procedure reference

1. Connect to your PostgreSQL DB instance on the Babelfish database in the T-SQL port.

   ```
   sqlcmd -S your-DB-instance.aws-region.rds.amazonaws.com -U test -P password
   ```

2. Install the `tds_fdw` extension.

   ```
   1>EXEC sp_execute_postgresql N'CREATE EXTENSION tds_fdw';
   2>go
   ```

Supported functionality

Babelfish supports adding remote RDS for SQL Server or Babelfish for Aurora PostgreSQL endpoints as the linked server. You can also add other remote SQL Server instances as linked servers. Then, use `OPENQUERY()` to retrieve data from these linked servers. Starting from Babelfish version 3.2.0, four-part names are also supported.

The following stored procedures and catalog views are supported in order to use the linked servers.

Stored procedures

- **sp_addlinkedserver –** Babelfish doesn't support the `@provstr` parameter.
- **sp_addlinkedsrvlogin**
  - You must provide an explicit remote username and password to connect to the remote data source. You can't connect with the user's self credentials. Babelfish supports only `@useself = false`.
  - Babelfish doesn't support the `@locallogin` parameter since configuring remote server access specific to local login isn't supported.
- **sp_linkedservers**
- **sp_helplinkedsrvlogin**
- **sp_dropserver**
- **sp_droplinkedsrvlogin –** Babelfish doesn't support the `@locallogin` parameter since configuring remote server access specific to local login isn't supported.
- **sp_serveroption –** Babelfish supports the following server options:
  - query timeout (from Babelfish version 3.2.0)
  - connect timeout (from Babelfish version 3.3.0)
- **sp_testlinkedserver** (from Babelfish version 3.3.0)
- **sp_enum_oledb_providers** (from Babelfish version 3.3.0)

Catalog views

- **sys.servers**
- **sys.linked_logins**

Using encryption in transit for the connection

The connection from the source Babelfish for Aurora PostgreSQL server to the target remote server uses encryption in transit (TLS/SSL), depending on the remote server database configuration. If the remote server isn't configured for encryption, the Babelfish server making the request to the remote database falls back to unencrypted.

To enforce connection encryption

- If the target linked server is an RDS for SQL Server instance, set `rds.force_ssl = on` for the target SQL Server instance. For more information about SSL/TLS configuration for RDS for SQL Server, see Using SSL with a Microsoft SQL Server DB instance
- If the target linked server is a Babelfish for Aurora PostgreSQL cluster, set `babelfishpg_tsql.tds_ssl_encrypt = on` and `ssl = on` for the target server. For more information about SSL/TLS, see Babelfish SSL settings and client connections.

Adding Babelfish as a linked server from SQL Server

Babelfish for Aurora PostgreSQL can be added as a linked server from a SQL Server. On a SQL Server database, you can add Babelfish as a linked server using Microsoft OLE DB provider for ODBC : MSDASQL.

There are two ways to configure Babelfish as a linked server from SQL Server using MSDASQL provider:

- Providing ODBC connection string as the provider string.
- Provide the System DSN of ODBC data source while adding the linked server.

Limitations

- OPENQUERY() works only for SELECT and doesn't work for DML.
- Four-part object names work only for reading and doesn't work for modifying the remote table. An UPDATE can reference a remote table in the FROM clause without modifying it.
- Executing stored procedures against Babelfish linked servers isn't supported.
- Babelfish major version upgrade might not work if there are objects dependent on `OPENQUERY()` or objects referenced through four-part names. You must ensure that any objects referencing `OPENQUERY()` or four-part names are dropped before a major version upgrade.
- The following datatypes don't work as expected against remote Babelfish server: `nvarchar(max)`, `varchar(max)`, `varbinary(max)`, `binary(max)` and `time`. We recommend using the CAST function to convert these to the supported datatypes.

Example

In the following example, a Babelfish for Aurora PostgreSQL instance is connecting to an instance of RDS for SQL Server in the cloud.

```
EXEC master.dbo.sp_addlinkedserver @server=N'rds_sqlserver', @srvproduct=N'',
 @provider=N'SQLNCLI', @datasrc=N'myserver.CB2XKFSFFMY7.US-WEST-2.RDS.AMAZONAWS.COM';
EXEC master.dbo.sp_addlinkedsrvlogin
 @rmtsrvname=N'rds_sqlserver',@useself=N'False',@locallogin=NULL,@rmtuser=N'username',@rmtpassw
```

When the linked server is in place, you can then use T-SQL OPENQUERY() or standard four-part naming to reference a table, view, or other supported objects, on the remote server:

```
SELECT * FROM OPENQUERY(rds_sqlserver, 'SELECT * FROM TestDB.dbo.t1');
SELECT * FROM rds_sqlserver.TestDB.dbo.t1;
```

To drop the linked server and all associated logins:

```
EXEC master.dbo.sp_dropserver @server=N'rds_sqlserver', @droplogins=N'droplogins';
```

Troubleshooting

You can use the same security group for both source and remote servers to allow them to communicate with each other. Security group should allow only inbound traffic on TDS port (1433 by default) and source IP in security group can be set as the security group ID itself. For more information on how to set the rules for connecting to an instance from another instance with the same security group, see Rules to connect to instances from an instance with the same security group.

If access isn't configured correctly, an error message similar to the following example appears when you try to query the remote server.

```
TDS client library error: DB #: 20009, DB Msg: Unable to connect: server is unavailable
 or does not exist (mssql2019.aws-region.rds.amazonaws.com), OS #: 110, OS Msg:
 Connection timed out, Level: 9
```

Using Full Text Search in Babelfish

Starting with version 4.0.0, Babelfish provides limited support for Full Text Search (FTS). FTS is a powerful feature in relational databases that enables efficient searching and indexing of text-heavy data. It allows you to perform complex text searches and retrieve relevant results quickly. FTS is particularly valuable for applications that deal with large volumes of textual data, such as content management systems, e-commerce platforms, and document archives.

Understanding Babelfish Full Text Search supported features

Babelfish supports the following Full Text Search features:

- CONTAINS Clause:
  - Basic support for the CONTAINS clause.

    ```
    CONTAINS (
         {
            column_name
         }
         , '<contains_search_condition>'
    )
    ```

    > **Note**
    >
    > Currently, only English language is supported.

  - Comprehensive handling and translation of `simple_term` search strings.
- `FULLTEXT INDEX` Clause:
  - Supports only `CREATE FULLTEXT INDEX ON table_name(column_name [...n]) KEY INDEX index_name` statement.
  - Supports full `DROP FULLTEXT INDEX` statement.

> **Note**
>
> In order to re-index the Full Text Index, you need to drop the Full Text Index and create a new one on the same column.

- Special characters in search condition:
  - Babelfish ensures that single occurrences of special characters in search strings are handled effectively.

    > **Note**
    >
    > While Babelfish now identifies special characters in search string, it's essential to recognize that the results obtained may vary compared to those obtained with T-SQL.

- Table alias in column_name:
  - With table alias support, users can create more concise and readable SQL queries for Full-Text Search.

Limitations in Babelfish Full Text Search

- Currently, the following options aren't supported in Babelfish for `CONTAINS` Clause.
  - Special characters and Languages other than English aren't supported. You will receive the generic error message for unsupported characters and language

    ```
    Full-text search conditions with special characters or languages other than English
     are not currently supported in Babelfish
    ```

  - Multiple columns like `column_list`
  - PROPERTY attribute
  - `prefix_term`, `generation_term`, `generic_proximity_term`, `custom_proximity_term`, and `weighted_term`
  - Boolean operators aren't supported and you will receive the following error message when used:

```
boolean operators not supported
```

  - Identifier names with dots aren't supported.
- Currently, the following options aren't supported in Babelfish for `CREATE FULLTEXT INDEX` Clause.
  - [TYPE COLUMN type_column_name]
  - [LANGUAGE language_term]
  - [STATISTICAL_SEMANTICS]
  - catalog filegroup options
  - with options
- Creating a full text catalog isn't supported. Creating a full text index doesn't require a full text catalog.
- `CREATE FULLTEXT INDEX` doesn't support identifier names with dots.
- Babelfish doesn't currently support consecutive special characters in search strings. You will receive the following error message when used:

```
Consecutive special characters in the full-text search condition are not currently
 supported in Babelfish
```

Babelfish supports Geospatial data types

Starting with versions 3.5.0 and 4.1.0, Babelfish includes support for the following two spatial data types:

- **Geometry data type –** This data type is intended for storing planar or Euclidean (flat-earth) data.
- **Geography data type –** This data type is intended for storing ellipsoidal or round-earth data, such as GPS latitude and longitude coordinates.

These data types allow for the storage and manipulation of spatial data, but with limitations.

Understanding the Geospatial data types in Babelfish

- Geospatial data types are supported in various database objects such as views, procedures, and tables.

- Supports 2-D point data type to store location data as points defined by latitude, longitude, and a valid Spatial Reference System Identifier (SRID).
- Applications connecting to Babelfish through drivers like JDBC, ODBC, DOTNET, and PYTHON can utilize this Geospatial feature.

Geometry data type functions supported in Babelfish

- **STGeomFromText (*`geometry_tagged_text`*, SRID)** – Creates a geometry instance using Well-Known Text (WKT) representation.
- **STPointFromText (*`point_tagged_text`*, SRID)** – Creates a point instance using WKT representation.
- **Point (X, Y, SRID)** – Creates a point instance using float values of x and y coordinates.
- **<geometry_instance>.STAsText ( )** – Extracts WKT representation from geometry instance.
- **<geometry_instance>.STDistance (other_geometry)** – Calculates the distance between two geometry instances.
- **<geometry_instance>.STX** – Extracts the X coordinate (longitude) for geometry instance.
- **<geometry_instance>.STY** – Extracts the Y coordinate (latitude) for geometry instance.

Geography data type functions supported in Babelfish

- **STGeomFromText (*`geography_tagged_text`*, SRID)** – Creates a geography instance using WKT representation.
- **STPointFromText (*`point_tagged_text`*, SRID)** – Creates a point instance using WKT representation.
- **Point (Lat, Long, SRID)** – Creates a point instance using float values of Latitude and Longitude.
- **<geography_instance>.STAsText ( )** – Extracts WKT representation from geography instance.
- **<geography_instance>.STDistance (other_geography)** – Calculates the distance between two geography instances.
- **<geography_instance>.Lat** – Extracts the Latitude value for geography instance.
- **<geography_instance>.Long** – Extracts the Longitude value for geography instance.

Limitations in Babelfish for Geospatial data types

- Currently, Babelfish doesn't support more advanced features like Z-M flags for point instances of Geospatial data types.
- Geometry types other than point instance aren't currently supported:
  - LineString
  - CircularString
  - CompoundCurve
  - Polygon
  - CurvePolygon
  - MultiPoint
  - MultiLineString
  - MultiPolygon
  - GeometryCollection
- Currently, spatial indexing isn't supported for Geospatial data types.
- Only the listed functions are currently supported for these data types. For more information, see Geometry data type functions supported in Babelfish and Geography data type functions supported in Babelfish.
- STDistance function output for Geography data might have minor precision variations compared to T-SQL. This is due to the underlying PostGIS implementation. For more information, see ST_Distance
- For optimal performance, use built-in Geospatial data types, without creating additional layers of abstraction in Babelfish.

> **Tip**
>
> While you can create custom data types, it's not recommended to create it on top of Geospatial data. This could introduce complexities, potentially leading to unexpected behavior due to the limited support.

- In Babelfish, Geospatial function names are used as keywords and will perform spatial operations only if used in the intended way.

Tip

When creating user-defined functions and procedures in Babelfish, avoid using the same names as built-in Geospatial functions. If you have any existing database objects with the same names, use `sp_rename` to rename them.

Troubleshooting Babelfish

Following, you can find troubleshooting ideas and workarounds for some Babelfish DB cluster issues.

Topics

Connection failure

Common causes of connection failures to a new Aurora DB cluster with Babelfish include the following:

- **Security group doesn't allow access** – If you're having trouble connecting to a Babelfish, make sure that you added your IP address to the default Amazon EC2 security group. You can use https://checkip.amazonaws.com/ to determine your IP address and then add it to your in-bound rule for the TDS port and the PostgreSQL port. For more information, see Add rules to a security group in the *Amazon EC2 User Guide*.
- **Mismatching SSL configurations** – If the rds.force_ssl parameter is turned on (set to 1) on Aurora PostgreSQL, then clients must connect to Babelfish over SSL. If your client isn't set up correctly, you see an error message such as the following:

  ```
  Cannot connect to your-Babelfish-DB-cluster, 1433
  ---------------------
  ADDITIONAL INFORMATION:
  no pg_hba_conf entry for host "256.256.256.256", user "your-user-name",
  "database babelfish_db", SSL off (Microsoft SQL Server, Error: 33557097)
  ...
  ```

  This error indicates a possible SSL configuration issue between your local client and the Babelfish DB cluster, and that the cluster requires clients to use SSL (its rds.force_ssl parameter is set to 1). For more information about configuring SSL, see Using SSL with a PostgreSQL DB instance in the *Amazon RDS User Guide*.

  If you are using SQL Server Management Studio (SSMS) to connect to Babelfish and you see this error, you can choose **Encrypt connection** and **Trust server certificate** connection options on the Connection Properties pane and try again. These settings handle the SSL connection requirement for SSMS.

For more information about troubleshooting Aurora connection issues, see Can't connect to Amazon RDS DB instance.

Turning off Babelfish

When you no longer need Babelfish, you can turn off Babelfish functionality.

Be aware of some considerations:

- In some cases, you might turn off Babelfish before completing a migration to Aurora PostgreSQL. If you do and your DDL depends on SQL Server data types or you use any T-SQL functionality in your code, your code fails.
- If after provisioning a Babelfish instance you turn off the Babelfish extension, you can't provision that same database again on the same cluster.

To turn off Babelfish, modify your parameter group, setting `rds.babelfish_status` to `OFF`. You can continue to use your SQL Server data types with Babelfish off, by setting `rds.babelfish_status` to `datatypeonly`.

If you turn off Babelfish in parameter group, all clusters that use that parameter group lose Babelfish functionality.

For more information about modifying parameter groups, see Working with parameter groups. For information about Babelfish–specific parameters, see DB cluster parameter group settings for Babelfish.

Babelfish version updates

Babelfish is an option available with Aurora PostgreSQL version 13.4 and higher releases. Updates to Babelfish become available with certain new releases of the Aurora PostgreSQL database engine. For more information, see the *Release Notes for Aurora PostgreSQL*.

> **Note**
>
> Babelfish DB clusters running on any version of Aurora PostgreSQL 13 can't be upgraded to Aurora PostgreSQL 14.3, 14.4, and 14.5. Also, Babelfish doesn't support a direct upgrade from 13.x to 15.x. You must first upgrade your 13.x DB cluster to 14.6 and higher version and then upgrade to 15.x version.

For a list of supported functionality across different Babelfish releases, see Supported functionality in Babelfish by version.

For a list of currently unsupported functionality, see Unsupported functionality in Babelfish.

You can use the describe-db-engine-versions AWS CLI command to get a list of Aurora PostgreSQL versions in your AWS Region that support Babelfish, as shown in the following example.

For Linux, macOS, or Unix:

```
$ aws rds describe-db-engine-versions --region us-east-1 \
    --engine aurora-postgresql \
    --query '*[]|[?SupportsBabelfish==`true`].[EngineVersion]' \
    --output text
13.4
13.5
13.6
13.7
13.8
14.3
14.4
14.5
14.6
14.7
14.8
14.9
14.10
15.2
```

```
15.3
15.4
15.5
16.1
```

For more information, see describe-db-engine-versions in the *AWS CLI Command Reference*.

In the following topics, you can learn how to identify the version of Babelfish running on your Aurora PostgreSQL DB cluster, and how to upgrade to a new version.

Contents

Identifying your version of Babelfish

You can query Babelfish to find details about the Babelfish version, the Aurora PostgreSQL version, and the compatible Microsoft SQL Server version. You can use the TDS port or the PostgreSQL port.

- To use the TDS port to query for version information
- To use the PostgreSQL port to query for version information

To use the TDS port to query for version information

1. Use `sqlcmd` or `ssms` to connect to the endpoint for your Babelfish DB cluster.

```
sqlcmd -S bfish_db.cluster-123456789012.aws-region.rds.amazonaws.com,1433 -U
    login-id -P password -d db_name
```

2. To identify the Babelfish version, run the following query:

```
1> SELECT CAST(serverproperty('babelfishversion') AS VARCHAR)
2> GO
```

   The query returns results similar to the following:

```
serverproperty
------------------------------
3.4.0

(1 rows affected)
```

3. To identify the version of the Aurora PostgreSQL DB cluster, run the following query:

```
1> SELECT aurora_version() AS aurora_version
2> GO
```

   The query returns results similar to the following:

```
aurora_version


-------------------------------------------------
15.5.0

(1 rows affected)
```

4. To identify the compatible Microsoft SQL Server version, run the following query:

```
1> SELECT @@VERSION AS version
2> GO
```

   The query returns results similar to the following:

```
Babelfish for Aurora PostgreSQL with SQL Server Compatibility - 12.0.2000.8
Dec 7 2023 09:43:06
```

```
Copyright (c) Amazon Web Services
PostgreSQL 15.5 on x86_64-pc-linux-gnu (Babelfish 3.4.0)

(1 rows affected)
```

As an example that shows one minor difference between Babelfish and Microsoft SQL Server, you can run the following query. On Babelfish, the query returns 1, while on Microsoft SQL Server, the query returns NULL.

```
SELECT CAST(serverproperty('babelfish') AS VARCHAR) AS runs_on_babelfish
```

You can also use the PostgreSQL port to obtain version information, as shown in the following procedure.

To use the PostgreSQL port to query for version information

1. Use psql or pgAdmin to connect to the endpoint for your Babelfish DB cluster.

   ```
   psql host=bfish_db.cluster-123456789012.aws-region.rds.amazonaws.com
        port=5432 dbname=babelfish_db user=sa
   ```

2. Turn on the extended feature (\x) of psql for more readable output.

   ```
   babelfish_db=> \x
   babelfish_db=> SELECT
   babelfish_db=> aurora_version() AS aurora_version,
   babelfish_db=> version() AS postgresql_version,
   babelfish_db=> sys.version() AS Babelfish_compatibility,
   babelfish_db=> sys.SERVERPROPERTY('BabelfishVersion') AS Babelfish_Version;
   ```

   The query returns output similar to the following:

   ```

   -[ RECORD 1 ]-----------
   +-------------------------------------------------------------------------------------------
   aurora_version          | 15.5.0
   postgresql_version      | PostgreSQL 15.5 on x86_64-pc-linux-gnu, compiled by
    x86_64-pc-linux-gnu-gcc (GCC) 9.5.0, 64-bit
   babelfish_compatibility | Babelfish for Aurora Postgres with SQL Server
    Compatibility - 12.0.2000.8                     +
   ```

```
                        | Dec 7 2023 09:43:06
                                    +
                        | Copyright (c) Amazon Web Services
                                     +
                        | PostgreSQL 15.5 on x86_64-pc-linux-gnu (Babelfish 3.4.0)
babelfish_version       | 3.4.0
```

Upgrading your Babelfish cluster to a new version

New versions of Babelfish become available with some new releases of the Aurora PostgreSQL database engine after version 13.4. Each new release of Babelfish has its own version number. As with Aurora PostgreSQL, Babelfish uses the *major*.*minor*.*patch* naming scheme for versions. For example, the first Babelfish release, Babelfish version 1.0.0, became available as part of Aurora PostgreSQL 13.4.0.

Babelfish doesn't require a separate installation process. As discussed in Creating a Babelfish for Aurora PostgreSQL DB cluster, **Turn on Babelfish** is an option that you choose when you create an Aurora PostgreSQL DB cluster.

Likewise, you can't upgrade Babelfish independently from the supporting Aurora DB cluster. To upgrade an existing Babelfish for Aurora PostgreSQL DB cluster to a new version of Babelfish, you upgrade the Aurora PostgreSQL DB cluster to a new version that supports the version of Babelfish that you want to use. The procedure that you follow for the upgrade depends on the version of Aurora PostgreSQL that's supporting your Babelfish deployment, as follows.

Major version upgrades

You must upgrade the following Aurora PostgreSQL versions to Aurora PostgreSQL 14.6 and higher version before upgrading to Aurora PostgreSQL 15.2 version.

- Aurora PostgreSQL 13.8 and all higher versions
- Aurora PostgreSQL 13.7.1 and all higher minor versions
- Aurora PostgreSQL 13.6.4 and all higher minor versions

You can upgrade Aurora PostgreSQL 14.6 and higher versions to Aurora PostgreSQL 15.2 and higher versions.

Upgrading an Aurora PostgreSQL DB cluster to a new major version involves several preliminary tasks. For more information, see How to perform a major version upgrade. To successfully upgrade your Babelfish for Aurora PostgreSQL DB cluster, you need to create a custom DB cluster parameter group for the new Aurora PostgreSQL version. This new parameter group must contain the same Babelfish parameter settings as that of the cluster that you're upgrading. For more information and for a table of major version upgrade sources and targets, see Upgrading Babelfish to a new major version.

Minor version upgrades and patches

Minor versions and patches don't require the creation of a new DB cluster parameter group for the upgrade. Minor versions and patches can use the minor version upgrade process, whether that's applied automatically or manually. For more information and a table of version sources and targets, see Upgrading Babelfish to a new minor version.

> **Note**
>
> Before performing a major or a minor upgrade, apply all pending maintenance tasks to your Babelfish for Aurora PostgreSQL cluster.

Topics

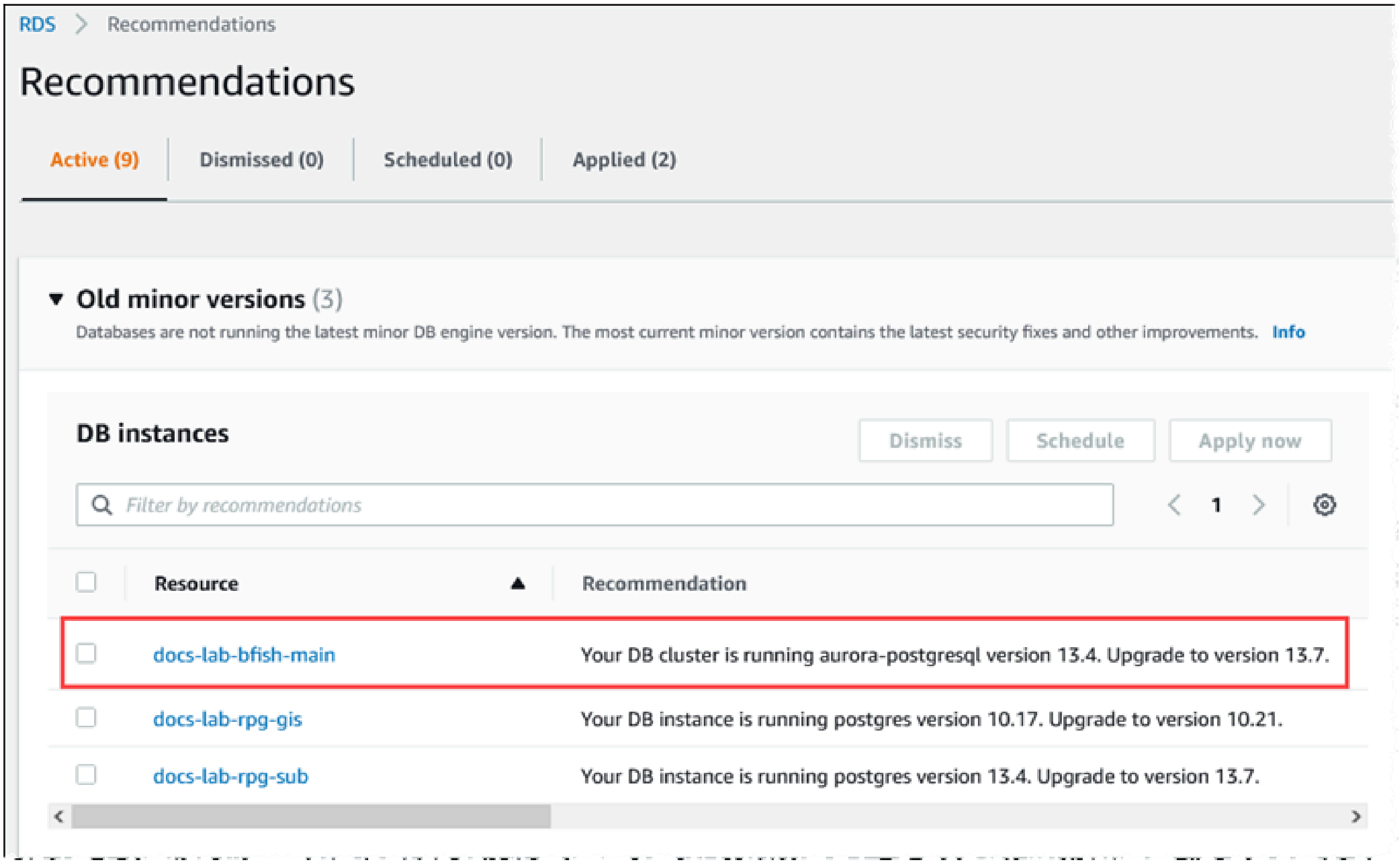
- Upgrading Babelfish to a new minor version
- Upgrading Babelfish to a new major version

Upgrading Babelfish to a new minor version

A new minor version includes only changes that are backward compatible. A *patch* version includes important fixes for a minor version after its release. For example, the version label for the first release of Aurora PostgreSQL 13.4 was Aurora PostgreSQL 13.4.0. Several patches for that minor version have been released to date, including Aurora PostgreSQL 13.4.1, 13.4.2, and 13.4.4. You can find the patches available for each Aurora PostgreSQL version in the **Patch releases** list at the top of the Aurora PostgreSQL release notes for that version. For an example, see PostgreSQL 14.3 in the *Release Notes for Aurora PostgreSQL*.

If your Aurora PostgreSQL DB cluster is configured with the **Auto minor version upgrade** option, your Babelfish for Aurora PostgreSQL DB cluster is upgraded automatically during the cluster's maintenance window. To learn more about auto minor version upgrade (AmVU) and how to use it, see Automatic minor version upgrades for Aurora DB clusters. If your cluster isn't using AmVU, you can manually upgrade your Babelfish for Aurora PostgreSQL DB cluster to new minor versions either by responding to maintenance tasks, or by modifying the cluster to use the new version.

When you choose an Aurora PostgreSQL version to install and when you view an existing Aurora PostgreSQL DB cluster in the AWS Management Console, the version displays the *major.minor* digits only. For example, the following image from the Console for an existing Babelfish for Aurora PostgreSQL DB cluster with Aurora PostgreSQL 13.4 recommends upgrading the cluster to version 13.7, a new minor release of Aurora PostgreSQL.


RDS > Recommendations
Recommendations
Active (9)
Dismissed (0)
Scheduled (0)
Applied (2)
▼ Old minor versions (3)
Databases are not running the latest minor DB engine version. The most current minor version contains the latest security fixes and other improvements. Info
DB instances
Dismiss
Schedule
Apply now
Filter by recommendations
1

□	Resource ▲	Recommendation
□	docs-lab-bfish-main	Your DB cluster is running aurora-postgresql version 13.4. Upgrade to version 13.7.
□	docs-lab-rpg-gis	Your DB instance is running postgres version 10.17. Upgrade to version 10.21.
□	docs-lab-rpg-sub	Your DB instance is running postgres version 13.4. Upgrade to version 13.7.

To get complete version details, including the *patch* level, you can query the Aurora PostgreSQL DB cluster using the `aurora_version` Aurora PostgreSQL function. For more information, see [aurora_version](#) in the [Aurora PostgreSQL functions reference](#). You can find an example of using the function in the [To use the PostgreSQL port to query for version information](#) procedure in [Identifying your version of Babelfish](#).

The following table shows Aurora PostgreSQL and Babelfish version and the available target versions that can support the minor version upgrade process.

Current source versions		Newest upgrade targets		Other available upgrade versions			
Aurora PostgreSQL	Babelfish	Aurora PostgreSQL	Babelfish	Aurora PostgreSQL versions with Babelfish option			
15.4	3.3.0	15.5	3.4.0				
15.3.2	3.2.1	15.5	3.4.0	15.4			
15.2.4	3.1.3	15.5	3.4.0	15.4	15.3		
14.9.1	2.6.0	14.10	2.7.0				
14.8.2	2.5.1	14.10	2.7.0	14.9.1			
14.7.4	2.4.3	14.10	2.7.0	14.9.1	14.8.2		
14.6.4	2.3.3	14.10	2.7.0	14.9.1	14.8.2	14.7.4	
14.5.3	2.2.3	14.10	2.7.0	14.9.1	14.8.2	14.7.4	14.6.4
14.3.1	2.1.1	14.6	2.3.0				
14.3.0	2.1.0	14.6	2.3.0	14.3.1			
13.8	1.4.0	13.9	1.5				
13.7.1	1.3.1	13.9	1.5	13.8			
13.7.0	1.3.0	13.9	1.5	13.7.1			
13.6.4	1.2.4	13.9	1.5	13.7			
13.6.3	1.2.1	13.9	1.5	13.7	13.6.4		
13.6.2	1.2.1	13.9	1.5	13.7	13.6.4		
13.6.1	1.2.0	13.9	1.5	13.7	13.6.4		

Current source versions		Newest upgrade targets		Other available upgrade versions			
13.6.0	1.2.0	13.9	1.5	13.7	13.6.4		
13.5	1.1.0	13.9	1.5	13.7	13.6		
13.4	1.0.0	13.9	1.5	13.7	13.6	13.5	

Upgrading Babelfish to a new major version

For a major version upgrade, you need to first upgrade your Babelfish for Aurora PostgreSQL DB cluster to a version that supports the major version upgrade. To achieve this, apply patch updates or minor version upgrades to your DB cluster. For more information,see Upgrading Babelfish to a new minor version.

The following table shows Aurora PostgreSQL version and Babelfish version that can support a major version upgrade.

Current source versions		Newest available upgrade target		Other available versions (minor version upgrades)		
Aurora PostgreSQL	Babelfish	Aurora PostgreSQL	Babelfish	Aurora PostgreSQL version (Babelfish version)		
15.5	3.4.0	16.1	4.0.0			
15.4	3.3.0	16.1	4.0.0			
15.3	3.2.0	16.1	4.0.0			
15.2	3.1.0	16.1	4.0.0			
14.10	2.7.0	15.5	3.4.0			
14.9	2.6.0	15.5	3.4.0	15.4(3.3.0)		
14.8	2.5.0	15.5	3.4.0	15.4(3.3.0)	15.3(3.2.0)	

Current source versions		Newest available upgrade target		Other available versions (minor version upgrades)		
14.7	2.4.0	15.5	3.4.0	15.4(3.3.0)	15.3(3.2.0)	15.2(3.1.0)
14.6	2.3.0	15.5	3.4.0	15.4(3.3.0)	15.3(3.2.0)	15.2(3.1.0)
13.9	1.5.0	14.6	2.3.0			
13.8	1.4.0	14.6	2.3.0			
13.7.1	1.3.1	14.6	2.3.0	13.8 (1.4)		
13.6.4	1.2.2	14.6	2.3.0	13.8 (1.4)	13.7 (1.3)	

Before upgrading Babelfish to a new major version

An upgrade might involve brief outages. For that reason, we recommend that you perform or schedule upgrades during your maintenance window or during other periods of low usage.

Before you perform a major version upgrade

1. Identify the Babelfish version of your existing Aurora PostgreSQL DB cluster by using the commands outlined in Identifying your version of Babelfish. The Aurora PostgreSQL version and Babelfish version information is handled by PostgreSQL, so follow the steps detailed in the To use the PostgreSQL port to query for version information procedure to get the details.
2. Verify if your version supports the major version upgrade. For the list of versions that support the major version upgrade feature, see Upgrading Babelfish to a new minor version and perform the necessary pre-upgrade tasks.

   For example, if your Babelfish version is running on an Aurora PostgreSQL 13.5 DB cluster and you want to upgrade to Aurora PostgreSQL 15.2, then first apply all the minor releases and patches to upgrade your cluster to Aurora PostgreSQL 14.6 or higher version. When your cluster is at version 14.6 or higher, continue with the major version upgrade process.
3. Create a manual snapshot of your current Babelfish DB cluster as a backup. The backup lets you restore the cluster to its Aurora PostgreSQL version, Babelfish version, and restore all data to the state before the upgrade. For more information, see Creating a DB cluster snapshot. Be sure to keep your existing custom DB cluster parameter group to use again if you decide to

restore this cluster to its pre-upgraded state. For more information, see Restoring from a DB cluster snapshot and Parameter group considerations.

4. Prepare a custom DB cluster parameter group for the target Aurora PostgreSQL DB version. Duplicate the settings for the Babelfish parameters from your current Babelfish for Aurora PostgreSQL DB cluster. To find a list of all Babelfish parameters, see DB cluster parameter group settings for Babelfish. For a major version upgrade, the following parameters require the same settings as the source DB cluster. For the upgrade to succeed, all the settings must be the same.

   - rds.babelfish_status
   - babelfishpg_tds.tds_default_numeric_precision
   - babelfishpg_tds.tds_default_numeric_scale
   - babelfishpg_tsql.database_name
   - babelfishpg_tsql.default_locale
   - babelfishpg_tsql.migration_mode
   - babelfishpg_tsql.server_collation_name

   > **Warning**
   >
   > If the settings for the Babelfish parameters in the custom DB cluster parameter group for the new Aurora PostgreSQL version don't match the parameter values of the cluster that you're upgrading, the `ModifyDBCluster` operation fails. An `InvalidParameterCombination` error message appears in the AWS Management Console or in the output from the `modify-db-cluster` AWS CLI command.

5. Use the AWS Management Console or the AWS CLI to create the custom DB cluster parameter group. Choose the applicable Aurora PostgreSQL family for the version of Aurora PostgreSQL that you want for the upgrade.

   > **Tip**
   >
   > Parameter groups are managed at the AWS Region level. When you work with AWS CLI, you can configure with a default Region instead of specifying the `--region` in the command. To learn more about using the AWS CLI, see Quick setup in the *AWS Command Line Interface User Guide*.

Performing major version upgrade

1. Upgrade Aurora PostgreSQL DB cluster to a new major version. For more information, see Upgrading the Aurora PostgreSQL engine to a new major version.
2. Reboot the writer instance of the cluster, so that the parameter settings can take effect.

After upgrading to a new major version

After a major version upgrade to a new Aurora PostgreSQL version, the IDENTITY value in tables with an IDENTITY column might be larger (+32) than the value was before the upgrade. The result is that when the next row is inserted into such tables, the generated identity column value jumps to the +32 number and starts the sequence from there. This condition won't negatively affect the functions of your Babelfish DB cluster. However, if you want, you can reset the sequence object based on the maximum value of the column. To do so, connect to the T-SQL port on your Babelfish writer instance using sqlcmd or another SQL Server client. For more information, see Using a SQL Server client to connect to your DB cluster.

```
sqlcmd -S bfish-db.cluster-123456789012.aws-region.rds.amazonaws.com,1433 -U
     sa -P ******** -d dbname
```

When connected, use the following SQL command to generate statements that you can use to seed the associated sequence object. This SQL command works for both single database and multiple database Babelfish configurations. For more information about these two deployment models, see Using Babelfish with a single database or multiple databases.

```
DECLARE @schema_prefix NVARCHAR(200) = ''
IF current_setting('babelfishpg_tsql.migration_mode') = 'multi-db'
    SET @schema_prefix = db_name() + '_'
SELECT 'SELECT setval(pg_get_serial_sequence(''' + @schema_prefix +
 schema_name(tables.schema_id)
    + '.' + tables.name + ''', ''' + columns.name + '''),(select max(' + columns.name +
 ')
    FROM ' + schema_name(tables.schema_id) + '.' + tables.name + '));
    'FROM sys.tables tables JOIN sys.columns
    columns ON tables.object_id = columns.object_id
    WHERE columns.is_identity = 1
GO
```

The query generates a series of SELECT statements that you can then run to reset the maximum IDENTITY value and close any gap. The following shows the output when using the sample SQL Server database, Northwind, running on a Babelfish cluster.

```
--------------------------------------------------------
SELECT setval(pg_get_serial_sequence('northwind_dbo.categories', 'categoryid'),(select
 max(categoryid)
    FROM dbo.categories));

SELECT setval(pg_get_serial_sequence('northwind_dbo.orders', 'orderid'),(select
 max(orderid)
    FROM dbo.orders));


SELECT setval(pg_get_serial_sequence('northwind_dbo.products', 'productid'),(select
 max(productid)
    FROM dbo.products));


SELECT setval(pg_get_serial_sequence('northwind_dbo.shippers', 'shipperid'),(select
 max(shipperid)
    FROM dbo.shippers));


SELECT setval(pg_get_serial_sequence('northwind_dbo.suppliers', 'supplierid'),(select
 max(supplierid)
    FROM dbo.suppliers));


(5 rows affected)
```

Run the statements one by one to reset the sequence values.

Example: Upgrading the Babelfish DB cluster to a major release

In this example, you can find the series of AWS CLI commands that explains how to upgrade an Aurora PostgreSQL 13.6.4 DB cluster running Babelfish version 1.2.2 to Aurora PostgreSQL 14.6. First, you create a custom DB cluster parameter group for Aurora PostgreSQL 14. Next, you modify the parameter values to match those of your Aurora PostgreSQL version 13 source. Finally, you perform the upgrade by modifying the source cluster. For more information, see DB cluster parameter group settings for Babelfish. In that topic, you can also find information about using the AWS Management Console to perform the upgrade.

Use the create-db-cluster-parameter-group CLI command to create the DB cluster parameter group for the new version.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster-parameter-group \
    --db-cluster-parameter-group-name docs-lab-babelfish-apg-14 \
    --db-parameter-group-family aurora-postgresql14 \
    --description 'New custom parameter group for upgrade to new major version' \
    --region us-west-1
```

When you issue this command, the custom DB cluster parameter group is created in the AWS Region. You see output similar to the following.

```
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupName": "docs-lab-babelfish-apg-14",
        "DBParameterGroupFamily": "aurora-postgresql14",
        "Description": "New custom parameter group for upgrade to new major version",
        "DBClusterParameterGroupArn": "arn:aws:rds:us-west-1:111122223333:cluster-
pg:docs-lab-babelfish-apg-14"
    }
}
```

For more information, see Creating a DB cluster parameter group.

Use the modify-db-cluster-parameter-group CLI command to modify the settings so that they match the source cluster.

For Windows:

```
aws rds modify-db-cluster-parameter-group --db-cluster-parameter-group-name docs-lab-
babelfish-apg-14 ^
  --parameters
 "ParameterName=rds.babelfish_status,ParameterValue=on,ApplyMethod=pending-reboot" ^
 "ParameterName=babelfishpg_tds.tds_default_numeric_precision,ParameterValue=38,ApplyMethod=pen
reboot" ^
 "ParameterName=babelfishpg_tds.tds_default_numeric_scale,ParameterValue=8,ApplyMethod=pending-
reboot" ^
 "ParameterName=babelfishpg_tsql.database_name,ParameterValue=babelfish_db,ApplyMethod=pending-
reboot" ^
```

```
 "ParameterName=babelfishpg_tsql.default_locale,ParameterValue=en-
US,ApplyMethod=pending-reboot" ^
 "ParameterName=babelfishpg_tsql.migration_mode,ParameterValue=single-
db,ApplyMethod=pending-reboot" ^
 "ParameterName=babelfishpg_tsql.server_collation_name,ParameterValue=sql_latin1_general_cp1_ci
reboot"
```

The response looks similar to the following.

```
{
    "DBClusterParameterGroupName": "docs-lab-babelfish-apg-14"
}
```

Use the modify-db-cluster CLI command to modify the cluster to use the new version and the new custom DB cluster parameter group. You also specify the `--allow-major-version-upgrade` argument, as shown in the following sample.

```
aws rds modify-db-cluster \
--db-cluster-identifier docs-lab-bfish-apg-14 \
--engine-version 14.6 \
--db-cluster-parameter-group-name docs-lab-babelfish-apg-14 \
--allow-major-version-upgrade \
--region us-west-1 \
--apply-immediately
```

Use the reboot-db-instance CLI command to reboot the writer instance of the cluster, so that the parameter settings can take effect.

```
aws rds reboot-db-instance \
--db-instance-identifier docs-lab-bfish-apg-14-instance-1\
--region us-west-1
```

Using Babelfish product version parameter

A new Grand Unified Configuration (GUC) parameter called `babelfishpg_tds.product_version` is introduced from Babelfish 2.4.0 and 3.1.0 versions. This parameter allows you to set the SQL Server product version number as the output of Babelfish.

The parameter is a 4-part version ID string, and each part should be separated by ".".

Syntax

```
Major.Minor.Build.Revision
```

- Major Version: A number between 11 and 16.
- Minor Version: A number between 0 and 255.
- Build Version: A number between 0 and 65535.
- Revision: 0 and any positive number.

Configuring Babelfish product version parameter

You must use the cluster parameter group to set the babelfishpg_tds.product_version parameter in the console. For more information on how to modify the DB cluster parameter, see [Modifying parameters in a DB cluster parameter group](#).

When you set the product version parameter to an invalid value, the change will not take effect. Although the console might show you the new value, the parameter retains the previous value. Check the engine log file for details about the error messages.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
--db-cluster-parameter-group-name mydbparametergroup \
--parameters
 "ParameterName=babelfishpg_tds.product_version,ParameterValue=15.2.4000.1,ApplyMethod=immediat
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
--db-cluster-parameter-group-name mydbparametergroup ^
--parameters
 "ParameterName=babelfishpg_tds.product_version,ParameterValue=15.2.4000.1,ApplyMethod=immediat
```

Affected queries and parameter

Query/Parameter	Result	Effective time
SELECT @@VERSION	Returns user defined SQL Server version (babelfishpg_tsql.version value = Default)	Immediately
SELECT SERVERPROPERTY('ProductVersion')	Returns user defined SQL Server version	Immediately
SELECT SERVERPROPERTY('ProductMajorVersion')	Returns Major Version of the user defined SQL Server version	Immediately
VERSION tokens in PRELOGIN Response Message	Server returns PRELOGIN messages with user defined SQL Server version	Takes effect when a user creates a new session
SQLServerVersion in LoginAck when using JDBC	DatabaseMetaData.getDatabaseProductVersion() returns user defined SQL Server version	Takes effect when a user creates a new session

Interface with babelfishpg_tsql.version parameter

You can set the output of the @@VERSION using the parameters babelfishpg_tsql.version and babelfishpg_tds.product_version. The following examples show how these two parameters interface.

- When babelfishpg_tsql.version parameter is "default" and babelfishpg_tds.product_version is 15.0.2000.8.
  - Output of @@version – 15.0.2000.8.
- When babelfishpg_tsql.version parameter is set to 13.0.2000.8 and babelfishpg_tds.product_version parameter is 15.0.2000.8.
  - Output of @@version – 13.0.2000.8.

Babelfish for Aurora PostgreSQL reference

Topics

Unsupported functionality in Babelfish

In the following table and lists, you can find functionality that isn't currently supported in Babelfish. Updates to Babelfish are included in Aurora PostgreSQL versions. For more information, see the *Release Notes for Aurora PostgreSQL*.

Topics

Functionality that isn't currently supported

In the table you can find information about certain functionality that isn't currently supported.

Functionality or syntax	Description
Assembly modules and SQL Common Language Runtime (CLR) routines	Functionality related to assembly modules and CLR routines isn't supported.
Column attributes	ROWGUIDCOL, SPARSE, FILESTREAM, and MASKED aren't supported.

Functionality or syntax	Description
Contained databases	Contained databases with logins authenticated at the database level rather than at the server level aren't supported.
Cursors (updatable)	Updatable cursors aren't supported.
Cursors (global)	GLOBAL cursors aren't supported.
Cursor (fetch behaviors)	The following cursor fetch behaviors aren't supported: FETCH PRIOR, FIRST, LAST, ABSOLUTE, abd RELATIVE
Cursor-typed output parameters	Cursor-typed variables and parameters aren't supported for output parameters (an error is raised).
Cursor options	SCROLL, KEYSET, DYNAMIC, FAST_FORWARD, SCROLL_LOCKS, OPTIMISTIC, TYPE_WARNING, and FOR UPDATE
Data encryption	Data encryption isn't supported.
Data-tier applications (DAC)	Data-tier applications (DAC) import or export operations with DAC package (.dacpac) or DAC backup (.bacpac) files aren't supported.
DBCC commands	Microsoft SQL Server Database Console Commands (DBCC) aren't supported. DBCC CHECKIDENT is supported in Babelfish 3.4.0 and higher releases.
DROP IF EXISTS	This syntax isn't supported for USER and SCHEMA objects. It's supported for the objects TABLE, VIEW, PROCEDURE, FUNCTION, and DATABASE.
Encryption	Built-in functions and statements don't support encryption.
ENCRYPT_CLIENT_CERT connections	Client certificate connections aren't supported.
EXECUTE AS statement	This statement isn't supported.
EXECUTE AS SELF clause	This clause isn't supported in functions, procedures, or triggers.

Functionality or syntax	Description
EXECUTE AS USER clause	This clause isn't supported in functions, procedures, or triggers.
Foreign key constraints referencing database name	Foreign key constraints that reference the database name aren't supported.
FORMAT	User-defined types aren't supported.
Function declarations with greater than 100 parameters	Function declarations that contain more than 100 parameters aren't supported.
Function calls that include DEFAULT as a parameter value	DEFAULT isn't a supported parameter value for a function call. DEFAULT as a parameter value for a function call is supported from Babelfish 3.4.0 and higher releases.
Functions, externally defined	External functions, including SQL CLR functions, aren't supported.
Global temporary tables (tables with names that start with ##)	Global temporary tables aren't supported.
Graph functionality	All SQL graph functionality isn't supported.
Identifiers (variables or parameters) with multiple leading @ characters	Identifiers that start with more than one leading @ aren't supported.
Identifiers, table or column names that contain @ or]] characters	Table or column names that contain an @ sign or square brackets aren't supported.
Inline indexes	Inline indexes aren't supported.
Invoking a procedure whose name is in a variable	Using a variable as a procedure name isn't supported.
Materialized views	Materialized views aren't supported.

Functionality or syntax	Description
NOT FOR REPLICATION clause	This syntax is accepted and ignored.
ODBC escape functions	ODBC escape functions aren't supported.
Partitioning	Table and index partitioning isn't supported.
Procedure calls that includes DEFAULT as a parameter value	DEFAULT isn't a supported parameter value. DEFAULT as a parameter value for a function call is supported from Babelfish 3.4.0 and higher releases.
Procedure declarations with more than 100 parameters	Declarations with more than 100 parameters aren't supported.
Procedures, externally defined	Externally defined procedures, including SQL CLR procedures, aren't supported.
Procedure versioning	Procedure versioning isn't supported.
Procedures WITH RECOMPILE	WITH RECOMPILE (when used in conjunction with the DECLARE and EXECUTE statements) isn't supported.
Remote object references	Executing procedures and functions using four-part names aren't supported. In remote objects, supports four-part object names for selected queries. For more information, see DB cluster parameter group settings for Babelfish.
Row-level security	Row-level security with CREATE SECURITY POLICY and inline table-valued functions isn't supported.
Service broker functionality	Service broker functionality isn't supported.
SESSIONPROPERTY	Unsupported properties: ANSI_NULLS, ANSI_PADDING, ANSI_WARNINGS, ARITHABORT, CONCAT_NULL_YIELDS_NULL, and NUMERIC_ROUNDABORT
SET LANGUAGE	This syntax isn't supported with any value other than `english` or `us_english` .

Functionality or syntax	Description
SP_CONFIGURE	This system stored procedure isn't supported.
SQL keyword SPARSE	The keyword SPARSE is accepted and ignored.
Table value constructor syntax (FROM clause)	The unsupported syntax is for a derived table constructed with the FROM clause.
Temporal tables	Temporal tables aren't supported.
Temporary procedures aren't dropped automatically	This functionality isn't supported.
Triggers, externally defined	These triggers aren't supported, including SQL Common Language Runtime (CLR).
Without SCHEMABINDING clause	Creating a view without SCHEMABINDING isn't supported, but the view is created as if WITH SCHEMABINDING was specified . Using SCHEMABINDING when creating functions, procedures, triggers is silently ignored.

Settings that aren't supported

The following settings aren't supported:

- SET ANSI_NULL_DFLT_OFF ON
- SET ANSI_NULL_DFLT_ON OFF
- SET ANSI_PADDING OFF
- SET ANSI_WARNINGS OFF
- SET ARITHABORT OFF
- SET ARITHIGNORE ON
- SET CURSOR_CLOSE_ON_COMMIT ON
- SET NUMERIC_ROUNDABORT ON
- SET PARSEONLY ON (command doesn't work as expected)
- SET FMTONLY ON (command doesn't work as expected. It suppresses only the execution of SELECT statements but not others.)

Commands that aren't supported

Certain functionality for the following commands isn't supported:

- ADD SIGNATURE
- ALTER DATABASE, ALTER DATABASE SET
- BACKUP/RESTORE DATABASE/LOG
- BACPAC and DACPAC FILES RESTORE
- CREATE, ALTER, DROP AUTHORIZATION. ALTER AUTHORIZATION is supported for database objects.
- CREATE, ALTER, DROP AVAILABILITY GROUP
- CREATE, ALTER, DROP BROKER PRIORITY
- CREATE, ALTER, DROP COLUMN ENCRYPTION KEY
- CREATE, ALTER, DROP DATABASE ENCRYPTION KEY
- CREATE, ALTER, DROP, BACKUP CERTIFICATE
- CREATE AGGREGATE
- CREATE CONTRACT
- CHECKPOINT

Column names or attributes that aren't supported

The following column names aren't supported:

- $IDENTITY
- $ROWGUID
- IDENTITYCOL

Data types that aren't supported

The following data types aren't supported:

- Geospatial (GEOGRAPHY and GEOMETRY)
- HIERARCHYID

Object types that aren't supported

The following object types aren't supported:

- COLUMN MASTER KEY
- CREATE, ALTER EXTERNAL DATA SOURCE
- CREATE, ALTER, DROP DATABASE AUDIT SPECIFICATION
- CREATE, ALTER, DROP EXTERNAL LIBRARY
- CREATE, ALTER, DROP SERVER AUDIT
- CREATE, ALTER, DROP SERVER AUDIT SPECIFICATION
- CREATE, ALTER, DROP, OPEN/CLOSE SYMMETRIC KEY
- CREATE, DROP DEFAULT
- CREDENTIAL
- CRYPTOGRAPHIC PROVIDER
- DIAGNOSTIC SESSION
- Indexed views
- SERVICE MASTER KEY
- SYNONYM

Functions that aren't supported

The following built-in functions aren't supported:

Aggregate functions

- APPROX_COUNT_DISTINCT
- CHECKSUM_AGG
- GROUPING_ID
- STRING_AGG using the WITHIN GROUP clause

Cryptographic functions

- CERTENCODED function

- CERTID function
- CERTPROPERTY function

Metadata functions

- COLUMNPROPERTY
- TYPEPROPERTY
- SERVERPROPERTY function – The following properties aren't supported:
  - BuildClrVersion
  - ComparisonStyle
  - ComputerNamePhysicalNetBIOS
  - HadrManagerStatus
  - InstanceDefaultDataPath
  - InstanceDefaultLogPath
  - IsClustered
  - IsHadrEnabled
  - LCID
  - NumLicenses
  - ProcessID
  - ProductBuild
  - ProductBuildType
  - ProductUpdateReference
  - ResourceLastUpdateDateTime
  - ResourceVersion
  - ServerName
  - SqlCharSet
  - SqlCharSetName
  - SqlSortOrder
  - SqlSortOrderName
  - FilestreamShareName
  - FilestreamConfiguredLevel

- FilestreamEffectiveLevel

Security functions

- CERTPRIVATEKEY
- LOGINPROPERTY

Statements, operators, other functions

- EVENTDATA function
- GET_TRANSMISSION_STATUS
- OPENXML

Syntax that isn't supported

The following syntax isn't supported:

- ALTER DATABASE
- ALTER DATABASE SCOPED CONFIGURATION
- ALTER DATABASE SCOPED CREDENTIAL
- ALTER DATABASE SET HADR
- ALTER FUNCTION
- ALTER INDEX
- ALTER PROCEDURE statement
- ALTER SCHEMA
- ALTER SERVER CONFIGURATION
- ALTER SERVICE, BACKUP/RESTORE SERVICE MASTER KEY clause
- ALTER VIEW
- BEGIN CONVERSATION TIMER
- BEGIN DISTRIBUTED TRANSACTION
- BEGIN DIALOG CONVERSATION
- BULK INSERT
- CREATE COLUMNSTORE INDEX

- CREATE EXTERNAL FILE FORMAT
- CREATE EXTERNAL TABLE
- CREATE, ALTER, DROP APPLICATION ROLE
- CREATE, ALTER, DROP ASSEMBLY
- CREATE, ALTER, DROP ASYMMETRIC KEY
- CREATE, ALTER, DROP CREDENTIAL
- CREATE, ALTER, DROP CRYPTOGRAPHIC PROVIDER
- CREATE, ALTER, DROP ENDPOINT
- CREATE, ALTER, DROP EVENT SESSION
- CREATE, ALTER, DROP EXTERNAL LANGUAGE
- CREATE, ALTER, DROP EXTERNAL RESOURCE POOL
- CREATE, ALTER, DROP FULLTEXT CATALOG
- CREATE, ALTER, DROP FULLTEXT INDEX
- CREATE, ALTER, DROP FULLTEXT STOPLIST
- CREATE, ALTER, DROP MESSAGE TYPE
- CREATE, ALTER, DROP, OPEN/CLOSE, BACKUP/RESTORE MASTER KEY
- CREATE, ALTER, DROP PARTITION FUNCTION
- CREATE, ALTER, DROP PARTITION SCHEME
- CREATE, ALTER, DROP QUEUE
- CREATE, ALTER, DROP RESOURCE GOVERNOR
- CREATE, ALTER, DROP RESOURCE POOL
- CREATE, ALTER, DROP ROUTE
- CREATE, ALTER, DROP SEARCH PROPERTY LIST
- CREATE, ALTER, DROP SECURITY POLICY
- CREATE, ALTER, DROP SELECTIVE XML INDEX clause
- CREATE, ALTER, DROP SERVICE
- CREATE, ALTER, DROP SPATIAL INDEX
- CREATE, ALTER, DROP TYPE
- CREATE, ALTER, DROP XML INDEX
- CREATE, ALTER, DROP XML SCHEMA COLLECTION

- CREATE/DROP RULE
- CREATE, DROP WORKLOAD CLASSIFIER
- CREATE, ALTER, DROP WORKLOAD GROUP
- ALTER TRIGGER
- CREATE TABLE... GRANT clause
- CREATE TABLE... IDENTITY clause
- CREATE USER – This syntax isn't supported. The PostgreSQL statement CREATE USER doesn't create a user that is equivalent to the SQL Server CREATE USER syntax. For more information, see T-SQL differences in Babelfish.
- DENY
- END, MOVE CONVERSATION
- EXECUTE with AS LOGIN or AT option
- GET CONVERSATION GROUP
- GROUP BY ALL clause
- GROUP BY CUBE clause
- GROUP BY ROLLUP clause
- INSERT... DEFAULT VALUES
- MERGE
- READTEXT
- REVERT
- SELECT PIVOT(supported from 3.4.0 and higher releases except when used in a view definition, a common table expression, or a join)/UNPIVOT
- SELECT TOP x PERCENT WHERE x <> 100
- SELECT TOP... WITH TIES
- SELECT... FOR BROWSE
- SELECT... FOR XML AUTO
- SELECT... FOR XML EXPLICIT
- SEND
- SET DATEFORMAT
- SET DEADLOCK_PRIORITY
- SET FMTONLY

- SET FORCEPLAN
- SET NUMERIC_ROUNDABORT ON
- SET OFFSETS
- SET REMOTE_PROC_TRANSACTIONS
- SET SHOWPLAN_TEXT
- SET SHOWPLAN_XML
- SET STATISTICS
- SET STATISTICS PROFILE
- SET STATISTICS TIME
- SET STATISTICS XML
- SHUTDOWN statement
- UPDATE STATISTICS
- UPDATETEXT
- Using EXECUTE to call a SQL function
- VIEW... CHECK OPTION clause
- VIEW... VIEW_METADATA clause
- WAITFOR DELAY
- WAITFOR TIME
- WAITFOR, RECEIVE
- WITH XMLNAMESPACES construct
- WRITETEXT
- XPATH expressions

Supported functionality in Babelfish by version

In the following table you can find T-SQL functionality supported by different Babelfish versions. For lists of unsupported functionality, see Unsupported functionality in Babelfish. For information about various Babelfish releases, see the *Release Notes for Aurora PostgreSQL*.

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
4 parts object name references for SELECT statements	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
AS keyword in CREATE FUNCTION	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
ALTER AUTHORIZATION syntax to change	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
database owner															
ALTER ROLE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ALTER USER...WITH LOGIN	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
AT TIME ZONE clause	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
Babelfish instance as a linked server	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
Comparison operators !< and !>	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
CREATE Instead of Triggers (DML) on SQL Server Views	–	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
CREATE ROLE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CREATE TRIGGER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Create unique indexes	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cross-dat abase procedure execution	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
Cross-database references SELECT,SELECT..INTO, INSERT, UPDATE, DELETE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cursor-typed parameters for input parameters only (not output)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
Data migration using the bcp client utility	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Datatypes TIMESTAMP , ROWVERSION (for usage information, see Features with limited implementation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
DEFAULT keyword in calls to stored procedures and functions	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
DBCC CHECKIDENT	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
DROP DATABASE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
DROP IF EXISTS (for SCHEMA, DATABASE, and USER objects)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
DROP INDEX index ON schema.table	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
DROP INDEX schema.table.index	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
DROP ROLE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ENABLE/DISABLE TRIGGER	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
FULL TEXT SEARCH	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–
Full Text Search with CONTAINS clause	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
GRANT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Geometry and Geography spatial datatypes	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
GUC babelfish pg_tds.product_version	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
Identifiers with leading dot character	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
INSTEAD OF triggers on tables	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
INSTEAD OF triggers on views	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–	–
KILL	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
PIVOT (supported from 3.4.0 and higher releases except when used in a view definition, a common table expression, or a join)	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
REVOKE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SELECT... OFFSET... FETCH clauses	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SELECT FOR JSON AUTO	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
SET BABELFISH_SHOWPLAN_ALL ON (and OFF)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SET BABELFISH_STATISTICS PROFILE ON (OFF)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SET CONTEXT_INFO	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
SET LOCK_TIMEOUT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SET NO_BROWSETABLE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
SET rowcount	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
SET SHOWPLAN_ALL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
SET STATISTICS IO	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
SET TRANSACTION ISOLATION LEVEL syntax	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
SSMS: Connecting with the object explorer connection dialog	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SSMS: Data migration with the Import/Export Wizard	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SSMS: Partial support for the object explorer	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STDEV	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
STDEVP	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
Triggers with multiple DML actions can reference transition tables	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL hints (join methods, index usage, MAXDOP)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
T-SQL square bracket syntax with the LIKE predicate	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
Unquoted string values in stored procedure calls and default values	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
VAR	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
VARP	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
Aurora and PostgreSQL features:															
Aurora ML services	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
Database authentication with Kerberos using AWS Directory Service	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
Dump and restore	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
pg_stat_statement extension	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
pgvector	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
Zero-downtime patching (ZDP)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
T-SQL Built-in functions:															
APP_NAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
ATN2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
CHARINDEX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CHOOSE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
COL_LENGTH	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–
COL_NAME	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–	–
COLUMNS_UPDATED	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
COLUMNPROPERTY (CharMaxLen, AllowsNull only)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CONCAT_WS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CONTEXT_INFO	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
CURSOR_STATUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DATABASE_PRINCIPAL_ID	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
DATEADD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
DATEDIFF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
DATEDIFF_BIG	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
DATEFROMPARTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DATENAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
DATEPART	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
DATETIMEFROMPARTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DATETIME2FROMPARTS	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
DATETIMEOFFSETFROMPARTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
DATETRUNC	✓	✓	✓	✓	–	–	–	✓	–	–	–	–	–	–	–
DATE_BUCKET	✓	✓	✓	✓	–	–	–	✓	–	–	–	–	–	–	–
EOMONTH	✓	✓	✓	✓	–	–	–	✓	–	–	–	–	–	–	–
EXECUTE AS CALLER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
fn_listextendedproperty	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
FOR JSON	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
FULLTEXTSERVICEPROPERTY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HAS_DBACCESS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HAS_PERMS_BY_NAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
HOST_NAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
HOST_ID	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
IDENTITY	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
IS_MEMBER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IS_ROLEMEMBER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IS_SRVROLEMEMBER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ISJSON	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
JSON_MODIFY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
JSON_QUERY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
JSON_VALUE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NEXT VALUE FOR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
OBJECT_DEFINITION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
OBJECT_SCHEMA_NAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
OPENJSON	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OPENQUERY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ORIGINAL_LOGIN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
PARSENAME	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
PATINDEX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ROWCOUNT_BIG	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
SCHEMA_NAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SESSION_CONTEXT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SESSION_USER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SID_BINARY (returns NULL always)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
SMALLDATETIMEFROMPARTS	✓	✓	✓	✓	✓	–	–	✓	✓	✓	–	–	–	–	–
SQUARE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
STR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
STRING_AGG	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
STRING_SPLIT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SUSER_SID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SUSER_SNAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SWITCHOFFSET	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
SYSTEM_USER	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
TIMEFROMPARTS	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
TODATETIMEOFFSET	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
TO_CHAR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
TRIGGER_NESTLEVEL (without arguments only)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TRY_CONVERT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
TYPE_ID	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
TYPE_NAME	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
UPDATE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL INFORMATION_SCHEMA catalogs															
CHECK_CONSTRAINTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
COLUMN_DOMAIN_USAGE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
COLUMNS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CONSTRAINT_COLUMN_USAGE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DOMAINS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
KEY_COLUMN_USAGE	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
ROUTINES	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
TABLES	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
TABLE_CONSTRAINTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
VIEWS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL System-defined @@ variables:															
@@CURSOR_ROWS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@DATEFIRST	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@DBTS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
@@ERROR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@ERROR=213	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@FETCH_STATUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@IDENTITY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@LANGUAGE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@LOCK_TIMEOUT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@MAX_CONNECTIONS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@MAX_PRECISION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@MICROSOFTVERSION	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@NESTLEVEL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@PROCID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@ROWCOUNT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
@@SERVERNAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@SERVICENAME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@SRID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@TRANCOUNT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
@@VERSION (note format difference as described in T-SQL differences in Babelfish.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL System stored procedures:

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_addextendedproperty	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_addlinkedserver	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_addlinkedsrvlogin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_addrole	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_addrolemember	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_babelfish_volatility	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_column_privileges	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_columns	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_columns_100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_columns_managed	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursor_list	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorclose	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorexecute	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorfetch	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursoropen	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursoroption	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorprepare	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorprepexec	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_cursorunprepare	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_databases	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_datatype_info	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_datatype_info_100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_describe_cursor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_describe_first_result_set	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_describe_undeclared_parameters	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_dropextendedproperty	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_droplinkedsrvlogin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_droprole	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_droprolemember	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_dropserver	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_enum_oledb_providers	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_execute	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_execute_postgresql(CREATE, ALTER, DROP)	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_executesql	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_fkeys	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_getapplock	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_helpdb	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_helpdbfixedrole	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_helplinkedsrvlogin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sp_helprole	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_helprolemember	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_helpsrvrolemember	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
sp_helpuser	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_linkedservers	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_oledb_ro_usrname	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_pkeys	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_prefix	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_prepare	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_procedure_params_100_managed	✓	–	✓	–	–	–	–	–	–	–	–	–	–	–	–
sp_releaseapplock	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_rename	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_serveroption(connect_timeout option)	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_set_session_context	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sp_special_columns	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_sproc_columns	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_sproc_columns_100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sp_statistics	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_statistics_100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_stored_procedures	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_table_privileges	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_tablecollations_100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_tables	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_testlinkedserver	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_unprepare	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sp_updateextendedproperty	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sp_who	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
xp_qv	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL Properties supported on the CONNECTIONPROPERTY system function															
auth_scheme	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
client_net_address	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
local_net_address	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
local_tcp_port	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
net_transport	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
protocol_type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
physical_net_transport	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL Properties supported on the OBJECTPROPERTY system function															
IsInlineFunction	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsScalarFunction	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
IsTableFunction	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
T-SQL Properties supported on the SERVERPROPERTY function															
Babelfish	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Collation	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CollationID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Edition	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EditionID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EngineEdition	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
InstanceName	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–
IsAdvancedAnalyticsInstalled	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsBigDataCluster	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsFullTextInstalled	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
IsIntegratedSecurityOnly	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsLocalDB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsPolyBaseInstalled	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsSingleUser	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IsXTPSupported	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Japanese_CI_AI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Japanese_CI_AS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Japanese_CS_AS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LicenseType	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
MachineName	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
ProductLevel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
ProductMajorVersion	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ProductMinorVersion	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ProductUpdateLevel	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
ProductVersion	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ServerName	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SQL Server views supported by Babelfish															
information_schema.key_column_usage	✓	✓	✓	✓	–	–	–	✓	–	–	–	–	–	–	–
information_schema.routines	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
information_schema.schemata	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
information_schema.sequences	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sys.all_columns	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.all_objects	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.all_parameters	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–
sys.all_sql_modules	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.all_views	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.columns	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.configurations	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sys.data_spaces	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.database_files	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.database_mirroring	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.database_principals	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.database_role_members	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.databases	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.dm_exec_connections	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.dm_exec_sessions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sys.dm_hadr_database_replica_states	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.dm_os_host_info	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.endpoints	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.extended_properties	✓	✓	✓	✓	✓	–	–	✓	✓	–	–	–	–	–	–
sys.indexes	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.schemas	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.server_principals	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.server_role_members	✓	✓	✓	✓	–	–	–	–	–	–	–	–	–	–	–
sys.sql_modules	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

T-SQL Functionality or syntax	4.1.0	4.0.0	3.5.0	3.4.0	3.3.0	3.2.0	3.1.0	2.8.0	2.7.0	2.6.0	2.5.0	2.4.0	2.3.0	2.2.0	2.1.0
sys.sysconfigures	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.syscurconfigs	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.syslogins	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
sys.sysprocesses	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.sysusers	✓	✓	✓	✓	✓	✓	–	✓	✓	–	–	–	–	–	–
sys.table_types	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.tables	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sys.types	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	–
sys.xml_schema_collections	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
syslanguages	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
sysobjects.crdate	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–

Babelfish for Aurora PostgreSQL procedure reference

Overview

You can use the following procedure for Amazon RDS DB instances running Babelfish for Aurora PostgreSQL for a better query performance:

- sp_babelfish_volatility
- sp_execute_postgresql

sp_babelfish_volatility

PostgreSQL function volatility helps the optimizer for a better query execution which when used in parts of certain clauses has a significant impact on query performance.

Syntax

```
sp_babelfish_volatility 'function_name', 'volatility'
```

Arguments

function_name (optional)

You can either specify the value of this argument with a two-part name as `schema_name.function_name`or only the `function_name`. If you specify only the `function_name`, the schema name is the default schema for the current user.

volatility (optional)

The valid PostgreSQL values of volatility are `stable`, `volatile`, or `immutable`. For more information, see https://www.postgresql.org/docs/current/xfunc-volatility.html

> **Note**
>
> When `sp_babelfish_volatility` is called with `function_name` which has multiple definitions, it will throw an error.

Result set

If the parameters are not mentioned then the result set is displayed under the following columns: `schemaname`, `functionname`, `volatility`.

Usage notes

PostgreSQL function volatility helps the optimizer for a better query execution which when used in parts of certain clauses has a significant impact on query performance.

Examples

The following examples shows how to create simple functions and later explains how to use `sp_babelfish_volatility` on these functions using different methods.

```
1> create function f1() returns int as begin return 0 end
2> go
```

```
1> create schema test_schema
2> go
```

```
1> create function test_schema.f1() returns int as begin return 0 end
2> go
```

The following example displays volatility of the functions:

```
1> exec sp_babelfish_volatility
2> go

schemaname  functionname volatility
----------- ------------ ----------
dbo         f1           volatile
test_schema f1           volatile
```

The following example shows how to change the volatility of the functions:

```
1> exec sp_babelfish_volatility 'f1','stable'
2> go
1> exec sp_babelfish_volatility 'test_schema.f1','immutable'
2> go
```

When you specify only the function_name, it displays the schema name, function name and volatility of that function. The following example displays volatility of functions after changing the values:

```
1> exec sp_babelfish_volatility 'test_schema.f1'
2> go

schemaname  functionname volatility
----------- ------------ ----------
test_schema f1           immutable
```

```
1> exec sp_babelfish_volatility 'f1'
2> go

schemaname  functionname volatility
----------- ------------ ----------
dbo         f1           stable
```

When you don't specify any argument, it displays a list of functions (schema name, function name, volatility of the functions) present in the current database:

```
1> exec sp_babelfish_volatility
2> go

schemaname  functionname volatility
----------- ------------ ----------
dbo         f1           stable
test_schema f1           immutable
```

sp_execute_postgresql

You can execute PostgreSQL statements from T-SQL endpoint. This simplifies your applications as you don’t need to exit T-SQL port to execute these statements.

Syntax

```
sp_execute_postgresql [ @stmt = ] statement
```

Arguments

[@stmt] statement

The argument is of datatype varchar. This argument accept PG dialect statements.

> **Note**
>
> You can only pass one PG dialect statement as an argument otherwise it will raise the following error.

```
1>exec sp_execute_postgresql 'create extension pg_stat_statements; drop extension
 pg_stat_statements'
2>go
```

```

Msg 33557097, Level 16, State 1, Server BABELFISH, Line 1
expected 1 statement but got 2 statements after parsing
```

Usage notes

CREATE EXTENSION

Creates and loads a new extension into the current database.

```
1>EXEC sp_execute_postgresql 'create extension [ IF NOT EXISTS ] <extension name>
 [ WITH ] [SCHEMA schema_name] [VERSION version]';
2>go
```

The following example shows how to create an extension:

```
1>EXEC sp_execute_postgresql 'create extension pg_stat_statements with schema sys
 version "1.10"';
2>go
```

Use the following command to access extension objects:

```
1>select * from pg_stat_statements;
2>go
```

> **Note**
>
> If schema name is not provided explicitly during extension creation, by default the extensions are installed in the public schema. You must provide the schema qualifier to access the extension objects as mentioned below:

```
1>select * from [public].pg_stat_statements;
2>go
```

Supported extensions

The following extensions available with Aurora PostgreSQL works with Babelfish.

- `pg_stat_statements`
- `tds_fdw`
- `fuzzystrmatch`

Limitations

- Users need to have sysadmin role on T-SQL and rds_superuser on postgres to install the extenstions.
- Extensions cannot be installed in user created schemas and also in dbo and guest schemas for master, tempdb and msdb database.
- CASCADE option is not supported.

ALTER EXTENSION

You can upgrade to a new extension version using ALTER extension.

```
1>EXEC sp_execute_postgresql 'alter extension <extension name> UPDATE TO
 <new_version>';
2>go
```

Limitations

- You can upgrade the version of your extension only using the ALTER Extension statement. Other operations aren't supported.

DROP EXTENSION

Drops the specified extension. You can also use `if exists` or `restrict` options to drop the extension.

```
1>EXEC sp_execute_postgresql 'drop extension <extension name>';
2>go
```

Limitations

- CASCADE option is not supported.

Managing Amazon Aurora PostgreSQL

The following section discusses managing performance and scaling for an Amazon Aurora PostgreSQL DB cluster. It also includes information about other maintenance tasks.

Topics

- Scaling Aurora PostgreSQL DB instances
- Maximum connections to an Aurora PostgreSQL DB instance
- Temporary storage limits for Aurora PostgreSQL
- Huge pages for Aurora PostgreSQL
- Testing Amazon Aurora PostgreSQL by using fault injection queries
- Displaying volume status for an Aurora PostgreSQL DB cluster
- Specifying the RAM disk for the stats_temp_directory
- Managing temporary files with PostgreSQL

Scaling Aurora PostgreSQL DB instances

You can scale Aurora PostgreSQL DB instances in two ways, instance scaling and read scaling. For more information about read scaling, see Read scaling.

You can scale your Aurora PostgreSQL DB cluster by modifying the DB instance class for each DB instance in the DB cluster. Aurora PostgreSQL supports several DB instance classes optimized for Aurora. Don't use db.t2 or db.t3 instance classes for larger Aurora clusters of size greater than 40 terabytes (TB).

> **Note**
>
> We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more details on the T instance classes, see DB instance class types.

Scaling isn't instantaneous. It can take 15 minutes or more to complete the change to a different DB instance class. If you use this approach to modify the DB instance class, you apply the change during the next scheduled maintenance window (rather than immediately) to avoid affecting users.

As an alternative to modifying the DB instance class directly, you can minimize downtime by using the high availability features of Amazon Aurora. First, add an Aurora Replica to your cluster. When creating the replica, choose the DB instance class size that you want to use for your cluster. When the Aurora Replica is synchronized with the cluster, you then failover to the newly added Replica. To learn more, see Aurora Replicas and Fast failover with Amazon Aurora PostgreSQL.

For detailed specifications of the DB instance classes supported by Aurora PostgreSQL, see Supported DB engines for DB instance classes.

Maximum connections to an Aurora PostgreSQL DB instance

An Aurora PostgreSQL DB cluster allocates resources based on the DB instance class and its available memory. Each connection to the DB cluster consumes incremental amounts of these resources, such as memory and CPU. Memory consumed per connection varies based on query type, count, and whether temporary tables are used. Even an idle connection consumes memory and CPU. That's because when queries run on a connection, more memory is allocated for each query and it's not released completely, even when processing stops. Thus, we recommend that you make sure your applications aren't holding on to idle connections: each one of these wastes resources and affects performance negatively. For more information, see Resources consumed by idle PostgreSQL connections.

The maximum number of connections allowed by an Aurora PostgreSQL DB instance is determined by the `max_connections` parameter value specified in the parameter group for that DB instance. The ideal setting for the `max_connections` parameter is one that supports all the client connections your application needs, without an excess of unused connections, plus at least 3 more connections to support AWS automation. Before modifying the `max_connections` parameter setting, we recommend that you consider the following:

- If the `max_connections` value is too low, the Aurora PostgreSQL DB instance might not have sufficient connections available when clients attempt to connect. If this happens, attempts to connect using `psql` raise error messages such as the following:

  ```
  psql: FATAL: remaining connection slots are reserved for non-replication superuser
   connections
  ```
- If the `max_connections` value exceeds the number of connections that are actually needed, the unused connections can cause performance to degrade.

The default value of `max_connections` is derived from the following Aurora PostgreSQL `LEAST` function:

`LEAST({DBInstanceClassMemory/9531392},5000)`.

If you want to change the value for `max_connections`, you need to create a custom DB cluster parameter group and change its value there. After applying your custom DB parameter group to your cluster, be sure to reboot the primary instance so the new value takes effect. For more information, see Amazon Aurora PostgreSQL parameters and Creating a DB cluster parameter group.

> **Tip**
>
> If your applications frequently open and close connections, or keep a large number of long-lived connections open, we recommend that you use Amazon RDS Proxy. RDS Proxy is a fully managed, highly available database proxy that uses connection pooling to share database connections securely and efficiently. To learn more about RDS Proxy, see Using Amazon RDS Proxy for Aurora.

For details about how Aurora Serverless v2 instances handle this parameter, see Maximum connections for Aurora Serverless v2.

Temporary storage limits for Aurora PostgreSQL

Aurora PostgreSQL stores tables and indexes in the Aurora storage subsystem. Aurora PostgreSQL uses separate temporary storage for non-persistent temporary files. This includes files that are used for such purposes as sorting large data sets during query processing or for index build operations. For more information, see the article How can I troubleshoot local storage issues in Aurora PostgreSQL-Compatible instances?.

These local storage volumes are backed by Amazon Elastic Block Store and can be extended by using a larger DB instance class. For more information about storage, see Amazon Aurora storage and reliability. You can also increase your local storage for temporary objects by using an NVMe enabled instance type and Aurora Optimized Reads-enabled temporary objects. For more information, see Improving query performance for Aurora PostgreSQL with Aurora Optimized Reads.

Note

You might see `storage-optimization` events when scaling DB instances, for example, from db.r5.2xlarge to db.r5.4xlarge.

The following table shows the maximum amount of temporary storage available for each Aurora PostgreSQL DB instance class. For more information on DB instance class support for Aurora, see Aurora DB instance classes.

DB instance class	Maximum temporary storage available (GiB)
db.x2g.16xlarge	1829
db.x2g.12xlarge	1606
db.x2g.8xlarge	1071
db.x2g.4xlarge	535
db.x2g.2xlarge	268

DB instance class	Maximum temporary storage available (GiB)
db.x2g.xlarge	134
db.x2g.large	67
db.r7g.16xlarge	1008
db.r7g.12xlarge	756
db.r7g.8xlarge	504
db.r7g.4xlarge	252
db.r7g.2xlarge	126
db.r7g.xlarge	63
db.r7g.large	32
db.r6g.16xlarge	1008
db.r6g.12xlarge	756
db.r6g.8xlarge	504
db.r6g.4xlarge	252
db.r6g.2xlarge	126
db.r6g.xlarge	63
db.r6g.large	32
db.r6i.32xlarge	1829
db.r6i.24xlarge	1500
db.r6i.16xlarge	1008
db.r6i.12xlarge	748

DB instance class	Maximum temporary storage available (GiB)
db.r6i.8xlarge	504
db.r6i.4xlarge	249
db.r6i.2xlarge	124
db.r6i.xlarge	62
db.r6i.large	31
db.r5.24xlarge	1500
db.r5.16xlarge	1008
db.r5.12xlarge	748
db.r5.8xlarge	504
db.r5.4xlarge	249
db.r5.2xlarge	124
db.r5.xlarge	62
db.r5.large	31
db.r4.16xlarge	960
db.r4.8xlarge	480
db.r4.4xlarge	240
db.r4.2xlarge	120
db.r4.xlarge	60
db.r4.large	30
db.t4g.large	16.5

DB instance class	Maximum temporary storage available (GiB)
db.t4g.medium	8.13
db.t3.large	16
db.t3.medium	7.5

Note

NVMe enabled instance types can increase the temporary space available by up to the total NVMe size. For more information, see Improving query performance for Aurora PostgreSQL with Aurora Optimized Reads.

You can monitor the temporary storage available for a DB instance with the `FreeLocalStorage` CloudWatch metric, --> described in Amazon CloudWatch metrics for Amazon Aurora. (This doesn't apply to Aurora Serverless v2.)

For some workloads, you can reduce the amount of temporary storage by allocating more memory to the processes that are performing the operation. To increase the memory available to an operation, increasing the values of the work_mem or maintenance_work_mem PostgreSQL parameters.

Huge pages for Aurora PostgreSQL

Huge pages are a memory management feature that reduces overhead when a DB instance is working with large contiguous chunks of memory, such as that used by shared buffers. This PostgreSQL feature is supported by all currently available Aurora PostgreSQL versions.

`Huge_pages` parameter is turned on by default for all DB instance classes other than t3.medium,db.t3.large,db.t4g.medium,db.t4g.large instance classes. You can't change the `huge_pages` parameter value or turn off this feature in the supported instance classes of Aurora PostgreSQL.

Testing Amazon Aurora PostgreSQL by using fault injection queries

You can test the fault tolerance of your Aurora PostgreSQL DB cluster by using fault injection queries. Fault injection queries are issued as SQL commands to an Amazon Aurora instance. Fault injection queries let you crash the instance so that you can test failover and recovery. You can also simulate Aurora Replica failure, disk failure, and disk congestion. Fault injection queries are supported by all available Aurora PostgreSQL versions, as follows.

- Aurora PostgreSQL versions 12, 13, 14, and higher
- Aurora PostgreSQL version 11.7 and higher
- Aurora PostgreSQL version 10.11 and higher

Topics

When a fault injection query specifies a crash, it forces a crash of the Aurora PostgreSQL DB instance. The other fault injection queries result in simulations of failure events, but don't cause the event to occur. When you submit a fault injection query, you also specify an amount of time for the failure event simulation to occur.

You can submit a fault injection query to one of your Aurora Replica instances by connecting to the endpoint for the Aurora Replica. For more information, see Amazon Aurora connection management.

Testing an instance crash

You can force a crash of an Aurora PostgreSQL instance by using the fault injection query function `aurora_inject_crash()`.

For this fault injection query, a failover does not occur. If you want to test a failover, then you can choose the **Failover** instance action for your DB cluster in the RDS console, or use the failover-db-cluster AWS CLI command or the FailoverDBCluster RDS API operation.

Syntax

```
SELECT aurora_inject_crash ('instance' | 'dispatcher' | 'node');
```

Options

This fault injection query takes one of the following crash types. The crash type is not case sensitive:

'instance'

A crash of the PostgreSQL-compatible database for the Amazon Aurora instance is simulated.

'dispatcher'

A crash of the dispatcher on the primary instance for the Aurora DB cluster is simulated. The *dispatcher* writes updates to the cluster volume for an Amazon Aurora DB cluster.

'node'

A crash of both the PostgreSQL-compatible database and the dispatcher for the Amazon Aurora instance is simulated.

Testing an Aurora Replica failure

You can simulate the failure of an Aurora Replica by using the fault injection query function `aurora_inject_replica_failure()`.

An Aurora Replica failure blocks replication to the Aurora Replica or all Aurora Replicas in the DB cluster by the specified percentage for the specified time interval. When the time interval completes, the affected Aurora Replicas are automatically synchronized with the primary instance.

Syntax

```
SELECT aurora_inject_replica_failure(
   percentage_of_failure,
   time_interval,
   'replica_name'
);
```

Options

This fault injection query takes the following parameters:

percentage_of_failure

The percentage of replication to block during the failure event. This value can be a double between 0 and 100. If you specify 0, then no replication is blocked. If you specify 100, then all replication is blocked.

time_interval

The amount of time to simulate the Aurora Replica failure. The interval is in seconds. For example, if the value is 20, the simulation runs for 20 seconds.

> **Note**
>
> Take care when specifying the time interval for your Aurora Replica failure event. If you specify too long an interval, and your writer instance writes a large amount of data during the failure event, then your Aurora DB cluster might assume that your Aurora Replica has crashed and replace it.

replica_name

The Aurora Replica in which to inject the failure simulation. Specify the name of an Aurora Replica to simulate a failure of the single Aurora Replica. Specify an empty string to simulate failures for all Aurora Replicas in the DB cluster.

To identify replica names, see the `server_id` column from the `aurora_replica_status()` function. For example:

```
postgres=> SELECT server_id FROM aurora_replica_status();
```

Testing a disk failure

You can simulate a disk failure for an Aurora PostgreSQL DB cluster by using the fault injection query function `aurora_inject_disk_failure()`.

During a disk failure simulation, the Aurora PostgreSQL DB cluster randomly marks disk segments as faulting. Requests to those segments are blocked for the duration of the simulation.

Syntax

```
SELECT aurora_inject_disk_failure(
    percentage_of_failure,
    index,
    is_disk,
    time_interval
);
```

Options

This fault injection query takes the following parameters:

percentage_of_failure

The percentage of the disk to mark as faulting during the failure event. This value can be a double between 0 and 100. If you specify 0, then none of the disk is marked as faulting. If you specify 100, then the entire disk is marked as faulting.

index

A specific logical block of data in which to simulate the failure event. If you exceed the range of available logical blocks or storage nodes data, you receive an error that tells you the maximum index value that you can specify. To avoid this error, see Displaying volume status for an Aurora PostgreSQL DB cluster.

is_disk

Indicates whether the injection failure is to a logical block or a storage node. Specifying true means injection failures are to a logical block. Specifying false means injection failures are to a storage node.

time_interval

The amount of time to simulate the disk failure. The interval is in seconds. For example, if the value is 20, the simulation runs for 20 seconds.

Testing disk congestion

You can simulate a disk congestion for an Aurora PostgreSQL DB cluster by using the fault injection query function `aurora_inject_disk_congestion()`.

During a disk congestion simulation, the Aurora PostgreSQL DB cluster randomly marks disk segments as congested. Requests to those segments are delayed between the specified minimum and maximum delay time for the duration of the simulation.

Syntax

```
SELECT aurora_inject_disk_congestion(
   percentage_of_failure,
   index,
   is_disk,
   time_interval,
   minimum,
   maximum
);
```

Options

This fault injection query takes the following parameters:

percentage_of_failure

The percentage of the disk to mark as congested during the failure event. This is a double value between 0 and 100. If you specify 0, then none of the disk is marked as congested. If you specify 100, then the entire disk is marked as congested.

index

A specific logical block of data or storage node to use to simulate the failure event.

If you exceed the range of available logical blocks or storage nodes of data, you receive an error that tells you the maximum index value that you can specify. To avoid this error, see [Displaying volume status for an Aurora PostgreSQL DB cluster](#).

is_disk

Indicates whether the injection failure is to a logical block or a storage node. Specifying true means injection failures are to a logical block. Specifying false means injection failures are to a storage node.

time_interval

The amount of time to simulate the disk congestion. The interval is in seconds. For example, if the value is 20, the simulation runs for 20 seconds.

minimum, maximum

The minimum and maximum amount of congestion delay, in milliseconds. Valid values range from 0.0 to 100.0 milliseconds. Disk segments marked as congested are delayed for a random amount of time within the minimum and maximum range for the duration of the simulation. The maximum value must be greater than the minimum value.

Displaying volume status for an Aurora PostgreSQL DB cluster

In Amazon Aurora, a DB cluster volume consists of a collection of logical blocks. Each of these represents 10 gigabytes of allocated storage. These blocks are called *protection groups*.

The data in each protection group is replicated across six physical storage devices, called *storage nodes*. These storage nodes are allocated across three Availability Zones (AZs) in the region where the DB cluster resides. In turn, each storage node contains one or more logical blocks of data for the DB cluster volume. For more information about protection groups and storage nodes, see Introducing the Aurora storage engine on the AWS Database Blog. To learn more about Aurora cluster volumes in general, see Amazon Aurora storage and reliability.

Use the `aurora_show_volume_status()` function to return the following server status variables:

- `Disks` — The total number of logical blocks of data for the DB cluster volume.
- `Nodes` — The total number of storage nodes for the DB cluster volume.

You can use the `aurora_show_volume_status()` function to help avoid an error when using the `aurora_inject_disk_failure()` fault injection function. The `aurora_inject_disk_failure()` fault injection function simulates the failure of an entire storage node, or a single logical block of data within a storage node. In the function, you specify the index value of a specific logical block of data or storage node. However, the statement returns an error if you specify an index value greater than the number of logical blocks of data or storage nodes used by the DB cluster volume. For more information about fault injection queries, see Testing Amazon Aurora PostgreSQL by using fault injection queries.

> **Note**
>
> The `aurora_show_volume_status()` function is available for Aurora PostgreSQL version 10.11. For more information about Aurora PostgreSQL versions, see Amazon Aurora PostgreSQL releases and engine versions.

Syntax

```
SELECT * FROM aurora_show_volume_status();
```

Example

```
customer_database=> SELECT * FROM aurora_show_volume_status();
 disks | nodes
-------+-------
    96 |    45
```

Specifying the RAM disk for the stats_temp_directory

You can use the Aurora PostgreSQL parameter, `rds.pg_stat_ramdisk_size`, to specify the system memory allocated to a RAM disk for storing the PostgreSQL `stats_temp_directory`. The RAM disk parameter is only available in Aurora PostgreSQL 14 and lower versions.

Under certain workloads, setting this parameter can improve performance and decrease IO requirements. For more information about the `stats_temp_directory`, see Run-time Statistics in the PostgreSQL documentation. From PostgreSQL version 15, the PostgreSQL community switched to use dynamic shared memory. So, there is no need to set `stats_temp_directory`.

To enable a RAM disk for your `stats_temp_directory`, set the `rds.pg_stat_ramdisk_size` parameter to a non-zero value in the DB cluster parameter group used by your DB cluster. This parameter denotes MB, so you must use an integer value. Expressions, formulas, and functions aren't valid for the `rds.pg_stat_ramdisk_size` parameter. Be sure to restart the DB cluster so that the change takes effect. For information about setting parameters, see Working with parameter groups. For more information about restarting the DB cluster, see Rebooting an Amazon Aurora DB cluster or Amazon Aurora DB instance.

As an example, the following AWS CLI command sets the RAM disk parameter to 256 MB.

```
aws rds modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name db-cl-pg-ramdisk-testing \
    --parameters "ParameterName=rds.pg_stat_ramdisk_size, ParameterValue=256,
 ApplyMethod=pending-reboot"
```

After you restart the DB cluster, run the following command to see the status of the `stats_temp_directory`:

```
postgres=> SHOW stats_temp_directory;
```

The command should return the following:

```
stats_temp_directory
---------------------------
/rdsdbramdisk/pg_stat_tmp
(1 row)
```

Managing temporary files with PostgreSQL

In PostgreSQL, a query performing sort and hash operations uses the instance memory to store results up to the value specified in the `work_mem` parameter. When the instance memory is not sufficient, temporary files are created to store the results. These are written to disk to complete the query execution. Later, these files are automatically removed after the query completes. In Aurora PostgreSQL, these files share local storage with other log files. You can monitor your Aurora PostgreSQL DB cluster's local storage space by watching the Amazon CloudWatch metric for `FreeLocalStorage`. For more information, see Troubleshoot local storage issues.

You can use the following parameters and functions to manage the temporary files in your instance.

- **`temp_file_limit`** – This parameter cancels any query exceeding the size of temp_files in KB. This limit prevents any query from running endlessly and consuming disk space with temporary files. You can estimate the value using the results from the `log_temp_files` parameter. As a best practice, examine the workload behavior and set the limit according to the estimation. The following example shows how a query is canceled when it exceeds the limit.

  ```
  postgres=> select * from pgbench_accounts, pg_class, big_table;
  ```

  ```


  ERROR: temporary file size exceeds temp_file_limit (64kB)
  ```

- **`log_temp_files`** – This parameter sends messages to the postgresql.log when the temporary files of a session are removed. This parameter produces logs after a query successfully completes. Therefore, it might not help in troubleshooting active, long-running queries.

  The following example shows that when the query successfully completes, the entries are logged in the postgresql.log file while the temporary files are cleaned up.

```
2023-02-06 23:48:35 UTC:205.251.233.182(12456):adminuser@postgres:[31236]:LOG:
 temporary file: path "base/pgsql_tmp/pgsql_tmp31236.5", size 140353536
2023-02-06 23:48:35 UTC:205.251.233.182(12456):adminuser@postgres:[31236]:STATEMENT:
  select a.aid from pgbench_accounts a, pgbench_accounts b where a.bid=b.bid order by
 a.bid limit 10;
2023-02-06 23:48:35 UTC:205.251.233.182(12456):adminuser@postgres:[31236]:LOG:
 temporary file: path "base/pgsql_tmp/pgsql_tmp31236.4", size 180428800
2023-02-06 23:48:35 UTC:205.251.233.182(12456):adminuser@postgres:[31236]:STATEMENT:
  select a.aid from pgbench_accounts a, pgbench_accounts b where a.bid=b.bid order by
 a.bid limit 10;
```

- **pg_ls_tmpdir** – This function that is available from RDS for PostgreSQL 13 and above provides visibility into the current temporary file usage. The completed query doesn't appear in the results of the function. In the following example, you can view the results of this function.

```
postgres=> select * from pg_ls_tmpdir();
```

```
      name       |    size    |      modification
-----------------+------------+------------------------
 pgsql_tmp8355.1 | 1072250880 | 2023-02-06 22:54:56+00
 pgsql_tmp8351.0 | 1072250880 | 2023-02-06 22:54:43+00
 pgsql_tmp8327.0 | 1072250880 | 2023-02-06 22:54:56+00
 pgsql_tmp8351.1 |  703168512 | 2023-02-06 22:54:56+00
 pgsql_tmp8355.0 | 1072250880 | 2023-02-06 22:54:00+00
 pgsql_tmp8328.1 |  835031040 | 2023-02-06 22:54:56+00
 pgsql_tmp8328.0 | 1072250880 | 2023-02-06 22:54:40+00
(7 rows)
```

```
postgres=> select query from pg_stat_activity where pid = 8355;

query
----------------------------------------------------------------------------------------
select a.aid from pgbench_accounts a, pgbench_accounts b where a.bid=b.bid order by
 a.bid
(1 row)
```

The file name includes the processing ID (PID) of the session that generated the temporary file. A more advanced query, such as in the following example, performs a sum of the temporary files for each PID.

```
postgres=> select replace(left(name, strpos(name, '.')-1),'pgsql_tmp','') as pid,
 count(*), sum(size) from pg_ls_tmpdir() group by pid;
```

```


 pid  | count |   sum
------+-------------------
 8355 |     2 | 2144501760
 8351 |     2 | 2090770432
 8327 |     1 | 1072250880
 8328 |     2 | 2144501760
(4 rows)
```

- **`pg_stat_statements`** – If you activate the pg_stat_statements parameter, then you can view the average temporary file usage per call. You can identify the query_id of the query and use it to examine the temporary file usage as shown in the following example.

```
postgres=> select queryid from pg_stat_statements where query like 'select a.aid from
 pgbench%';
```

```


       queryid
----------------------
 -7170349228837045701
(1 row)
```

```
postgres=> select queryid, substr(query,1,25), calls, temp_blks_read/calls
 temp_blks_read_per_call, temp_blks_written/calls temp_blks_written_per_call from
 pg_stat_statements where queryid = -7170349228837045701;
```

```


       queryid        |          substr           | calls | temp_blks_read_per_call |
 temp_blks_written_per_call
```

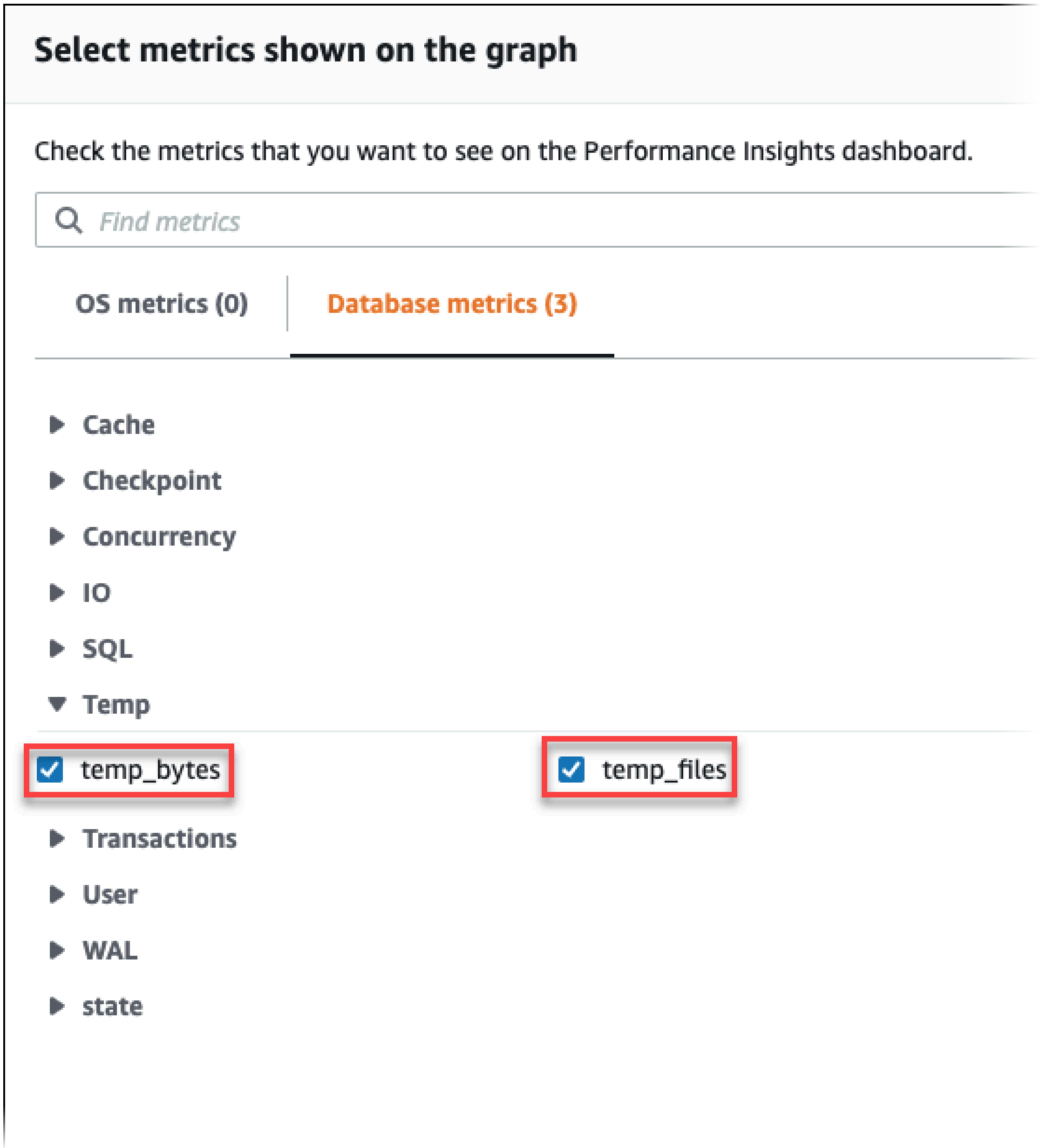
```
----------------------+---------------------------+-------+-------------------------
+----------------------------
 -7170349228837045701 | select a.aid from pgbench |    50 |                  239226 |
                    388678
(1 row)
```

- **Performance Insights** – In the Performance Insights dashboard, you can view temporary file usage by turning on the metrics **temp_bytes** and **temp_files**. Then, you can see the average of both of these metrics and see how they correspond to the query workload. The view within Performance Insights doesn't show specifically the queries that are generating the temporary files. However, when you combine Performance Insights with the query shown for `pg_ls_tmpdir`, you can troubleshoot, analyze, and determine the changes in your query workload.

  For more information about how to analyze metrics and queries with Performance Insights, see Analyzing metrics with the Performance Insights dashboard

  To view the temporary file usage with Performance Insights

  1. In the Performance Insights dashboard, choose **Manage Metrics**.
  2. Choose **Database metrics**, and select the **temp_bytes** and **temp_files** metrics as shown in the following image.



Select metrics shown on the graph

Check the metrics that you want to see on the Performance Insights dashboard.

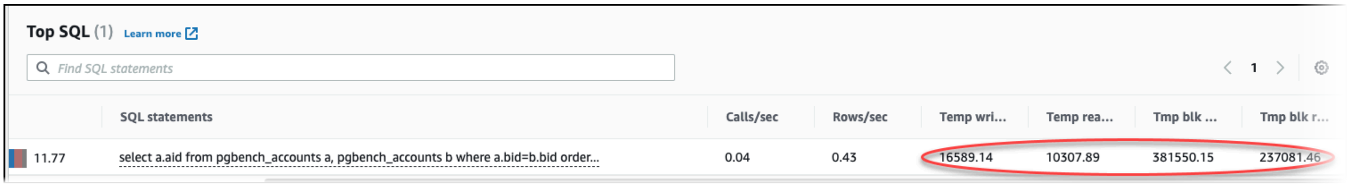
☑ temp_bytes ☑ temp_files

3. In the **Top SQL** tab, choose the **Preferences** icon.

4. In the **Preferences** window, turn on the following statistics to appear in the **Top SQL**tab and choose **Continue**.

   - Temp writes/sec
   - Temp reads/sec
   - Tmp blk write/call
   - Tmp blk read/call

5. The temporary file is broken out when combined with the query shown for `pg_ls_tmpdir`, as shown in the following example.



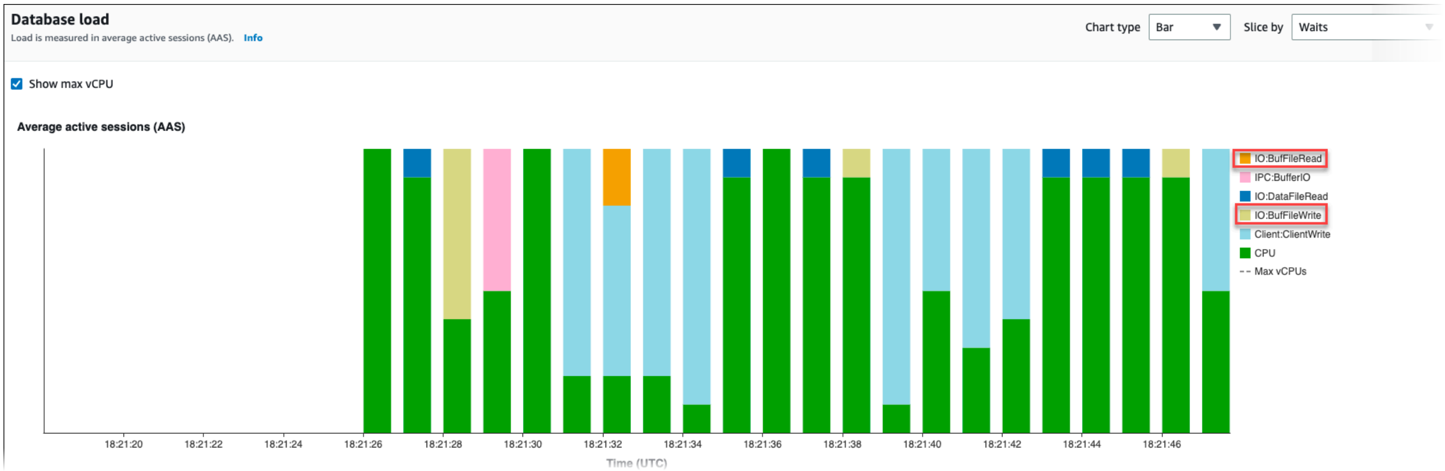
Top SQL (1) Learn more

Find SQL statements

< 1 >

	SQL statements	Calls/sec	Rows/sec	Temp wri...	Temp rea...	Tmp blk ...	Tmp blk r...
11.77	select a.aid from pgbench_accounts a, pgbench_accounts b where a.bid=b.bid order...	0.04	0.43	16589.14	10307.89	381550.15	237081.46

The `IO:BufFileRead` and `IO:BufFileWrite` events occur when the top queries in your workload often create temporary files. You can use Performance Insights to identify top queries waiting on `IO:BufFileRead` and `IO:BufFileWrite` by reviewing Average Active Session (AAS) in Database Load and Top SQL sections.



Database load

Load is measured in average active sessions (AAS). Info

Chart type: Bar

Slice by: Waits

☑ Show max vCPU

For more information on how to analyze top queries and load by wait event with Performance Insights, see Overview of the Top SQL tab. You should identify and tune the queries that cause increase in temporary file usage and related wait events. For more information on these wait events and remediation, see IO:BufFileRead and IO:BufFileWrite.

Note

The `work_mem` parameter controls when the sort operation runs out of memory and results are written into temporary files. We recommend that you don't change the setting of this parameter higher than the default value because it would permit every database session to consume more memory. Also, a single session that performs complex joins and sorts can perform parallel operations in which each operation consumes memory.

As a best practice, when you have a large report with multiple joins and sorts, set this parameter at the session level by using the `SET work_mem` command. Then the change is only applied to the current session and doesn't change the value globally.

Tuning with wait events for Aurora PostgreSQL

Wait events are an important tuning tool for Aurora PostgreSQL. When you can find out why sessions are waiting for resources and what they are doing, you're better able to reduce bottlenecks. You can use the information in this section to find possible causes and corrective actions. Before delving into this section, we strongly recommend that you understand basic Aurora concepts, especially the following topics:

- Amazon Aurora storage and reliability
- Managing performance and scaling for Aurora DB clusters

⚠ **Important**

The wait events in this section are specific to Aurora PostgreSQL. Use the information in this section to tune Amazon Aurora only, not RDS for PostgreSQL.
Some wait events in this section have no analogs in the open source versions of these database engines. Other wait events have the same names as events in open source engines, but behave differently. For example, Amazon Aurora storage works differently from open source storage, so storage-related wait events indicate different resource conditions.

Topics

- Lock:Relation
- Lock:transactionid
- Lock:tuple
- LWLock:buffer_content (BufferContent)
- LWLock:buffer_mapping
- LWLock:BufferIO (IPC:BufferIO)
- LWLock:lock_manager
- LWLock:MultiXact
- Timeout:PgSleep

Essential concepts for Aurora PostgreSQL tuning

Before you tune your Aurora PostgreSQL database, make sure to learn what wait events are and why they occur. Also review the basic memory and disk architecture of Aurora PostgreSQL. For a helpful architecture diagram, see the PostgreSQL wikibook.

Topics

- Aurora PostgreSQL wait events
- Aurora PostgreSQL memory
- Aurora PostgreSQL processes

Aurora PostgreSQL wait events

A *wait event* indicates a resource for which a session is waiting. For example, the wait event `Client:ClientRead` occurs when Aurora PostgreSQL is waiting to receive data from the client. Typical resources that a session waits for include the following:

- Single-threaded access to a buffer, for example, when a session is attempting to modify a buffer
- A row that is currently locked by another session
- A data file read
- A log file write

For example, to satisfy a query, the session might perform a full table scan. If the data isn't already in memory, the session waits for the disk I/O to complete. When the buffers are read into memory, the session might need to wait because other sessions are accessing the same buffers. The database records the waits by using a predefined wait event. These events are grouped into categories.

A wait event doesn't by itself show a performance problem. For example, if requested data isn't in memory, reading data from disk is necessary. If one session locks a row for an update, another session waits for the row to be unlocked so that it can update it. A commit requires waiting for the write to a log file to complete. Waits are integral to the normal functioning of a database.

Large numbers of wait events typically show a performance problem. In such cases, you can use wait event data to determine where sessions are spending time. For example, if a report that typically runs in minutes now runs for hours, you can identify the wait events that contribute the most to total wait time. If you can determine the causes of the top wait events, you can sometimes make changes that improve performance. For example, if your session is waiting on a row that has been locked by another session, you can end the locking session.

Aurora PostgreSQL memory

Aurora PostgreSQL memory is divided into shared and local.

Topics

- Shared memory in Aurora PostgreSQL
- Local memory in Aurora PostgreSQL

Shared memory in Aurora PostgreSQL

Aurora PostgreSQL allocates shared memory when the instance starts. Shared memory is divided into multiple subareas. Following, you can find a description of the most important ones.

Topics

- Shared buffers
- Write ahead log (WAL) buffers

Shared buffers

The *shared buffer pool* is an Aurora PostgreSQL memory area that holds all pages that are or were being used by application connections. A *page* is the memory version of a disk block. The shared buffer pool caches the data blocks read from disk. The pool reduces the need to reread data from disk, making the database operate more efficiently.

Every table and index is stored as an array of pages of a fixed size. Each block contains multiple tuples, which correspond to rows. A tuple can be stored in any page.

The shared buffer pool has finite memory. If a new request requires a page that isn't in memory, and no more memory exists, Aurora PostgreSQL evicts a less frequently used page to accommodate the request. The eviction policy is implemented by a clock sweep algorithm.

The `shared_buffers` parameter determines how much memory the server dedicates to caching data.

Write ahead log (WAL) buffers

A *write-ahead log (WAL) buffer* holds transaction data that Aurora PostgreSQL later writes to persistent storage. Using the WAL mechanism, Aurora PostgreSQL can do the following:

- Recover data after a failure
- Reduce disk I/O by avoiding frequent writes to disk

When a client changes data, Aurora PostgreSQL writes the changes to the WAL buffer. When the client issues a `COMMIT`, the WAL writer process writes transaction data to the WAL file.

The `wal_level` parameter determines how much information is written to the WAL.

Local memory in Aurora PostgreSQL

Every backend process allocates local memory for query processing.

Topics

Work memory area

The *work memory area* holds temporary data for queries that performs sorts and hashes. For example, a query with an `ORDER BY` clause performs a sort. Queries use hash tables in hash joins and aggregations.

The `work_mem` parameter the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. The default value is 4 MB. Multiple sessions can run simultaneously, and each session can run maintenance operations in parallel. For this reason, the total work memory used can be multiples of the `work_mem` setting.

Maintenance work memory area

The *maintenance work memory area* caches data for maintenance operations. These operations include vacuuming, creating an index, and adding foreign keys.

The `maintenance_work_mem` parameter specifies the maximum amount of memory to be used by maintenance operations. The default value is 64 MB. A database session can only run one maintenance operation at a time.

Temporary buffer area

The *temporary buffer area* caches temporary tables for each database session.

Each session allocates temporary buffers as needed up to the limit you specify. When the session ends, the server clears the buffers.

The `temp_buffers` parameter sets the maximum number of temporary buffers used by each session. Before the first use of temporary tables within a session, you can change the `temp_buffers` value.

Aurora PostgreSQL processes

Aurora PostgreSQL uses multiple processes.

Topics

Postmaster process

The *postmaster process* is the first process started when you start Aurora PostgreSQL. The postmaster process has the following primary responsibilities:

- Fork and monitor background processes
- Receive authentication requests from client processes, and authenticate them before allowing the database to service requests

Backend processes

If the postmaster authenticates a client request, the postmaster forks a new backend process, also called a postgres process. One client process connects to exactly one backend process. The client process and the backend process communicate directly without intervention by the postmaster process.

Background processes

The postmaster process forks several processes that perform different backend tasks. Some of the more important include the following:

- WAL writer

  Aurora PostgreSQL writes data in the WAL (write ahead logging) buffer to the log files. The principle of write ahead logging is that the database can't write changes to the data files until after the database writes log records describing those changes to disk. The WAL mechanism reduces disk I/O, and allows Aurora PostgreSQL to use the logs to recover the database after a failure.
- Background writer

  This process periodically write dirty (modified) pages from the memory buffers to the data files. A page becomes dirty when a backend process modifies it in memory.
- Autovacuum daemon

  The daemon consists of the following:

  - The autovacuum launcher
  - The autovacuum worker processes

  When autovacuum is turned on, it checks for tables that have had a large number of inserted, updated, or deleted tuples. The daemon has the following responsibilities:

  - Recover or reuse disk space occupied by updated or deleted rows
  - Update statistics used by the planner
  - Protect against loss of old data because of transaction ID wraparound

The autovacuum feature automates the execution of VACUUM and ANALYZE commands. VACUUM has the following variants: standard and full. Standard vacuum runs in parallel with other database operations. VACUUM FULL requires an exclusive lock on the table it is working on. Thus, it can't run in parallel with operations that access the same table. VACUUM creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions.

Aurora PostgreSQL wait events

The following table lists the wait events for Aurora PostgreSQL that most commonly indicate performance problems, and summarizes the most common causes and corrective actions. The following wait events are a subset of the list in Amazon Aurora PostgreSQL wait events.

Wait event	Definition
Client:ClientRead	This event occurs when Aurora PostgreSQL is waiting to receive data from the client.
Client:ClientWrite	This event occurs when Aurora PostgreSQL is waiting to write data to the client.
CPU	This event occurs when a thread is active in CPU or is waiting for CPU.
IO:BufFileRead and IO:BufFileWrite	These events occur when Aurora PostgreSQL creates temporary files.
IO:DataFileRead	This event occurs when a connection waits on a backend process to read a required page from storage because the page isn't available in shared memory.
IO:XactSync	This event occurs when the database is waiting for the Aurora storage subsystem to acknowledge the commit of a regular transaction, or the commit or rollback of a prepared transaction.
IPC:DamRecordTxAck	This event occurs when Aurora PostgreSQL in a session using database activity streams generates

Wait event	Definition
	an activity stream event, then waits for that event to become durable.
Lock:advisory	This event occurs when a PostgreSQL application uses a lock to coordinate activity across multiple sessions.
Lock:extend	This event occurs when a backend process is waiting to lock a relation to extend it while another process has a lock on that relation for the same purpose.
Lock:Relation	This event occurs when a query is waiting to acquire a lock on a table or view that's currently locked by another transaction.
Lock:transactionid	This event occurs when a transaction is waiting for a row-level lock.
Lock:tuple	This event occurs when a backend process is waiting to acquire a lock on a tuple.
LWLock:buffer_content (BufferContent)	This event occurs when a session is waiting to read or write a data page in memory while another session has that page locked for writing.
LWLock:buffer_mapping	This event occurs when a session is waiting to associate a data block with a buffer in the shared buffer pool.
LWLock:BufferIO (IPC:BufferIO)	This event occurs when Aurora PostgreSQL or RDS for PostgreSQL is waiting for other processes to finish their input/output (I/O) operations when concurrently trying to access a page.

Wait event	Definition
LWLock:lock_manager	This event occurs when the Aurora PostgreSQL engine maintains the shared lock's memory area to allocate, check, and deallocate a lock when a fast path lock isn't possible.
LWLock:MultiXact	This type of event occurs when Aurora PostgreSQL is keeping a session open to complete multiple transactions that involve the same row in a table. The wait event denotes which aspect of multiple transaction processing is generating the wait event, that is, LWLock:MultiXactOffsetSLRU, LWLock:MultiXactOffsetBuffer, LWLock:MultiXactMemberSLRU, or LWLock:MultiXactMemberBuffer.
Timeout:PgSleep	This event occurs when a server process has called the `pg_sleep` function and is waiting for the sleep timeout to expire.

Client:ClientRead

The `Client:ClientRead` event occurs when Aurora PostgreSQL is waiting to receive data from the client.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for Aurora PostgreSQL version 10 and higher.

Context

An Aurora PostgreSQL DB cluster is waiting to receive data from the client. The Aurora PostgreSQL DB cluster must receive the data from the client before it can send more data to the client. The time that the cluster waits before receiving data from the client is a `Client:ClientRead` event.

Likely causes of increased waits

Common causes for the `Client:ClientRead` event to appear in top waits include the following:

Increased network latency

There might be increased network latency between the Aurora PostgreSQL DB cluster and client. Higher network latency increases the time required for DB cluster to receive data from the client.

Increased load on the client

There might be CPU pressure or network saturation on the client. An increase in load on the client can delay transmission of data from the client to the Aurora PostgreSQL DB cluster.

Excessive network round trips

A large number of network round trips between the Aurora PostgreSQL DB cluster and the client can delay transmission of data from the client to the Aurora PostgreSQL DB cluster.

Large copy operation

During a copy operation, the data is transferred from the client's file system to the Aurora PostgreSQL DB cluster. Sending a large amount of data to the DB cluster can delay transmission of data from the client to the DB cluster.

Idle client connection

A connection to an Aurora PostgreSQL DB instance is in idle in transaction state and is waiting for a client to send more data or issue a command. This state can lead to an increase in `Client:ClientRead` events.

PgBouncer used for connection pooling

PgBouncer has a low-level network configuration setting called `pkt_buf`, which is set to 4,096 by default. If the workload is sending query packets larger than 4,096 bytes through PgBouncer, we recommend increasing the `pkt_buf` setting to 8,192. If the new setting doesn't decrease the number of `Client:ClientRead` events, we recommend increasing the `pkt_buf` setting to larger values, such as 16,384 or 32,768. If the query text is large, the larger setting can be particularly helpful.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Place the clients in the same Availability Zone and VPC subnet as the cluster

To reduce network latency and increase network throughput, place clients in the same Availability Zone and virtual private cloud (VPC) subnet as the Aurora PostgreSQL DB cluster. Make sure that the clients are as geographically close to the DB cluster as possible.

Scale your client

Using Amazon CloudWatch or other host metrics, determine if your client is currently constrained by CPU or network bandwidth, or both. If the client is constrained, scale your client accordingly.

Use current generation instances

In some cases, you might not be using a DB instance class that supports jumbo frames. If you're running your application on Amazon EC2, consider using a current generation instance for the client. Also, configure the maximum transmission unit (MTU) on the client operating system. This technique might reduce the number of network round trips and increase network throughput. For more information, see Jumbo frames (9001 MTU) in the *Amazon EC2 User Guide*.

For information about DB instance classes, see Aurora DB instance classes. To determine the DB instance class that is equivalent to an Amazon EC2 instance type, place `db.` before the Amazon EC2 instance type name. For example, the `r5.8xlarge` Amazon EC2 instance is equivalent to the `db.r5.8xlarge` DB instance class.

Increase network bandwidth

Use NetworkReceiveThroughput and NetworkTransmitThroughput Amazon CloudWatch metrics to monitor incoming and outgoing network traffic on the DB cluster. These metrics can help you to determine if network bandwidth is sufficient for your workload.

If your network bandwidth isn't enough, increase it. If the AWS client or your DB instance is reaching the network bandwidth limits, the only way to increase the bandwidth is to increase your DB instance size.

For more information about CloudWatch metrics, see Amazon CloudWatch metrics for Amazon Aurora.

Monitor maximums for network performance

If you are using Amazon EC2 clients, Amazon EC2 provides maximums for network performance metrics, including aggregate inbound and outbound network bandwidth. It also provides connection tracking to ensure that packets are returned as expected and link-local services access for services such as the Domain Name System (DNS). To monitor these maximums, use a current enhanced networking driver and monitor network performance for your client.

For more information, see Monitor network performance for your Amazon EC2 instance in the *Amazon EC2 User Guide* and Monitor network performance for your Amazon EC2 instance in the *Amazon EC2 User Guide*.

Monitor for transactions in the "idle in transaction" state

Check whether you have an increasing number of idle in transaction connections. To do this, monitor the state column in the pg_stat_activity table. You might be able to identify the connection source by running a query similar to the following.

```
select client_addr, state, count(1) from pg_stat_activity
where state like 'idle in transaction%'
group by 1,2
order by 3 desc
```

Client:ClientWrite

The Client:ClientWrite event occurs when Aurora PostgreSQL is waiting to write data to the client.

Topics

Supported engine versions

This wait event information is supported for Aurora PostgreSQL version 10 and higher.

Context

A client process must read all of the data received from an Aurora PostgreSQL DB cluster before the cluster can send more data. The time that the cluster waits before sending more data to the client is a `Client:ClientWrite` event.

Reduced network throughput between the Aurora PostgreSQL DB cluster and the client can cause this event. CPU pressure and network saturation on the client can also cause this event. *CPU pressure* is when the CPU is fully utilized and there are tasks waiting for CPU time. *Network saturation* is when the network between the database and client is carrying more data than it can handle.

Likely causes of increased waits

Common causes for the `Client:ClientWrite` event to appear in top waits include the following:

Increased network latency

> There might be increased network latency between the Aurora PostgreSQL DB cluster and client. Higher network latency increases the time required for the client to receive the data.

Increased load on the client

> There might be CPU pressure or network saturation on the client. An increase in load on the client delays the reception of data from the Aurora PostgreSQL DB cluster.

Large volume of data sent to the client

> The Aurora PostgreSQL DB cluster might be sending a large amount of data to the client. A client might not be able to receive the data as fast as the cluster is sending it. Activities such as a copy of a large table can result in an increase in `Client:ClientWrite` events.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Place the clients in the same Availability Zone and VPC subnet as the cluster
- Use current generation instances
- Reduce the amount of data sent to the client
- Scale your client

Place the clients in the same Availability Zone and VPC subnet as the cluster

To reduce network latency and increase network throughput, place clients in the same Availability Zone and virtual private cloud (VPC) subnet as the Aurora PostgreSQL DB cluster.

Use current generation instances

In some cases, you might not be using a DB instance class that supports jumbo frames. If you're running your application on Amazon EC2, consider using a current generation instance for the client. Also, configure the maximum transmission unit (MTU) on the client operating system. This technique might reduce the number of network round trips and increase network throughput. For more information, see Jumbo frames (9001 MTU) in the *Amazon EC2 User Guide*.

For information about DB instance classes, see Aurora DB instance classes. To determine the DB instance class that is equivalent to an Amazon EC2 instance type, place `db.` before the Amazon EC2 instance type name. For example, the `r5.8xlarge` Amazon EC2 instance is equivalent to the `db.r5.8xlarge` DB instance class.

Reduce the amount of data sent to the client

When possible, adjust your application to reduce the amount of data that the Aurora PostgreSQL DB cluster sends to the client. Making such adjustments relieves CPU and network contention on the client.

Scale your client

Using Amazon CloudWatch or other host metrics, determine if your client is currently constrained by CPU or network bandwidth, or both. If the client is constrained, scale your client accordingly.

CPU

This event occurs when a thread is active in CPU or is waiting for CPU.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is relevant for Aurora PostgreSQL version 9.6 and higher.

Context

The *central processing unit (CPU)* is the component of a computer that runs instructions. For example, CPU instructions perform arithmetic operations and exchange data in memory. If a query increases the number of instructions that it performs through the database engine, the time spent running the query increases. *CPU scheduling* is giving CPU time to a process. Scheduling is orchestrated by the kernel of the operating system.

Topics

- How to tell when this wait occurs
- DBLoadCPU metric
- os.cpuUtilization metrics
- Likely cause of CPU scheduling

How to tell when this wait occurs

This CPU wait event indicates that a backend process is active in CPU or is waiting for CPU. You know that it's occurring when a query shows the following information:

- The `pg_stat_activity.state` column has the value `active`.
- The `wait_event_type` and `wait_event` columns in `pg_stat_activity` are both `null`.

To see the backend processes that are using or waiting on CPU, run the following query.

```
SELECT *
FROM   pg_stat_activity
WHERE  state = 'active'
AND    wait_event_type IS NULL
AND    wait_event IS NULL;
```

DBLoadCPU metric

The Performance Insights metric for CPU is `DBLoadCPU`. The value for `DBLoadCPU` can differ from the value for the Amazon CloudWatch metric `CPUUtilization`. The latter metric is collected from the HyperVisor for a database instance.

os.cpuUtilization metrics

Performance Insights operating-system metrics provide detailed information about CPU utilization. For example, you can display the following metrics:

- `os.cpuUtilization.nice.avg`
- `os.cpuUtilization.total.avg`
- `os.cpuUtilization.wait.avg`
- `os.cpuUtilization.idle.avg`

Performance Insights reports the CPU usage by the database engine as `os.cpuUtilization.nice.avg`.

Likely cause of CPU scheduling

From an operating system perspective, the CPU is active when it isn't running the idle thread. The CPU is active while it performs a computation, but it's also active when it waits on memory I/O. This type of I/O dominates a typical database workload.

Processes are likely to wait to get scheduled on a CPU when the following conditions are met:

- The CloudWatch `CPUUtilization` metric is near 100 percent.
- The average load is greater than the number of vCPUs, indicating a heavy load. You can find the `loadAverageMinute` metric in the OS metrics section in Performance Insights.

Likely causes of increased waits

When the CPU wait event occurs more than normal, possibly indicating a performance problem, typical causes include the following.

Topics

Likely causes of sudden spikes

The most likely causes of sudden spikes are as follows:

- Your application has opened too many simultaneous connections to the database. This scenario is known as a "connection storm."
- Your application workload changed in any of the following ways:
  - New queries
  - An increase in the size of your dataset
  - Index maintenance or creation
  - New functions
  - New operators
  - An increase in parallel query execution
- Your query execution plans have changed. In some cases, a change can cause an increase in buffers. For example, the query is now using a sequential scan when it previously used an index. In this case, the queries need more CPU to accomplish the same goal.

Likely causes of long-term high frequency

The most likely causes of events that recur over a long period:

- Too many backend processes are running concurrently on CPU. These processes can be parallel workers.
- Queries are performing suboptimally because they need a large number of buffers.

Corner cases

If none of the likely causes turn out to be actual causes, the following situations might be occurring:

- The CPU is swapping processes in and out.
- CPU context switching has increased.
- Aurora PostgreSQL code is missing wait events.

Actions

If the CPU wait event dominates database activity, it doesn't necessarily indicate a performance problem. Respond to this event only when performance degrades.

Topics

Investigate whether the database is causing the CPU increase

Examine the `os.cpuUtilization.nice.avg` metric in Performance Insights. If this value is far less than the CPU usage, nondatabase processes are the main contributor to CPU.

Determine whether the number of connections increased

Examine the `DatabaseConnections` metric in Amazon CloudWatch. Your action depends on whether the number increased or decreased during the period of increased CPU wait events.

The connections increased

If the number of connections went up, compare the number of backend processes consuming CPU to the number of vCPUs. The following scenarios are possible:

- The number of backend processes consuming CPU is less than the number of vCPUs.

  In this case, the number of connections isn't an issue. However, you might still try to reduce CPU utilization.

- The number of backend processes consuming CPU is greater than the number of vCPUs.

  In this case, consider the following options:

  - Decrease the number of backend processes connected to your database. For example, implement a connection pooling solution such as RDS Proxy. To learn more, see Using Amazon RDS Proxy for Aurora.
  - Upgrade your instance size to get a higher number of vCPUs.
  - Redirect some read-only workloads to reader nodes, if applicable.

The connections didn't increase

Examine the `blks_hit` metrics in Performance Insights. Look for a correlation between an increase in `blks_hit` and CPU usage. The following scenarios are possible:

- CPU usage and `blks_hit` are correlated.

  In this case, find the top SQL statements that are linked to the CPU usage, and look for plan changes. You can use either of the following techniques:

  - Explain the plans manually and compare them to the expected execution plan.
  - Look for an increase in block hits per second and local block hits per second. In the **Top SQL** section of Performance Insights dashboard, choose **Preferences**.
- CPU usage and `blks_hit` aren't correlated.

  In this case, determine whether any of the following occurs:

  - The application is rapidly connecting to and disconnecting from the database.

    Diagnose this behavior by turning on `log_connections` and `log_disconnections`, then analyzing the PostgreSQL logs. Consider using the `pgbadger` log analyzer. For more information, see https://github.com/darold/pgbadger.
  - The OS is overloaded.

    In this case, Performance Insights shows that backend processes are consuming CPU for a longer time than usual. Look for evidence in the Performance Insights `os.cpuUtilization` metrics or the CloudWatch `CPUUtilization` metric. If the operating system is overloaded, look at Enhanced Monitoring metrics to diagnose further. Specifically, look at the process list and the percentage of CPU consumed by each process.
  - Top SQL statements are consuming too much CPU.

Examine statements that are linked to the CPU usage to see whether they can use less CPU. Run an `EXPLAIN` command, and focus on the plan nodes that have the most impact. Consider using a PostgreSQL execution plan visualizer. To try out this tool, see http://explain.dalibo.com/.

Respond to workload changes

If your workload has changed, look for the following types of changes:

New queries

Check whether the new queries are expected. If so, ensure that their execution plans and the number of executions per second are expected.

An increase in the size of the data set

Determine whether partitioning, if it's not already implemented, might help. This strategy might reduce the number of pages that a query needs to retrieve.

Index maintenance or creation

Check whether the schedule for the maintenance is expected. A best practice is to schedule maintenance activities outside of peak activities.

New functions

Check whether these functions perform as expected during testing. Specifically, check whether the number of executions per second is expected.

New operators

Check whether they perform as expected during the testing.

An increase in running parallel queries

Determine whether any of the following situations has occurred:

- The relations or indexes involved have suddenly grown in size so that they differ significantly from `min_parallel_table_scan_size` or `min_parallel_index_scan_size`.
- Recent changes have been made to `parallel_setup_cost` or `parallel_tuple_cost`.
- Recent changes have been made to `max_parallel_workers` or `max_parallel_workers_per_gather`.

IO:BufFileRead and IO:BufFileWrite

The IO:BufFileRead and IO:BufFileWrite events occur when Aurora PostgreSQL creates temporary files. When operations require more memory than the working memory parameters currently define, they write temporary data to persistent storage. This operation is sometimes called "spilling to disk."

Topics

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

IO:BufFileRead and IO:BufFileWrite relate to the work memory area and maintenance work memory area. For more information about these local memory areas, see Work memory area and Maintenance work memory area.

The default value for work_mem is 4 MB. If one session performs operations in parallel, each worker handling the parallelism uses 4 MB of memory. For this reason, set work_mem carefully. If you increase the value too much, a database running many sessions might consume too much memory. If you set the value too low, Aurora PostgreSQL creates temporary files in local storage. The disk I/O for these temporary files can reduce performance.

If you observe the following sequence of events, your database might be generating temporary files:

1. Sudden and sharp decreases in availability
2. Fast recovery for the free space

You might also see a "chainsaw" pattern. This pattern can indicate that your database is creating small files constantly.

Likely causes of increased waits

In general, these wait events are caused by operations that consume more memory than the `work_mem` or `maintenance_work_mem` parameters allocate. To compensate, the operations write to temporary files. Common causes for the `IO:BufFileRead` and `IO:BufFileWrite` events include the following:

Queries that need more memory than exists in the work memory area

Queries with the following characteristics use the work memory area:

- Hash joins
- `ORDER BY` clause
- `GROUP BY` clause
- `DISTINCT`
- Window functions
- `CREATE TABLE AS SELECT`
- Materialized view refresh

Statements that need more memory than exists in the maintenance work memory area

The following statements use the maintenance work memory area:

- `CREATE INDEX`
- `CLUSTER`

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Identify the problem
- Examine your join queries
- Examine your ORDER BY and GROUP BY queries
- Avoid using the DISTINCT operation
- Consider using window functions instead of GROUP BY functions

- Investigate materialized views and CTAS statements
- Use pg_repack when you create indexes
- Increase maintenance_work_mem when you cluster tables
- Tune memory to prevent IO:BufFileRead and IO:BufFileWrite

Identify the problem

Assume a situation in which Performance Insights isn't turned on and you suspect that `IO:BufFileRead` and `IO:BufFileWrite` are occurring more frequently than is normal. Do the following:

1. Examine the `FreeLocalStorage` metric in Amazon CloudWatch.
2. Look for a chainsaw pattern, which is a series of jagged spikes.

A chainsaw pattern indicates a quick consumption and release of storage, often associated with temporary files. If you notice this pattern, turn on Performance Insights. When using Performance Insights, you can identify when the wait events occur and which queries are associated with them. Your solution depends on the specific query causing the events.

Or set the parameter `log_temp_files`. This parameter logs all queries generating more than threshold KB of temporary files. If the value is `0`, Aurora PostgreSQL logs all temporary files. If the value is `1024`, Aurora PostgreSQL logs all queries that produces temporary files larger than 1 MB. For more information about `log_temp_files`, see Error Reporting and Logging in the PostgreSQL documentation.

Examine your join queries

Your application probably use joins. For example, the following query joins four tables.

```
SELECT *
       FROM order
 INNER JOIN order_item
       ON (order.id = order_item.order_id)
 INNER JOIN customer
       ON (customer.id = order.customer_id)
 INNER JOIN customer_address
       ON (customer_address.customer_id = customer.id AND
           order.customer_address_id = customer_address.id)
 WHERE customer.id = 1234567890;
```

A possible cause of spikes in temporary file usage is a problem in the query itself. For example, a broken clause might not filter the joins properly. Consider the second inner join in the following example.

```
SELECT *
       FROM order
 INNER JOIN order_item
       ON (order.id = order_item.order_id)
 INNER JOIN customer
       ON (customer.id = customer.id)
 INNER JOIN customer_address
       ON (customer_address.customer_id = customer.id AND
           order.customer_address_id = customer_address.id)
 WHERE customer.id = 1234567890;
```

The preceding query mistakenly joins `customer.id` to `customer.id`, generating a Cartesian product between every customer and every order. This type of accidental join generates large temporary files. Depending on the size of the tables, a Cartesian query can even fill up storage. Your application might have Cartesian joins when the following conditions are met:

- You see large, sharp decreases in storage availability, followed by fast recovery.
- No indexes are being created.
- No `CREATE TABLE FROM SELECT` statements are being issued.
- No materialized views are being refreshed.

To see whether the tables are being joined using the proper keys, inspect your query and object-relational mapping directives. Bear in mind that certain queries of your application are not called all the time, and some queries are dynamically generated.

Examine your ORDER BY and GROUP BY queries

In some cases, an `ORDER BY` clause can result in excessive temporary files. Consider the following guidelines:

- Only include columns in an `ORDER BY` clause when they need to be ordered. This guideline is especially important for queries that return thousands of rows and specify many columns in the `ORDER BY` clause.

- Considering creating indexes to accelerate ORDER BY clauses when they match columns that have the same ascending or descending order. Partial indexes are preferable because they are smaller. Smaller indexes are read and traversed more quickly.
- If you create indexes for columns that can accept null values, consider whether you want the null values stored at the end or at the beginning of the indexes.

  If possible, reduce the number of rows that need to be ordered by filtering the result set. If you use WITH clause statements or subqueries, remember that an inner query generates a result set and passes it to the outside query. The more rows that a query can filter out, the less ordering the query needs to do.
- If you don't need to obtain the full result set, use the LIMIT clause. For example, if you only want the top five rows, a query using the LIMIT clause doesn't keep generating results. In this way, the query requires less memory and temporary files.

A query that uses a GROUP BY clause can also require temporary files. GROUP BY queries summarize values by using functions such as the following:

- COUNT
- AVG
- MIN
- MAX
- SUM
- STDDEV

To tune GROUP BY queries, follow the recommendations for ORDER BY queries.

Avoid using the DISTINCT operation

If possible, avoid using the DISTINCT operation to remove duplicated rows. The more unnecessary and duplicated rows that your query returns, the more expensive the DISTINCT operation becomes. If possible, add filters in the WHERE clause even if you use the same filters for different tables. Filtering the query and joining correctly improves your performance and reduces resource use. It also prevents incorrect reports and results.

If you need to use DISTINCT for multiple rows of a same table, consider creating a composite index. Grouping multiple columns in an index can improve the time to evaluate distinct rows. Also, if you use Amazon Aurora PostgreSQL version 10 or higher, you can correlate statistics among multiple columns by using the CREATE STATISTICS command.

Consider using window functions instead of GROUP BY functions

Using GROUP BY, you change the result set, and then retrieve the aggregated result. Using window functions, you aggregate data without changing the result set. A window function uses the OVER clause to perform calculations across the sets defined by the query, correlating one row with another. You can use all the GROUP BY functions in window functions, but also use functions such as the following:

- RANK
- ARRAY_AGG
- ROW_NUMBER
- LAG
- LEAD

To minimize the number of temporary files generated by a window function, remove duplications for the same result set when you need two distinct aggregations. Consider the following query.

```
SELECT sum(salary) OVER (PARTITION BY dept ORDER BY salary DESC) as sum_salary
     , avg(salary) OVER (PARTITION BY dept ORDER BY salary ASC) as avg_salary
  FROM empsalary;
```

You can rewrite the query with the WINDOW clause as follows.

```
SELECT sum(salary) OVER w as sum_salary
         , avg(salary) OVER w as_avg_salary
    FROM empsalary
  WINDOW w AS (PARTITION BY dept ORDER BY salary DESC);
```

By default, the Aurora PostgreSQL execution planner consolidates similar nodes so that it doesn't duplicate operations. However, by using an explicit declaration for the window block, you can maintain the query more easily. You might also improve performance by preventing duplication.

Investigate materialized views and CTAS statements

When a materialized view refreshes, it runs a query. This query can contain an operation such as GROUP BY, ORDER BY, or DISTINCT. During a refresh, you might observe large numbers of temporary files and the wait events `IO:BufFileWrite` and `IO:BufFileRead`. Similarly, when you create a table based on a `SELECT` statement, the `CREATE TABLE` statement runs a query. To reduce the temporary files needed, optimize the query.

Use pg_repack when you create indexes

When you create an index, the engine orders the result set. As tables grow in size, and as values in the indexed column become more diverse, the temporary files require more space. In most cases, you can't prevent the creation of temporary files for large tables without modifying the maintenance work memory area. For more information, see Maintenance work memory area.

A possible workaround when recreating a large index is to use the pg_repack tool. For more information, see Reorganize tables in PostgreSQL databases with minimal locks in the pg_repack documentation.

Increase maintenance_work_mem when you cluster tables

The `CLUSTER` command clusters the table specified by *table_name* based on an existing index specified by *index_name*. Aurora PostgreSQL physically recreates the table to match the order of a given index.

When magnetic storage was prevalent, clustering was common because storage throughput was limited. Now that SSD-based storage is common, clustering is less popular. However, if you cluster tables, you can still increase performance slightly depending on the table size, index, query, and so on.

If you run the `CLUSTER` command and observe the wait events `IO:BufFileWrite` and `IO:BufFileRead`, tune `maintenance_work_mem`. Increase the memory size to a fairly large amount. A high value means that the engine can use more memory for the clustering operation.

Tune memory to prevent IO:BufFileRead and IO:BufFileWrite

In some situation, you need to tune memory. Your goal is to balance the following requirements:

- The `work_mem` value (see Work memory area)
- The memory remaining after discounting the `shared_buffers` value (see Buffer pool)
- The maximum connections opened and in use, which is limited by `max_connections`

Increase the size of the work memory area

In some situations, your only option is to increase the memory used by your session. If your queries are correctly written and are using the correct keys for joins, consider increasing the `work_mem` value. For more information, see Work memory area.

To find out how many temporary files a query generates, set `log_temp_files` to `0`. If you increase the `work_mem` value to the maximum value identified in the logs, you prevent the query from generating temporary files. However, `work_mem` sets the maximum per plan node for each connection or parallel worker. If the database has 5,000 connections, and if each one uses 256 MiB memory, the engine needs 1.2 TiB of RAM. Thus, your instance might run out of memory.

Reserve sufficient memory for the shared buffer pool

Your database uses memory areas such as the shared buffer pool, not just the work memory area. Consider the requirements of these additional memory areas before you increase `work_mem`. For more information about the buffer pool, see Buffer pool.

For example, assume that your Aurora PostgreSQL instance class is db.r5.2xlarge. This class has 64 GiB of memory. By default, 75 percent of the memory is reserved for the shared buffer pool. After you subtract the amount allocated to the shared memory area, 16,384 MB remains. Don't allocate the remaining memory exclusively to the work memory area because the operating system and the engine also require memory.

The memory that you can allocate to `work_mem` depends on the instance class. If you use a larger instance class, more memory is available. However, in the preceding example, you can't use more than 16 GiB. Otherwise, your instance becomes unavailable when it runs out of memory. To recover the instance from the unavailable state, the Aurora PostgreSQL automation services automatically restart.

Manage the number of connections

Suppose that your database instance has 5,000 simultaneous connections. Each connection uses at least 4 MiB of `work_mem`. The high memory consumption of the connections is likely to degrade performance. In response, you have the following options:

- Upgrade to a larger instance class.
- Decrease the number of simultaneous database connections by using a connection proxy or pooler.

For proxies, consider Amazon RDS Proxy, pgBouncer, or a connection pooler based on your application. This solution alleviates the CPU load. It also reduces the risk when all connections require the work memory area. When fewer database connections exist, you can increase the value of `work_mem`. In this way, you reduce the occurrence of the `IO:BufFileRead` and `IO:BufFileWrite` wait events. Also, the queries waiting for the work memory area speed up significantly.

IO:DataFileRead

The `IO:DataFileRead` event occurs when a connection waits on a backend process to read a required page from storage because the page isn't available in shared memory.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

All queries and data manipulation (DML) operations access pages in the buffer pool. Statements that can induce reads include `SELECT`, `UPDATE`, and `DELETE`. For example, an `UPDATE` can read pages from tables or indexes. If the page being requested or updated isn't in the shared buffer pool, this read can lead to the `IO:DataFileRead` event.

Because the shared buffer pool is finite, it can fill up. In this case, requests for pages that aren't in memory force the database to read blocks from disk. If the `IO:DataFileRead` event occurs frequently, your shared buffer pool might be too small to accommodate your workload. This problem is acute for `SELECT` queries that read a large number of rows that don't fit in the buffer pool. For more information about the buffer pool, see Buffer pool.

Likely causes of increased waits

Common causes for the `IO:DataFileRead` event include the following:

Connection spikes

You might find multiple connections generating the same number of IO:DataFileRead wait events. In this case, a spike (sudden and large increase) in `IO:DataFileRead` events can occur.

SELECT and DML statements performing sequential scans

Your application might be performing a new operation. Or an existing operation might change because of a new execution plan. In such cases, look for tables (particularly large tables) that have a greater `seq_scan` value. Find them by querying `pg_stat_user_tables`. To track queries that are generating more read operations, use the extension `pg_stat_statements`.

CTAS and CREATE INDEX for large data sets

A *CTAS* is a `CREATE TABLE AS SELECT` statement. If you run a CTAS using a large data set as a source, or create an index on a large table, the `IO:DataFileRead` event can occur. When you create an index, the database might need to read the entire object using a sequential scan. A CTAS generates `IO:DataFile` reads when pages aren't in memory.

Multiple vacuum workers running at the same time

Vacuum workers can be triggered manually or automatically. We recommend adopting an aggressive vacuum strategy. However, when a table has many updated or deleted rows, the `IO:DataFileRead` waits increase. After space is reclaimed, the vacuum time spent on `IO:DataFileRead` decreases.

Ingesting large amounts of data

When your application ingests large amounts of data, `ANALYZE` operations might occur more often. The `ANALYZE` process can be triggered by an autovacuum launcher or invoked manually.

The `ANALYZE` operation reads a subset of the table. The number of pages that must be scanned is calculated by multiplying 30 by the `default_statistics_target` value. For more information, see the PostgreSQL documentation. The `default_statistics_target` parameter accepts values between 1 and 10,000, where the default is 100.

Resource starvation

If instance network bandwidth or CPU are consumed, the `IO:DataFileRead` event might occur more frequently.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Check predicate filters for queries that generate waits

Assume that you identify specific queries that are generating `IO:DataFileRead` wait events. You might identify them using the following techniques:

- Performance Insights
- Catalog views such as the one provided by the extension `pg_stat_statements`
- The catalog view `pg_stat_all_tables`, if it periodically shows an increased number of physical reads
- The `pg_statio_all_tables` view, if it shows that `_read` counters are increasing

We recommend that you determine which filters are used in the predicate (`WHERE` clause) of these queries. Follow these guidelines:

- Run the `EXPLAIN` command. In the output, identify which types of scans are used. A sequential scan doesn't necessarily indicate a problem. Queries that use sequential scans naturally produce more `IO:DataFileRead` events when compared to queries that use filters.

  Find out whether the column listed in the `WHERE` clause is indexed. If not, consider creating an index for this column. This approach avoids the sequential scans and reduces the `IO:DataFileRead` events. If a query has restrictive filters and still produces sequential scans, evaluate whether the proper indexes are being used.
- Find out whether the query is accessing a very large table. In some cases, partitioning a table can improve performance, allowing the query to only read necessary partitions.
- Examine the cardinality (total number of rows) from your join operations. Note how restrictive the values are that you're passing in the filters for your `WHERE` clause. If possible, tune your query to reduce the number of rows that are passed in each step of the plan.

Minimize the effect of maintenance operations

Maintenance operations such as VACUUM and ANALYZE are important. We recommend that you don't turn them off because you find IO:DataFileRead wait events related to these maintenance operations. The following approaches can minimize the effect of these operations:

- Run maintenance operations manually during off-peak hours. This technique prevents the database from reaching the threshold for automatic operations.
- For very large tables, consider partitioning the table. This technique reduces the overhead of maintenance operations. The database only accesses the partitions that require maintenance.
- When you ingest large amounts of data, consider disabling the autoanalyze feature.

The autovacuum feature is automatically triggered for a table when the following formula is true.

```
pg_stat_user_tables.n_dead_tup > (pg_class.reltuples x autovacuum_vacuum_scale_factor)
 + autovacuum_vacuum_threshold
```

The view pg_stat_user_tables and catalog pg_class have multiple rows. One row can correspond to one row in your table. This formula assumes that the reltuples are for a specific table. The parameters autovacuum_vacuum_scale_factor (0.20 by default) and autovacuum_vacuum_threshold (50 tuples by default) are usually set globally for the whole instance. However, you can set different values for a specific table.

Topics

Find tables consuming space unnecessarily

To find tables consuming more space than necessary, run the following query. When this query is run by a database user role that doesn't have the rds_superuser role, it returns information about only those tables that the user role has permissions to read. This query is supported by PostgreSQL version 12 and later versions.

```
WITH report AS (
   SELECT   schemaname
```

```
           ,tblname
           ,n_dead_tup
           ,n_live_tup
           ,block_size*tblpages AS real_size
           ,(tblpages-est_tblpages)*block_size AS extra_size
           ,CASE WHEN tblpages - est_tblpages > 0
              THEN 100 * (tblpages - est_tblpages)/tblpages::float
              ELSE 0
            END AS extra_ratio, fillfactor, (tblpages-est_tblpages_ff)*block_size AS
 bloat_size
           ,CASE WHEN tblpages - est_tblpages_ff > 0
              THEN 100 * (tblpages - est_tblpages_ff)/tblpages::float
              ELSE 0
            END AS bloat_ratio
           ,is_na
    FROM (
           SELECT  ceil( reltuples / ( (block_size-page_hdr)/tpl_size ) ) +
 ceil( toasttuples / 4 ) AS est_tblpages
                  ,ceil( reltuples / ( (block_size-page_hdr)*fillfactor/
(tpl_size*100) ) ) + ceil( toasttuples / 4 ) AS est_tblpages_ff
                  ,tblpages
                  ,fillfactor
                  ,block_size
                  ,tblid
                  ,schemaname
                  ,tblname
                  ,n_dead_tup
                  ,n_live_tup
                  ,heappages
                  ,toastpages
                  ,is_na
             FROM (
                    SELECT ( 4 + tpl_hdr_size + tpl_data_size + (2*ma)
                               - CASE WHEN tpl_hdr_size%ma = 0 THEN ma ELSE
 tpl_hdr_size%ma END
                               - CASE WHEN ceil(tpl_data_size)::int%ma = 0 THEN ma ELSE
 ceil(tpl_data_size)::int%ma END
                           ) AS tpl_size
                           ,block_size - page_hdr AS size_per_block
                           ,(heappages + toastpages) AS tblpages
                           ,heappages
                           ,toastpages
                           ,reltuples
                           ,toasttuples
                           ,block_size
                           ,page_hdr
                           ,tblid
                           ,schemaname
                           ,tblname
                           ,fillfactor
                           ,is_na
                           ,n_dead_tup
                           ,n_live_tup
                          FROM (
                                SELECT  tbl.oid                       AS tblid
                                       ,ns.nspname                    AS schemaname
                                       ,tbl.relname                   AS tblname
                                       ,tbl.reltuples                 AS reltuples
                                       ,tbl.relpages                  AS heappages
                                       ,coalesce(toast.relpages, 0)   AS toastpages
                                       ,coalesce(toast.reltuples, 0)  AS toasttuples
                                       ,psat.n_dead_tup               AS n_dead_tup
                                       ,psat.n_live_tup               AS n_live_tup
                                       ,24                            AS page_hdr
                                       ,current_setting('block_size')::numeric AS
 block_size

 ,coalesce(substring( array_to_string(tbl.reloptions, ' ') FROM
 'fillfactor=([0-9]+)')::smallint, 100) AS fillfactor
                                       ,CASE WHEN version()~'mingw32' OR version()~'64-
bit|x86_64|ppc64|ia64|amd64' THEN 8 ELSE 4 END        AS ma
                                       ,23 + CASE WHEN MAX(coalesce(null_frac,0)) > 0
 THEN ( 7 + count(*) ) / 8 ELSE 0::int END              AS tpl_hdr_size
                                       ,sum( (1-coalesce(s.null_frac, 0)) *
 coalesce(s.avg_width, 1024) )                                    AS tpl_data_size
                                       ,bool_or(att.atttypid =
 'pg_catalog.name'::regtype) OR count(att.attname) <> count(s.attname)         AS is_na
                                  FROM  pg_attribute       AS att
                                  JOIN  pg_class           AS tbl    ON (att.attrelid =
 tbl.oid)
                                  JOIN  pg_stat_all_tables AS psat   ON (tbl.oid =
 psat.relid)
                                  JOIN  pg_namespace       AS ns     ON (ns.oid =
 tbl.relnamespace)
                             LEFT JOIN  pg_stats           AS s      ON
 (s.schemaname=ns.nspname AND s.tablename = tbl.relname AND s.inherited=false AND
 s.attname=att.attname)
```

```
                             LEFT JOIN  pg_class           AS toast  ON
 (tbl.reltoastrelid = toast.oid)
                                 WHERE  att.attnum > 0
                                   AND  NOT att.attisdropped
                                   AND  tbl.relkind = 'r'
                              GROUP BY  tbl.oid, ns.nspname, tbl.relname,
 tbl.reltuples, tbl.relpages, toastpages, toasttuples, fillfactor, block_size, ma,
 n_dead_tup, n_live_tup
                              ORDER BY  schemaname, tblname
                           ) AS s
                 ) AS s2
       ) AS s3
 ORDER BY bloat_size DESC
)
  SELECT *
    FROM report
   WHERE bloat_ratio != 0
 -- AND schemaname = 'public'
 -- AND tblname = 'pgbench_accounts'
;

-- WHERE NOT is_na
--   AND tblpages*((pst).free_percent + (pst).dead_tuple_percent)::float4/100 >= 1
```

You can check for table and index bloat in your application. For more information, see

You can use PostgreSQL Multiversion Concurrency Control (MVCC) to help preserve data integrity. PostgreSQL MVCC works by saving an internal copy of updated or deleted rows (also called *tuples*) until a transaction is either committed or rolled back. This saved internal copy is invisible to users. However, table bloat can occur when those invisible copies aren't cleaned up regularly by the VACUUM or AUTOVACUUM utilities. Unchecked, table bloat can incur increased storage costs and slow your processing speed.

In many cases, the default settings for VACUUM or AUTOVACUUM on Aurora are sufficient for handling unwanted table bloat. However, you may want to check for bloat if your application is experiencing the following conditions:

- Processes a large number of transactions in a relatively short time between VACUUM processes.
- Performs poorly and runs out of storage.

To get started, gather the most accurate information about how much space is used by dead tuples and how much you can recover by cleaning up the table and index bloat. To do so, use the `pgstattuple` extension to gather statistics on your Aurora cluster. For more information, see pgstattuple. Privileges to use the `pgstattuple` extension are limited to the `pg_stat_scan_tables` role and database superusers.

To create the `pgstattuple` extension on Aurora, connect a client session to the cluster, for example, psql or pgAdmin, and use the following command:

```
CREATE EXTENSION pgstattuple;
```

Create the extension in each database that you want to profile. After creating the extension, use the command line interface (CLI) to measure how much unusable space you can reclaim. Before gathering statistics, modify the cluster parameter group by setting AUTOVACUUM to 0. A setting of 0 prevents Aurora from automatically cleaning up any dead tuples left behind by your application, which can impact the accuracy of the results. Enter the following command to create a simple table:

```
postgres=> CREATE TABLE lab AS SELECT generate_series (0,100000);
SELECT 100001
```

In the following example, we run the query with AUTOVACUUM turned on for the DB cluster. The `dead_tuple_count` is 0, which indicates that the AUTOVACUUM has deleted the obsolete data or tuples from the PostgreSQL database.

To use `pgstattuple` to gather information about the table, specify the name of a table or an object identifier (OID) in the query:

```
postgres=> SELECT * FROM pgstattuple('lab');
```

```
table_len  | tuple_count | tuple_len | tuple_percent | dead_tuple_count |
 dead_tuple_len | dead_tuple_percent | free_space | free_percent
-----------+-------------+-----------+---------------+------------------
+----------------+--------------------+------------+--------------
3629056    | 100001      | 2800028   | 77.16         | 0                | 0
```

In the following query, we turn off AUTOVACUUM and enter a command that deletes 25,000 rows from the table. As a result, the `dead_tuple_count` increases to 25000.

```
postgres=> DELETE FROM lab WHERE generate_series < 25000;

DELETE 25000
```

```
SELECT * FROM pgstattuple('lab');
```

```
table_len | tuple_count | tuple_len | tuple_percent | dead_tuple_count | dead_tuple_len
 | dead_tuple_percent | free_space | free_percent
-----------+-------------+-----------+---------------+------------------
+-----------------+--------------------+-----------+--------------
3629056 | 75001 | 2100028 | 57.87 | 25000 | 700000 | 19.29 | 16616 | 0.46
(1 row)
```

To reclaim those dead tuples, start a VACUUM process.

Observing bloat without interrupting your application

After running your applications and viewing the result, use pg_repack or VACUUM FULL on the restored copy and compare the differences. If you see a significant drop in the dead_tuple_count, dead_tuple_len, or dead_tuple_percent, then adjust the vacuum schedule on your production cluster to minimize the bloat.

Avoiding bloat in temporary tables

If your application creates temporary tables, make sure that your application removes those temporary tables when they're no longer needed. Autovacuum processes don't locate temporary tables. Left unchecked, temporary tables can quickly create database bloat. Moreover, the bloat can extend into the system tables, which are the internal tables that track PostgreSQL objects and attributes, like pg_attribute and pg_depend.

When a temporary table is no longer needed, you can use a TRUNCATE statement to empty the table and free up the space. Then, manually vacuum the pg_attribute and pg_depend tables. Vacuuming these tables ensures that creating and truncating/deleting temporary tables continually isn't adding tuples and contributing to system bloat.

You can avoid this problem while creating a temporary table by including the following syntax that deletes the new rows when content is committed:

```
CREATE TEMP TABLE IF NOT EXISTS table_name(table_description) ON COMMIT DELETE ROWS;
```

The `ON COMMIT DELETE ROWS` clause truncates the temporary table when the transaction is committed.

Avoiding bloat in indexes

When you change an indexed field in a table, the index update results in one or more dead tuples in that index. By default, the autovacuum process cleans up bloat in indexes, but that cleanup uses a significant amount of time and resources. To specify index cleanup preferences when you create a table, include the vacuum_index_cleanup clause. By default, at table creation time, the clause is set to AUTO, which means that the server decides if your index requires cleanup when it vacuums the table. You can set the clause to ON to turn on index cleanup for a specific table, or OFF to turn off index cleanup for that table. Remember, turning off index cleanup might save time, but can potentially lead to a bloated index.

You can manually control index cleanup when you VACUUM a table at the command line. To vacuum a table and remove dead tuples from the indexes, include the INDEX_CLEANUP clause with a value of ON and the table name:

```
acctg=> VACUUM (INDEX_CLEANUP ON) receivables;
```

```

```

```
INFO: aggressively vacuuming "public.receivables"
```

```
VACUUM
```

To vacuum a table without cleaning the indexes, specify a value of OFF:

```
acctg=> VACUUM (INDEX_CLEANUP OFF) receivables;
```

```

```

```
INFO: aggressively vacuuming "public.receivables"
```

```
VACUUM
```

.

Find indexes consuming unnecessary space

To find indexes consuming unnecessary space, run the following query.

```
-- WARNING: run with a nonsuperuser role, the query inspects
-- only indexes on tables you have permissions to read.
-- WARNING: rows with is_na = 't' are known to have bad statistics ("name" type is not
 supported).
-- This query is compatible with PostgreSQL 8.2 and later.

SELECT current_database(), nspname AS schemaname, tblname, idxname,
 bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_ratio,
  fillfactor, bs*(relpages-est_pages_ff) AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_ratio,
  is_na
  -- , 100-(sub.pst).avg_leaf_density, est_pages, index_tuple_hdr_bm,
  -- maxalign, pagehdr, nulldatawidth, nulldatahdrwidth, sub.reltuples, sub.relpages
  -- (DEBUG INFO)
FROM (
  SELECT coalesce(1 +
       ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)), 0
       -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
    ) AS est_pages,
    coalesce(1 +
       ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/
(100*(4+nulldatahdrwidth)::float))), 0
```

```
   bs, nspname, table_oid, tblname, idxname, relpages, fillfactor, is_na
   -- , stattuple.pgstatindex(quote_ident(nspname)||'.'||quote_ident(idxname)) AS
pst,
   -- index_tuple_hdr_bm, maxalign, pagehdr, nulldatawidth, nulldatahdrwidth,
reltuples
   -- (DEBUG INFO)
 FROM (
   SELECT maxalign, bs, nspname, tblname, idxname, reltuples, relpages, relam,
table_oid, fillfactor,
     ( index_tuple_hdr_bm +
         maxalign - CASE -- Add padding to the index tuple header to align on MAXALIGN
           WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
           ELSE index_tuple_hdr_bm%maxalign
         END
       + nulldatawidth + maxalign - CASE -- Add padding to the data to align on
MAXALIGN
           WHEN nulldatawidth = 0 THEN 0
           WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
           ELSE nulldatawidth::integer%maxalign
         END
     )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
     -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
   FROM (
     SELECT
       i.nspname, i.tblname, i.idxname, i.reltuples, i.relpages, i.relam, a.attrelid
AS table_oid,
       current_setting('block_size')::numeric AS bs, fillfactor,
       CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
         WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|ppc64|ia64|amd64'
THEN 8
         ELSE 4
       END AS maxalign,
       /* per page header, fixed size: 20 for 7.X, 24 for others */
       24 AS pagehdr,
       /* per page btree opaque data */
       16 AS pageopqdata,
       /* per tuple header: add IndexAttributeBitMapData if some cols are null-able */
       CASE WHEN max(coalesce(s.null_frac,0)) = 0
         THEN 2 -- IndexTupleData size
         ELSE 2 + (( 32 + 8 - 1 ) / 8)
         -- IndexTupleData size + IndexAttributeBitMapData size ( max num filed per
index + 8 - 1 /8)
       END AS index_tuple_hdr_bm,
        /* data len: we remove null values save space using it fractionnal part from
 stats */
        sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 1024)) AS
 nulldatawidth,
        max( CASE WHEN a.atttypid = 'pg_catalog.name'::regtype THEN 1 ELSE 0 END ) > 0
 AS is_na
      FROM pg_attribute AS a
        JOIN (
          SELECT nspname, tbl.relname AS tblname, idx.relname AS idxname,
            idx.reltuples, idx.relpages, idx.relam,
            indrelid, indexrelid, indkey::smallint[] AS attnum,
            coalesce(substring(
              array_to_string(idx.reloptions, ' ')
               from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor
          FROM pg_index
            JOIN pg_class idx ON idx.oid=pg_index.indexrelid
            JOIN pg_class tbl ON tbl.oid=pg_index.indrelid
            JOIN pg_namespace ON pg_namespace.oid = idx.relnamespace
          WHERE pg_index.indisvalid AND tbl.relkind = 'r' AND idx.relpages > 0
        ) AS i ON a.attrelid = i.indexrelid
        JOIN pg_stats AS s ON s.schemaname = i.nspname
          AND ((s.tablename = i.tblname AND s.attname =
 pg_catalog.pg_get_indexdef(a.attrelid, a.attnum, TRUE))
          -- stats from tbl
          OR  (s.tablename = i.idxname AND s.attname = a.attname))
          -- stats from functionnal cols
        JOIN pg_type AS t ON a.atttypid = t.oid
      WHERE a.attnum > 0
      GROUP BY 1, 2, 3, 4, 5, 6, 7, 8, 9
    ) AS s1
  ) AS s2
    JOIN pg_am am ON s2.relam = am.oid WHERE am.amname = 'btree'
) AS sub
-- WHERE NOT is_na
ORDER BY 2,3,4;
```

Find tables that are eligible to be autovacuumed

To find tables that are eligible to be autovacuumed, run the following query.

```
--This query shows tables that need vacuuming and are eligible candidates.
--The following query lists all tables that are due to be processed by autovacuum.
-- During normal operation, this query should return very little.
WITH  vbt AS (SELECT setting AS autovacuum_vacuum_threshold
              FROM pg_settings WHERE name = 'autovacuum_vacuum_threshold')
    , vsf AS (SELECT setting AS autovacuum_vacuum_scale_factor
              FROM pg_settings WHERE name = 'autovacuum_vacuum_scale_factor')
    , fma AS (SELECT setting AS autovacuum_freeze_max_age
              FROM pg_settings WHERE name = 'autovacuum_freeze_max_age')
    , sto AS (SELECT opt_oid, split_part(setting, '=', 1) as param,
                split_part(setting, '=', 2) as value
              FROM (SELECT oid opt_oid, unnest(reloptions) setting FROM pg_class) opt)
SELECT
    '"'||ns.nspname||'"."'||c.relname||'"' as relation
    , pg_size_pretty(pg_table_size(c.oid)) as table_size
    , age(relfrozenxid) as xid_age
    , coalesce(cfma.value::float, autovacuum_freeze_max_age::float)
 autovacuum_freeze_max_age
    , (coalesce(cvbt.value::float, autovacuum_vacuum_threshold::float) +
         coalesce(cvsf.value::float,autovacuum_vacuum_scale_factor::float) *
 c.reltuples)
         as autovacuum_vacuum_tuples
    , n_dead_tup as dead_tuples
FROM pg_class c
JOIN pg_namespace ns ON ns.oid = c.relnamespace
JOIN pg_stat_all_tables stat ON stat.relid = c.oid
JOIN vbt on (1=1)
JOIN vsf ON (1=1)
JOIN fma on (1=1)
LEFT JOIN sto cvbt ON cvbt.param = 'autovacuum_vacuum_threshold' AND c.oid =
 cvbt.opt_oid
LEFT JOIN sto cvsf ON cvsf.param = 'autovacuum_vacuum_scale_factor' AND c.oid =
 cvsf.opt_oid
LEFT JOIN sto cfma ON cfma.param = 'autovacuum_freeze_max_age' AND c.oid = cfma.opt_oid
WHERE c.relkind = 'r'
AND nspname <> 'pg_catalog'
AND (
    age(relfrozenxid) >= coalesce(cfma.value::float, autovacuum_freeze_max_age::float)
    or
    coalesce(cvbt.value::float, autovacuum_vacuum_threshold::float) +
      coalesce(cvsf.value::float,autovacuum_vacuum_scale_factor::float) * c.reltuples
 <= n_dead_tup
    -- or 1 = 1
)
ORDER BY age(relfrozenxid) DESC;
```

Respond to high numbers of connections

When you monitor Amazon CloudWatch, you might find that the `DatabaseConnections` metric spikes. This increase indicates an increased number of connections to your database. We recommend the following approach:

- Limit the number of connections that the application can open with each instance. If your application has an embedded connection pool feature, set a reasonable number of connections. Base the number on what the vCPUs in your instance can parallelize effectively.

  If your application doesn't use a connection pool feature, considering using Amazon RDS Proxy or an alternative. This approach lets your application open multiple connections with the load balancer. The balancer can then open a restricted number of connections with the database. As fewer connections are running in parallel, your DB instance performs less context switching in the kernel. Queries should progress faster, leading to fewer wait events. For more information, see Using Amazon RDS Proxy for Aurora.
- Whenever possible, take advantage of reader nodes for Aurora PostgreSQL and read replicas for RDS for PostgreSQL. When your application runs a read-only operation, send these requests to the reader-only endpoint. This technique spreads application requests across all reader nodes, reducing the I/O pressure on the writer node.
- Consider scaling up your DB instance. A higher-capacity instance class gives more memory, which gives Aurora PostgreSQL a larger shared buffer pool to hold pages. The larger size also gives the DB instance more vCPUs to handle connections. More vCPUs are particularly helpful when the operations that are generating `IO:DataFileRead` wait events are writes.

IO:XactSync

The `IO:XactSync` event occurs when the database is waiting for the Aurora storage subsystem to acknowledge the commit of a regular transaction, or the commit or rollback of a prepared transaction. A prepared transaction is part of PostgreSQL's support for a two-phase commit.

Topics

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

The event `IO:XactSync` indicates that the instance is spending time waiting for the Aurora storage subsystem to confirm that transaction data was processed.

Likely causes of increased waits

When the `IO:XactSync` event appears more than normal, possibly indicating a performance problem, typical causes include the following:

Network saturation

> Traffic between clients and the DB instance or traffic to the storage subsystem might be too heavy for the network bandwidth.

CPU pressure

> A heavy workload might be preventing the Aurora storage daemon from getting sufficient CPU time.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Monitor your resources
- Scale up the CPU
- Increase network bandwidth
- Reduce the number of commits

Monitor your resources

To determine the cause of the increased `IO:XactSync` events, check the following metrics:

- `WriteThroughput` and `CommitThroughput` – Changes in write throughput or commit throughput can show an increase in workload.

- WriteLatency and CommitLatency – Changes in write latency or commit latency can show that the storage subsystem is being asked to do more work.
- CPUUtilization – If the instance's CPU utilization is above 90 percent, the Aurora storage daemon might not be getting sufficient time on the CPU. In this case, I/O performance degrades.

For information about these metrics, see Instance-level metrics for Amazon Aurora.

Scale up the CPU

To address CPU starvation issues, consider changing to an instance type with more CPU capacity. For information about CPU capacity for a DB instance class, see Hardware specifications for DB instance classes for Aurora.

Increase network bandwidth

To determine whether the instance is reaching its network bandwidth limits, check for the following other wait events:

- IO:DataFileRead, IO:BufferRead, IO:BufferWrite, and IO:XactWrite – Queries using large amounts of I/O can generate more of these wait events.
- Client:ClientRead and Client:ClientWrite – Queries with large amounts of client communication can generate more of these wait events.

If network bandwidth is an issue, consider changing to an instance type with more network bandwidth. For information about network performance for a DB instance class, see Hardware specifications for DB instance classes for Aurora.

Reduce the number of commits

To reduce the number of commits, combine statements into transaction blocks.

IPC:DamRecordTxAck

The IPC:DamRecordTxAck event occurs when Aurora PostgreSQL in a session using database activity streams generates an activity stream event, then waits for that event to become durable.

Topics

- Relevant engine versions
- Context

- Causes
- Actions

Relevant engine versions

This wait event information is relevant for all Aurora PostgreSQL 10.7 and higher 10 versions, 11.4 and higher 11 versions, and all 12 and 13 versions.

Context

In synchronous mode, durability of activity stream events is favored over database performance. While waiting for a durable write of the event, the session blocks other database activity, causing the `IPC:DamRecordTxAck` wait event.

Causes

The most common cause for the `IPC:DamRecordTxAck` event to appear in top waits is that the Database Activity Streams (DAS) feature is a holistic audit. Higher SQL activity generates activity stream events that need to be recorded.

Actions

We recommend different actions depending on the causes of your wait event:

- Reduce the number of SQL statements or turn off database activity streams. Doing this reduces the number of events that require durable writes.
- Change to asynchronous mode. Doing this helps to reduce contention on the `IPC:DamRecordTxAck` wait event.

  However, the DAS feature can't guarantee the durability of every event in asynchronous mode.

Lock:advisory

The `Lock:advisory` event occurs when a PostgreSQL application uses a lock to coordinate activity across multiple sessions.

Topics

- Relevant engine versions
- Context

- Causes
- Actions

Relevant engine versions

This wait event information is relevant for Aurora PostgreSQL versions 9.6 and higher.

Context

PostgreSQL advisory locks are application-level, cooperative locks explicitly locked and unlocked by the user's application code. An application can use PostgreSQL advisory locks to coordinate activity across multiple sessions. Unlike regular, object- or row-level locks, the application has full control over the lifetime of the lock. For more information, see Advisory Locks in the PostgreSQL documentation.

Advisory locks can be released before a transaction ends or be held by a session across transactions. This isn't true for implicit, system-enforced locks, such as an access-exclusive lock on a table acquired by a `CREATE INDEX` statement.

For a description of the functions used to acquire (lock) and release (unlock) advisory locks, see Advisory Lock Functions in the PostgreSQL documentation.

Advisory locks are implemented on top of the regular PostgreSQL locking system and are visible in the `pg_locks` system view.

Causes

This lock type is exclusively controlled by an application explicitly using it. Advisory locks that are acquired for each row as part of a query can cause a spike in locks or a long-term buildup.

These effects happen when the query is run in a way that acquires locks on more rows than are returned by the query. The application must eventually release every lock, but if locks are acquired on rows that aren't returned, the application can't find all of the locks.

The following example is from Advisory Locks in the PostgreSQL documentation.

```
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100;
```

In this example, the `LIMIT` clause can only stop the query's output after the rows have already been internally selected and their ID values locked. This can happen suddenly when a growing data volume causes the planner to choose a different execution plan that wasn't tested during development. The buildup in this case happens because the application explicitly calls `pg_advisory_unlock` for every ID value that was locked. However, in this case it can't find the set of locks acquired on rows that weren't returned. Because the locks are acquired on the session level, they aren't released automatically at the end of the transaction.

Another possible cause for spikes in blocked lock attempts is unintended conflicts. In these conflicts, unrelated parts of the application share the same lock ID space by mistake.

Actions

Review application usage of advisory locks and detail where and when in the application flow each type of advisory lock is acquired and released.

Determine whether a session is acquiring too many locks or a long-running session isn't releasing locks early enough, leading to a slow buildup of locks. You can correct a slow buildup of session-level locks by ending the session using `pg_terminate_backend(pid)`.

A client waiting for an advisory lock appears in `pg_stat_activity` with `wait_event_type=Lock` and `wait_event=advisory`. You can obtain specific lock values by querying the `pg_locks` system view for the same `pid`, looking for `locktype=advisory` and `granted=f`.

You can then identify the blocking session by querying `pg_locks` for the same advisory lock having `granted=t`, as shown in the following example.

```
SELECT blocked_locks.pid AS blocked_pid,
         blocking_locks.pid AS blocking_pid,
         blocked_activity.usename AS blocked_user,
         blocking_activity.usename AS blocking_user,
         now() - blocked_activity.xact_start AS blocked_transaction_duration,
         now() - blocking_activity.xact_start AS blocking_transaction_duration,
         concat(blocked_activity.wait_event_type,':',blocked_activity.wait_event) AS
 blocked_wait_event,
         concat(blocking_activity.wait_event_type,':',blocking_activity.wait_event) AS
 blocking_wait_event,
         blocked_activity.state AS blocked_state,
         blocking_activity.state AS blocking_state,
         blocked_locks.locktype AS blocked_locktype,
         blocking_locks.locktype AS blocking_locktype,
         blocked_activity.query AS blocked_statement,
         blocking_activity.query AS blocking_statement
    FROM pg_catalog.pg_locks blocked_locks
    JOIN pg_catalog.pg_stat_activity blocked_activity ON blocked_activity.pid =
 blocked_locks.pid
    JOIN pg_catalog.pg_locks blocking_locks
        ON blocking_locks.locktype = blocked_locks.locktype
        AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
        AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
        AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
        AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
        AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
        AND blocking_locks.transactionid IS NOT DISTINCT FROM
 blocked_locks.transactionid
        AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
        AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
        AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
        AND blocking_locks.pid != blocked_locks.pid
    JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid =
 blocking_locks.pid
    WHERE NOT blocked_locks.GRANTED;
```

All of the advisory lock API functions have two sets of arguments, either one `bigint` argument or two `integer` arguments:

- For the API functions with one `bigint` argument, the upper 32 bits are in `pg_locks.classid` and the lower 32 bits are in `pg_locks.objid`.
- For the API functions with two `integer` arguments, the first argument is `pg_locks.classid` and the second argument is `pg_locks.objid`.

The `pg_locks.objsubid` value indicates which API form was used: 1 means one `bigint` argument; 2 means two `integer` arguments.

Lock:extend

The `Lock:extend` event occurs when a backend process is waiting to lock a relation to extend it while another process has a lock on that relation for the same purpose.

Topics

- Supported engine versions
- Context

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

The event `Lock:extend` indicates that a backend process is waiting to extend a relation that another backend process holds a lock on while it's extending that relation. Because only one process at a time can extend a relation, the system generates a `Lock:extend` wait event. `INSERT`, `COPY`, and `UPDATE` operations can generate this event.

Likely causes of increased waits

When the `Lock:extend` event appears more than normal, possibly indicating a performance problem, typical causes include the following:

Surge in concurrent inserts or updates to the same table

> There might be an increase in the number of concurrent sessions with queries that insert into or update the same table.

Insufficient network bandwidth

> The network bandwidth on the DB instance might be insufficient for the storage communication needs of the current workload. This can contribute to storage latency that causes an increase in `Lock:extend` events.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Reduce concurrent inserts and updates to the same relation

First, determine whether there's an increase in `tup_inserted` and `tup_updated` metrics and an accompanying increase in this wait event. If so, check which relations are in high contention for insert and update operations. To determine this, query the `pg_stat_all_tables` view for the values in `n_tup_ins` and `n_tup_upd` fields. For information about the `pg_stat_all_tables` view, see [pg_stat_all_tables](#) in the PostgreSQL documentation.

To get more information about blocking and blocked queries, query `pg_stat_activity` as in the following example:

```
SELECT
    blocked.pid,
    blocked.usename,
    blocked.query,
    blocking.pid AS blocking_id,
    blocking.query AS blocking_query,
    blocking.wait_event AS blocking_wait_event,
    blocking.wait_event_type AS blocking_wait_event_type
FROM pg_stat_activity AS blocked
JOIN pg_stat_activity AS blocking ON blocking.pid = ANY(pg_blocking_pids(blocked.pid))
where
blocked.wait_event = 'extend'
and blocked.wait_event_type = 'Lock';

   pid  | usename  |            query             | blocking_id |
             blocking_query                         | blocking_wait_event |
 blocking_wait_event_type
  ------+----------+------------------------------+-------------
+-------------------------------------------------------------------
+---------------------+-------------------------
   7143 |  myuser  | insert into tab1 values (1); |        4600 | INSERT INTO tab1 (a)
 SELECT s FROM generate_series(1,1000000) s; | DataFileExtend      | IO
```

After you identify relations that contribute to increase `Lock:extend` events, use the following techniques to reduce the contention:

- Find out whether you can use partitioning to reduce contention for the same table. Separating inserted or updated tuples into different partitions can reduce contention. For information about partitioning, see [Managing PostgreSQL partitions with the pg_partman extension](#).

- If the wait event is mainly due to update activity, consider reducing the relation's fillfactor value. This can reduce requests for new blocks during the update. The fillfactor is a storage parameter for a table that determines the maximum amount of space for packing a table page. It's expressed as a percentage of the total space for a page. For more information about the fillfactor parameter, see CREATE TABLE in the PostgreSQL documentation.

> ⚠ **Important**
>
> We highly recommend that you test your system if you change the fillfactor because changing this value can negatively impact performance, depending on your workload.

Increase network bandwidth

To see whether there's an increase in write latency, check the `WriteLatency` metric in CloudWatch. If there is, use the `WriteThroughput` and `ReadThroughput` Amazon CloudWatch metrics to monitor the storage related traffic on the DB cluster. These metrics can help you to determine if network bandwidth is sufficient for the storage activity of your workload.

If your network bandwidth isn't enough, increase it. If your DB instance is reaching the network bandwidth limits, the only way to increase the bandwidth is to increase your DB instance size.

For more information about CloudWatch metrics, see Amazon CloudWatch metrics for Amazon Aurora. For information about network performance for each DB instance class, see Hardware specifications for DB instance classes for Aurora.

Lock:Relation

The `Lock:Relation` event occurs when a query is waiting to acquire a lock on a table or view (relation) that's currently locked by another transaction.

Topics

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

Most PostgreSQL commands implicitly use locks to control concurrent access to data in tables. You can also use these locks explicitly in your application code with the `LOCK` command. Many lock modes aren't compatible with each other, and they can block transactions when they're trying to access the same object. When this happens, Aurora PostgreSQL generates a `Lock:Relation` event. Some common examples are the following:

- Exclusive locks such as `ACCESS EXCLUSIVE` can block all concurrent access. Data definition language (DDL) operations such as `DROP TABLE`, `TRUNCATE`, `VACUUM FULL`, and `CLUSTER` acquire `ACCESS EXCLUSIVE` locks implicitly. `ACCESS EXCLUSIVE` is also the default lock mode for `LOCK TABLE` statements that don't specify a mode explicitly.
- Using `CREATE INDEX (without CONCURRENT)` on a table conflicts with data manipulation language (DML) statements `UPDATE`, `DELETE`, and `INSERT`, which acquire `ROW EXCLUSIVE` locks.

For more information about table-level locks and conflicting lock modes, see Explicit Locking in the PostgreSQL documentation.

Blocking queries and transactions typically unblock in one of the following ways:

- Blocking query – The application can cancel the query or the user can end the process. The engine can also force the query to end because of a session's statement-timeout or a deadlock detection mechanism.
- Blocking transaction – A transaction stops blocking when it runs a `ROLLBACK` or `COMMIT` statement. Rollbacks also happen automatically when sessions are disconnected by a client or by network issues, or are ended. Sessions can be ended when the database engine is shut down, when the system is out of memory, and so forth.

Likely causes of increased waits

When the `Lock:Relation` event occurs more frequently than normal, it can indicate a performance issue. Typical causes include the following:

Increased concurrent sessions with conflicting table locks

There might be an increase in the number of concurrent sessions with queries that lock the same table with conflicting locking modes.

Maintenance operations

Health maintenance operations such as `VACUUM` and `ANALYZE` can significantly increase the number of conflicting locks. `VACUUM FULL` acquires an `ACCESS EXCLUSIVE` lock, and `ANALYSE` acquires a `SHARE UPDATE EXCLUSIVE` lock. Both types of locks can cause a `Lock:Relation` wait event. Application data maintenance operations such as refreshing a materialized view can also increase blocked queries and transactions.

Locks on reader instances

There might be a conflict between the relation locks held by the writer and readers. Currently, only `ACCESS EXCLUSIVE` relation locks are replicated to reader instances. However, the `ACCESS EXCLUSIVE` relation lock will conflict with any `ACCESS SHARE` relation locks held by the reader. This can cause an increase in lock relation wait events on the reader.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Reduce the impact of blocking SQL statements

To reduce the impact of blocking SQL statements, modify your application code where possible. Following are two common techniques for reducing blocks:

- Use the `NOWAIT` option – Some SQL commands, such as `SELECT` and `LOCK` statements, support this option. The `NOWAIT` directive cancels the lock-requesting query if the lock can't be acquired immediately. This technique can help prevent a blocking session from causing a pile-up of blocked sessions behind it.

For example: Assume that transaction A is waiting on a lock held by transaction B. Now, if B requests a lock on a table that’s locked by transaction C, transaction A might be blocked until transaction C completes. But if transaction B uses a NOWAIT when it requests the lock on C, it can fail fast and ensure that transaction A doesn't have to wait indefinitely.

- Use SET lock_timeout – Set a lock_timeout value to limit the time a SQL statement waits to acquire a lock on a relation. If the lock isn't acquired within the timeout specified, the transaction requesting the lock is cancelled. Set this value at the session level.

Minimize the effect of maintenance operations

Maintenance operations such as VACUUM and ANALYZE are important. We recommend that you don't turn them off because you find Lock:Relation wait events related to these maintenance operations. The following approaches can minimize the effect of these operations:

- Run maintenance operations manually during off-peak hours.
- To reduce Lock:Relation waits caused by autovacuum tasks, perform any needed autovacuum tuning. For information about tuning autovacuum, see Working with PostgreSQL autovacuum on Amazon RDS in the *Amazon RDS User Guide*.

Check for reader locks

You can see how concurrent sessions on a writer and readers might be holding locks that block each other. One way to do this is by running queries that return the lock type and relation. In the table you can find a sequence of queries to two such concurrent sessions, a writer session (left-hand column) and a reader session (right-hand column).

The replay process waits for the duration of max_standby_streaming_delay before cancelling the reader query. As shown in the example, the lock timeout of 100ms is well below the default max_standby_streaming_delay of 30 seconds. The lock times out before it's an issue.

Writer session

```
export WRITER=aurorapg1.1234567891
0.us-west-1.rds.amazonaws.com

psql -h $WRITER
psql (15devel, server 10.14)
```

Reader session

```
export READER=aurorapg2.1234567891
0.us-west-1.rds.amazonaws.com

psql -h $READER
psql (15devel, server 10.14)
```

Writer session	Reader session
`Type "help" for help.`	`Type "help" for help.`

The writer session creates table `t1` on the writer instance. The `ACCESS EXCLUSIVE` lock is acquired on the writer immediately, assuming that there are no conflicting queries on the writer.

```
postgres=> CREATE TABLE t1(b
 integer);
CREATE TABLE
```

The reader session sets a lock timeout interval of 100 milliseconds.

```
postgres=> SET lock_timeout=100;
SET
```

The reader session tries to read data from table `t1` on the reader instance.

```
postgres=> SELECT * FROM t1;
 b
---
(0 rows)
```

The writer session drops `t1`.

```
postgres=> BEGIN;
BEGIN
postgres=> DROP TABLE t1;
DROP TABLE
postgres=>
```

The query times out and is canceled on the reader.

```
postgres=> SELECT * FROM t1;
ERROR:  canceling statement due to
 lock timeout
LINE 1: SELECT * FROM t1;
                      ^
```

Writer session	Reader session

The reader session queries pg_locks and pg_stat_activity to determine the cause of the error. The result indicates that the aurora wal replay process is holding an ACCESS EXCLUSIVE lock on table t1.

```
postgres=> SELECT locktype, relation,
 mode, backend_type
postgres-> FROM pg_locks l, pg_stat_a
ctivity t1
postgres-> WHERE l.pid=t1.pid AND
 relation = 't1'::regclass::oid;
 locktype | relation |        mode
    |   backend_type
----------+----------+----------
-----------+------------------
 relation | 68628525 | AccessExc
lusiveLock | aurora wal replay
(1 row)
```

Lock:transactionid

The `Lock:transactionid` event occurs when a transaction is waiting for a row-level lock.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

The event `Lock:transactionid` occurs when a transaction is trying to acquire a row-level lock that has already been granted to a transaction that is running at the same time. The session that shows the `Lock:transactionid` wait event is blocked because of this lock. After the blocking transaction ends in either a `COMMIT` or `ROLLBACK` statement, the blocked transaction can proceed.

The multiversion concurrency control semantics of Aurora PostgreSQL guarantee that readers don't block writers and writers don't block readers. For row-level conflicts to occur, blocking and blocked transactions must issue conflicting statements of the following types:

- `UPDATE`
- `SELECT … FOR UPDATE`
- `SELECT … FOR KEY SHARE`

The statement `SELECT … FOR KEY SHARE` is a special case. The database uses the clause `FOR KEY SHARE` to optimize the performance of referential integrity. A row-level lock on a row can block `INSERT`, `UPDATE`, and `DELETE` commands on other tables that reference the row.

Likely causes of increased waits

When this event appears more than normal, the cause is typically `UPDATE`, `SELECT … FOR UPDATE`, or `SELECT … FOR KEY SHARE` statements combined with the following conditions.

Topics

- High concurrency
- Idle in transaction
- Long-running transactions

High concurrency

Aurora PostgreSQL can use granular row-level locking semantics. The probability of row-level conflicts increases when the following conditions are met:

- A highly concurrent workload contends for the same rows.
- Concurrency increases.

Idle in transaction

Sometimes the `pg_stat_activity.state` column shows the value `idle in transaction`. This value appears for sessions that have started a transaction, but haven't yet issued a `COMMIT` or ROLLBACK. If the pg_stat_activity.state value isn't active, the query shown in pg_stat_activity is the most recent one to finish running. The blocking session isn't actively processing a query because an open transaction is holding a lock.

If an idle transaction acquired a row-level lock, it might be preventing other sessions from acquiring it. This condition leads to frequent occurrence of the wait event Lock:transactionid. To diagnose the issue, examine the output from pg_stat_activity and pg_locks.

Long-running transactions

Transactions that run for a long time get locks for a long time. These long-held locks can block other transactions from running.

Actions

Row-locking is a conflict among UPDATE, SELECT … FOR UPDATE, or SELECT … FOR KEY SHARE statements. Before attempting a solution, find out when these statements are running on the same row. Use this information to choose a strategy described in the following sections.

Topics

Respond to high concurrency

If concurrency is the issue, try one of the following techniques:

- Lower the concurrency in the application. For example, decrease the number of active sessions.
- Implement a connection pool. To learn how to pool connections with RDS Proxy, see Using Amazon RDS Proxy for Aurora.
- Design the application or data model to avoid contending UPDATE and SELECT … FOR UPDATE statements. You can also decrease the number of foreign keys accessed by SELECT … FOR KEY SHARE statements.

Respond to idle transactions

If pg_stat_activity.state shows idle in transaction, use the following strategies:

- Turn on autocommit wherever possible. This approach prevents transactions from blocking other transactions while waiting for a `COMMIT` or `ROLLBACK`.
- Search for code paths that are missing `COMMIT`, `ROLLBACK`, or `END`.
- Make sure that the exception handling logic in your application always has a path to a valid `end of transaction`.
- Make sure that your application processes query results after ending the transaction with `COMMIT` or `ROLLBACK`.

Respond to long-running transactions

If long-running transactions are causing the frequent occurrence of `Lock:transactionid`, try the following strategies:

- Keep row locks out of long-running transactions.
- Limit the length of queries by implementing autocommit whenever possible.

Lock:tuple

The `Lock:tuple` event occurs when a backend process is waiting to acquire a lock on a tuple.

Topics

- Supported engine versions
- Context
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

The event `Lock:tuple` indicates that a backend is waiting to acquire a lock on a tuple while another backend holds a conflicting lock on the same tuple. The following table illustrates a scenario in which sessions generate the `Lock:tuple` event.

Time	Session 1	Session 2	Session 3
t1	Starts a transaction.		
t2	Updates row 1.		
t3		Updates row 1. The session acquires an exclusive lock on the tuple and then waits for session 1 to release the lock by committing or rolling back.	
t4			Updates row 1. The session waits for session 2 to release the exclusive lock on the tuple.

Or you can simulate this wait event by using the benchmarking tool `pgbench`. Configure a high number of concurrent sessions to update the same row in a table with a custom SQL file.

To learn more about conflicting lock modes, see Explicit Locking in the PostgreSQL documentation. To learn more about `pgbench`, see pgbench in the PostgreSQL documentation.

Likely causes of increased waits

When this event appears more than normal, possibly indicating a performance problem, typical causes include the following:

- A high number of concurrent sessions are trying to acquire a conflicting lock for the same tuple by running `UPDATE` or `DELETE` statements.
- Highly concurrent sessions are running a `SELECT` statement using the `FOR UPDATE` or `FOR NO KEY UPDATE` lock modes.
- Various factors drive application or connection pools to open more sessions to execute the same operations. As new sessions are trying to modify the same rows, DB load can spike, and `Lock:tuple` can appear.

For more information, see Row-Level Locks in the PostgreSQL documentation.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Investigate your application logic
- Find the blocker session
- Reduce concurrency when it is high
- Troubleshoot bottlenecks

Investigate your application logic

Find out whether a blocker session has been in the `idle in transaction` state for long time. If so, consider ending the blocker session as a short-term solution. You can use the `pg_terminate_backend` function. For more information about this function, see Server Signaling Functions in the PostgreSQL documentation.

For a long-term solution, do the following:

- Adjust the application logic.
- Use the `idle_in_transaction_session_timeout` parameter. This parameter ends any session with an open transaction that has been idle for longer than the specified amount of time. For more information, see Client Connection Defaults in the PostgreSQL documentation.
- Use autocommit as much as possible. For more information, see SET AUTOCOMMIT in the PostgreSQL documentation.

Find the blocker session

While the `Lock:tuple` wait event is occurring, identify the blocker and blocked session by finding out which locks depend on one another. For more information, see Lock dependency information in the PostgreSQL wiki. To analyze past `Lock:tuple` events, use the Aurora function `aurora_stat_backend_waits`.

The following example queries all sessions, filtering on `tuple` and ordering by `wait_time`.

```
--AURORA_STAT_BACKEND_WAITS
      SELECT a.pid,
             a.usename,
             a.app_name,
             a.current_query,
             a.current_wait_type,
             a.current_wait_event,
             a.current_state,
             wt.type_name AS wait_type,
             we.event_name AS wait_event,
             a.waits,
             a.wait_time
        FROM (SELECT pid,
                     usename,
                     left(application_name,16) AS app_name,
                     coalesce(wait_event_type,'CPU') AS current_wait_type,
                     coalesce(wait_event,'CPU') AS current_wait_event,
                     state AS current_state,
                     left(query,80) as current_query,
                     (aurora_stat_backend_waits(pid)).*
                FROM pg_stat_activity
               WHERE pid <> pg_backend_pid()
                 AND usename<>'rdsadmin') a
NATURAL JOIN aurora_stat_wait_type() wt
NATURAL JOIN aurora_stat_wait_event() we
WHERE we.event_name = 'tuple'
    ORDER BY a.wait_time;

  pid  | usename | app_name |                 current_query                  |
 current_wait_type | current_wait_event | current_state | wait_type | wait_event |
 waits | wait_time
-------+---------+----------+------------------------------------------------
+-------------------+--------------------+---------------+-----------+------------
+-------+-----------
 32136 | sys     | psql     | /*session3*/ update tab set col=1 where col=1; | Lock
          | tuple              | active        | Lock      | tuple      |     1 |
 1000018
 11999 | sys     | psql     | /*session4*/ update tab set col=1 where col=1; | Lock
          | tuple              | active        | Lock      | tuple      |     1 |
 1000024
```

Reduce concurrency when it is high

The `Lock:tuple` event might occur constantly, especially in a busy workload time. In this situation, consider reducing the high concurrency for very busy rows. Often, just a few rows control a queue or the Boolean logic, which makes these rows very busy.

You can reduce concurrency by using different approaches based in the business requirement, application logic, and workload type. For example, you can do the following:

- Redesign your table and data logic to reduce high concurrency.
- Change the application logic to reduce high concurrency at the row level.
- Leverage and redesign queries with row-level locks.
- Use the `NOWAIT` clause with retry operations.
- Consider using optimistic and hybrid-locking logic concurrency control.
- Consider changing the database isolation level.

Troubleshoot bottlenecks

The `Lock:tuple` can occur with bottlenecks such as CPU starvation or maximum usage of Amazon EBS bandwidth. To reduce bottlenecks, consider the following approaches:

- Scale up your instance class type.
- Optimize resource-intensive queries.
- Change the application logic.
- Archive data that is rarely accessed.

LWLock:buffer_content (BufferContent)

The `LWLock:buffer_content` event occurs when a session is waiting to read or write a data page in memory while another session has that page locked for writing. In Aurora PostgreSQL 13 and higher, this wait event is called `BufferContent`.

Topics

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

To read or manipulate data, PostgreSQL accesses it through shared memory buffers. To read from the buffer, a process gets a lightweight lock (LWLock) on the buffer content in shared mode. To write to the buffer, it gets that lock in exclusive mode. Shared locks allow other processes to concurrently acquire shared locks on that content. Exclusive locks prevent other processes from getting any type of lock on it.

The `LWLock:buffer_content` (`BufferContent`) event indicates that multiple processes are attempting to get a lock on contents of a specific buffer.

Likely causes of increased waits

When the `LWLock:buffer_content` (`BufferContent`) event appears more than normal, possibly indicating a performance problem, typical causes include the following:

Increased concurrent updates to the same data

> There might be an increase in the number of concurrent sessions with queries that update the same buffer content. This contention can be more pronounced on tables with a lot of indexes.

Workload data is not in memory

> When data that the active workload is processing is not in memory, these wait events can increase. This effect is because processes holding locks can keep them longer while they perform disk I/O operations.

Excessive use of foreign key constraints

> Foreign key constraints can increase the amount of time a process holds onto a buffer content lock. This effect is because read operations require a shared buffer content lock on the referenced key while that key is being updated.

Actions

We recommend different actions depending on the causes of your wait event. You might identify `LWLock:buffer_content` (`BufferContent`) events by using Amazon RDS Performance Insights or by querying the view `pg_stat_activity`.

Topics

- Improve in-memory efficiency
- Reduce usage of foreign key constraints
- Remove unused indexes

Improve in-memory efficiency

To increase the chance that active workload data is in memory, partition tables or scale up your instance class. For information about DB instance classes, see Aurora DB instance classes.

Reduce usage of foreign key constraints

Investigate workloads experiencing high numbers of `LWLock:buffer_content` (`BufferContent`) wait events for usage of foreign key constraints. Remove unnecessary foreign key constraints.

Remove unused indexes

For workloads experiencing high numbers of `LWLock:buffer_content` (`BufferContent`) wait events, identify unused indexes and remove them.

LWLock:buffer_mapping

This event occurs when a session is waiting to associate a data block with a buffer in the shared buffer pool.

> **Note**
>
> This event appears as `LWLock:buffer_mapping` in Aurora PostgreSQL version 12 and lower, and `LWLock:BufferMapping` in version 13 and higher.

Topics

- Supported engine versions
- Context
- Causes
- Actions

Supported engine versions

This wait event information is relevant for Aurora PostgreSQL version 9.6 and higher.

Context

The *shared buffer pool* is an Aurora PostgreSQL memory area that holds all pages that are or were being used by processes. When a process needs a page, it reads the page into the shared buffer pool. The `shared_buffers` parameter sets the shared buffer size and reserves a memory area to store the table and index pages. If you change this parameter, make sure to restart the database. For more information, see Shared buffers.

The `LWLock:buffer_mapping` wait event occurs in the following scenarios:

- A process searches the buffer table for a page and acquires a shared buffer mapping lock.
- A process loads a page into the buffer pool and acquires an exclusive buffer mapping lock.
- A process removes a page from the pool and acquires an exclusive buffer mapping lock.

Causes

When this event appears more than normal, possibly indicating a performance problem, the database is paging in and out of the shared buffer pool. Typical causes include the following:

- Large queries
- Bloated indexes and tables
- Full table scans
- A shared pool size that is smaller than the working set

Actions

We recommend different actions depending on the causes of your wait event.

Topics

- Monitor buffer-related metrics
- Assess your indexing strategy
- Reduce the number of buffers that must be allocated quickly

Monitor buffer-related metrics

When `LWLock:buffer_mapping` waits spike, investigate the buffer hit ratio. You can use these metrics to get a better understanding of what is happening in the buffer cache. Examine the following metrics:

`BufferCacheHitRatio`

> This Amazon CloudWatch metric measures the percentage of requests that are served by the buffer cache of a DB instance in your DB cluster. You might see this metric decrease in the lead-up to the `LWLock:buffer_mapping` wait event.

`blks_hit`

> This Performance Insights counter metric indicates the number of blocks that were retrieved from the shared buffer pool. After the `LWLock:buffer_mapping` wait event appears, you might observe a spike in `blks_hit`.

`blks_read`

> This Performance Insights counter metric indicates the number of blocks that required I/O to be read into the shared buffer pool. You might observe a spike in `blks_read` in the lead-up to the `LWLock:buffer_mapping` wait event.

Assess your indexing strategy

To confirm that your indexing strategy is not degrading performance, check the following:

Index bloat

> Ensure that index and table bloat aren't leading to unnecessary pages being read into the shared buffer. If your tables contain unused rows, consider archiving the data and removing the rows from the tables. You can then rebuild the indexes for the resized tables.

Indexes for frequently used queries

> To determine whether you have the optimal indexes, monitor DB engine metrics in Performance Insights. The `tup_returned` metric shows the number of rows read. The `tup_fetched` metric shows the number of rows returned to the client. If `tup_returned` is significantly larger than `tup_fetched`, the data might not be properly indexed. Also, your table statistics might not be current.

Reduce the number of buffers that must be allocated quickly

To reduce the `LWLock:buffer_mapping` wait events, try to reduce the number of buffers that must be allocated quickly. One strategy is to perform smaller batch operations. You might be able to achieve smaller batches by partitioning your tables.

LWLock:BufferIO (IPC:BufferIO)

The `LWLock:BufferIO` event occurs when Aurora PostgreSQL or RDS for PostgreSQL is waiting for other processes to finish their input/output (I/O) operations when concurrently trying to access a page. Its purpose is for the same page to be read into the shared buffer.

Topics

- Relevant engine versions
- Context
- Causes
- Actions

Relevant engine versions

This wait event information is relevant for all Aurora PostgreSQL versions. For Aurora PostgreSQL 12 and earlier versions this wait event is named as lwlock:buffer_io whereas in Aurora PostgreSQL 13 version it is named as lwlock:bufferio. From Aurora PostgreSQL 14 version BufferIO wait event moved from `LWLock` to `IPC` wait event type (IPC:BufferIO).

Context

Each shared buffer has an I/O lock that is associated with the `LWLock:BufferIO` wait event, each time a block (or a page) has to be retrieved outside the shared buffer pool.

This lock is used to handle multiple sessions that all require access to the same block. This block has to be read from outside the shared buffer pool, defined by the `shared_buffers` parameter.

As soon as the page is read inside the shared buffer pool, the `LWLock:BufferIO` lock is released.

Note

The `LWLock:BufferIO` wait event precedes the IO:DataFileRead wait event. The `IO:DataFileRead` wait event occurs while data is being read from storage.

For more information on lightweight locks, see Locking Overview.

Causes

Common causes for the `LWLock:BufferIO` event to appear in top waits include the following:

- Multiple backends or connections trying to access the same page that's also pending an I/O operation
- The ratio between the size of the shared buffer pool (defined by the `shared_buffers` parameter) and the number of buffers needed by the current workload
- The size of the shared buffer pool not being well balanced with the number of pages being evicted by other operations
- Large or bloated indexes that require the engine to read more pages than necessary into the shared buffer pool
- Lack of indexes that forces the DB engine to read more pages from the tables than necessary
- Sudden spikes for database connections trying to perform operations on the same page

Actions

We recommend different actions depending on the causes of your wait event:

- Observe Amazon CloudWatch metrics for correlation between sharp decreases in the `BufferCacheHitRatio` and `LWLock:BufferIO` wait events. This effect can mean that you have a small shared buffers setting. You might need to increase it or scale up your DB instance class. You can split your workload into more reader nodes.
- Tune `max_wal_size` and `checkpoint_timeout` based on your workload peak time if you see `LWLock:BufferIO` coinciding with `BufferCacheHitRatio` metric dips. Then identify which query might be causing it.
- Verify whether you have unused indexes, then remove them.

- Use partitioned tables (which also have partitioned indexes). Doing this helps to keep index reordering low and reduces its impact.
- Avoid indexing columns unnecessarily.
- Prevent sudden database connection spikes by using a connection pool.
- Restrict the maximum number of connections to the database as a best practice.

LWLock:lock_manager

This event occurs when the Aurora PostgreSQL engine maintains the shared lock's memory area to allocate, check, and deallocate a lock when a fast path lock isn't possible.

Topics

Supported engine versions

This wait event information is relevant for Aurora PostgreSQL version 9.6 and higher.

Context

When you issue a SQL statement, Aurora PostgreSQL records locks to protect the structure, data, and integrity of your database during concurrent operations. The engine can achieve this goal using a fast path lock or a path lock that isn't fast. A path lock that isn't fast is more expensive and creates more overhead than a fast path lock.

Fast path locking

To reduce the overhead of locks that are taken and released frequently, but that rarely conflict, backend processes can use fast path locking. The database uses this mechanism for locks that meet the following criteria:

- They use the DEFAULT lock method.
- They represent a lock on a database relation rather than a shared relation.

- They are weak locks that are unlikely to conflict.
- The engine can quickly verify that no conflicting locks can possibly exist.

The engine can't use fast path locking when either of the following conditions is true:

- The lock doesn't meet the preceding criteria.
- No more slots are available for the backend process.

For more information about fast path locking, see fast path in the PostgreSQL lock manager README and pg-locks in the PostgreSQL documentation.

Example of a scaling problem for the lock manager

In this example, a table named `purchases` stores five years of data, partitioned by day. Each partition has two indexes. The following sequence of events occurs:

1. You query many days worth of data, which requires the database to read many partitions.
2. The database creates a lock entry for each partition. If partition indexes are part of the optimizer access path, the database creates a lock entry for them, too.
3. When the number of requested locks entries for the same backend process is higher than 16, which is the value of `FP_LOCK_SLOTS_PER_BACKEND`, the lock manager uses the non–fast path lock method.

Modern applications might have hundreds of sessions. If concurrent sessions are querying the parent without proper partition pruning, the database might create hundreds or even thousands of non–fast path locks. Typically, when this concurrency is higher than the number of vCPUs, the `LWLock:lock_manager` wait event appears.

Note

The `LWLock:lock_manager` wait event isn't related to the number of partitions or indexes in a database schema. Instead, it's related to the number of non–fast path locks that the database must control.

Likely causes of increased waits

When the `LWLock:lock_manager` wait event occurs more than normal, possibly indicating a performance problem, the most likely causes of sudden spikes are as follows:

- Concurrent active sessions are running queries that don't use fast path locks. These sessions also exceed the maximum vCPU.
- A large number of concurrent active sessions are accessing a heavily partitioned table. Each partition has multiple indexes.
- The database is experiencing a connection storm. By default, some applications and connection pool software create more connections when the database is slow. This practice makes the problem worse. Tune your connection pool software so that connection storms don't occur.
- A large number of sessions query a parent table without pruning partitions.
- A data definition language (DDL), data manipulation language (DML), or a maintenance command exclusively locks either a busy relation or tuples that are frequently accessed or modified.

Actions

We recommend different actions depending on the causes of your wait event.

Topics

Use partition pruning

Partition pruning is a query optimization strategy that excludes unneeded partitions from table scans, thereby improving performance. Partition pruning is turned on by default. If it is turned off, turn it on as follows.

```
SET enable_partition_pruning = on;
```

Queries can take advantage of partition pruning when their WHERE clause contains the column used for the partitioning. For more information, see Partition Pruning in the PostgreSQL documentation.

Remove unnecessary indexes

Your database might contain unused or rarely used indexes. If so, consider deleting them. Do either of the following:

- Learn how to find unnecessary indexes by reading Unused Indexes in the PostgreSQL wiki.
- Run PG Collector. This SQL script gathers database information and presents it in a consolidated HTML report. Check the "Unused indexes" section. For more information, see pg-collector in the AWS Labs GitHub repository.

Tune your queries for fast path locking

To find out whether your queries use fast path locking, query the `fastpath` column in the `pg_locks` table. If your queries aren't using fast path locking, try to reduce number of relations per query to fewer than 16.

Tune for other wait events

If `LWLock:lock_manager` is first or second in the list of top waits, check whether the following wait events also appear in the list:

- `Lock:Relation`
- `Lock:transactionid`
- `Lock:tuple`

If the preceding events appear high in the list, consider tuning these wait events first. These events can be a driver for `LWLock:lock_manager`.

Reduce hardware bottlenecks

You might have a hardware bottleneck, such as CPU starvation or maximum usage of your Amazon EBS bandwidth. In these cases, consider reducing the hardware bottlenecks. Consider the following actions:

- Scale up your instance class.
- Optimize queries that consume large amounts of CPU and memory.
- Change your application logic.
- Archive your data.

For more information about CPU, memory, and EBS network bandwidth, see Amazon RDS Instance Types.

Use a connection pooler

If your total number of active connections exceeds the maximum vCPU, more OS processes require CPU than your instance type can support. In this case, consider using or tuning a connection pool. For more information about the vCPUs for your instance type, see Amazon RDS Instance Types.

For more information about connection pooling, see the following resources:

- Using Amazon RDS Proxy for Aurora
- pgbouncer
- Connection Pools and Data Sources in the *PostgreSQL Documentation*

Upgrade your Aurora PostgreSQL version

If your current version of Aurora PostgreSQL is lower than 12, upgrade to version 12 or higher. PostgreSQL versions 12 and 13 have an improved partition mechanism. For more information about version 12, see PostgreSQL 12.0 Release Notes. For more information about upgrading Aurora PostgreSQL, see Amazon Aurora PostgreSQL updates.

LWLock:MultiXact

The `LWLock:MultiXactMemberBuffer`, `LWLock:MultiXactOffsetBuffer`, `LWLock:MultiXactMemberSLRU`, and `LWLock:MultiXactOffsetSLRU` wait events indicate that a session is waiting to retrieve a list of transactions that modifies the same row in a given table.

- `LWLock:MultiXactMemberBuffer` – A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a multixact member.
- `LWLock:MultiXactMemberSLRU` – A process is waiting to access the simple least-recently used (SLRU) cache for a multixact member.

- `LWLock:MultiXactOffsetBuffer` – A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a multixact offset.
- `LWLock:MultiXactOffsetSLRU` – A process is waiting to access the simple least-recently used (SLRU) cache for a multixact offset.

Topics

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Context

A *multixact* is a data structure that stores a list of transaction IDs (XIDs) that modify the same table row. When a single transaction references a row in a table, the transaction ID is stored in the table header row. When multiple transactions reference the same row in a table, the list of transaction IDs is stored in the multixact data structure. The multixact wait events indicate that a session is retrieving from the data structure the list of transactions that refer to a given row in a table.

Likely causes of increased waits

Three common causes of multixact use are as follows:

- **Sub-transactions from explicit savepoints** – Explicitly creating a savepoint in your transactions spawns new transactions for the same row. For example, using `SELECT FOR UPDATE`, then `SAVEPOINT`, and then `UPDATE`.

  Some drivers, object-relational mappers (ORMs), and abstraction layers have configuration options for automatically wrapping all operations with savepoints. This can generate many multixact wait events in some workloads. The PostgreSQL JDBC Driver's `autosave` option is an example of this. For more information, see pgJDBC in the PostgreSQL JDBC documentation. Another example is the PostgreSQL ODBC driver and its `protocol` option. For more information, see psqlODBC Configuration Options in the PostgreSQL ODBC driver documentation.

- **Sub-transactions from PL/pgSQL EXCEPTION clauses** – Each `EXCEPTION` clause that you write in your PL/pgSQL functions or procedures creates a `SAVEPOINT` internally.
- **Foreign keys** – Multiple transactions acquire a shared lock on the parent record.

When a given row is included in a multiple transaction operation, processing the row requires retrieving transaction IDs from the `multixact` listings. If lookups can't get the multixact from the memory cache, the data structure must be read from the Aurora storage layer. This I/O from storage means that SQL queries can take longer. Memory cache misses can start occurring with heavy usage due to a large number of multiple transactions. All these factors contribute to an increase in this wait event.

Actions

We recommend different actions depending on the causes of your wait event. Some of these actions can help in immediate reduction of the wait events. But, others might require investigation and correction to scale your workload.

Topics

Perform vacuum freeze on tables with this wait event

If this wait event spikes suddenly and affects your production environment, you can use any of the following temporary methods to reduce its count.

- Use *VACUUM FREEZE* on the affected table or table partition to resolve the issue immediately. For more information, see VACUUM.
- Use the VACUUM (FREEZE, INDEX_CLEANUP FALSE) clause to perform a quick vacuum by skipping the indexes. For more information, see Vacuuming a table as quickly as possible .

Increase autovacuum frequency on tables with this wait event

After scanning all tables in all databases, VACUUM will eventually remove multixacts, and their oldest multixact values are advanced. For more information, see Multixacts and Wraparound. To keep the LWLock:MultiXact wait events to its minimum, you must run the VACUUM as often as necessary. To do so, ensure that the VACUUM in your Aurora PostgreSQL DB cluster is configured optimally.

If using VACUUM FREEZE on the affected table or table partition resolves the wait event issue, we recommend using a scheduler, such as `pg_cron`, to perform the VACUUM instead of adjusting autovacuum at the instance level.

For the autovacuum to happen more frequently, you can reduce the value of the storage parameter `autovacuum_multixact_freeze_max_age` in the affected table. For more information, see autovacuum_multixact_freeze_max_age.

Increase memory parameters

You can set the following parameters at the cluster level so that all instances in your cluster remain consistent. This helps in reducing the wait events in your workload. We recommend you to not set these values so high that you run out of memory.

- `multixact_offsets_cache_size` to 128
- `multixact_members_cache_size` to 256

You must reboot the instance for the parameter change to take affect. With these parameters, you can use more of the instance RAM to store the multixact structure in memory before spilling to disk.

Reduce long-running transactions

Long-running transaction causes the vacuum to retain its information until the transaction is committed or until the read-only transaction is closed. We recommend that you proactively monitor and manage long-running transactions. For more information, see Database has long running idle in transaction connection. Try to modify your application to avoid or minimize your use of long-running transactions.

Long term actions

Examine your workload to discover the cause for the multixact spillover. You must fix the issue in order to scale your workload and reduce the wait event.

- You must analyze the DDL (data definition language) used to create your tables. Make sure that the table structures and indexes are well designed.
- When the affected tables have foreign keys, determine whether they are needed or if there is another way to enforce referential integrity.
- When a table has large unused indexes, it can cause autovacuum to not fit your workload and might block it from running. To avoid this, check for unused indexes and remove them completely. For more information, see Managing autovacuum with large indexes.
- Reduce the use of savepoints in your transactions.

Timeout:PgSleep

The `Timeout:PgSleep` event occurs when a server process has called the `pg_sleep` function and is waiting for the sleep timeout to expire.

Topics

- Supported engine versions
- Likely causes of increased waits
- Actions

Supported engine versions

This wait event information is supported for all versions of Aurora PostgreSQL.

Likely causes of increased waits

This wait event occurs when an application, stored function, or user issues a SQL statement that calls one of the following functions:

- `pg_sleep`
- `pg_sleep_for`
- `pg_sleep_until`

The preceding functions delay execution until the specified number of seconds have elapsed. For example, `SELECT pg_sleep(1)` pauses for 1 second. For more information, see Delaying Execution in the PostgreSQL documentation.

Actions

Identify the statement that was running the `pg_sleep` function. Determine if the use of the function is appropriate.

Tuning Aurora PostgreSQL with Amazon DevOps Guru proactive insights

DevOps Guru proactive insights detects conditions on your Aurora PostgreSQL DB clusters that can cause problems, and lets you know about them before they occur. DevOps Guru can do the following:

- Prevent many common database issues by cross-checking your database configuration against common recommended settings.
- Alert you to critical issues in your fleet that, if left unchecked, can lead to larger problems later.
- Alert you to newly discovered problems.

Every proactive insight contains an analysis of the cause of the problem and recommendations for corrective actions.

Topics

Database has long running idle in transaction connection

A connection to the database has been in the `idle in transaction` state for more than 1800 seconds.

Topics

- Likely causes for this issue
- Actions
- Relevant metrics

Supported engine versions

This insight information is supported for all versions of Aurora PostgreSQL.

Context

A transaction in the `idle in transaction` state can hold locks that block other queries. It can also prevent `VACUUM` (including autovacuum) from cleaning up dead rows, leading to index or table bloat or transaction ID wraparound.

Likely causes for this issue

A transaction initiated in an interactive session with BEGIN or START TRANSACTION hasn't ended by using a COMMIT, ROLLBACK, or END command. This causes the transaction to move to `idle in transaction` state.

Actions

You can find idle transactions by querying `pg_stat_activity`.

In your SQL client, run the following query to list all connections in `idle in transaction` state and to order them by duration:

```
SELECT now() - state_change as idle_in_transaction_duration, now() - xact_start as
 xact_duration,*
FROM  pg_stat_activity
WHERE state  = 'idle in transaction'
AND   xact_start is not null
ORDER BY 1 DESC;
```

We recommend different actions depending on the causes of your insight.

Topics

- End transaction

- Terminate the connection
- Configure the idle_in_transaction_session_timeout parameter
- Check the AUTOCOMMIT status
- Check the transaction logic in your application code

End transaction

When you initiate a transaction in an interactive session with BEGIN or START TRANSACTION, it moves to `idle in transaction` state. It remains in this state until you end the transaction by issuing a COMMIT, ROLLBACK, END command or disconnect the connection completely to roll back the transaction.

Terminate the connection

Terminate the connection with an idle transaction using the following query:

```
SELECT pg_terminate_backend(pid);
```

pid is the process ID of the connection.

Configure the idle_in_transaction_session_timeout parameter

Configure the `idle_in_transaction_session_timeout` parameter in the parameter group. The advantage of configuring this parameter is that it does not require a manual intervention to terminate the long idle in transaction. For more information on this parameter, see the PostgreSQL documentation.

The following message will be reported in the PostgreSQL log file after the connection is terminated, when a transaction is in the idle_in_transaction state for longer than the specified time.

```
FATAL: terminating connection due to idle in transaction timeout
```

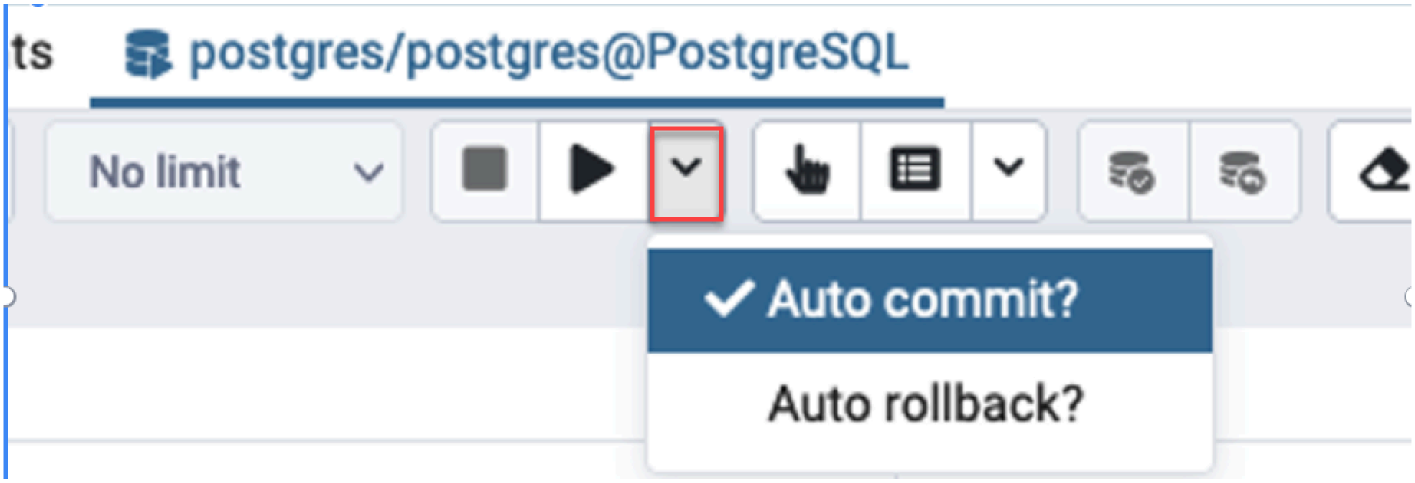
Check the AUTOCOMMIT status

AUTOCOMMIT is turned on by default. But if it is accidentally turned off in the client ensure that you turn it back on.

- In your psql client, run the following command:

```
postgres=> \set AUTOCOMMIT on
```

- In pgadmin, turn it on by choosing the AUTOCOMMIT option from the down arrow.



Check the transaction logic in your application code

Investigate your application logic for possible problems. Consider the following actions:

- Check if the JDBC auto commit is set true in your application. Also, consider using explicit `COMMIT` commands in your code.
- Check your error handling logic to see whether it closes a transaction after errors.
- Check whether your application is taking long to process the rows returned by a query while the transaction is open. If so, consider coding the application to close the transaction before processing the rows.
- Check whether a transaction contains many long-running operations. If so, divide a single transaction into multiple transactions.

Relevant metrics

The following PI metrics are related to this insight:

- idle_in_transaction_count - Number of sessions in `idle in transaction` state.
- idle_in_transaction_max_time - The duration of the longest running transaction in the `idle in transaction` state.

Best practices with Amazon Aurora PostgreSQL

Following, you can find several best practices for managing your Amazon Aurora PostgreSQL DB cluster. Be sure to also review basic maintenance tasks. For more information, see Managing Amazon Aurora PostgreSQL.

Topics

Avoiding slow performance, automatic restart, and failover for Aurora PostgreSQL DB instances

If you're running a heavy workload or workloads that spike beyond the allocated resources of your DB instance, you can exhaust the resources on which you're running your application and Aurora database. To get metrics on your database instance such as CPU utilization, memory usage, and number of database connections used, you can refer to the metrics provided by Amazon CloudWatch, Performance Insights, and Enhanced Monitoring. For more information on monitoring your DB instance, see Monitoring metrics in an Amazon Aurora cluster.

If your workload exhausts the resources you're using, your DB instance might slow down, restart, or even fail over to another DB instance. To avoid this, monitor your resource utilization, examine the workload running on your DB instance, and make optimizations where necessary. If optimizations don't improve the instance metrics and mitigate the resource exhaustion, consider scaling up your DB instance before you reach its limits. For more information on available DB instance classes and their specifications, see Aurora DB instance classes.

Diagnosing table and index bloat

You can use PostgreSQL Multiversion Concurrency Control (MVCC) to help preserve data integrity. PostgreSQL MVCC works by saving an internal copy of updated or deleted rows (also called *tuples*) until a transaction is either committed or rolled back. This saved internal copy is invisible to users. However, table bloat can occur when those invisible copies aren't cleaned up regularly by the VACUUM or AUTOVACUUM utilities. Unchecked, table bloat can incur increased storage costs and slow your processing speed.

In many cases, the default settings for VACUUM or AUTOVACUUM on Aurora are sufficient for handling unwanted table bloat. However, you may want to check for bloat if your application is experiencing the following conditions:

- Processes a large number of transactions in a relatively short time between VACUUM processes.
- Performs poorly and runs out of storage.

To get started, gather the most accurate information about how much space is used by dead tuples and how much you can recover by cleaning up the table and index bloat. To do so, use the `pgstattuple` extension to gather statistics on your Aurora cluster. For more information, see pgstattuple. Privileges to use the `pgstattuple` extension are limited to the `pg_stat_scan_tables` role and database superusers.

To create the `pgstattuple` extension on Aurora, connect a client session to the cluster, for example, psql or pgAdmin, and use the following command:

```
CREATE EXTENSION pgstattuple;
```

Create the extension in each database that you want to profile. After creating the extension, use the command line interface (CLI) to measure how much unusable space you can reclaim. Before gathering statistics, modify the cluster parameter group by setting AUTOVACUUM to 0. A setting of 0 prevents Aurora from automatically cleaning up any dead tuples left behind by your application, which can impact the accuracy of the results. Enter the following command to create a simple table:

```
postgres=> CREATE TABLE lab AS SELECT generate_series (0,100000);
SELECT 100001
```

In the following example, we run the query with AUTOVACUUM turned on for the DB cluster. The `dead_tuple_count` is 0, which indicates that the AUTOVACUUM has deleted the obsolete data or tuples from the PostgreSQL database.

To use `pgstattuple` to gather information about the table, specify the name of a table or an object identifier (OID) in the query:

```
postgres=> SELECT * FROM pgstattuple('lab');
```

```
table_len  | tuple_count | tuple_len | tuple_percent | dead_tuple_count |
 dead_tuple_len | dead_tuple_percent | free_space | free_percent
-----------+-------------+-----------+---------------+------------------
+----------------+--------------------+------------+--------------
3629056    | 100001      | 2800028   | 77.16         | 0                | 0
  | 0                  | 16616      | 0.46
(1 row)
```

In the following query, we turn off AUTOVACUUM and enter a command that deletes 25,000 rows from the table. As a result, the `dead_tuple_count` increases to 25000.

```
postgres=> DELETE FROM lab WHERE generate_series < 25000;

DELETE 25000
```

```
SELECT * FROM pgstattuple('lab');
```

```
table_len | tuple_count | tuple_len | tuple_percent | dead_tuple_count | dead_tuple_len
 | dead_tuple_percent | free_space | free_percent
-----------+-------------+-----------+---------------+------------------
+----------------+--------------------+------------+--------------
3629056 | 75001 | 2100028 | 57.87 | 25000 | 700000 | 19.29 | 16616 | 0.46
```

```
(1 row)
```

To reclaim those dead tuples, start a VACUUM process.

Observing bloat without interrupting your application

Settings on an Aurora cluster are optimized to provide the best practices for most workloads. However, you might want to optimize a cluster to better suit your applications and use patterns. In this case, you can use the `pgstattuple` extension without disrupting a busy application. To do so, perform the following steps:

1. Clone your Aurora instance.
2. Modify the parameter file to turn off AUTOVACUUM in the clone.
3. Perform a `pgstattuple` query while testing the clone with a sample workload or with pgbench, which is a program for running benchmark tests on PostgreSQL. For more information, see pgbench.

After running your applications and viewing the result, use pg_repack or VACUUM FULL on the restored copy and compare the differences. If you see a significant drop in the dead_tuple_count, dead_tuple_len, or dead_tuple_percent, then adjust the vacuum schedule on your production cluster to minimize the bloat.

Avoiding bloat in temporary tables

If your application creates temporary tables, make sure that your application removes those temporary tables when they're no longer needed. Autovacuum processes don't locate temporary tables. Left unchecked, temporary tables can quickly create database bloat. Moreover, the bloat can extend into the system tables, which are the internal tables that track PostgreSQL objects and attributes, like pg_attribute and pg_depend.

When a temporary table is no longer needed, you can use a TRUNCATE statement to empty the table and free up the space. Then, manually vacuum the pg_attribute and pg_depend tables. Vacuuming these tables ensures that creating and truncating/deleting temporary tables continually isn't adding tuples and contributing to system bloat.

You can avoid this problem while creating a temporary table by including the following syntax that deletes the new rows when content is committed:

```
CREATE TEMP TABLE IF NOT EXISTS table_name(table_description) ON COMMIT DELETE ROWS;
```

The `ON COMMIT DELETE ROWS` clause truncates the temporary table when the transaction is committed.

Avoiding bloat in indexes

When you change an indexed field in a table, the index update results in one or more dead tuples in that index. By default, the autovacuum process cleans up bloat in indexes, but that cleanup uses a significant amount of time and resources. To specify index cleanup preferences when you create a table, include the vacuum_index_cleanup clause. By default, at table creation time, the clause is set to AUTO, which means that the server decides if your index requires cleanup when it vacuums the table. You can set the clause to ON to turn on index cleanup for a specific table, or OFF to turn off index cleanup for that table. Remember, turning off index cleanup might save time, but can potentially lead to a bloated index.

You can manually control index cleanup when you VACUUM a table at the command line. To vacuum a table and remove dead tuples from the indexes, include the INDEX_CLEANUP clause with a value of ON and the table name:

```
acctg=> VACUUM (INDEX_CLEANUP ON) receivables;

INFO: aggressively vacuuming "public.receivables"
VACUUM
```

To vacuum a table without cleaning the indexes, specify a value of OFF:

```
acctg=> VACUUM (INDEX_CLEANUP OFF) receivables;

INFO: aggressively vacuuming "public.receivables"
VACUUM
```

Improved memory management in Aurora PostgreSQL

Customer workloads exhausting the free memory available in the DB instance leads to database restart by the operating system causing database unavailability. Aurora PostgreSQL has introduced improved memory management capabilities that proactively prevents stability issues and database restarts caused by insufficient free memory. This improvement is available by default in the following versions:

- 15.3 and higher 15 versions
- 14.8 and higher 14 versions
- 13.11 and higher 13 versions
- 12.15 and higher 12 versions
- 11.20 and higher 11 versions

To improve the memory management, it does the following:

- Cancelling database transactions that request more memory when the system is approaching critical memory pressure.
- The system is said to be under critical memory pressure, when it exhausts all the physical memory and is about to exhaust the swap. In these circumstances, any transaction that requests memory will be cancelled in an effort to immediately reduce memory pressure in the DB instance.
- Essential PostgreSQL launchers and background workers such as autovacuum workers are always protected.

Configuring memory management parameters

To turn on memory management

This feature is turned on by default. An error message is displayed when a transaction is cancelled due to insufficient memory as shown in the following example:

```
ERROR: out of memory Detail: Failed on request of size 16777216.
```

To turn off memory management

To turn off this feature, connect to the Aurora PostgreSQL DB cluster with psql and use the SET statement for the parameter values as mentioned below.

For Aurora PostgreSQL versions 11.21, 12.16, 13.12, 14.9, 15.4, and older versions:

```
postgres=>SET rds.memory_allocation_guard = true;
```

The default value of `rds.memory_allocation_guard` parameter is set to `false` in the Parameter group.

For Aurora PostgreSQL 12.17, 13.13, 14.10, 15.5, and higher versions:

```
postgres=>rds.enable_memory_management = false;
```

The default value of `rds.enable_memory_management` parameter is set to `true` in the Parameter group.

Setting the values of these parameters in the DB cluster parameter group prevents the queries from being canceled. For more information about DB cluster parameter group, see Working with parameter groups.

The value of these dynamic parameters can also be set at a session level to include or exclude a session in improved memory management.

> **Note**
>
> We don't recommend turning off this feature as it might lead to out-of-memory error that can cause workload induced database restart due to the memory exhaustion in the system.

Fast failover with Amazon Aurora PostgreSQL

Following, you can learn how to make sure that failover occurs as fast as possible. To recover quickly after failover, you can use cluster cache management for your Aurora PostgreSQL DB cluster. For more information, see Fast recovery after failover with cluster cache management for Aurora PostgreSQL.

Some of the steps that you can take to make failover perform fast include the following:

- Set Transmission Control Protocol (TCP) keepalives with short time frames, to stop longer running queries before the read timeout expires if there's a failure.
- Set timeouts for Java Domain Name System (DNS) caching aggressively. Doing this helps ensure the Aurora read-only endpoint can properly cycle through read-only nodes on later connection attempts.
- Set the timeout variables used in the JDBC connection string as low as possible. Use separate connection objects for short- and long-running queries.
- Use the read and write Aurora endpoints that are provided to connect to the cluster.

- Use RDS API operations to test application response on server-side failures. Also, use a packet dropping tool to test application response for client-side failures.
- Use the AWS JDBC Driver to take full advantage of the failover capabilities of Aurora PostgreSQL. For more information about the AWS JDBC Driver and complete instructions for using it, see the Amazon Web Services (AWS) JDBC Driver GitHub repository.

These are covered in more detail following.

Topics

Setting TCP keepalives parameters

When you set up a TCP connection, a set of timers is associated with the connection. When the keepalive timer reaches zero, a keepalive probe packet is sent to the connection endpoint. If the probe receives a reply, you can assume that the connection is still up and running.

Turning on TCP keepalive parameters and setting them aggressively ensures that if your client can't connect to the database, any active connections are quickly closed. The application can then connect to a new endpoint.

Make sure to set the following TCP keepalive parameters:

- `tcp_keepalive_time` controls the time, in seconds, after which a keepalive packet is sent when no data has been sent by the socket. ACKs aren't considered data. We recommend the following setting:

  ```
  tcp_keepalive_time = 1
  ```
- `tcp_keepalive_intvl` controls the time, in seconds, between sending subsequent keepalive packets after the initial packet is sent. Set this time by using the `tcp_keepalive_time` parameter. We recommend the following setting:

  ```
  tcp_keepalive_intvl = 1
  ```

- tcp_keepalive_probes is the number of unacknowledged keepalive probes that occur before the application is notified. We recommend the following setting:

  ```
  tcp_keepalive_probes = 5
  ```

These settings should notify the application within five seconds when the database stops responding. If keepalive packets are often dropped within the application's network, you can set a higher tcp_keepalive_probes value. Doing this allows for more buffer in less reliable networks, although it increases the time that it takes to detect an actual failure.

To set TCP keepalive parameters on Linux

1. Test how to configure your TCP keepalive parameters.

   We recommend doing so by using the command line with the following commands. This suggested configuration is system-wide. In other words, it also affects all other applications that create sockets with the SO_KEEPALIVE option on.

   ```
   sudo sysctl net.ipv4.tcp_keepalive_time=1
   sudo sysctl net.ipv4.tcp_keepalive_intvl=1
   sudo sysctl net.ipv4.tcp_keepalive_probes=5
   ```

2. After you've found a configuration that works for your application, persist these settings by adding the following lines to /etc/sysctl.conf, including any changes you made:

   ```
   tcp_keepalive_time = 1
   tcp_keepalive_intvl = 1
   tcp_keepalive_probes = 5
   ```

Configuring your application for fast failover

Following, you can find a discussion of several configuration changes for Aurora PostgreSQL that you can make for fast failover. To learn more about PostgreSQL JDBC driver setup and configuration, see the PostgreSQL JDBC Driver documentation.

Topics

- Other options for obtaining the host string

Reducing DNS cache timeouts

When your application tries to establish a connection after a failover, the new Aurora PostgreSQL writer will be a previous reader. You can find it by using the Aurora read-only endpoint before DNS updates have fully propagated. Setting the java DNS time to live (TTL) to a low value, such as under 30 seconds, helps cycle between reader nodes on later connection attempts.

```
// Sets internal TTL to match the Aurora RO Endpoint TTL
java.security.Security.setProperty("networkaddress.cache.ttl" , "1");
// If the lookup fails, default to something like small to retry
java.security.Security.setProperty("networkaddress.cache.negative.ttl" , "3");
```

Setting an Aurora PostgreSQL connection string for fast failover

To use Aurora PostgreSQL fast failover, make sure that your application's connection string has a list of hosts instead of just a single host. Following is an example connection string that you can use to connect to an Aurora PostgreSQL cluster. In this example, the hosts are in bold.

```
jdbc:postgresql://myauroracluster.cluster-c9bfei4hjlrd.us-east-1-
beta.rds.amazonaws.com:5432,
myauroracluster.cluster-ro-c9bfei4hjlrd.us-east-1-beta.rds.amazonaws.com:5432
/postgres?user=<primaryuser>&password=<primarypw>&loginTimeout=2
&connectTimeout=2&cancelSignalTimeout=2&socketTimeout=60
&tcpKeepAlive=true&targetServerType=primary
```

For best availability and to avoid a dependency on the RDS API, we recommend that you maintain a file to connect with. This file contains a host string that your application reads from when you establish a connection to the database. This host string has all the Aurora endpoints available for the cluster. For more information about Aurora endpoints, see Amazon Aurora connection management.

For example, you might store your endpoints in a local file as shown following.

```
myauroracluster.cluster-c9bfei4hjlrd.us-east-1-beta.rds.amazonaws.com:5432,
myauroracluster.cluster-ro-c9bfei4hjlrd.us-east-1-beta.rds.amazonaws.com:5432
```

Your application reads from this file to populate the host section of the JDBC connection string. Renaming the DB cluster causes these endpoints to change. Make sure that your application handles this event if it occurs.

Another option is to use a list of DB instance nodes, as follows.

```
my-node1.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432,
my-node2.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432,
my-node3.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432,
my-node4.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432
```

The benefit of this approach is that the PostgreSQL JDBC connection driver loops through all nodes on this list to find a valid connection. In contrast, when you use the Aurora endpoints only two nodes are tried in each connection attempt. However, there's a downside to using DB instance nodes. If you add or remove nodes from your cluster and the list of instance endpoints becomes stale, the connection driver might never find the correct host to connect to.

To help ensure that your application doesn't wait too long to connect to any one host, set the following parameters aggressively:

- `targetServerType` – Controls whether the driver connects to a write or read node. To ensure that your applications reconnect only to a write node, set the `targetServerType` value to `primary`.

  Values for the `targetServerType` parameter include `primary`, `secondary`, `any`, and `preferSecondary`. The `preferSecondary` value attempts to establish a connection to a reader first. It connects to the writer if no reader connection can be established.
- `loginTimeout` – Controls how long your application waits to log in to the database after a socket connection has been established.
- `connectTimeout` – Controls how long the socket waits to establish a connection to the database.

You can modify other application parameters to speed up the connection process, depending on how aggressive you want your application to be:

- `cancelSignalTimeout` – In some applications, you might want to send a "best effort" cancel signal on a query that has timed out. If this cancel signal is in your failover path, consider setting it aggressively to avoid sending this signal to a dead host.

- socketTimeout – This parameter controls how long the socket waits for read operations. This parameter can be used as a global "query timeout" to ensure no query waits longer than this value. A good practice is to have two connection handlers. One connection handler runs short-lived queries and sets this value lower. Another connection handler, for long-running queries, has this value set much higher. With this approach, you can rely on TCP keepalive parameters to stop long-running queries if the server goes down.
- tcpKeepAlive – Turn on this parameter to ensure the TCP keepalive parameters that you set are respected.
- loadBalanceHosts – When set to true, this parameter has the application connect to a random host chosen from a list of candidate hosts.

Other options for obtaining the host string

You can get the host string from several sources, including the aurora_replica_status function and by using the Amazon RDS API.

In many cases, you need to determine who the writer of the cluster is or to find other reader nodes in the cluster. To do this, your application can connect to any DB instance in the DB cluster and query the aurora_replica_status function. You can use this function to reduce the amount of time it takes to find a host to connect to. However, in certain network failure scenarios the aurora_replica_status function might show out-of-date or incomplete information.

A good way to ensure that your application can find a node to connect to is to try to connect to the cluster writer endpoint and then the cluster reader endpoint. You do this until you can establish a readable connection. These endpoints don't change unless you rename your DB cluster. Thus, you can generally leave them as static members of your application or store them in a resource file that your application reads from.

After you establish a connection using one of these endpoints, you can get information about the rest of the cluster. To do this, call the aurora_replica_status function. For example, the following command retrieves information with aurora_replica_status.

```
postgres=> SELECT server_id, session_id, highest_lsn_rcvd, cur_replay_latency_in_usec,
 now(), last_update_timestamp
FROM aurora_replica_status();

server_id | session_id | highest_lsn_rcvd | cur_replay_latency_in_usec | now |
 last_update_timestamp
```

```
-----------+--------------------------------------+------------------
+----------------------------+-------------------------------+------------------------
mynode-1 | 3e3c5044-02e2-11e7-b70d-95172646d6ca | 594221001 | 201421 | 2017-03-07
 19:50:24.695322+00 | 2017-03-07 19:50:23+00
mynode-2 | 1efdd188-02e4-11e7-becd-f12d7c88a28a | 594221001 | 201350 | 2017-03-07
 19:50:24.695322+00 | 2017-03-07 19:50:23+00
mynode-3 | MASTER_SESSION_ID | | | 2017-03-07 19:50:24.695322+00 | 2017-03-07
 19:50:23+00
(3 rows)
```

For example, the hosts section of your connection string might start with both the writer and reader cluster endpoints, as shown following.

```
myauroracluster.cluster-c9bfei4hjlrd.us-east-1-beta.rds.amazonaws.com:5432,
myauroracluster.cluster-ro-c9bfei4hjlrd.us-east-1-beta.rds.amazonaws.com:5432
```

In this scenario, your application attempts to establish a connection to any node type, primary or secondary. When your application is connected, a good practice is to first examine the read/write status of the node. To do this, query for the result of the command `SHOW transaction_read_only`.

If the return value of the query is `OFF`, then you successfully connected to the primary node. However, suppose that the return value is `ON` and your application requires a read/write connection. In this case, you can call the `aurora_replica_status` function to determine the `server_id` that has `session_id='MASTER_SESSION_ID'`. This function gives you the name of the primary node. You can use this with the `endpointPostfix` described following.

Make sure that you're aware when you connect to a replica that has stale data. When this happens, the `aurora_replica_status` function might show out-of-date information. You can set a threshold for staleness at the application level. To check this, you can look at the difference between the server time and the `last_update_timestamp` value. In general, your application should avoid flipping between two hosts due to conflicting information returned by the `aurora_replica_status` function. Your application should try all known hosts first instead of following the data returned by `aurora_replica_status`.

Listing instances using the DescribeDBClusters API operation, example in Java

You can programmatically find the list of instances by using the AWS SDK for Java, specifically the DescribeDBClusters API operation.

Following is a small example of how you might do this in Java 8.

```
AmazonRDS client = AmazonRDSClientBuilder.defaultClient();
DescribeDBClustersRequest request = new DescribeDBClustersRequest()
   .withDBClusterIdentifier(clusterName);
DescribeDBClustersResult result =
rdsClient.describeDBClusters(request);

DBCluster singleClusterResult = result.getDBClusters().get(0);

String pgJDBCEndpointStr =
singleClusterResult.getDBClusterMembers().stream()
   .sorted(Comparator.comparing(DBClusterMember::getIsClusterWriter)
   .reversed()) // This puts the writer at the front of the list
   .map(m -> m.getDBInstanceIdentifier() + endpointPostfix + ":" +
 singleClusterResult.getPort()))
   .collect(Collectors.joining(","));
```

Here, `pgJDBCEndpointStr` contains a formatted list of endpoints, as shown following.

```
my-node1.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432,
my-node2.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com:5432
```

The variable `endpointPostfix` can be a constant that your application sets. Or your application can get it by querying the `DescribeDBInstances` API operation for a single instance in your cluster. This value remains constant within an AWS Region and for an individual customer. So it saves an API call to simply keep this constant in a resource file that your application reads from. In the example preceding, it's set to the following.

```
.cksc6xlmwcyw.us-east-1-beta.rds.amazonaws.com
```

For availability purposes, a good practice is to default to using the Aurora endpoints of your DB cluster if the API isn't responding or takes too long to respond. The endpoints are guaranteed to be up to date within the time it takes to update the DNS record. Updating the DNS record with an endpoint typically takes less than 30 seconds. You can store the endpoint in a resource file that your application consumes.

Testing failover

In all cases you must have a DB cluster with two or more DB instances in it.

From the server side, certain API operations can cause an outage that can be used to test how your applications responds:

- FailoverDBCluster – This operation attempts to promote a new DB instance in your DB cluster to writer.

  The following code example shows how you can use `failoverDBCluster` to cause an outage. For more details about setting up an Amazon RDS client, see Using the AWS SDK for Java.

  ```
  public void causeFailover() {

      final AmazonRDS rdsClient = AmazonRDSClientBuilder.defaultClient();

      FailoverDBClusterRequest request = new FailoverDBClusterRequest();
      request.setDBClusterIdentifier("cluster-identifier");

      rdsClient.failoverDBCluster(request);
  }
  ```

- RebootDBInstance – Failover isn't guaranteed with this API operation. It shuts down the database on the writer, however. You can use it to test how your application responds to connections dropping. The `ForceFailover` parameter doesn't apply for Aurora engines. Instead, use the `FailoverDBCluster` API operation.
- ModifyDBCluster – Modifying the `Port` parameter causes an outage when the nodes in the cluster begin listening on a new port. In general, your application can respond to this failure first by ensuring that only your application controls port changes. Also, ensure that it can appropriately update the endpoints it depends on. You can do this by having someone manually update the port when they make modifications at the API level. Or you can do this by using the RDS API in your application to determine if the port has changed.
- ModifyDBInstance – Modifying the `DBInstanceClass` parameter causes an outage.
- DeleteDBInstance – Deleting the primary (writer) causes a new DB instance to be promoted to writer in your DB cluster.

From the application or client side, if you use Linux, you can test how the application responds to sudden packet drops. You can do this based on whether port, host, or if TCP keepalive packets are sent or received by using the iptables command.

Fast failover example in Java

The following code example shows how an application might set up an Aurora PostgreSQL driver manager.

The application calls the `getConnection` function when it needs a connection. A call to `getConnection` can fail to find a valid host. An example is when no writer is found but the `targetServerType` parameter is set to `primary`. In this case, the calling application should simply retry calling the function.

To avoid pushing the retry behavior onto the application, you can wrap this retry call into a connection pooler. With most connection poolers, you can specify a JDBC connection string. So your application can call into `getJdbcConnectionString` and pass that to the connection pooler. Doing this means you can use faster failover with Aurora PostgreSQL.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import org.joda.time.Duration;

public class FastFailoverDriverManager {
   private static Duration LOGIN_TIMEOUT = Duration.standardSeconds(2);
   private static Duration CONNECT_TIMEOUT = Duration.standardSeconds(2);
   private static Duration CANCEL_SIGNAL_TIMEOUT = Duration.standardSeconds(1);
   private static Duration DEFAULT_SOCKET_TIMEOUT = Duration.standardSeconds(5);

   public FastFailoverDriverManager() {
       try {
            Class.forName("org.postgresql.Driver");
       } catch (ClassNotFoundException e) {
            e.printStackTrace();
       }

       /*
         * RO endpoint has a TTL of 1s, we should honor that here. Setting this
 aggressively makes sure that when
```

```
        * the PG JDBC driver creates a new connection, it will resolve a new different
RO endpoint on subsequent attempts
        * (assuming there is > 1 read node in your cluster)
        */
       java.security.Security.setProperty("networkaddress.cache.ttl" , "1");
      // If the lookup fails, default to something like small to retry
      java.security.Security.setProperty("networkaddress.cache.negative.ttl" , "3");
  }

  public Connection getConnection(String targetServerType) throws SQLException {
      return getConnection(targetServerType, DEFAULT_SOCKET_TIMEOUT);
  }

  public Connection getConnection(String targetServerType, Duration queryTimeout)
throws SQLException {
       Connection conn =
DriverManager.getConnection(getJdbcConnectionString(targetServerType, queryTimeout));

      /*
        * A good practice is to set socket and statement timeout to be the same thing
since both
        * the client AND server will stop the query at the same time, leaving no
running queries
        * on the backend
        */
       Statement st = conn.createStatement();
       st.execute("set statement_timeout to " + queryTimeout.getMillis());
       st.close();

      return conn;
  }

  private static String urlFormat = "jdbc:postgresql://%s"
          + "/postgres"
          + "?user=%s"
          + "&password=%s"
          + "&loginTimeout=%d"
          + "&connectTimeout=%d"
          + "&cancelSignalTimeout=%d"
          + "&socketTimeout=%d"
          + "&targetServerType=%s"
          + "&tcpKeepAlive=true"
          + "&ssl=true"
          + "&loadBalanceHosts=true";
```

```
   public String getJdbcConnectionString(String targetServerType, Duration
 queryTimeout) {
       return String.format(urlFormat,
                getFormattedEndpointList(getLocalEndpointList()),
                CredentialManager.getUsername(),
                CredentialManager.getPassword(),
                LOGIN_TIMEOUT.getStandardSeconds(),
                CONNECT_TIMEOUT.getStandardSeconds(),
                CANCEL_SIGNAL_TIMEOUT.getStandardSeconds(),
                queryTimeout.getStandardSeconds(),
                targetServerType
       );
   }

   private List<String> getLocalEndpointList() {
       /*
         * As mentioned in the best practices doc, a good idea is to read a local
 resource file and parse the cluster endpoints.
         * For illustration purposes, the endpoint list is hardcoded here
         */
        List<String> newEndpointList = new ArrayList<>();
        newEndpointList.add("myauroracluster.cluster-c9bfei4hjlrd.us-east-1-
beta.rds.amazonaws.com:5432");
        newEndpointList.add("myauroracluster.cluster-ro-c9bfei4hjlrd.us-east-1-
beta.rds.amazonaws.com:5432");

       return newEndpointList;
   }

   private static String getFormattedEndpointList(List<String> endpoints) {
       return IntStream.range(0, endpoints.size())
               .mapToObj(i -> endpoints.get(i).toString())
               .collect(Collectors.joining(","));
   }
}
```

Fast recovery after failover with cluster cache management for Aurora PostgreSQL

For fast recovery of the writer DB instance in your Aurora PostgreSQL clusters if there's a failover, use cluster cache management for Amazon Aurora PostgreSQL. Cluster cache management ensures that application performance is maintained if there's a failover.

In a typical failover situation, you might see a temporary but large performance degradation after failover. This degradation occurs because when the failover DB instance starts, the buffer cache is empty. An empty cache is also known as a *cold cache*. A cold cache degrades performance because the DB instance has to read from the slower disk, instead of taking advantage of values stored in the buffer cache.

With cluster cache management, you set a specific reader DB instance as the failover target. Cluster cache management ensures that the data in the designated reader's cache is kept synchronized with the data in the writer DB instance's cache. The designated reader's cache with prefilled values is known as a *warm cache*. If a failover occurs, the designated reader uses values in its warm cache immediately when it's promoted to the new writer DB instance. This approach provides your application much better recovery performance.

Cluster cache management requires that the designated reader instance have the same instance class type and size (`db.r5.2xlarge` or `db.r5.xlarge`, for example) as the writer. Keep this in mind when you create your Aurora PostgreSQL DB clusters so that your cluster can recover during a failover. For a listing of instance class types and sizes, see Hardware specifications for DB instance classes for Aurora.

> **Note**
>
> Cluster cache management is not supported for Aurora PostgreSQL DB clusters that are part of Aurora global databases. It is recommended that no workload should run on the designated tier-0 reader.

Contents

Configuring cluster cache management

To configure cluster cache management, do the following processes in order.

Topics

- Enabling cluster cache management
- Setting the promotion tier priority for the writer DB instance
- Setting the promotion tier priority for a reader DB instance

> **Note**
>
> Allow at least 1 minute after completing these steps for cluster cache management to be fully operational.

Enabling cluster cache management

To enable cluster cache management, take the steps described following.

Console

To enable cluster cache management

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose the parameter group for your Aurora PostgreSQL DB cluster.

   The DB cluster must use a parameter group other than the default, because you can't change values in a default parameter group.
4. For **Parameter group actions**, choose **Edit**.
5. Set the value of the `apg_ccm_enabled` cluster parameter to **1**.
6. Choose **Save changes**.

AWS CLI

To enable cluster cache management for an Aurora PostgreSQL DB cluster, use the AWS CLI `modify-db-cluster-parameter-group` command with the following required parameters:

- `--db-cluster-parameter-group-name`

- `--parameters`

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name my-db-cluster-parameter-group \
    --parameters "ParameterName=apg_ccm_enabled,ParameterValue=1,ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name my-db-cluster-parameter-group ^
    --parameters "ParameterName=apg_ccm_enabled,ParameterValue=1,ApplyMethod=immediate"
```

Setting the promotion tier priority for the writer DB instance

For cluster cache management, make sure that the promotion priority is **tier-0** for the writer DB instance of the Aurora PostgreSQL DB cluster. The *promotion tier priority* is a value that specifies the order in which an Aurora reader is promoted to the writer DB instance after a failure. Valid values are 0–15, where 0 is the first priority and 15 is the last priority. For more information about the promotion tier, see Fault tolerance for an Aurora DB cluster.

Console

To set the promotion priority for the writer DB instance to tier-0

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the **Writer** DB instance of the Aurora PostgreSQL DB cluster.
4. Choose **Modify**. The **Modify DB Instance** page appears.
5. On the **Additional configuration** panel, choose **tier-0** for the **Failover priority**.
6. Choose **Continue** and check the summary of modifications.
7. To apply the changes immediately after you save them, choose **Apply immediately**.
8. Choose **Modify DB Instance** to save your changes.

AWS CLI

To set the promotion tier priority to 0 for the writer DB instance using the AWS CLI, call the modify-db-instance command with the following required parameters:

- `--db-instance-identifier`
- `--promotion-tier`
- `--apply-immediately`

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier writer-db-instance \
    --promotion-tier 0 \
    --apply-immediately
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier writer-db-instance ^
    ---promotion-tier 0  ^
    --apply-immediately
```

Setting the promotion tier priority for a reader DB instance

You must set only one reader DB instance for cluster cache management. To do so, choose a reader from the Aurora PostgreSQL cluster that is the same instance class and size as the writer DB instance. For example, if the writer uses `db.r5.xlarge`, choose a reader that uses this same instance class type and size. Then set its promotion tier priority to 0.

The *promotion tier priority* is a value that specifies the order in which an Aurora reader is promoted to the writer DB instance after a failure. Valid values are 0–15, where 0 is the first priority and 15 is the last priority.

Console

To set the promotion priority of the reader DB instance to tier-0

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose a **Reader** DB instance of the Aurora PostgreSQL DB cluster that is the same instance class as the writer DB instance.
4. Choose **Modify**. The **Modify DB Instance** page appears.
5. On the **Additional configuration** panel, choose **tier-0** for the **Failover priority**.
6. Choose **Continue** and check the summary of modifications.
7. To apply the changes immediately after you save them, choose **Apply immediately**.
8. Choose **Modify DB Instance** to save your changes.

AWS CLI

To set the promotion tier priority to 0 for the reader DB instance using the AWS CLI, call the modify-db-instance command with the following required parameters:

- `--db-instance-identifier`
- `--promotion-tier`
- `--apply-immediately`

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier reader-db-instance \
    --promotion-tier 0 \
    --apply-immediately
```

For Windows:

```
aws rds modify-db-instance ^
```

```
    --db-instance-identifier reader-db-instance ^
    ---promotion-tier 0  ^
    --apply-immediately
```

Monitoring the buffer cache

After setting up cluster cache management, you can monitor the state of synchronization between the writer DB instance's buffer cache and the designated reader's warm buffer cache. To examine the buffer cache contents on both the writer DB instance and the designated reader DB instance, use the PostgreSQL `pg_buffercache` module. For more information, see the [PostgreSQL `pg_buffercache` documentation].

Using the `aurora_ccm_status` Function

Cluster cache management also provides the `aurora_ccm_status` function. Use the `aurora_ccm_status` function on the writer DB instance to get the following information about the progress of cache warming on the designated reader:

- `buffers_sent_last_minute` – How many buffers have been sent to the designated reader in the last minute.
- `buffers_found_last_minute` – The number of frequently accessed buffers identified during the past minute.
- `buffers_sent_last_scan` – How many buffers have been sent to the designated reader during the last complete scan of the buffer cache.
- `buffers_found_last_scan` – How many buffers have been identified as frequently accessed and needed to be sent during the last complete scan of the buffer cache. Buffers already cached on the designated reader aren't sent.
- `buffers_sent_current_scan` – How many buffers have been sent so far during the current scan.
- `buffers_found_current_scan` – How many buffers have been identified as frequently accessed in the current scan.
- `current_scan_progress` – How many buffers have been visited so far during the current scan.

The following example shows how to use the `aurora_ccm_status` function to convert some of its output into a warm rate and warm percentage.

```
SELECT buffers_sent_last_minute*8/60 AS warm_rate_kbps,
    100*(1.0-buffers_sent_last_scan::float/buffers_found_last_scan) AS warm_percent
    FROM aurora_ccm_status();
```

Troubleshooting CCM configuration

When you enable `apg_ccm_enabled` cluster parameter, cluster cache management is automatically turned on at the instance level on the writer DB instance and one reader DB instance on the Aurora PostgreSQL DB cluster. The writer and reader instance should use the same instance class type and size. Their promotion tier priority is set to `0`. Other reader instances in the DB cluster should have a non-zero promotion tier and cluster cache management is disabled for those instances.

The following reasons may lead to issues in the configuration and disable cluster cache management:

- When there is no single reader DB instance set to promotion tier 0.
- When the writer DB instance is not set to promotion tier 0.
- When more than one reader DB instances are set to promotion tier 0.
- When the writer and one reader DB instances with promotion tier 0 doesn't have the same instance size.

Managing Aurora PostgreSQL connection churn with pooling

When client applications connect and disconnect so often that Aurora PostgreSQL DB cluster response time slows, the cluster is said to be experiencing *connection churn*. Each new connection to the Aurora PostgreSQL DB cluster endpoint consumes resources, thus reducing the resources that can be used to process the actual workload. Connection churn is an issue that we recommend that you manage by following some of the best practices discussed following.

For starters, you can improve response times on Aurora PostgreSQL DB clusters that have high rates of connection churn. To do this, you can use a connection pooler, such as RDS Proxy. A *connection pooler* provides a cache of ready to use connections for clients. Almost all versions of Aurora PostgreSQL support RDS Proxy. For more information, see Amazon RDS Proxy with Aurora PostgreSQL.

If your specific version of Aurora PostgreSQL doesn't support RDS Proxy, you can use another PostgreSQL–compatible connection pooler, such as PgBouncer. To learn more, see the PgBouncer website.

To see if your Aurora PostgreSQL DB cluster can benefit from connection pooling, you can check the `postgresql.log` file for connections and disconnections. You can also use Performance Insights to find out how much connection churn your Aurora PostgreSQL DB cluster is experiencing. Following, you can find information about both topics.

Logging connections and disconnections

The PostgreSQL `log_connections` and `log_disconnections` parameters can capture connections and disconnections to the writer instance of the Aurora PostgreSQL DB cluster. By default, these parameters are turned off. To turn these parameters on, use a custom parameter group and turn on by changing the value to 1. For more information about custom parameter groups, see [Working with DB cluster parameter groups](). To check the settings, connect to your DB cluster endpoint for Aurora PostgreSQL by using psql and query as follows.

```
labdb=> SELECT setting FROM pg_settings
  WHERE name = 'log_connections';
 setting
---------
on
(1 row)
labdb=> SELECT setting FROM pg_settings
  WHERE name = 'log_disconnections';
setting
---------
on
(1 row)
```

With both of these parameters turned on, the log captures all new connections and disconnections. You see the user and database for each new authorized connection. At disconnection time, the session duration is also logged, as shown in the following example.

```
2022-03-07 21:44:53.978 UTC [16641] LOG: connection authorized: user=labtek
 database=labdb application_name=psql
2022-03-07 21:44:55.718 UTC [16641] LOG: disconnection: session time: 0:00:01.740
 user=labtek database=labdb host=[local]
```

To check your application for connection churn, turn on these parameters if they're not on already. Then gather data in the PostgreSQL log for analysis by running your application with a realistic workload and time period. You can view the log file in the RDS console. Choose the writer instance of your Aurora PostgreSQL DB cluster, and then choose the **Logs & events** tab. For more information, see Viewing and listing database log files.

Or you can download the log file from the console and use the following command sequence. This sequence finds the total number of connections authorized and dropped per minute.

```
grep "connection authorized\|disconnection: session time:"
 postgresql.log.2022-03-21-16|\
awk {'print $1,$2}' |\
sort |\
uniq -c |\
sort -n -k1
```

In the example output, you can see a spike in authorized connections followed by disconnections starting at 16:12:10.

```
.....
,......
.........
5 2022-03-21 16:11:55 connection authorized:
9 2022-03-21 16:11:55 disconnection: session
5 2022-03-21 16:11:56 connection authorized:
5 2022-03-21 16:11:57 connection authorized:
5 2022-03-21 16:11:57 disconnection: session
32 2022-03-21 16:12:10 connection authorized:
30 2022-03-21 16:12:10 disconnection: session
31 2022-03-21 16:12:11 connection authorized:
27 2022-03-21 16:12:11 disconnection: session
27 2022-03-21 16:12:12 connection authorized:
27 2022-03-21 16:12:12 disconnection: session
41 2022-03-21 16:12:13 connection authorized:
47 2022-03-21 16:12:13 disconnection: session
46 2022-03-21 16:12:14 connection authorized:
41 2022-03-21 16:12:14 disconnection: session
24 2022-03-21 16:12:15 connection authorized:
29 2022-03-21 16:12:15 disconnection: session
28 2022-03-21 16:12:16 connection authorized:
24 2022-03-21 16:12:16 disconnection: session
40 2022-03-21 16:12:17 connection authorized:
42 2022-03-21 16:12:17 disconnection: session
40 2022-03-21 16:12:18 connection authorized:
40 2022-03-21 16:12:18 disconnection: session
```

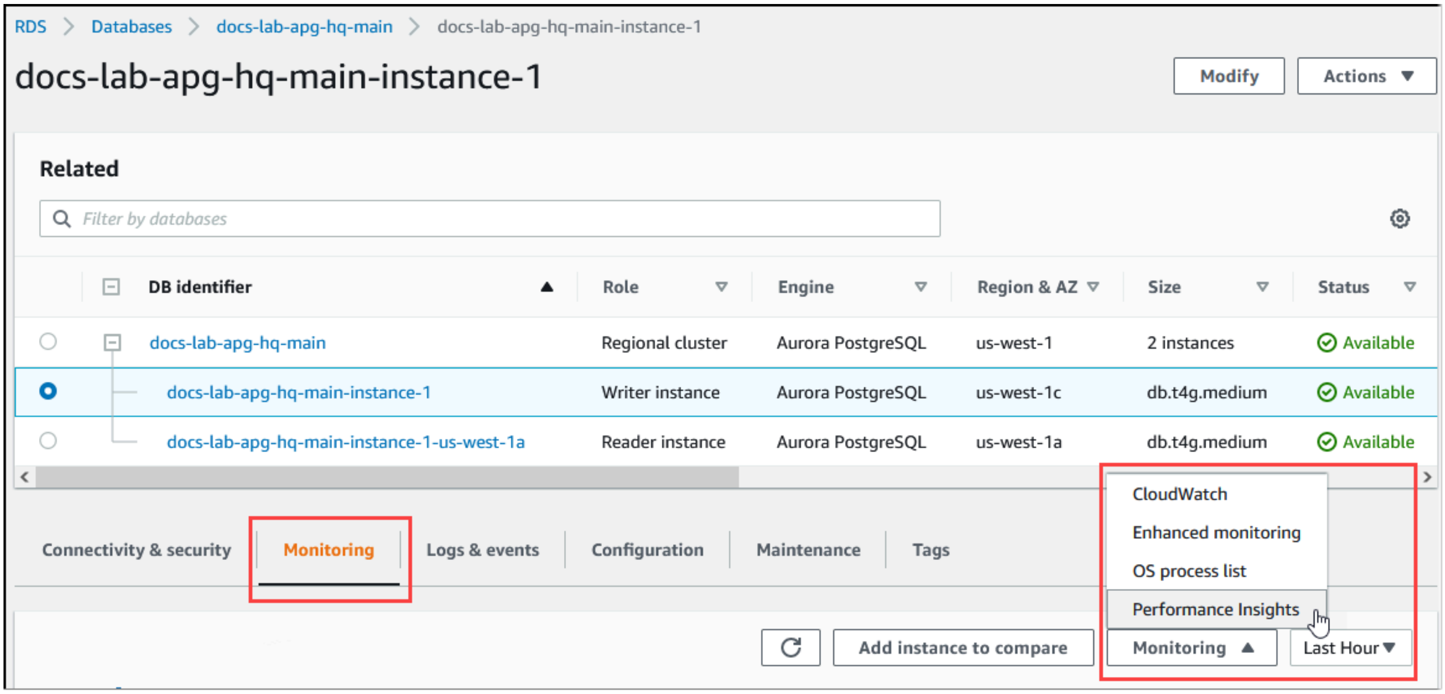
```
.....
,......
.........
1 2022-03-21 16:14:10 connection authorized:
1 2022-03-21 16:14:10 disconnection: session
1 2022-03-21 16:15:00 connection authorized:
1 2022-03-21 16:16:00 connection authorized:
```

With this information, you can decide if your workload can benefit from a connection pooler. For more detailed analysis, you can use Performance Insights.

Detecting connection churn with Performance Insights

You can use Performance Insights to assess the amount of connection churn on your Aurora PostgreSQL-Compatible Edition DB cluster. When you create an Aurora PostgreSQL DB cluster, the setting for Performance Insights is turned on by default. If you cleared this choice when you created your DB cluster, modify your cluster to turn on the feature. For more information, see Modifying an Amazon Aurora DB cluster.

With Performance Insights running on your Aurora PostgreSQL DB cluster, you can choose the metrics that you want to monitor. You can access Performance Insights from the navigation pane in the console. You can also access Performance Insights from the **Monitoring** tab of the writer instance for your Aurora PostgreSQL DB cluster, as shown in the following image.



RDS > Databases > docs-lab-apg-hq-main > docs-lab-apg-hq-main-instance-1

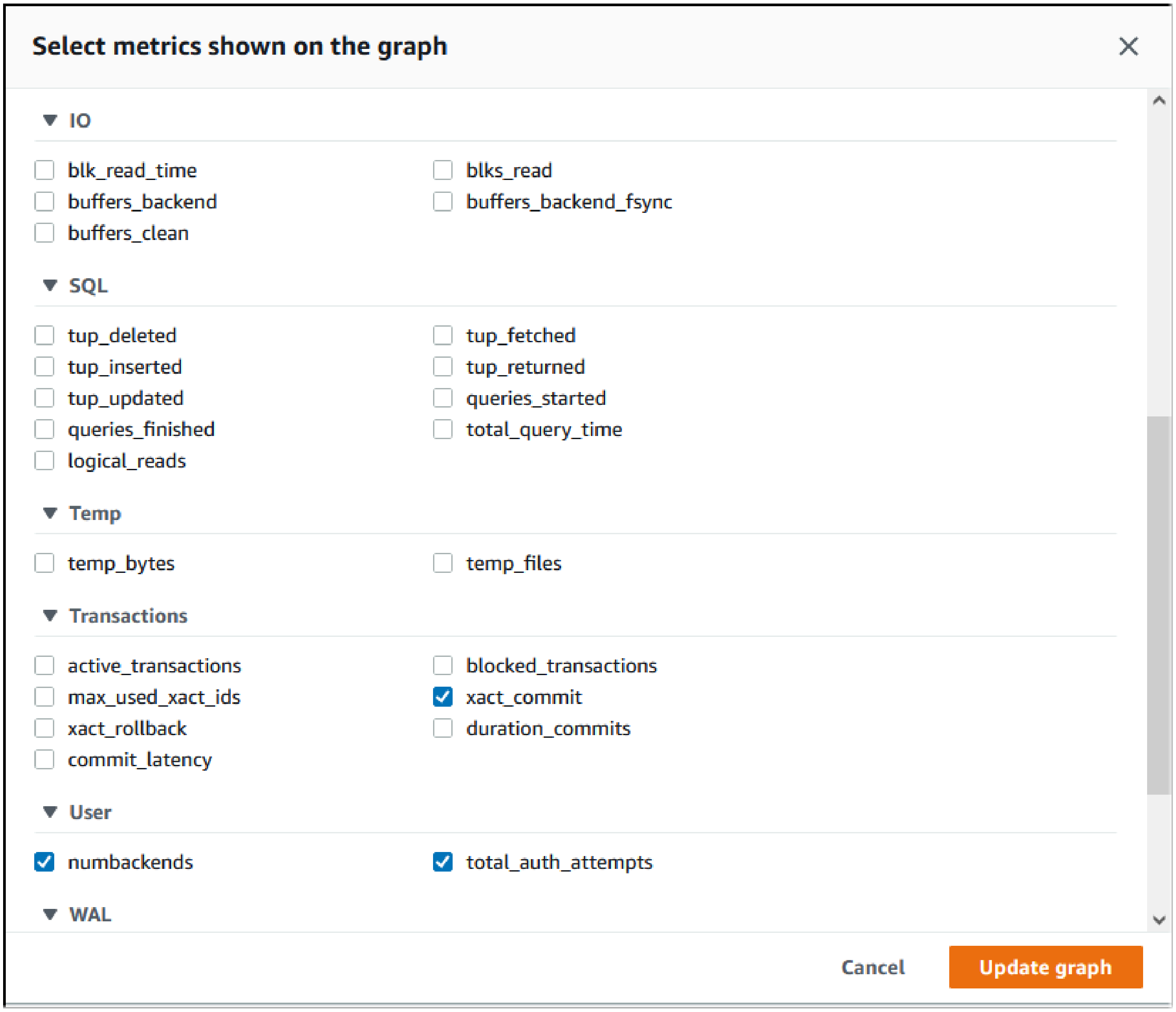
docs-lab-apg-hq-main-instance-1

Related

	DB identifier	Role	Engine	Region & AZ	Size	Status
○	docs-lab-apg-hq-main	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances	Available
●	docs-lab-apg-hq-main-instance-1	Writer instance	Aurora PostgreSQL	us-west-1c	db.t4g.medium	Available
○	docs-lab-apg-hq-main-instance-1-us-west-1a	Reader instance	Aurora PostgreSQL	us-west-1a	db.t4g.medium	Available

From the Performance Insights console, choose **Manage metrics**. To analyze your Aurora PostgreSQL DB cluster's connection and disconnection activity, choose the following metrics. These are all metrics from PostgreSQL.

- `xact_commit` – The number of committed transactions.
- `total_auth_attempts` – The number of attempted authenticated user connections per minute.
- `numbackends` – The number of backends currently connected to the database.



Select metrics shown on the graph

▼ IO

□ blk_read_time
□ blks_read
□ buffers_backend
□ buffers_backend_fsync
□ buffers_clean

▼ SQL

□ tup_deleted
□ tup_fetched
□ tup_inserted
□ tup_returned
□ tup_updated
□ queries_started
□ queries_finished
□ total_query_time
□ logical_reads

▼ Temp

□ temp_bytes
□ temp_files

▼ Transactions

□ active_transactions
□ blocked_transactions
□ max_used_xact_ids
☑ xact_commit
□ xact_rollback
□ duration_commits
□ commit_latency

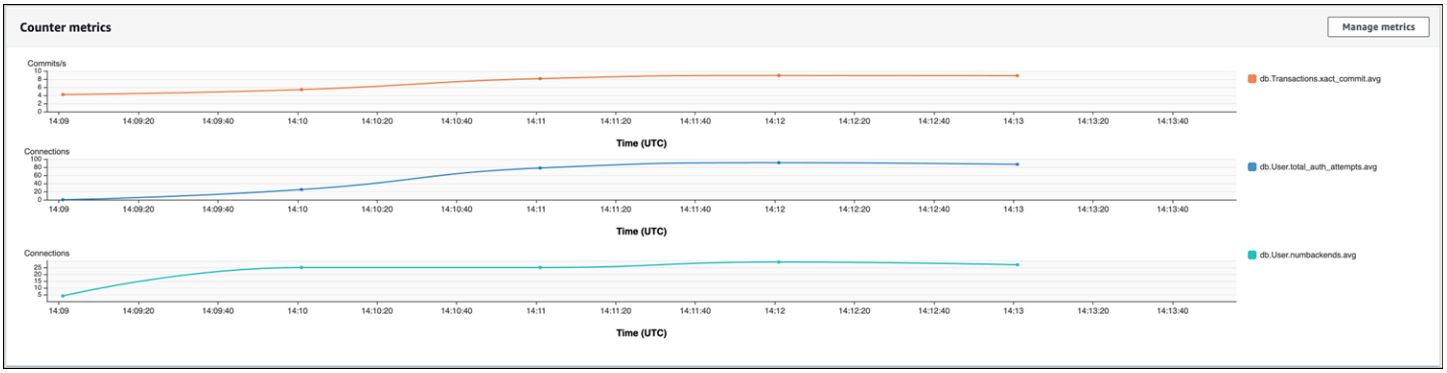
▼ User

☑ numbackends
☑ total_auth_attempts

▼ WAL

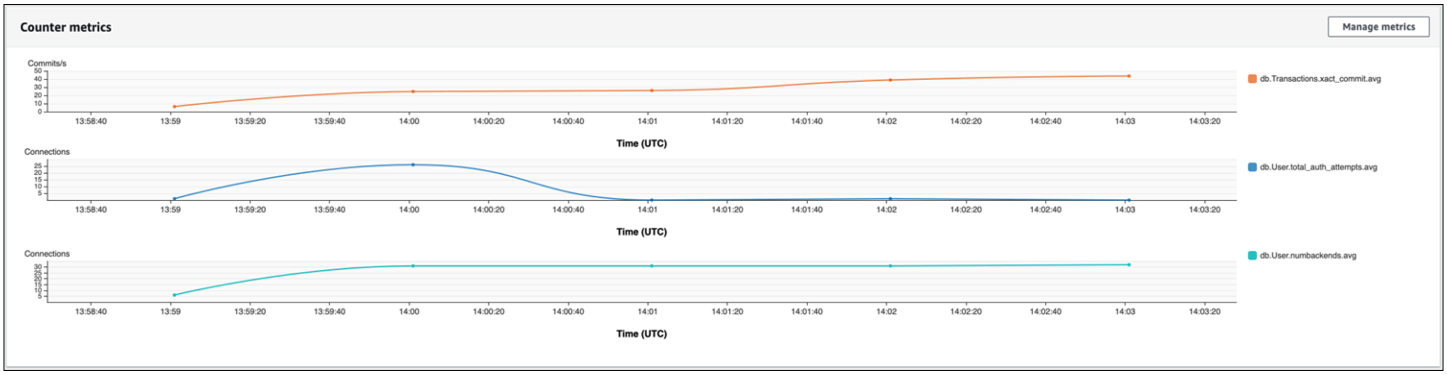
To save the settings and display connection activity, choose **Update graph**.

In the following image, you can see the impact of running pgbench with 100 users. The line showing connections is on a consistent upward slope. To learn more about pgbench and how to use it, see pgbench in PostgreSQL documentation.



The image shows that running a workload with as few as 100 users without a connection pooler can cause a significant increase in the number of `total_auth_attempts` throughout the duration of workload processing.

With RDS Proxy connection pooling, the connection attempts increase at the start of the workload. After setting up the connection pool, the average declines. The resources used by transactions and backend use stays consistent throughout workload processing.



For more information about using Performance Insights with your Aurora PostgreSQL DB cluster, see Monitoring DB load with Performance Insights on Amazon Aurora. To analyze the metrics, see Analyzing metrics with the Performance Insights dashboard.

Demonstrating the benefits of connection pooling

As mentioned previously, if you determine that your Aurora PostgreSQL DB cluster has a connection churn problem, you can use RDS Proxy for improved performance. Following, you can find an example that shows the differences in processing a workload when connections are pooled and when they're not. The example uses pgbench to model a transaction workload.

As is psql, pgbench is a PostgreSQL client application that you can install and run from your local client machine. You can also install and run it from the Amazon EC2 instance that you use for managing your Aurora PostgreSQL DB cluster. For more information, see pgbench in the PostgreSQL documentation.

To step through this example, you first create the pgbench environment in your database. The following command is the basic template for initializing the pgbench tables in the specified database. This example uses the default main user account, `postgres`, for the login. Change it as needed for your Aurora PostgreSQL DB cluster. You create the pgbench environment in a database on the writer instance of your cluster.

> **Note**
>
> The pgbench initialization process drops and recreates tables named `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`. Be sure that the database that you choose for *dbname* when you initialize pgbench doesn't use these names.

```
pgbench -U postgres -h db-cluster-instance-1.111122223333.aws-region.rds.amazonaws.com
 -p 5432 -d -i -s 50 dbname
```

For pgbench, specify the following parameters.

-d

Outputs a debugging report as pgbench runs.

-h

Specifies the endpoint of the Aurora PostgreSQL DB cluster's writer instance.

-i

Initializes the pgbench environment in the database for the benchmark tests.

-p

Identifies the port used for database connections. The default for Aurora PostgreSQL is typically 5432 or 5433.

-s

Specifies the scaling factor to use for populating the tables with rows. The default scaling factor is 1, which generates 1 row in the `pgbench_branches` table, 10 rows in the `pgbench_tellers` table, and 100000 rows in the `pgbench_accounts` table.

-U

Specifies the user account for the Aurora PostgreSQL DB cluster's writer instance.

After the pgbench environment is set up, you can then run benchmarking tests with and without connection pooling. The default test consists of a series of five SELECT, UPDATE, and INSERT commands per transaction that run repeatedly for the time specified. You can specify scaling factor, number of clients, and other details to model your own use cases.

As an example, the command that follows runs the benchmark for 60 seconds (-T option, for time) with 20 concurrent connections (the -c option). The -C option makes the test run using a new connection each time, rather than once per client session. This setting gives you an indication of the connection overhead.

```
pgbench -h docs-lab-apg-133-test-instance-1.c3zr2auzukpa.us-west-1.rds.amazonaws.com -U
 postgres -p 5432 -T 60 -c 20 -C labdb
Password:**********
pgbench (14.3, server 13.3)
  starting vacuum...end.
  transaction type: <builtin: TPC-B (sort of)>
    scaling factor: 50
    query mode: simple
    number of clients: 20
    number of threads: 1
    duration: 60 s
    number of transactions actually processed: 495
    latency average = 2430.798 ms
    average connection time = 120.330 ms
    tps = 8.227750 (including reconnection times)
```

Running pgbench on the writer instance of an Aurora PostgreSQL DB cluster without reusing connections shows that only about 8 transactions are processed each second. This gives a total of 495 transactions during the 1-minute test.

If you reuse connections, the response from Aurora PostgreSQL DB cluster for the number of users is almost 20 times faster. With reuse, a total of 9,042 transactions is processed compared to 495 in the same amount of time and for the same number of user connections. The difference is that in the following, each connection is being reused.

```
pgbench -h docs-lab-apg-133-test-instance-1.c3zr2auzukpa.us-west-1.rds.amazonaws.com -U
 postgres -p 5432 -T 60 -c 20 labdb
Password:*********
pgbench (14.3, server 13.3)
      starting vacuum...end.
      transaction type: <builtin: TPC-B (sort of)>
      scaling factor: 50
      query mode: simple
      number of clients: 20
      number of threads: 1
      duration: 60 s
      number of transactions actually processed: 9042
      latency average = 127.880 ms
      initial connection time = 2311.188 ms
      tps = 156.396765 (without initial connection time)
```

This example shows you that pooling connections can significantly improve response times. For information about setting up RDS Proxy for your Aurora PostgreSQL DB cluster, see Using Amazon RDS Proxy for Aurora.

Tuning memory parameters for Aurora PostgreSQL

In Amazon Aurora PostgreSQL, you can use several parameters that control the amount of memory used for various processing tasks. If a task takes more memory than the amount set for a given parameter, Aurora PostgreSQL uses other resources for processing, such as by writing to disk. This can cause your Aurora PostgreSQL DB cluster to slow or potentially halt, with an out-of-memory error.

The default setting for each memory parameter can usually handle its intended processing tasks. However, you can also tune your Aurora PostgreSQL DB cluster's memory-related parameters. You do this tuning to ensure that enough memory is allocated for processing your specific workload.

Following, you can find information about parameters that control memory management. You can also learn how to assess memory utilization.

Checking and setting parameter values

The parameters that you can set to manage memory and assess your Aurora PostgreSQL DB cluster's memory usage include the following:

- work_mem – Specifies the amount of memory that the Aurora PostgreSQL DB cluster uses for internal sort operations and hash tables before it writes to temporary disk files.
- log_temp_files – Logs temporary file creation, file names, and sizes. When this parameter is turned on, a log entry is stored for each temporary file that gets created. Turn this on to see how frequently your Aurora PostgreSQL DB cluster needs to write to disk. Turn it off again after you've gathered information about your Aurora PostgreSQL DB cluster's temporary file generation, to avoid excessive logging.
- logical_decoding_work_mem – Specifies the amount of memory (in megabytes) to use for logical decoding. *Logical decoding* is the process used to create a replica. This process is done by converting data from the write-ahead log (WAL) file to the logical streaming output needed by the target.

  The value of this parameter creates a single buffer of the size specified for each replication connection. By default, it's 65536 KB. After this buffer is filled, the excess is written to disk as a file. To minimize disk activity, you can set the value of this parameter to a much higher value than that of work_mem.

These are all dynamic parameters, so you can change them for the current session. To do this, connect to the Aurora PostgreSQL DB cluster with psql and using the SET statement, as shown following.

```
SET parameter_name TO parameter_value;
```

Session settings last for the duration of the session only. When the session ends, the parameter reverts to its setting in the DB cluster parameter group. Before changing any parameters, first check the current values by querying the pg_settings table, as follows.

```
SELECT unit, setting, max_val
   FROM pg_settings WHERE name='parameter_name';
```

For example, to find the value of the work_mem parameter, connect to the Aurora PostgreSQL DB cluster's writer instance and run the following query.

```
SELECT unit, setting, max_val, pg_size_pretty(max_val::numeric)
  FROM pg_settings WHERE name='work_mem';
unit  | setting  | max_val   | pg_size_pretty
------+----------+-----------+----------------
 kB   | 1024     | 2147483647| 2048 MB
(1 row)
```

Changing parameter settings so that they persist requires using a custom DB cluster parameter group. After exercising your Aurora PostgreSQL DB cluster with different values for these parameters using the SET statement, you can create a custom parameter group and apply to your Aurora PostgreSQL DB cluster. For more information, see Working with parameter groups.

Understanding the working memory parameter

The working memory parameter (work_mem) specifies the maximum amount of memory that Aurora PostgreSQL can use to process complex queries. Complex queries include those that involve sorting or grouping operations—in other words, queries that use the following clauses:

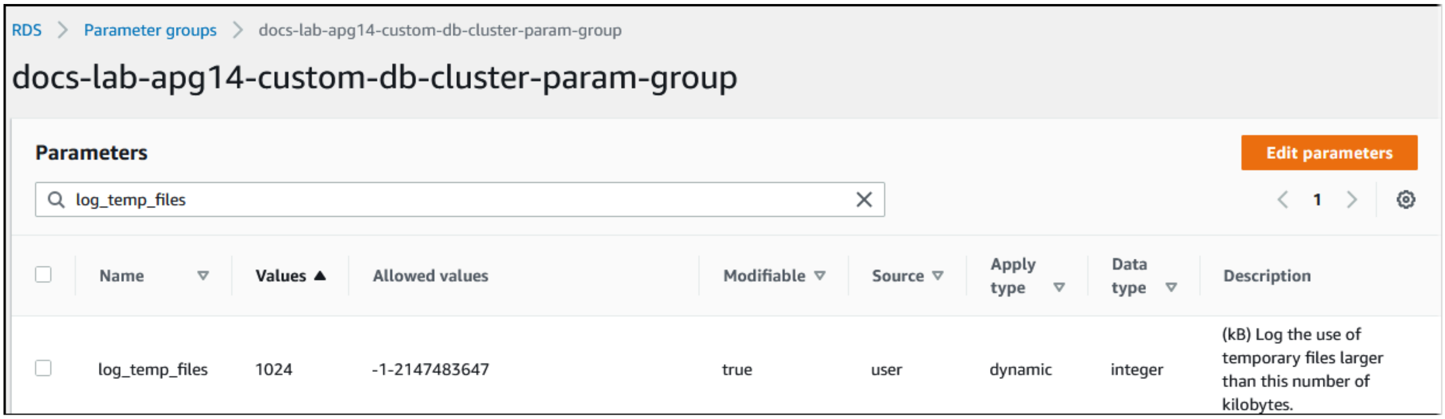
- ORDER BY
- DISTINCT
- GROUP BY
- JOIN (MERGE and HASH)

The query planner indirectly affects how your Aurora PostgreSQL DB cluster uses working memory. The query planner generates execution plans for processing SQL statements. A given plan might break up a complex query into multiple units of work that can be run in parallel. When possible, Aurora PostgreSQL uses the amount of memory specified in the work_mem parameter for each session before writing to disk for each parallel process.

Multiple database users running multiple operations concurrently and generating multiple units of work in parallel can exhaust your Aurora PostgreSQL DB cluster's allocated working memory. This can lead to excessive temporary file creation and disk I/O, or worse, it can lead to an out-of-memory error.

Identifying temporary file use

Whenever the memory required to process queries exceeds the value specified in the work_mem parameter, the working data is offloaded to disk in a temporary file. You can see how often this occurs by turning on the `log_temp_files` parameter. By default, this parameter is off (it's set to -1). To capture all temporary file information, set this parameter to 0. Set `log_temp_files` to any other positive integer to capture temporary file information for files equal to or greater than that amount of data (in kilobytes). In the following image, you can see an example from AWS Management Console.


RDS > Parameter groups > docs-lab-apg14-custom-db-cluster-param-group
Edit parameters
1

docs-lab-apg14-custom-db-cluster-param-group

Parameters

log_temp_files

☐	Name	Values ▲	Allowed values	Modifiable	Source	Apply type	Data type	Description
☐	log_temp_files	1024	-1-2147483647	true	user	dynamic	integer	(kB) Log the use of temporary files larger than this number of kilobytes.

After configuring temporary file logging, you can test with your own workload to see if your working memory setting is sufficient. You can also simulate a workload by using pgbench, a simple benchmarking application from the PostgreSQL community.

The following example initializes (`-i`) `pgbench` by creating the necessary tables and rows for running the tests. In this example, the scaling factor (`-s 50`) creates 50 rows in the `pgbench_branches` table, 500 rows in `pgbench_tellers`, and 5,000,000 rows in the `pgbench_accounts` table in the `labdb` database.

```
pgbench -U postgres -h your-cluster-instance-1.111122223333.aws-regionrds.amazonaws.com
 -p 5432 -i -s 50 labdb
Password:
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
5000000 of 5000000 tuples (100%) done (elapsed 15.46 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 61.13 s (drop tables 0.08 s, create tables 0.39 s, client-side generate 54.85
 s, vacuum 2.30 s, primary keys 3.51 s)
```

After initializing the environment, you can run the benchmark for a specific time (-T) and the number of clients (-c). This example also uses the -d option to output debugging information as the transactions are processed by the Aurora PostgreSQL DB cluster.

```
pgbench -h -U postgres your-cluster-instance-1.111122223333.aws-regionrds.amazonaws.com
 -p 5432 -d -T 60 -c 10 labdb
Password:*******
pgbench (14.3)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 50
query mode: simple
number of clients: 10
number of threads: 1
duration: 60 s
number of transactions actually processed: 1408
latency average = 398.467 ms
initial connection time = 4280.846 ms
tps = 25.096201 (without initial connection time)
```

For more information about pgbench, see pgbench in the PostgreSQL documentation.

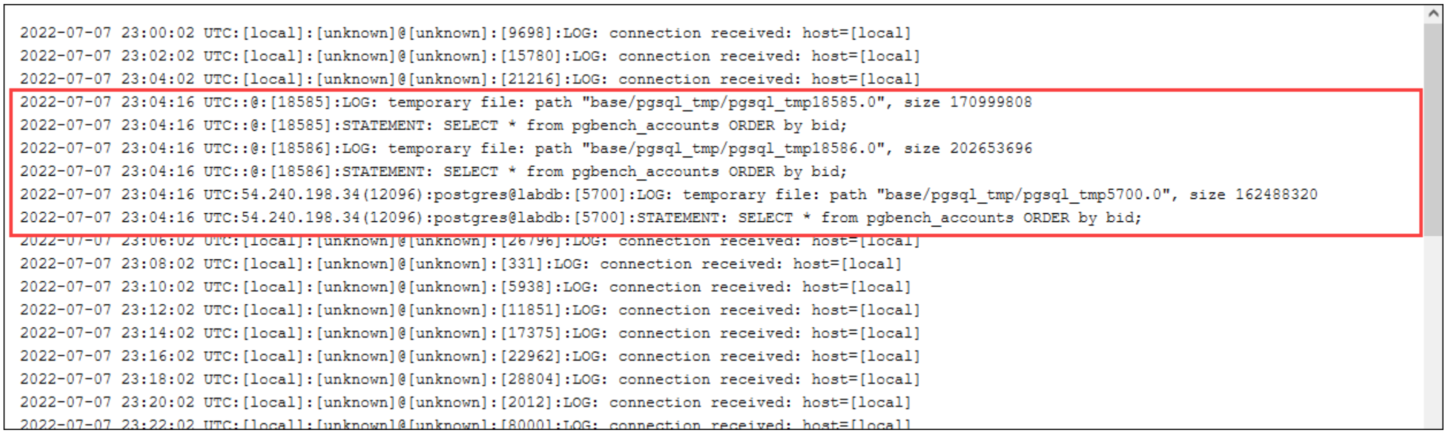
You can use the psql metacommand command (\d) to list the relations such as tables, views, and indexes created by pgbench.

```
labdb=>  \d pgbench_accounts
 Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
```

As shown in the output, the pgbench_accounts table is indexed on the aid column. To ensure that this next query uses working memory, query any nonindexed column, such as that shown in the following example.

```
postgres=> SELECT * FROM pgbench_accounts ORDER BY bid;
```

Check the log for the temporary files. To do so, open the AWS Management Console, choose the Aurora PostgreSQL DB cluster instance, and then choose the **Logs & Events** tab. View the logs in the console or download for further analysis. As shown in the following image, the size of the temporary files needed to process the query indicates that you should consider increasing the amount specified for the `work_mem` parameter.



```
2022-07-07 23:00:02 UTC:[local]:[unknown]@[unknown]:[9698]:LOG: connection received: host=[local]
2022-07-07 23:02:02 UTC:[local]:[unknown]@[unknown]:[15780]:LOG: connection received: host=[local]
2022-07-07 23:04:02 UTC:[local]:[unknown]@[unknown]:[21216]:LOG: connection received: host=[local]
2022-07-07 23:04:16 UTC::@:[18585]:LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp18585.0", size 170999808
2022-07-07 23:04:16 UTC::@:[18585]:STATEMENT: SELECT * from pgbench_accounts ORDER by bid;
2022-07-07 23:04:16 UTC::@:[18586]:LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp18586.0", size 202653696
2022-07-07 23:04:16 UTC::@:[18586]:STATEMENT: SELECT * from pgbench_accounts ORDER by bid;
2022-07-07 23:04:16 UTC:54.240.198.34(12096):postgres@labdb:[5700]:LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp5700.0", size 162488320
2022-07-07 23:04:16 UTC:54.240.198.34(12096):postgres@labdb:[5700]:STATEMENT: SELECT * from pgbench_accounts ORDER by bid;
2022-07-07 23:06:02 UTC:[local]:[unknown]@[unknown]:[26796]:LOG: connection received: host=[local]
2022-07-07 23:08:02 UTC:[local]:[unknown]@[unknown]:[331]:LOG: connection received: host=[local]
2022-07-07 23:10:02 UTC:[local]:[unknown]@[unknown]:[5938]:LOG: connection received: host=[local]
2022-07-07 23:12:02 UTC:[local]:[unknown]@[unknown]:[11851]:LOG: connection received: host=[local]
2022-07-07 23:14:02 UTC:[local]:[unknown]@[unknown]:[17375]:LOG: connection received: host=[local]
2022-07-07 23:16:02 UTC:[local]:[unknown]@[unknown]:[22962]:LOG: connection received: host=[local]
2022-07-07 23:18:02 UTC:[local]:[unknown]@[unknown]:[28804]:LOG: connection received: host=[local]
2022-07-07 23:20:02 UTC:[local]:[unknown]@[unknown]:[2012]:LOG: connection received: host=[local]
2022-07-07 23:22:02 UTC:[local]:[unknown]@[unknown]:[8000]:LOG: connection received: host=[local]
```

You can configure this parameter differently for individuals and groups, based on your operational needs. For example, you can set the `work_mem` parameter to 8 GB for the role named `dev_team`.

```
postgres=> ALTER ROLE dev_team SET work_mem='8GB';
```

With this setting for `work_mem`, any role that's a member of the `dev_team` role is allotted up to 8 GB of working memory.

Using indexes for faster response time

If your queries are taking too long to return results, you can verify that your indexes are being used as expected. First, turn on `\timing`, the psql metacommand, as follows.

```
postgres=>  \timing on
```

After turning on timing, use a simple SELECT statement.

```
postgres=> SELECT COUNT(*) FROM
  (SELECT * FROM pgbench_accounts
  ORDER BY bid)
  AS accounts;
count
-------
```

```
5000000
(1 row)
Time: 3119.049 ms (00:03.119)
```

As shown in the output, this query took just over 3 seconds to complete. To improve the response time, create an index on `pgbench_accounts`, as follows.

```
postgres=> CREATE INDEX ON pgbench_accounts(bid);
CREATE INDEX
```

Rerun the query, and notice the faster response time. In this example, the query completed about 5 times faster, in about half a second.

```
postgres=>  SELECT COUNT(*) FROM (SELECT * FROM pgbench_accounts ORDER BY bid) AS
 accounts;
 count
-------
 5000000
(1 row)
Time: 567.095 ms
```

Adjusting working memory for logical decoding

Logical replication has been available in all versions of Aurora PostgreSQL since its introduction in PostgreSQL version 10. When you configure logical replication, you can also set the `logical_decoding_work_mem` parameter to specify the amount of memory that the logical decoding process can use for the decoding and streaming process.

During logical decoding, write-ahead log (WAL) records are converted to SQL statements that are then sent to another target for logical replication or another task. When a transaction is written to the WAL and then converted, the entire transaction must fit into the value specified for `logical_decoding_work_mem`. By default, this parameter is set to 65536 KB. Any overflow is written to disk. This means that it must be reread from the disk before it can be sent to its destination, thus slowing the overall process.

You can assess the amount of transaction overflow in your current workload at a specific point in time by using the `aurora_stat_file` function as shown in the following example.

```
SELECT split_part (filename, '/', 2)
```

```
    AS slot_name, count(1) AS num_spill_files,
    sum(used_bytes) AS slot_total_bytes,
    pg_size_pretty(sum(used_bytes)) AS slot_total_size
    FROM aurora_stat_file()
    WHERE filename like '%spill%'
    GROUP BY 1;
   slot_name | num_spill_files | slot_total_bytes | slot_total_size
------------+-----------------+------------------+-----------------
 slot_name  |       590       |      411600000   | 393 MB
(1 row)
```

This query returns the count and size of spill files on your Aurora PostgreSQL DB cluster when the query is invoked. Longer running workloads might not have any spill files on disk yet. To profile long-running workloads, we recommend that you create a table to capture the spill file information as the workload runs. You can create the table as follows.

```
CREATE TABLE spill_file_tracking AS
    SELECT now() AS spill_time,*
    FROM aurora_stat_file()
    WHERE filename LIKE '%spill%';
```

To see how spill files are used during logical replication, set up a publisher and subscriber and then start a simple replication. For more information, see Setting up logical replication for your Aurora PostgreSQL DB cluster. With replication under way, you can create a job that captures the result set from the `aurora_stat_file()` spill file function, as follows.

```
INSERT INTO spill_file_tracking
  SELECT now(),*
  FROM aurora_stat_file()
  WHERE filename LIKE '%spill%';
```

Use the following psql command to run the job once per second.

```
\watch 0.5
```

As the job is running, connect to the writer instance from another psql session. Use the following series of statements to run a workload that exceeds the memory configuration and causes Aurora PostgreSQL to create a spill file.

```
labdb=> CREATE TABLE my_table (a int PRIMARY KEY, b int);
```

```
CREATE TABLE
labdb=> INSERT INTO my_table SELECT x,x FROM generate_series(0,10000000) x;
INSERT 0 10000001
labdb=> UPDATE my_table SET b=b+1;
UPDATE 10000001
```

These statements take several minutes to complete. When finished, press the Ctrl key and the C key together to stop the monitoring function. Then use the following command to create a table to hold the information about the Aurora PostgreSQL DB cluster's spill file usage.

```
SELECT spill_time, split_part (filename, '/', 2)
    AS slot_name, count(1)
    AS spills, sum(used_bytes)
    AS slot_total_bytes, pg_size_pretty(sum(used_bytes))
    AS slot_total_size FROM spill_file_tracking
  GROUP BY 1,2 ORDER BY 1;
                   spill_time | slot_name             | spills | slot_total_bytes |
 slot_total_size
------------------------------+-----------------------+--------+------------------
+-----------------
2022-04-15 13:42:52.528272+00 | replication_slot_name | 1      | 142352280        | 136
 MB
2022-04-15 14:11:33.962216+00 | replication_slot_name | 4      | 467637996        | 446
 MB
2022-04-15 14:12:00.997636+00 | replication_slot_name | 4      | 569409176        | 543
 MB
2022-04-15 14:12:03.030245+00 | replication_slot_name | 4      | 569409176        | 543
 MB
2022-04-15 14:12:05.059761+00 | replication_slot_name | 5      | 618410996        | 590
 MB
2022-04-15 14:12:07.22905+00  | replication_slot_name | 5      | 640585316        | 611
 MB
(6 rows)
```

The output shows that running the example created five spill files that used 611 MB of memory. To avoid writing to disk, we recommend setting the `logical_decoding_work_mem` parameter to the next highest memory size, 1024.

Using Amazon CloudWatch metrics to analyze resource usage for Aurora PostgreSQL

Aurora automatically sends metric data to CloudWatch in 1-minute periods. You can analyze resource usage for Aurora PostgreSQL using CloudWatch metrics. You can evaluate the network throughput and the network usage with the metrics.

Evaluating network throughput with CloudWatch

When your system usage approaches the resource limits for your instance type, the processing can slow down. You can use CloudWatch **Logs Insights** to monitor your storage resource usage and ensure that sufficient resources are available. When needed, you can modify the DB instance to a larger instance class.

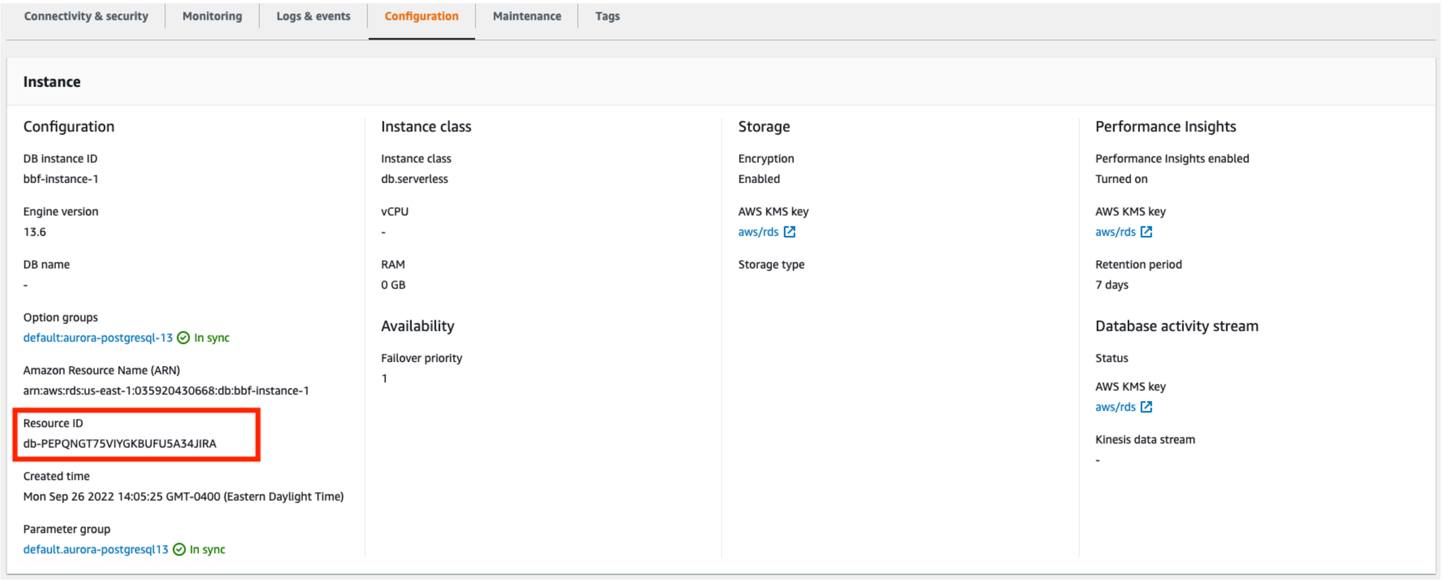
Aurora storage processing may be slow because of:

- Insufficient network bandwidth between the client and DB instance.
- Insufficient network bandwidth to the storage subsystem.
- A workload that is large for your instance type.

You can query CloudWatch **Logs Insights** to generate a graph of Aurora storage resource usage to monitor the resources. The graph shows the CPU utilization and metrics to help you decide whether to scale up to a larger instance size. For information about the query syntax for CloudWatch **Logs Insights**, see CloudWatch Logs Insights query syntax

To use CloudWatch, you need to export your Aurora PostgreSQL log files to CloudWatch. You can also modify your existing cluster to export logs to CloudWatch. For information about exporting logs to CloudWatch, see Turning on the option to publish logs to Amazon CloudWatch.

You need the **Resource ID** of your DB instance to query the CloudWatch **Logs Insights**. The **Resource ID** is available in the **Configuration** tab in your console:



Instance

Configuration

DB instance ID
bbf-instance-1

Engine version
13.6

DB name
-

Option groups
default:aurora-postgresql-13 In sync

Amazon Resource Name (ARN)
arn:aws:rds:us-east-1:035920430668:db:bbf-instance-1

Resource ID
db-PEPQNGT75VIYGKBUFU5A34JIRA

Created time
Mon Sep 26 2022 14:05:25 GMT-0400 (Eastern Daylight Time)

Parameter group
default.aurora-postgresql13 In sync

Instance class

Instance class
db.serverless

vCPU
-

RAM
0 GB

Availability

Failover priority
1

Storage

Encryption
Enabled

AWS KMS key
aws/rds

Storage type

Performance Insights

Performance Insights enabled
Turned on

AWS KMS key
aws/rds

Retention period
7 days

Database activity stream

Status

AWS KMS key
aws/rds

Kinesis data stream
-

To query your log files for resource storage metrics:

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.

   The CloudWatch overview home page appears.

2. If necessary, change the AWS Region. In the navigation bar, choose the AWS Region where your AWS resources are located. For more information, see Regions and endpoints.

3. In the navigation pane, choose **Logs** and then **Logs Insights**.

   The **Logs Insights** page appears.

4. Select the log files from the drop-down list to analyze.

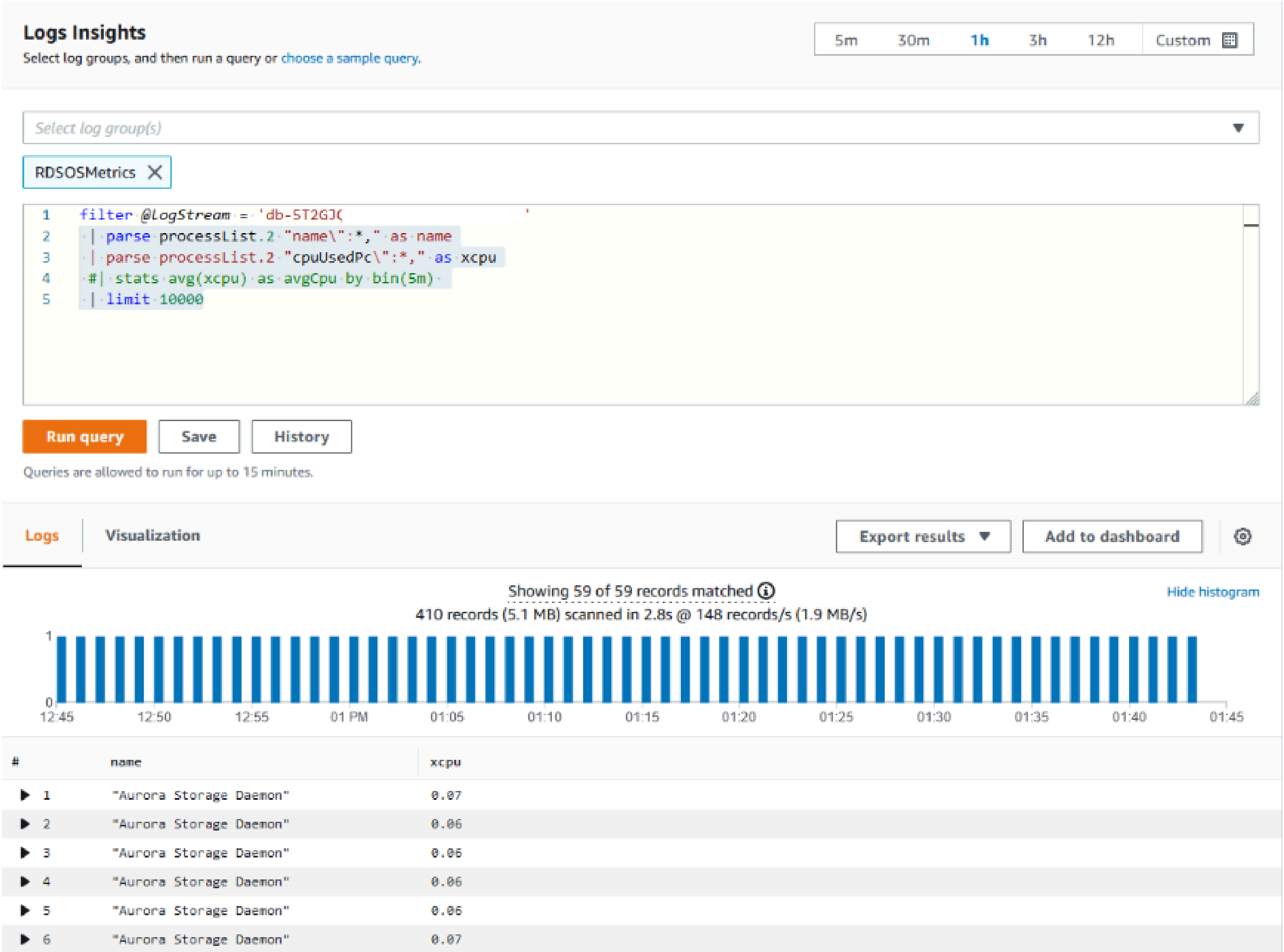
5. Enter the following query in the field, replacing `<resource ID>` with the resource ID of your DB cluster:

   ```
   filter @logStream = <resource ID> | parse @message "\"Aurora Storage
   Daemon\"*memoryUsedPc\":*,\"cpuUsedPc\":*," as a,memoryUsedPc,cpuUsedPc
   | display memoryUsedPc,cpuUsedPc #| stats avg(xcpu) as avgCpu by
   bin(5m) | limit 10000
   ```

6. Click **Run query**.

   The storage utilization graph is displayed.

   The following image provides the **Logs Insights** page and the graph display.



```
filter @LogStream = 'db-5T2GJ(                '
 | parse processList.2 "name\":*," as name
 | parse processList.2 "cpuUsedPc\":*," as xcpu
#| stats avg(xcpu) as avgCpu by bin(5m)
 | limit 10000
```

Showing 59 of 59 records matched

410 records (5.1 MB) scanned in 2.8s @ 148 records/s (1.9 MB/s)

#	name	xcpu
▶ 1	"Aurora Storage Daemon"	0.07
▶ 2	"Aurora Storage Daemon"	0.06
▶ 3	"Aurora Storage Daemon"	0.06
▶ 4	"Aurora Storage Daemon"	0.06
▶ 5	"Aurora Storage Daemon"	0.06
▶ 6	"Aurora Storage Daemon"	0.07

Evaluating DB instance usage with CloudWatch metrics

You can use CloudWatch metrics to watch your instance throughput and discover if your instance class provides sufficient resources for your applications. For information about your DB instance class limits, go to Hardware specifications for DB instance classes for Aurora and locate the specifications for your DB instance class to find your network performance.

If your DB instance usage is near the instance class limit, then performance may begin to slow. The CloudWatch metrics can confirm this situation so you can plan to manually scale-up to a larger instance class.

Combine the following CloudWatch metrics values to find out if you are nearing the instance class limit:

- **NetworkThroughput** – The amount of network throughput received and transmitted by the clients for each instance in the Aurora DB cluster. This throughput value doesn't include network traffic between instances in the DB cluster and the cluster volume.
- **StorageNetworkThroughput** – The amount of network throughput received and sent to the Aurora storage subsystem by each instance in the Aurora DB cluster.

Add the **NetworkThroughput** to the **StorageNetworkThroughput** to find the network throughput received from and sent to the Aurora storage subsystem by each instance in your Aurora DB cluster. The instance class limit for your instance should be greater than the sum of these two combined metrics.

You can use the following metrics to review additional details of the network traffic from your client applications when sending and receiving:

- **NetworkReceiveThroughput** – The amount of network throughput received from clients by each instance in the Aurora PostgreSQL DB cluster. This throughput doesn't include network traffic between instances in the DB cluster and the cluster volume.
- **NetworkTransmitThroughput** – The amount of network throughput sent to clients by each instance in the Aurora DB cluster. This throughput doesn't include network traffic between instances in the DB cluster and the cluster volume.
- **StorageNetworkReceiveThroughput** – The amount of network throughput received from the Aurora storage subsystem by each instance in the DB cluster.
- **StorageNetworkTransmitThroughput** – The amount of network throughput sent to the Aurora storage subsystem by each instance in the DB cluster.

Add all of these metrics together to evaluate how your network usage compares to the instance class limit. The instance class limit should be greater than the sum of these combined metrics.

The network limits and CPU utilization for storage are mutual. When the network throughput increases, then the CPU utilization also increases. Monitoring the CPU and network usage provides information about how and why the resources are being exhausted.

To help minimize network usage, you can consider:

- Using a larger instance class.
- Using `pg_partman` partitioning strategies.
- Dividing the write requests in batches to reduce overall transactions.

- Directing the read-only workload to a read-only instance.
- Deleting any unused indexes.
- Checking for bloated objects and VACUUM. In the case of severe bloat, use the PostgreSQL extension `pg_repack`. For more information about `pg_repack`, see Reorganize tables in PostgreSQL databases with minimal locks.

Using logical replication to perform a major version upgrade for Aurora PostgreSQL

Using logical replication and Aurora fast cloning, you can perform a major version upgrade that uses the current version of Aurora PostgreSQL database while gradually migrating the changing data to the new major version database. This low downtime upgrade process is referred to as a blue/green upgrade. The current version of the database is referred as the "blue" environment and the new database version is referred as the "green" environment.

Aurora fast cloning fully loads the existing data by taking a snapshot of the source database. Fast cloning uses a copy-on-write protocol built on top of the Aurora storage layer, which allows you to create a clone of database in a short time. This method is very effective when upgrading to a large database.

Logical replication in PostgreSQL tracks and transfers your data changes from initial instance to a new instance running in parallel until you move to the newer version of PostgreSQL. Logical replication uses a publish and subscribe model. For more information about Aurora PostgreSQL logical replication, see Replication with Amazon Aurora PostgreSQL.

Tip

You can minimize the downtime required for a major version upgrade by using the managed Amazon RDS Blue/Green Deployment feature. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.

Topics

- Requirements
- Limitations
- Setting and checking parameter values

- Upgrading Aurora PostgreSQL to a new major version
- Performing post-upgrade tasks

Requirements

You must meet the following requirements to perform this low downtime upgrade process:

- You must have rds_superuser permissions.
- The Aurora PostgreSQL DB cluster you intend to upgrade must be running a supported version that can perform major version upgrades using logical replication. Make sure to apply any minor version updates and patches to your DB cluster. The `aurora_volume_logical_start_lsn` function that is used in this technique is supported in the following versions of Aurora PostgreSQL:
  - 15.2 and higher 15 versions
  - 14.3 and higher 14 versions
  - 13.6 and higher 13 versions
  - 12.10 and higher 12 versions
  - 11.15 and higher 11 versions
  - 10.20 and higher 10 versions

  For more information on `aurora_volume_logical_start_lsn` function, see aurora_volume_logical_start_lsn.
- All of your tables must have a primary key or include a PostgreSQL identity column.
- Configure the security group for your VPC to allow inbound and outbound access between the two Aurora PostgreSQL DB clusters, both old and new. You can grant access to a specific classless inter-domain routing (CIDR) range or to another security group in your VPC or in a peer VPC. (Peer VPC requires a VPC peering connection.)

Note

For detailed information about the permissions required to configure and manage a running logical replication scenario, see the PostgreSQL core documentation.

Limitations

When you are performing low downtime upgrade on your Aurora PostgreSQL DB cluster to upgrade it to a new major version, you are using the native PostgreSQL logical replication feature. It has the same capabilities and limitations as the PostgreSQL logical replication. For more information, see PostgreSQL logical replication.

- Data definition language (DDL) commands are not replicated.
- Replication doesn't support schema changes in a live database. The schema is recreated in its original form during the cloning process. If you change the schema after cloning, but before completing the upgrade, it isn't reflected in the upgraded instance.
- Large objects are not replicated, but you can store data in normal tables.
- Replication is only supported by tables, including partitioned tables. Replication to other types of relations, such as views, materialized views, or foreign tables, is not supported.
- Sequence data is not replicated and requires a manual update post-failover.

> **Note**
>
> This upgrade doesn't support auto-scripting. You should perform all the steps manually.

Setting and checking parameter values

Before upgrading, configure the writer instance of your Aurora PostgreSQL DB cluster to act as a publication server. The instance should use a custom DB cluster parameter group with the following settings:

- `rds.logical_replication` – Set this parameter to 1. The `rds.logical_replication` parameter serves the same purpose as a standalone PostgreSQL server's `wal_level` parameter and other parameters that control the write-ahead log file management.
- `max_replication_slots` – Set this parameter to the total number of subscriptions that you plan to create. If you are using AWS DMS, set this parameter to the number of AWS DMS tasks that you plan to use for changed data capture from this DB cluster.
- `max_wal_senders` – Set to the number of concurrent connections, plus a few extra, to make available for management tasks and new sessions. If you are using AWS DMS, the number of

max_wal_senders should be equal to the number of concurrent sessions plus the number of AWS DMS tasks that may be working at any given time.

- `max_logical_replication_workers` – Set to the number of logical replication workers and table synchronization workers that you expect. It's generally safe to set the number of replication workers to the same value used for max_wal_senders. The workers are taken from the pool of background processes (max_worker_processes) allocated for the server.
- `max_worker_processes` – Set to the number of background processes for the server. This number should be large enough to allocate workers for replication, auto-vacuum processes, and other maintenance processes that may take place concurrently.

When you upgrade to a newer version of Aurora PostgreSQL, you need to duplicate any parameters that you modified in the earlier version of the parameter group. These parameters are applied to the upgraded version. You can query the `pg_settings` table to get a list of parameter settings so that you can re-create them on your new Aurora PostgreSQL DB cluster.

For example, to get the settings for replication parameters, run the following query:

```
SELECT name, setting FROM pg_settings WHERE name in
('rds.logical_replication', 'max_replication_slots',
'max_wal_senders', 'max_logical_replication_workers',
'max_worker_processes');
```

Upgrading Aurora PostgreSQL to a new major version

To prepare the publisher (blue)

1. In the example that follows, the source writer instance (blue) is an Aurora PostgreSQL DB cluster running PostgreSQL version 11.15. This is the publication node in our replication scenario. For this demonstration, our source writer instance hosts a sample table that holds a series of values:

   ```
   CREATE TABLE my_table (a int PRIMARY KEY);
   INSERT INTO my_table VALUES (generate_series(1,100));
   ```

2. To create a publication on the source instance, connect to the writer node of the instance with psql (the CLI for PostgreSQL) or with the client of your choice). Enter the following command in each database:

```
CREATE PUBLICATION publication_name FOR ALL TABLES;
```

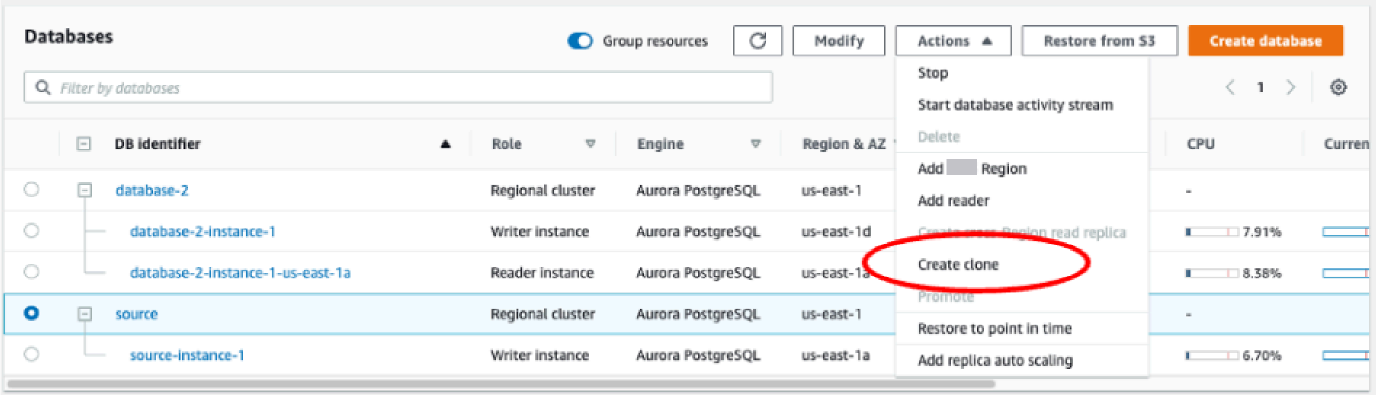
The publication_name specifies the name of the publication.

3. You also need to create a replication slot on the instance. The following command creates a replication slot and loads the `pgoutput` logical decoding plug-in. The plug-in changes content read from write-ahead logging (WAL) to the logical replication protocol, and filters the data according to the publication specification.

```
SELECT pg_create_logical_replication_slot('replication_slot_name', 'pgoutput');
```

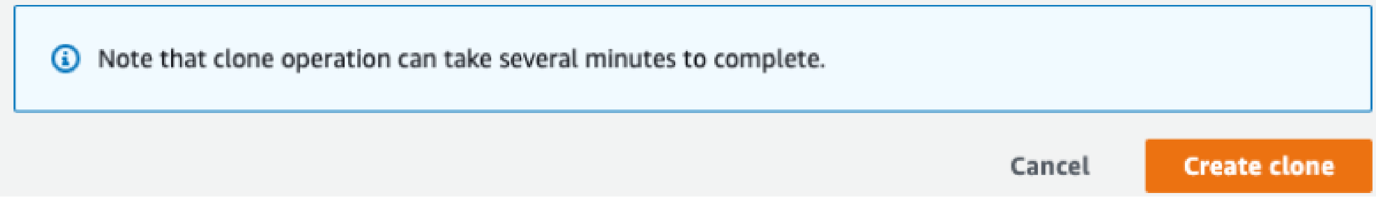
To clone the publisher

1. Use the Amazon RDS Console to create a clone of the source instance. Highlight the instance name in the Amazon RDS Console, and then choose **Create clone** in the **Actions** menu.



	DB identifier	Role	Engine	Region & AZ	CPU	Curren
○	database-2	Regional cluster	Aurora PostgreSQL	us-east-1	-	
○	database-2-instance-1	Writer instance	Aurora PostgreSQL	us-east-1d	7.91%	
○	database-2-instance-1-us-east-1a	Reader instance	Aurora PostgreSQL	us-east-1a	8.38%	
●	source	Regional cluster	Aurora PostgreSQL	us-east-1	-	
○	source-instance-1	Writer instance	Aurora PostgreSQL	us-east-1a	6.70%	

2. Provide a unique name for the instance. Most of the settings are defaults from the source instance. When you've made changes required for the new instance, choose **Create clone**.


Note that clone operation can take several minutes to complete.

3. While the target instance is initiating, the **Status** column of the writer node displays Creating in the **Status** column. When the instance is ready, the status changes to Available.

To prepare the clone for an upgrade

1. The clone is the ‘green’ instance in the deployment model. It is the host of the replication subscription node. When the node becomes available, connect with psql and query the new writer node to obtain the log sequence number (LSN). The LSN identifies the beginning of a record in the WAL stream.

   ```
   SELECT aurora_volume_logical_start_lsn();
   ```

2. In the response from the query, you find the LSN number. You need this number later in the process, so make a note of it.

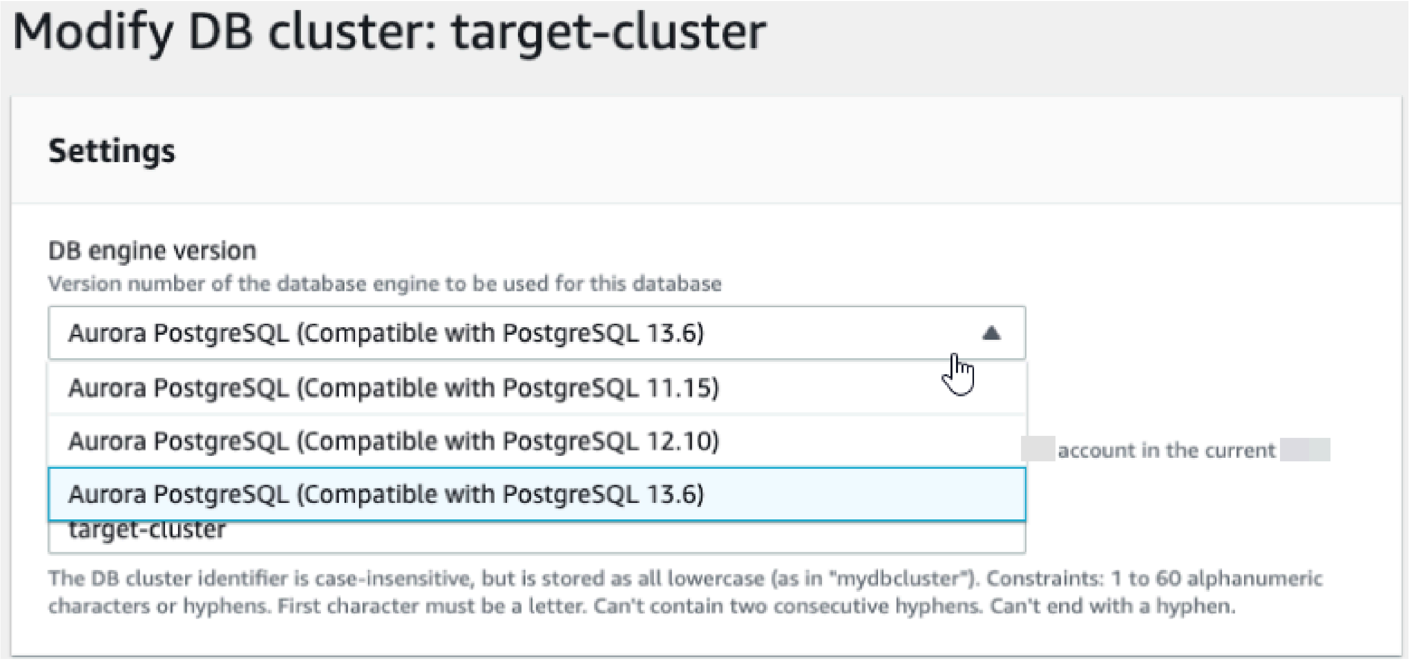
   ```
   postgres=> SELECT aurora_volume_logical_start_lsn();
   aurora_volume_logical_start_lsn
   ---------------
   0/402E2F0
   (1 row)
   ```

3. Before upgrading the clone, drop the clone's replication slot.

   ```
   SELECT pg_drop_replication_slot('replication_slot_name');
   ```

To upgrade the cluster to a new major version

- After cloning the provider node, use the Amazon RDS Console to initiate a major version upgrade on the subscription node. Highlight the instance name in the RDS console, and select the **Modify** button. Select the updated version and your updated parameter groups, and apply the settings immediately to upgrade the target instance.



Modify DB cluster: target-cluster

Settings

DB engine version
Version number of the database engine to be used for this database

Aurora PostgreSQL (Compatible with PostgreSQL 13.6)

- Aurora PostgreSQL (Compatible with PostgreSQL 11.15)
- Aurora PostgreSQL (Compatible with PostgreSQL 12.10)
- Aurora PostgreSQL (Compatible with PostgreSQL 13.6)

account in the current

target-cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

- You can also use the CLI to perform an upgrade:

```
aws rds modify-db-cluster —db-cluster-identifier $TARGET_Aurora_ID —engine-version
 13.6 —allow-major-version-upgrade —apply-immediately
```

To prepare the subscriber (green)

1. When the clone becomes available after the upgrade, connect with psql and define the subscription. To do so, you need to specify the following options in the `CREATE SUBSCRIPTION` command:

   - `subscription_name` – The name of the subscription.
   - `admin_user_name` – The name of an administrative user with rds_superuser permissions.
   - `admin_user_password` – The password associated with the administrative user.
   - `source_instance_URL` – The URL of the publication server instance.
   - `database` – The database that the subscription server will connect with.
   - `publication_name` – The name of the publication server.
   - `replication_slot_name` – The name of the replication slot.

   ```
   CREATE SUBSCRIPTION subscription_name
   ```

```
CONNECTION 'postgres://admin_user_name:admin_user_password@source_instance_URL/
database' PUBLICATION publication_name
WITH (copy_data = false, create_slot = false, enabled = false, connect = true,
 slot_name = 'replication_slot_name');
```

2. After creating the subscription, query the pg_replication_origin view to retrieve the roname value, which is the identifier of the replication origin. Each instance has one `roname`:

   ```
   SELECT * FROM pg_replication_origin;
   ```

   For example:

   ```
   postgres=>
   SELECT * FROM pg_replication_origin;

   roident | roname
   ---------+----------
   1 | pg_24586
   ```

3. Provide the LSN that you saved from the earlier query of the publication node and the `roname` returned from the subscription node [INSTANCE] in the command. This command uses the `pg_replication_origin_advance` function to specify the starting point in the log sequence for replication.

   ```
   SELECT pg_replication_origin_advance('roname', 'log_sequence_number');
   ```

   `roname` is the identifier returned by the pg_replication_origin view.

   `log_sequence_number` is the value returned by the earlier query of the `aurora_volume_logical_start_lsn` function.

4. Then, use the `ALTER SUBSCRIPTION... ENABLE` clause to turn on logical replication.

   ```
   ALTER SUBSCRIPTION subscription_name ENABLE;
   ```

5. At this point, you can confirm that replication is working. Add a value to the publication instance, then confirm that the value is replicated to the subscription node.

   Then, use the following command to monitor replication lag on the publication node:

```
SELECT now() AS CURRENT_TIME, slot_name, active, active_pid,
 pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),
confirmed_flush_lsn)) AS diff_size, pg_wal_lsn_diff(pg_current_wal_lsn(),
confirmed_flush_lsn) AS diff_bytes FROM pg_replication_slots WHERE slot_type =
 'logical';
```

For example:

```
postgres=> SELECT now() AS CURRENT_TIME, slot_name, active, active_pid,
 pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),
confirmed_flush_lsn)) AS diff_size, pg_wal_lsn_diff(pg_current_wal_lsn(),
 confirmed_flush_lsn) AS diff_bytes FROM pg_replication_slots WHERE slot_type =
 'logical';

current_time                   | slot_name              | active | active_pid |
 diff_size | diff_bytes
-------------------------------+------------------------+--------+------------
+-----------+------------
2022-04-13 15:11:00.243401+00  | replication_slot_name | t      | 21854      | 136
 bytes | 136
(1 row)
```

You can monitor the replication lag using `diff_size` and `diff_bytes` values. When these values reach 0, the replica has caught up to the source DB instance.

Performing post-upgrade tasks

When the upgrade is complete, the instance status displays as **Available** in the **Status** column of the console dashboard. On the new instance, we recommend you do the following:

- Redirect your applications to point to the writer node.
- Add reader nodes to manage the caseload and provide high-availability in the event of an issue with the writer node.
- Aurora PostgreSQL DB clusters occasionally require operating system updates. These updates might include a newer version of glibc library. During such updates, we recommend you to follow the guidelines as described in Collations supported in Aurora PostgreSQL.
- Update user permissions on the new instance to ensure access.

After testing your application and data on the new instance, we recommend that you make a final backup of your initial instance before removing it. For more information about using logical replication on an Aurora host, see Setting up logical replication for your Aurora PostgreSQL DB cluster.

Troubleshooting storage issues

If the amount of working memory needed for sort or index-creation operations exceeds the amount allocated by the `work_mem` parameter, Aurora PostgreSQL writes the excess data to temporary disk files. When it writes the data, Aurora PostgreSQL uses the same storage space that it uses for storing error and message logs, that is, *local storage*. Each instance in your Aurora PostgreSQL DB cluster has an amount of local storage available. The amount of storage is based on its DB instance class. To increase the amount of local storage, you need to modify the instance to use a larger DB instance class. For DB instance class specifications, see Hardware specifications for DB instance classes for Aurora.

You can monitor your Aurora PostgreSQL DB cluster's local storage space by watching the Amazon CloudWatch metric for `FreeLocalStorage`. This metric reports the amount of storage available to each DB instance in the Aurora DB cluster for temporary tables and logs. For more information, see Monitoring Amazon Aurora metrics with Amazon CloudWatch.

Sorting, indexing, and grouping operations start in working memory but often must be offloaded to local storage. If your Aurora PostgreSQL DB cluster runs out of local storage because of these types of operations, you can resolve the issue by taking one of the following actions.

- Increase the amount of working memory. This reduces the need to use local storage. By default, PostgreSQL allocates 4 MB for each sort, group, and index operation. To check the current working memory value for your Aurora PostgreSQL DB cluster's writer instance, connect to the instance using `psql` and run the following command.

  ```
  postgres=> SHOW work_mem;
  work_mem
  ----------
   4MB
  (1 row)
  ```

  You can increase the working memory at the session level before sort, group, and other operations, as follows.

```
SET work_mem TO '1 GB';
```

For more information about working memory, see Resource Consumption in the PostgreSQL documentation.

- Change the log retention period so that logs are stored for shorter timeframes. To learn how, see Aurora PostgreSQL database log files.

For Aurora PostgreSQL DB clusters larger than 40 TB, don't use db.t2, db.t3, or db.t4g instance classes. We recommend using the T DB instance classes only for development and test servers, or other non-production servers. For more information, see DB instance class types.

Replication with Amazon Aurora PostgreSQL

Following, you can find information about replication with Amazon Aurora PostgreSQL, including how to monitor replication.

Topics

Using Aurora Replicas

An *Aurora Replica* is an independent endpoint in an Aurora DB cluster, best used for scaling read operations and increasing availability. An Aurora DB cluster can include up to 15 Aurora Replicas located throughout the Availability Zones of the Aurora DB cluster's AWS Region.

The DB cluster volume is made up of multiple copies of the data for the DB cluster. However, the data in the cluster volume is represented as a single, logical volume to the primary writer DB instance and to Aurora Replicas in the DB cluster. For more information about Aurora Replicas, see Aurora Replicas.

Aurora Replicas work well for read scaling because they're fully dedicated to read operations on your cluster volume. The writer DB instance manages write operations. The cluster volume is shared among all instances in your Aurora PostgreSQL DB cluster. Thus, no extra work is needed to replicate a copy of the data for each Aurora Replica.

With Aurora PostgreSQL, when an Aurora Replica is deleted, its instance endpoint is removed immediately, and the Aurora Replica is removed from the reader endpoint. If there are statements running on the Aurora Replica that is being deleted, there is a three minute grace period. Existing statements can finish gracefully during the grace period. When the grace period ends, the Aurora Replica is shut down and deleted.

Aurora PostgreSQL DB clusters support Aurora Replicas in different AWS Regions, using Aurora global database. For more information, see Using Amazon Aurora global databases.

> **Note**
>
> With the improved read availability feature, if you want to reboot the Aurora Replicas in the DB cluster, you have to perform it manually. For the DB clusters created prior to this feature rebooting the writer DB instance automatically reboots the Aurora Replicas. The automatic reboot re-establishes an entry point that guarantees read/write consistency across the DB cluster.

Improving the read availability of Aurora Replicas

Aurora PostgreSQL improves the read availability in the DB cluster by continuously serving the read requests when the writer DB instance restarts or when the Aurora Replica is unable to keep up with the write traffic.

The read availability feature is available by default on the following versions of Aurora PostgreSQL:

- 15.2 and higher 15 versions
- 14.7 and higher 14 versions
- 13.10 and higher 13 versions
- 12.14 and higher 12 versions

The read availability feature is supported by Aurora global database in the following versions:

- 15.4 and higher 15 versions
- 14.9 and higher 14 versions

- 13.12 and higher 13 versions
- 12.16 and higher 12 versions

To use the read availability feature for a DB cluster created on one of these versions prior to this launch, restart the writer instance of the DB cluster.

When you modify static parameters of your Aurora PostgreSQL DB cluster, you must restart the writer instance so that the parameter changes take effect. For example, you must restart the writer instance when you set the value of `shared_buffers`. With the improved availability of Aurora Replicas, the DB cluster maintains read availability during these restarts, which reduces the impact of changes to the writer instance. The reader instances don't restart and continue to respond to the read requests. To apply static parameter changes, reboot each individual reader instance.

An Aurora PostgreSQL DB cluster's Aurora Replica can recover from replication errors such as writer restarts, failover, slow replication, and network issues by quickly recovering to in-memory database state after it reconnects with the writer. This approach allows Aurora Replica instances to reach consistency with the latest storage updates while the client database is still available.

The in-progress transactions that conflict with replication recovery might receive an error but the client can retry these transactions, after the readers catch up with the writer.

Monitoring Aurora Replicas

You can monitor the Aurora Replicas when recovering from a writer disconnect. Use the metrics below to check for the latest information about the reader instance and to track in-process read-only transactions.

- The `aurora_replica_status` function is updated to return the most up-to-date information for the reader instance when it is still connected. The last update time stamp in `aurora_replica_status` is always empty for the row corresponding to the DB instance that the query is executed on. This indicates that the reader instance has the latest data.
- When the Aurora replica disconnects from the writer instance and reconnects back, the following database event is emitted:

  ```
  Read replica has been disconnected from the writer instance and
  reconnected.
  ```
- When a read-only query is canceled due to a recovery conflict, you might see one or more of the following error messages in the database error log:

```
Canceling statement due to conflict with recovery.

User query may not have access to page data to replica disconnect.

User query might have tried to access a file that no longer exists.

When the replica reconnects, you will be able to repeat your command.
```

Limitations

The following limitations apply to Aurora Replicas with improved availability:

- Aurora Replicas of secondary DB cluster can restart if the data can't be streamed from the writer instance during replication recovery.
- Aurora Replicas don't support online replication recovery if one is already in progress and will restart.
- Aurora Replicas will restart when your DB instance is nearing the transaction ID wraparound. For more information on transaction ID wraparound, see Preventing Transaction ID Wraparound Failures.
- Aurora Replicas can restart when the replication process is blocked under certain circumstances.

Monitoring Aurora PostgreSQL replication

Read scaling and high availability depend on minimal lag time. You can monitor how far an Aurora Replica is lagging behind the writer DB instance of your Aurora PostgreSQL DB cluster by monitoring the Amazon CloudWatch `ReplicaLag` metric. Because Aurora Replicas read from the same cluster volume as the writer DB instance, the `ReplicaLag` metric has a different meaning for an Aurora PostgreSQL DB cluster. The `ReplicaLag` metric for an Aurora Replica indicates the lag for the page cache of the Aurora Replica compared to that of the writer DB instance.

For more information on monitoring RDS instances and CloudWatch metrics, see Monitoring metrics in an Amazon Aurora cluster.

Using PostgreSQL logical replication with Aurora

By using PostgreSQL's logical replication feature with your Aurora PostgreSQL DB cluster, you can replicate and synchronize individual tables rather than the entire database instance. Logical replication uses a publish and subscribe model to replicate changes from a source to one or more recipients. It works by using change records from the PostgreSQL write-ahead log (WAL). The source, or *publisher*, sends WAL data for the specified tables to one or more recipients (*subscriber*), thus replicating the changes and keeping a subscriber's table synchronized with the publisher's table. The set of changes from the publisher are identified using a *publication*. Subscribers get the changes by creating a *subscription* that defines the connection to the publisher's database and its publications. A *replication slot* is the mechanism used in this scheme to track progress of a subscription.

For Aurora PostgreSQL DB clusters, the WAL records are saved on Aurora storage. The Aurora PostgreSQL DB cluster that's acting as the publisher in a logical replication scenario reads the WAL data from Aurora storage, decodes it, and sends it to the subscriber so that the changes can be applied to the table on that instance. The publisher uses a *logical decoder* to decode the data for use by subscribers. By default, Aurora PostgreSQL DB clusters use the native PostgreSQL `pgoutput` plugin when sending data. Other logical decoders are available. For example, Aurora PostgreSQL also supports the `wal2json` plugin that converts WAL data to JSON.

As of Aurora PostgreSQL version 14.5, 13.8, 12.12, and 11.17, Aurora PostgreSQL augments the PostgreSQL logical replication process with a *write-through cache* to improve performance. The WAL transaction logs are cached locally, in a buffer, to reduce the amount of disk I/O, that is, reading from Aurora storage during logical decoding. The write-through cache is used by default whenever you use logical replication for your Aurora PostgreSQL DB cluster. Aurora provides several functions that you can use to manage the cache. For more information, see Managing the Aurora PostgreSQL logical replication write-through cache.

Logical replication is supported by all currently available Aurora PostgreSQL versions. For more information, Amazon Aurora PostgreSQL updates in the *Release Notes for Aurora PostgreSQL*.

Note

In addition to the native PostgreSQL logical replication feature introduced in PostgreSQL 10, Aurora PostgreSQL also supports the `pglogical` extension. For more information, see Using pglogical to synchronize data across instances.

For more information about PostgreSQL logical replication, see Logical replication and Logical decoding concepts in the PostgreSQL documentation.

In the following topics, you can find information about how to set up logical replication between your Aurora PostgreSQL DB clusters.

Topics

Setting up logical replication for your Aurora PostgreSQL DB cluster

Setting up logical replication requires `rds_superuser` privileges. Your Aurora PostgreSQL DB cluster must be configured to use a custom DB cluster parameter group so that you can set the necessary parameters as detailed in the procedure following. For more information, see Working with DB cluster parameter groups.

To set up PostgreSQL logical replication for an Aurora PostgreSQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose your Aurora PostgreSQL DB cluster.
3. Open the **Configuration** tab. Among the Instance details, find the **Parameter group** link with **DB cluster parameter group** for **Type**.
4. Choose the link to open the custom parameters associated with your Aurora PostgreSQL DB cluster.
5. In the **Parameters** search field, type `rds` to find the `rds.logical_replication` parameter. The default value for this parameter is `0`, meaning that it's turned off by default.
6. Choose **Edit parameters** to access the property values, and then choose 1 from the selector to turn on the feature. Depending on your expected usage, you might also need to change the settings for the following parameters. However, in many cases, the default values are sufficient.

   - `max_replication_slots` – Set this parameter to a value that's at least equal to your planned total number of logical replication publications and subscriptions. If you are using

AWS DMS, this parameter should equal at least your planned change data capture tasks from the cluster, plus logical replication publications and subscriptions.

- `max_wal_senders` and `max_logical_replication_workers` – Set these parameters to a value that's at least equal to the number of logical replication slots that you intend to be active, or the number of active AWS DMS tasks for change data capture. Leaving a logical replication slot inactive prevents the vacuum from removing obsolete tuples from tables, so we recommend that you monitor replication slots and remove inactive slots as needed.
- `max_worker_processes` – Set this parameter to a value that's at least equal to the total of the `max_logical_replication_workers`, `autovacuum_max_workers`, and `max_parallel_workers` values. On small DB instance classes, background worker processes can affect application workloads, so monitor the performance of your database if you set `max_worker_processes` higher than the default value. (The default value is the result of `GREATEST(${DBInstanceVCPU*2},8}`, which means that, by default, this is either 8 or twice the CPU equivalent of the DB instance class, whichever is greater).

> **Note**
>
> You can modify parameter values in a customer-created DB parameter group. you can't change the parameter values in a default DB parameter group.

7. Choose **Save changes**.
8. Reboot the writer instance of your Aurora PostgreSQL DB cluster so that your changes takes effect. In the Amazon RDS console, choose the primary DB instance of the cluster and choose **Reboot** from the **Actions** menu.
9. When the instance is available, you can verify that logical replication is turned on, as follows.

    a. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster.

    ```
    psql --host=your-db-cluster-instance-1.aws-region.rds.amazonaws.com --port=5432
     --username=postgres --password --dbname=labdb
    ```

    b. Verify that logical replication has been enabled by using the following command.

    ```
    labdb=> SHOW rds.logical_replication;
     rds.logical_replication
    -------------------------
     on
    ```

```
(1 row)
```

c. Verify that the `wal_level` is set to `logical`.

```
labdb=> SHOW wal_level;
  wal_level
-----------
 logical
(1 row)
```

For an example of using logical replication to keep a database table synchronized with changes from a source Aurora PostgreSQL DB cluster, see Example: Using logical replication with Aurora PostgreSQL DB clusters.

Turning off logical replication

After completing your replication tasks, you should stop the replication process, drop replication slots, and turn off logical replication. Before dropping slots, make sure that they're no longer needed. Active replication slots can't be dropped.

To turn off logical replication

1. Drop all replication slots.

   To drop all of the replication slots, connect to the publisher and run the following SQL command.

   ```
   SELECT pg_drop_replication_slot(slot_name)
     FROM pg_replication_slots
    WHERE slot_name IN (SELECT slot_name FROM pg_replication_slots);
   ```

   The replication slots can't be active when you run this command.

2. Modify the custom DB cluster parameter group associated with the publisher as detailed in Setting up logical replication for your Aurora PostgreSQL DB cluster, but set the `rds.logical_replication` parameter to 0.

   For more information about custom parameter groups, see Modifying parameters in a DB cluster parameter group.

3. Restart the publisher Aurora PostgreSQL DB cluster for the change to the `rds.logical_replication` parameter to take effect.

Managing the Aurora PostgreSQL logical replication write-through cache

By default, Aurora PostgreSQL versions 14.5, 13.8, 12.12, and 11.17 and higher use a write-through cache to improve the performance for logical replication. Without the write-through cache, Aurora PostgreSQL uses the Aurora storage layer in its implementation of the native PostgreSQL logical replication process. It does so by writing WAL data to storage and then reading the data back from storage to decode it and send (replicate) to its targets (subscribers). This can result in bottlenecks during logical replication for Aurora PostgreSQL DB clusters.

The write-through cache reduces the need to use the Aurora storage layer. Instead of always writing and reading from the Aurora storage layer, Aurora PostgreSQL uses a buffer to cache the logical WAL stream so that it can be used during the replication process, rather than always pulling from disk. This buffer is the PostgreSQL native cache used by logical replication, identified in Aurora PostgreSQL DB cluster parameters as `rds.logical_wal_cache`. By default, this cache uses 1/32 of the Aurora PostgreSQL DB cluster's buffer cache setting (`shared_buffers`) but not less than 64kB nor more than the size of one WAL segment, typically 16MB.

When you use logical replication with your Aurora PostgreSQL DB cluster (for the versions that support the write-through cache), you can monitor the cache hit ratio to see how well it's working for your use case. To do so, connect to your Aurora PostgreSQL DB cluster's write instance using `psql` and then use the Aurora function, `aurora_stat_logical_wal_cache`, as shown in the following example.

```
SELECT * FROM aurora_stat_logical_wal_cache();
```

The function returns output such as the following.

```
name       | active_pid | cache_hit | cache_miss | blks_read | hit_rate |
 last_reset_timestamp
-----------+------------+-----------+------------+-----------+----------+--------------
test_slot1 | 79183      | 24        | 0          | 24        | 100.00%  | 2022-08-05
 17:39...
test_slot2 |            | 1         | 0          |  1        | 100.00%  | 2022-08-05
 17:34...
(2 rows)
```

The last_reset_timestamp values have been shortened for readability. For more information about this function, see aurora_stat_logical_wal_cache.

Aurora PostgreSQL provides the following two functions for monitoring the write-through cache.

- The aurora_stat_logical_wal_cache function – For reference documentation, see aurora_stat_logical_wal_cache.
- The aurora_stat_reset_wal_cache function – For reference documentation, see aurora_stat_reset_wal_cache.

If you find that the automatically adjusted WAL cache size isn't sufficient for your workloads, you can change the the value of the rds.logical_wal_cache manually, by modifying the parameter in your custom DB cluster parameter group. Note that any positive value less than 32kB is treated as 32kB. For more information about the wal_buffers, see Write Ahead Log in the PostgreSQL documentation.

Managing logical slots for Aurora PostgreSQL

Streaming activity is captured in the pg_replication_origin_status view. To see the contents of this view, you can use the pg_show_replication_origin_status() function, as shown following:

```
SELECT * FROM pg_show_replication_origin_status();
```

You can get a list of your logical slots by using the following SQL query.

```
SELECT * FROM pg_replication_slots;
```

To drop a logical slot, use the pg_drop_replication_slot with the name of the slot, as shown in the following command.

```
SELECT pg_drop_replication_slot('test_slot');
```

Example: Using logical replication with Aurora PostgreSQL DB clusters

The following procedure shows you how to start logical replication between two Aurora PostgreSQL DB clusters. Both the publisher and the subscriber must be configured for logical replication as detailed in Setting up logical replication for your Aurora PostgreSQL DB cluster.

The Aurora PostgreSQL DB cluster that's the designated publisher must also allow access to the replication slot. To do so, modify the security group associated with the Aurora PostgreSQL DB cluster 's virtual public cloud (VPC) based on the Amazon VPC service. Allow inbound access by adding the security group associated with the subscriber's VPC to the publisher's security group. For more information, see Control traffic to resources using security groups in the *Amazon VPC User Guide*.

With these preliminary steps complete, you can use PostgreSQL commands `CREATE PUBLICATION` on the publisher and the `CREATE SUBSCRIPTION` on the subscriber, as detailed in the following procedure.

To start the logical replication process between two Aurora PostgreSQL DB clusters

These steps assume that your Aurora PostgreSQL DB clusters have a writer instance with a database in which to create the example tables.

1. **On the publisher Aurora PostgreSQL DB cluster**

   a. Create a table using the following SQL statement.

   ```
   CREATE TABLE LogicalReplicationTest (a int PRIMARY KEY);
   ```

   b. Insert data into the publisher database by using the following SQL statement.

   ```
   INSERT INTO LogicalReplicationTest VALUES (generate_series(1,10000));
   ```

   c. Verify that data exists in the table by using the following SQL statement.

   ```
   SELECT count(*) FROM LogicalReplicationTest;
   ```

   d. Create a publication for this table by using the `CREATE PUBLICATION` statement, as follows.

   ```
   CREATE PUBLICATION testpub FOR TABLE LogicalReplicationTest;
   ```

2. **On the subscriber Aurora PostgreSQL DB cluster**

   a. Create the same `LogicalReplicationTest` table on the subscriber that you created on the publisher, as follows.

```
CREATE TABLE LogicalReplicationTest (a int PRIMARY KEY);
```

b. Verify that this table is empty.

```
SELECT count(*) FROM LogicalReplicationTest;
```

c. Create a subscription to get the changes from the publisher. You need to use the following details about the publisher Aurora PostgreSQL DB cluster.

- **host –** The publisher Aurora PostgreSQL DB cluster's writer DB instance.
- **port –** The port on which the writer DB instance is listening. The default for PostgreSQL is 5432.
- **dbname –** The name of the database.

```
CREATE SUBSCRIPTION testsub CONNECTION
   'host=publisher-cluster-writer-endpoint port=5432 dbname=db-name user=user
 password=password'
   PUBLICATION testpub;
```

Note

Specify a password other than the prompt shown here as a security best practice.

After the subscription is created, a logical replication slot is created at the publisher.

d. To verify for this example that the initial data is replicated on the subscriber, use the following SQL statement on the subscriber database.

```
SELECT count(*) FROM LogicalReplicationTest;
```

Any further changes on the publisher are replicated to the subscriber.

Logical replication affects performance. We recommend that you turn off logical replication after your replication tasks are complete.

Example: Logical replication using Aurora PostgreSQL and AWS Database Migration Service

You can use the AWS Database Migration Service (AWS DMS) to replicate a database or a portion of a database. Use AWS DMS to migrate your data from an Aurora PostgreSQL database to another open source or commercial database. For more information about AWS DMS, see the AWS Database Migration Service User Guide.

The following example shows how to set up logical replication from an Aurora PostgreSQL database as the publisher and then use AWS DMS for migration. This example uses the same publisher and subscriber that were created in Example: Using logical replication with Aurora PostgreSQL DB clusters.

To set up logical replication with AWS DMS, you need details about your publisher and subscriber from Amazon RDS. In particular, you need details about the publisher's writer DB instance and the subscriber's DB instance.

Get the following information for the publisher's writer DB instance:

- The virtual private cloud (VPC) identifier
- The subnet group
- The Availability Zone (AZ)
- The VPC security group
- The DB instance ID

Get the following information for the subscriber's DB instance:

- The DB instance ID
- The source engine

To use AWS DMS for logical replication with Aurora PostgreSQL

1. Prepare the publisher database to work with AWS DMS.

   To do this, PostgreSQL 10.x and later databases require that you apply AWS DMS wrapper functions to the publisher database. For details on this and later steps, see the instructions in Using PostgreSQL version 10.x and later as a source for AWS DMS in the *AWS Database Migration Service User Guide.*

2. Sign in to the AWS Management Console and open the AWS DMS console at https://console.aws.amazon.com/dms/v2. At top right, choose the same AWS Region in which the publisher and subscriber are located.

3. Create an AWS DMS replication instance.

   Choose values that are the same as for your publisher's writer DB instance. These include the following settings:

   - For **VPC**, choose the same VPC as for the writer DB instance.
   - For **Replication Subnet Group**, choose a subnet group with the same values as the writer DB instance. Create a new one if necessary.
   - For **Availability zone**, choose the same zone as for the writer DB instance.
   - For **VPC Security Group**, choose the same group as for the writer DB instance.

4. Create an AWS DMS endpoint for the source.

   Specify the publisher as the source endpoint by using the following settings:

   - For **Endpoint type**, choose **Source endpoint**.
   - Choose **Select RDS DB Instance**.
   - For **RDS Instance**, choose the DB identifier of the publisher's writer DB instance.
   - For **Source engine**, choose **postgres**.

5. Create an AWS DMS endpoint for the target.

   Specify the subscriber as the target endpoint by using the following settings:

   - For **Endpoint type**, choose **Target endpoint**.
   - Choose **Select RDS DB Instance**.
   - For **RDS Instance**, choose the DB identifier of the subscriber DB instance.
   - Choose a value for **Source engine**. For example, if the subscriber is an RDS PostgreSQL database, choose **postgres**. If the subscriber is an Aurora PostgreSQL database, choose **aurora-postgresql**.

6. Create an AWS DMS database migration task.

   You use a database migration task to specify what database tables to migrate, to map data using the target schema, and to create new tables on the target database. At a minimum, use the following settings for **Task configuration**:

- For **Replication instance**, choose the replication instance that you created in an earlier step.
- For **Source database endpoint**, choose the publisher source that you created in an earlier step.
- For **Target database endpoint**, choose the subscriber target that you created in an earlier step.

The rest of the task details depend on your migration project. For more information about specifying all the details for DMS tasks, see Working with AWS DMS tasks in the *AWS Database Migration Service User Guide.*

After AWS DMS creates the task, it begins migrating data from the publisher to the subscriber.

Using Aurora PostgreSQL as a Knowledge Base for Amazon Bedrock

From Aurora PostgreSQL 15.4, 14.9, 13.12, 12.16 versions, you can use the Aurora PostgreSQL DB cluster as a Knowledge Base for Amazon Bedrock. For more information, see Create a vector store in Amazon Aurora. A Knowledge Base automatically takes unstructured text data stored in a Amazon S3 bucket, converts it to text chunks and vectors, and stores it in a PostgreSQL database. With the generative AI applications, you can use Agents for Amazon Bedrock to query the data stored in the Knowledge Base and use the results of those queries to augment answers provided by foundational models. This workflow is called Retrieval Augmented Generation (RAG). For more information on RAG, see Retrieval Augmented Generation (RAG).

Topics

Prerequisites

Familiarize yourself with the following prerequisites to use Aurora PostgreSQL cluster as a Knowledge Base for Amazon Bedrock. At a high-level, you need to configure the following services for use with Bedrock:

- Amazon Aurora PostgreSQL DB cluster created in the following versions:
  - 15.4 and higher versions
  - 14.9 and higher versions
  - 13.12 and higher versions
  - 12.16 and higher versions

> **Note**
>
> You must enable the `pgvector` extension in your target database and use version 0.5.0 or higher. For more information, see pgvector v0.5.0 with HNSW indexing.

- Data API
- A user managed in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.

Preparing Aurora PostgreSQL to be used as a Knowledge Base for Amazon Bedrock

You need to follow the steps below to create and configure an Aurora PostgreSQL DB cluster to use it as a Knowledge Base for Amazon Bedrock.

1. Create an Aurora PostgreSQL DB cluster. For more information, see Creating and connecting to an Aurora PostgreSQL DB cluster
2. Enable Data API while creating Aurora PostgreSQL DB cluster. For more information on the versions supported, see Using RDS Data API.
3. Note the Aurora PostgreSQL DB cluster Amazon Resource Names (ARN) to use it in the Amazon Bedrock. For more information, see Amazon Resource Names (ARNs)
4. Log in to the database with your master user and setup pgvector. Use the following command if the extension is not installed:

   ```
   CREATE EXTENSION IF NOT EXISTS vector;
   ```

   Use `pgvector` 0.5.0 and higher version that supports HNSW indexing. For more information, see pgvector v0.5.0 with HNSW indexing.

Use the following command to check the version of the pg_vector installed:

```
postgres=>SELECT extversion FROM pg_extension WHERE extname='vector';
```

5. Create a specific schema that Bedrock can use to query the data. Use the following command to create a schema:

```
CREATE SCHEMA bedrock_integration;
```

6. Create a new role that Bedrock can use to query the database. Use the following command to create a new role:

```
CREATE ROLE bedrock_user WITH PASSWORD password LOGIN;
```

Note

Make a note of this password as you would be using the same to create a Secrets Manager password.

7. To grant the bedrock_user permission to manage the bedrock_integration schema, so they can create tables or indexes in it.

```
GRANT ALL ON SCHEMA bedrock_integration to bedrock_user;
```

8. Login as the bedrock_user and create a table in the bedrock_integration schema.

```
CREATE TABLE bedrock_integration.bedrock_kb (id uuid PRIMARY KEY, embedding
 vector(1536), chunks text, metadata json);
```

9. We recommend you to create an index with the cosine operator which the bedrock can use to query the data.

```
CREATE INDEX on bedrock_integration.bedrock_kb USING hnsw (embedding
 vector_cosine_ops);
```

10. Create an AWS Secrets Manager database secret. For more information, see AWS Secrets Manager database secret.

Creating a knowledge base in the Bedrock console

While preparing Aurora PostgreSQL to be used as the vector store for a Knowledge Base, gather the following details that you need to supply to Amazon Bedrock console.

- Amazon Aurora DB cluster ARN
- Secret ARN
- Database name (e.g. postgres)
- Table name - Advise to provide a schema qualified name, ie. CREATE TABLE bedrock_integration.bedrock_kb; which will create the bedrock_kb table in the bedrock_integration schema
- Table fields:

  ID: (id)

  Text chunks (chunks)

  Vector embedding (embedding)

  Metadata (metadata)

With these details you can create a knowledge base in the Bedrock console. For more information, see Create a vector store in Amazon Aurora.

Once Aurora is added as a knowledge base, you ingest the data sources into it. For more information, see Ingest your data sources into the knowledge base.

Integrating Amazon Aurora PostgreSQL with other AWS services

Amazon Aurora integrates with other AWS services so that you can extend your Aurora PostgreSQL DB cluster to use additional capabilities in the AWS Cloud. Your Aurora PostgreSQL DB cluster can use AWS services to do the following:

- Quickly collect, view, and assess performance for your Aurora PostgreSQL DB instances with Amazon RDS Performance Insights. Performance Insights expands on existing Amazon RDS monitoring features to illustrate your database's performance and help you analyze any issues that affect it. With the Performance Insights dashboard, you can visualize the database load and filter the load by waits, SQL statements, hosts, or users. For more information about Performance Insights, see Monitoring DB load with Performance Insights on Amazon Aurora.

- Configure your Aurora PostgreSQL DB cluster to publish log data to Amazon CloudWatch Logs. CloudWatch Logs provide highly durable storage for your log records. With CloudWatch Logs, you can perform real-time analysis of the log data, and use CloudWatch to create alarms and view metrics. For more information, see Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs.
- Import data from an Amazon S3 bucket to an Aurora PostgreSQL DB cluster, or export data from an Aurora PostgreSQL DB cluster to an Amazon S3 bucket. For more information, see Importing data from Amazon S3 into an Aurora PostgreSQL DB cluster and Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3.
- Add machine learning-based predictions to database applications using the SQL language. Aurora machine learning uses a highly optimized integration between the Aurora database and the AWS machine learning (ML) services SageMaker and Amazon Comprehend. For more information, see Using Amazon Aurora machine learning with Aurora PostgreSQL.
- Invoke AWS Lambda functions from an Aurora PostgreSQL DB cluster. To do this, use the `aws_lambda` PostgreSQL extension provided with Aurora PostgreSQL. For more information, see Invoking an AWS Lambda function from an Aurora PostgreSQL DB cluster.
- Integrate queries from Amazon Redshift and Aurora PostgreSQL. For more information, see Getting started with using federated queries to PostgreSQL in the *Amazon Redshift Database Developer Guide*.

Importing data from Amazon S3 into an Aurora PostgreSQL DB cluster

You can import data that's been stored using Amazon Simple Storage Service into a table on an Aurora PostgreSQL DB cluster instance. To do this, you first install the Aurora PostgreSQL `aws_s3` extension. This extension provides the functions that you use to import data from an Amazon S3 bucket. A *bucket* is an Amazon S3 container for objects and files. The data can be in a comma-separate value (CSV) file, a text file, or a compressed (gzip) file. Following, you can learn how to install the extension and how to import data from Amazon S3 into a table.

Your database must be running PostgreSQL version 10.7 or higher to import from Amazon S3 into Aurora PostgreSQL.

If you don't have data stored on Amazon S3, you need to first create a bucket and store the data. For more information, see the following topics in the *Amazon Simple Storage Service User Guide*.

- Create a bucket
- Add an object to a bucket

Cross-account import from Amazon S3 is supported. For more information, see Granting cross-account permissions in the *Amazon Simple Storage Service User Guide*.

You can use the customer managed key for encryption while importing data from S3. For more information, see KMS keys stored in AWS KMS in the *Amazon Simple Storage Service User Guide*.

Note

Importing data from Amazon S3 isn't supported for Aurora Serverless v1. It is supported for Aurora Serverless v2.

Topics

Installing the aws_s3 extension

Before you can use Amazon S3 with your Aurora PostgreSQL DB cluster, you need to install the `aws_s3` extension. This extension provides functions for importing data from an Amazon S3. It also provides functions for exporting data from an instance of an Aurora PostgreSQL DB cluster to an Amazon S3 bucket. For more information, see Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3. The `aws_s3` extension depends on some of the helper functions in the `aws_commons` extension, which is installed automatically when needed.

To install the aws_s3 extension

1. Use psql (or pgAdmin) to connect to the writer instance of your Aurora PostgreSQL DB cluster as a user that has `rds_superuser` privileges. If you kept the default name during the setup process, you connect as `postgres`.

```
psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password
```

2. To install the extension, run the following command.

```
postgres=> CREATE EXTENSION aws_s3 CASCADE;
NOTICE: installing required extension "aws_commons"
CREATE EXTENSION
```

3. To verify that the extension is installed, you can use the psql \dx metacommand.

```
postgres=> \dx
       List of installed extensions
    Name     | Version |   Schema   |                 Description
-------------+---------+------------+---------------------------------------------
 aws_commons | 1.2     | public     | Common data types across AWS services
 aws_s3      | 1.1     | public     | AWS S3 extension for importing data from S3
 plpgsql     | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

The functions for importing data from Amazon S3 and exporting data to Amazon S3 are now available to use.

Overview of importing data from Amazon S3 data

To import S3 data into Aurora PostgreSQL

First, gather the details that you need to supply to the function. These include the name of the table on your Aurora PostgreSQL DB cluster's instance, and the bucket name, file path, file type, and AWS Region where the Amazon S3 data is stored. For more information, see View an object in the *Amazon Simple Storage Service User Guide*.

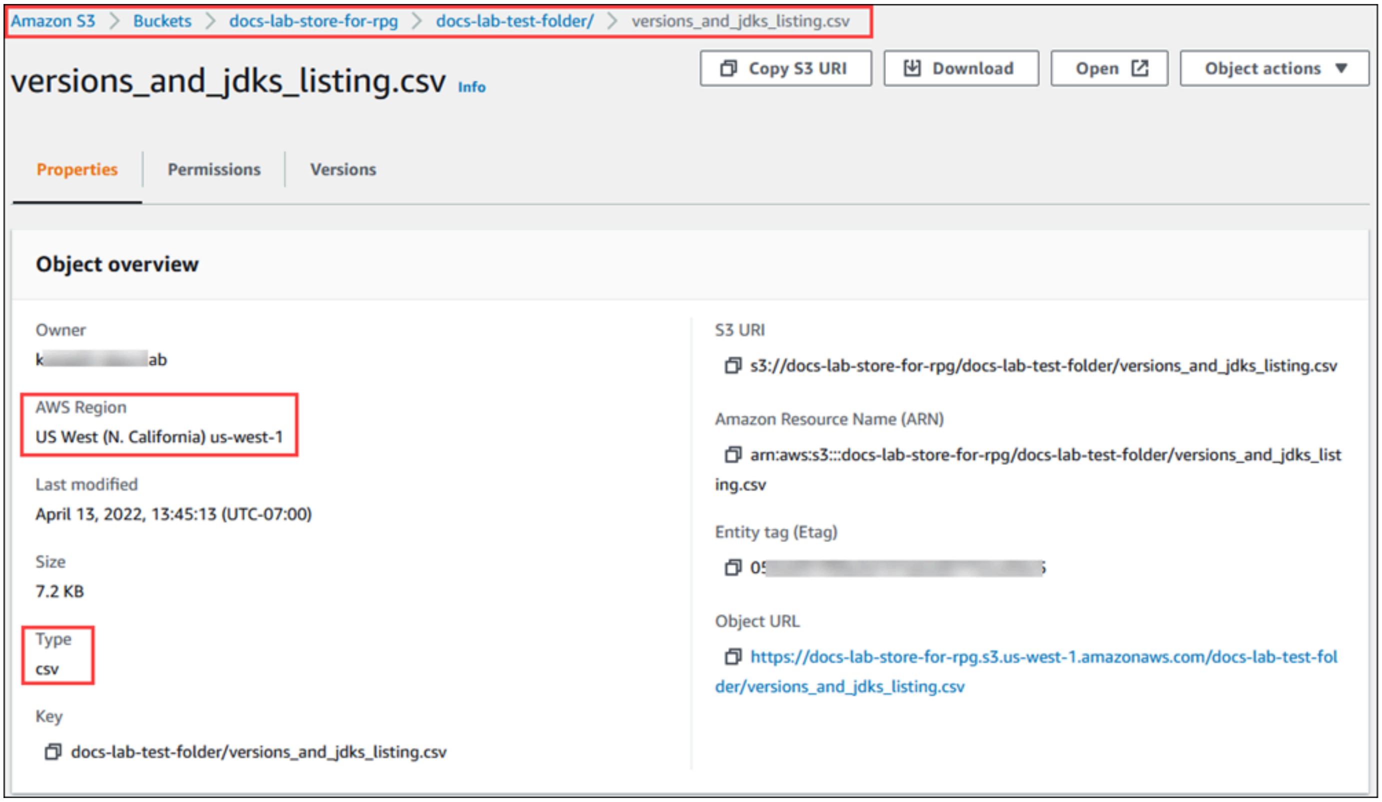
> **Note**
>
> Multi part data import from Amazon S3 isn't currently supported.

1. Get the name of the table into which the `aws_s3.table_import_from_s3` function is to import the data. As an example, the following command creates a table `t1` that can be used in later steps.

```
postgres=> CREATE TABLE t1
    (col1 varchar(80),
    col2 varchar(80),
    col3 varchar(80));
```

2. Get the details about the Amazon S3 bucket and the data to import. To do this, open the Amazon S3 console at https://console.aws.amazon.com/s3/, and choose **Buckets**. Find the bucket containing your data in the list. Choose the bucket, open its Object overview page, and then choose Properties.

   Make a note of the bucket name, path, the AWS Region, and file type. You need the Amazon Resource Name (ARN) later, to set up access to Amazon S3 through an IAM role. For more more information, see Setting up access to an Amazon S3 bucket. The image following shows an example.


Amazon S3 > Buckets > docs-lab-store-for-rpg > docs-lab-test-folder/ > versions_and_jdks_listing.csv
versions_and_jdks_listing.csv Info
Copy S3 URI
Download
Open
Object actions ▼
Properties
Permissions
Versions

Object overview

Owner
[redacted]

AWS Region
US West (N. California) us-west-1

Last modified
April 13, 2022, 13:45:13 (UTC-07:00)

Size
7.2 KB

Type
csv

Key
docs-lab-test-folder/versions_and_jdks_listing.csv

S3 URI
s3://docs-lab-store-for-rpg/docs-lab-test-folder/versions_and_jdks_listing.csv

Amazon Resource Name (ARN)
arn:aws:s3:::docs-lab-store-for-rpg/docs-lab-test-folder/versions_and_jdks_listing.csv

Entity tag (Etag)
[redacted]

Object URL
https://docs-lab-store-for-rpg.s3.us-west-1.amazonaws.com/docs-lab-test-folder/versions_and_jdks_listing.csv

3. You can verify the path to the data on the Amazon S3 bucket by using the AWS CLI command `aws s3 cp`. If the information is correct, this command downloads a copy of the Amazon S3 file.

```
aws s3 cp s3://DOC-EXAMPLE-BUCKET/sample_file_path ./
```

4. Set up permissions on your Aurora PostgreSQL DB cluster to allow access to the file on the Amazon S3 bucket. To do so, you use either an AWS Identity and Access Management (IAM) role or security credentials. For more information, see Setting up access to an Amazon S3 bucket.

5. Supply the path and other Amazon S3 object details gathered (see step 2) to the `create_s3_uri` function to construct an Amazon S3 URI object. To learn more about this function, see aws_commons.create_s3_uri. The following is an example of constructing this object during a psql session.

   ```
   postgres=> SELECT aws_commons.create_s3_uri(
      'docs-lab-store-for-rpg',
      'versions_and_jdks_listing.csv',
      'us-west-1'
   ) AS s3_uri \gset
   ```

   In the next step, you pass this object (`aws_commons._s3_uri_1`) to the `aws_s3.table_import_from_s3` function to import the data to the table.

6. Invoke the `aws_s3.table_import_from_s3` function to import the data from Amazon S3 into your table. For reference information, see aws_s3.table_import_from_s3. For examples, see Importing data from Amazon S3 to your Aurora PostgreSQL DB cluster.

Setting up access to an Amazon S3 bucket

To import data from an Amazon S3 file, give the Aurora PostgreSQL DB cluster permission to access the Amazon S3 bucket containing the file. You provide access to an Amazon S3 bucket in one of two ways, as described in the following topics.

Topics

Using an IAM role to access an Amazon S3 bucket

Before you load data from an Amazon S3 file, give your Aurora PostgreSQL DB cluster permission to access the Amazon S3 bucket the file is in. This way, you don't have to manage additional credential information or provide it in the aws_s3.table_import_from_s3 function call.

To do this, create an IAM policy that provides access to the Amazon S3 bucket. Create an IAM role and attach the policy to the role. Then assign the IAM role to your DB cluster.

> **Note**
>
> You can't associate an IAM role with an Aurora Serverless v1 DB cluster, so the following steps don't apply.

To give an Aurora PostgreSQL DB cluster access to Amazon S3 through an IAM role

1. Create an IAM policy.

   This policy provides the bucket and object permissions that allow your Aurora PostgreSQL DB cluster to access Amazon S3.

   Include in the policy the following required actions to allow the transfer of files from an Amazon S3 bucket to Aurora PostgreSQL:

   - `s3:GetObject`
   - `s3:ListBucket`

   Include in the policy the following resources to identify the Amazon S3 bucket and objects in the bucket. This shows the Amazon Resource Name (ARN) format for accessing Amazon S3.

   - arn:aws:s3:::*DOC-EXAMPLE-BUCKET*
   - arn:aws:s3:::*DOC-EXAMPLE-BUCKET*/*

   For more information on creating an IAM policy for Aurora PostgreSQL, see Creating and using an IAM policy for IAM database access. See also Tutorial: Create and attach your first customer managed policy in the *IAM User Guide*.

The following AWS CLI command creates an IAM policy named `rds-s3-import-policy` with these options. It grants access to a bucket named *`DOC-EXAMPLE-BUCKET`*.

Note

Make a note of the Amazon Resource Name (ARN) of the policy returned by this command. You need the ARN in a subsequent step when you attach the policy to an IAM role.

Example

For Linux, macOS, or Unix:

```
aws iam create-policy \
   --policy-name rds-s3-import-policy \
   --policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "s3import",
         "Action": [
           "s3:GetObject",
           "s3:ListBucket"
         ],
         "Effect": "Allow",
         "Resource": [
           "arn:aws:s3:::DOC-EXAMPLE-BUCKET",
           "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
         ]
       }
     ]
   }'
```

For Windows:

```
aws iam create-policy ^
   --policy-name rds-s3-import-policy ^
   --policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "s3import",
         "Action": [
           "s3:GetObject",
           "s3:ListBucket"
         ],
         "Effect": "Allow",
         "Resource": [
           "arn:aws:s3:::DOC-EXAMPLE-BUCKET",
           "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
         ]
       }
     ]
   }'
```

2. Create an IAM role.

   You do this so Aurora PostgreSQL can assume this IAM role to access your Amazon S3 buckets. For more information, see Creating a role to delegate permissions to an IAM user in the *IAM User Guide*.

   We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource-based policies to limit the service's permissions to a specific resource. This is the most effective way to protect against the confused deputy problem.

   If you use both global condition context keys and the `aws:SourceArn` value contains the account ID, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

   - Use `aws:SourceArn` if you want cross-service access for a single resource.
   - Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

   In the policy, be sure to use the `aws:SourceArn` global condition context key with the full ARN of the resource. The following example shows how to do so using the AWS CLI command to create a role named `rds-s3-import-role`.

   Example

   For Linux, macOS, or Unix:

```
aws iam create-role \
   --role-name rds-s3-import-role \
   --assume-role-policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole",
         "Condition": {
             "StringEquals": {
                "aws:SourceAccount": "111122223333",
                "aws:SourceArn": "arn:aws:rds:us-
east-1:111122223333:cluster:clustername"
                }
             }
       }
     ]
   }'
```

For Windows:

```
aws iam create-role ^
   --role-name rds-s3-import-role ^
   --assume-role-policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole",
         "Condition": {
             "StringEquals": {
                "aws:SourceAccount": "111122223333",
                "aws:SourceArn": "arn:aws:rds:us-
east-1:111122223333:cluster:clustername"
                }
             }
```

```
        }
      ]
    }'
```

3. Attach the IAM policy that you created to the IAM role that you created.

   The following AWS CLI command attaches the policy created in the previous step to the role named `rds-s3-import-role` Replace *`your-policy-arn`* with the policy ARN that you noted in an earlier step.

   Example

   For Linux, macOS, or Unix:

   ```
   aws iam attach-role-policy \
      --policy-arn your-policy-arn \
      --role-name rds-s3-import-role
   ```

   For Windows:

   ```
   aws iam attach-role-policy ^
      --policy-arn your-policy-arn ^
      --role-name rds-s3-import-role
   ```

4. Add the IAM role to the DB cluster.

   You do so by using the AWS Management Console or AWS CLI, as described following.

Console

To add an IAM role for a PostgreSQL DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose the PostgreSQL DB cluster name to display its details.
3. On the **Connectivity & security** tab, in the **Manage IAM roles** section, choose the role to add under **Add IAM roles to this cluster** .
4. Under **Feature**, choose **s3Import**.
5. Choose **Add role**.

AWS CLI

To add an IAM role for a PostgreSQL DB cluster using the CLI

- Use the following command to add the role to the PostgreSQL DB cluster named `my-db-cluster`. Replace *`your-role-arn`* with the role ARN that you noted in a previous step. Use `s3Import` for the value of the `--feature-name` option.

  Example

  For Linux, macOS, or Unix:

  ```
  aws rds add-role-to-db-cluster \
     --db-cluster-identifier my-db-cluster \
     --feature-name s3Import \
     --role-arn your-role-arn   \
     --region your-region
  ```

  For Windows:

  ```
  aws rds add-role-to-db-cluster ^
     --db-cluster-identifier my-db-cluster ^
     --feature-name s3Import ^
     --role-arn your-role-arn ^
     --region your-region
  ```

RDS API

To add an IAM role for a PostgreSQL DB cluster using the Amazon RDS API, call the AddRoleToDBCluster operation.

Using security credentials to access an Amazon S3 bucket

If you prefer, you can use security credentials to provide access to an Amazon S3 bucket instead of providing access with an IAM role. You do so by specifying the `credentials` parameter in the aws_s3.table_import_from_s3 function call.

The `credentials` parameter is a structure of type `aws_commons._aws_credentials_1`, which contains AWS credentials. Use the aws_commons.create_aws_credentials function to set the access key and secret key in an `aws_commons._aws_credentials_1` structure, as shown following.

```
postgres=> SELECT aws_commons.create_aws_credentials(
   'sample_access_key', 'sample_secret_key', '')
AS creds \gset
```

After creating the `aws_commons._aws_credentials_1` structure, use the aws_s3.table_import_from_s3 function with the `credentials` parameter to import the data, as shown following.

```
postgres=> SELECT aws_s3.table_import_from_s3(
   't', '', '(format csv)',
   :'s3_uri',
   :'creds'
);
```

Or you can include the aws_commons.create_aws_credentials function call inline within the `aws_s3.table_import_from_s3` function call.

```
postgres=> SELECT aws_s3.table_import_from_s3(
   't', '', '(format csv)',
   :'s3_uri',
   aws_commons.create_aws_credentials('sample_access_key', 'sample_secret_key', '')
);
```

Troubleshooting access to Amazon S3

If you encounter connection problems when attempting to import data from Amazon S3, see the following for recommendations:

- Troubleshooting Amazon Aurora identity and access
- Troubleshooting Amazon S3 in the *Amazon Simple Storage Service User Guide*
- Troubleshooting Amazon S3 and IAM in the *IAM User Guide*

Importing data from Amazon S3 to your Aurora PostgreSQL DB cluster

You import data from your Amazon S3 bucket by using the `table_import_from_s3` function of the aws_s3 extension. For reference information, see aws_s3.table_import_from_s3.

Note

The following examples use the IAM role method to allow access to the Amazon S3 bucket. Thus, the `aws_s3.table_import_from_s3` function calls don't include credential parameters.

The following shows a typical example.

```
postgres=> SELECT aws_s3.table_import_from_s3(
   't1',
   '',
   '(format csv)',
   :'s3_uri'
);
```

The parameters are the following:

- `t1` – The name for the table in the PostgreSQL DB cluster to copy the data into.
- `''` – An optional list of columns in the database table. You can use this parameter to indicate which columns of the S3 data go in which table columns. If no columns are specified, all the columns are copied to the table. For an example of using a column list, see Importing an Amazon S3 file that uses a custom delimiter.
- `(format csv)` – PostgreSQL COPY arguments. The copy process uses the arguments and format of the PostgreSQL COPY command to import the data. Choices for format include comma-separated value (CSV) as shown in this example, text, and binary. The default is text.
- `s3_uri` – A structure that contains the information identifying the Amazon S3 file. For an example of using the aws_commons.create_s3_uri function to create an `s3_uri` structure, see Overview of importing data from Amazon S3 data.

For more information about this function, see aws_s3.table_import_from_s3.

The `aws_s3.table_import_from_s3` function returns text. To specify other kinds of files for import from an Amazon S3 bucket, see one of the following examples.

Note

Importing 0 bytes file will cause an error.

Topics

Importing an Amazon S3 file that uses a custom delimiter

The following example shows how to import a file that uses a custom delimiter. It also shows how to control where to put the data in the database table using the `column_list` parameter of the aws_s3.table_import_from_s3 function.

For this example, assume that the following information is organized into pipe-delimited columns in the Amazon S3 file.

```
1|foo1|bar1|elephant1
2|foo2|bar2|elephant2
3|foo3|bar3|elephant3
4|foo4|bar4|elephant4
...
```

To import a file that uses a custom delimiter

1. Create a table in the database for the imported data.

   ```
   postgres=> CREATE TABLE test (a text, b text, c text, d text, e text);
   ```

2. Use the following form of the aws_s3.table_import_from_s3 function to import data from the Amazon S3 file.

   You can include the aws_commons.create_s3_uri function call inline within the `aws_s3.table_import_from_s3` function call to specify the file.

   ```
   postgres=> SELECT aws_s3.table_import_from_s3(
   ```

```
   'test',
   'a,b,d,e',
   'DELIMITER ''|''',
   aws_commons.create_s3_uri('DOC-EXAMPLE-BUCKET', 'pipeDelimitedSampleFile', 'us-
east-2')
);
```

The data is now in the table in the following columns.

```
postgres=> SELECT * FROM test;
a | b | c | d | e
---+------+---+---+------+-----------
1 | foo1 | | bar1 | elephant1
2 | foo2 | | bar2 | elephant2
3 | foo3 | | bar3 | elephant3
4 | foo4 | | bar4 | elephant4
```

Importing an Amazon S3 compressed (gzip) file

The following example shows how to import a file from Amazon S3 that is compressed with gzip. The file that you import needs to have the following Amazon S3 metadata:

- Key: `Content-Encoding`
- Value: `gzip`

If you upload the file using the AWS Management Console, the metadata is typically applied by the system. For information about uploading files to Amazon S3 using the AWS Management Console, the AWS CLI, or the API, see Uploading objects in the *Amazon Simple Storage Service User Guide*.

For more information about Amazon S3 metadata and details about system-provided metadata, see Editing object metadata in the Amazon S3 console in the *Amazon Simple Storage Service User Guide*.

Import the gzip file into your Aurora PostgreSQL DB cluster as shown following.

```
postgres=> CREATE TABLE test_gzip(id int, a text, b text, c text, d text);
postgres=> SELECT aws_s3.table_import_from_s3(
 'test_gzip', '', '(format csv)',
 'DOC-EXAMPLE-BUCKET', 'test-data.gz', 'us-east-2'
);
```

Importing an encoded Amazon S3 file

The following example shows how to import a file from Amazon S3 that has Windows-1252 encoding.

```
postgres=> SELECT aws_s3.table_import_from_s3(
 'test_table', '', 'encoding ''WIN1252''',
 aws_commons.create_s3_uri('DOC-EXAMPLE-BUCKET', 'SampleFile', 'us-east-2')
);
```

Function reference

Functions

- aws_s3.table_import_from_s3
- aws_commons.create_s3_uri
- aws_commons.create_aws_credentials

aws_s3.table_import_from_s3

Imports Amazon S3 data into an Aurora PostgreSQL table. The `aws_s3` extension provides the `aws_s3.table_import_from_s3` function. The return value is text.

Syntax

The required parameters are `table_name`, `column_list` and `options`. These identify the database table and specify how the data is copied into the table.

You can also use the following parameters:

- The `s3_info` parameter specifies the Amazon S3 file to import. When you use this parameter, access to Amazon S3 is provided by an IAM role for the PostgreSQL DB cluster.

  ```
  aws_s3.table_import_from_s3 (
     table_name text,
     column_list text,
     options text,
     s3_info aws_commons._s3_uri_1
  ```

```
)
```

- The `credentials` parameter specifies the credentials to access Amazon S3. When you use this parameter, you don't use an IAM role.

```
aws_s3.table_import_from_s3 (
   table_name text,
   column_list text,
   options text,
   s3_info aws_commons._s3_uri_1,
   credentials aws_commons._aws_credentials_1
)
```

Parameters

table_name

A required text string containing the name of the PostgreSQL database table to import the data into.

column_list

A required text string containing an optional list of the PostgreSQL database table columns in which to copy the data. If the string is empty, all columns of the table are used. For an example, see Importing an Amazon S3 file that uses a custom delimiter.

options

A required text string containing arguments for the PostgreSQL `COPY` command. These arguments specify how the data is to be copied into the PostgreSQL table. For more details, see the PostgreSQL COPY documentation.

s3_info

An `aws_commons._s3_uri_1` composite type containing the following information about the S3 object:

- `bucket` – The name of the Amazon S3 bucket containing the file.
- `file_path` – The Amazon S3 file name including the path of the file.
- `region` – The AWS Region that the file is in. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

credentials

An `aws_commons._aws_credentials_1` composite type containing the following credentials to use for the import operation:

- Access key
- Secret key
- Session token

For information about creating an `aws_commons._aws_credentials_1` composite structure, see aws_commons.create_aws_credentials.

Alternate syntax

To help with testing, you can use an expanded set of parameters instead of the `s3_info` and `credentials` parameters. Following are additional syntax variations for the `aws_s3.table_import_from_s3` function:

- Instead of using the `s3_info` parameter to identify an Amazon S3 file, use the combination of the `bucket`, `file_path`, and `region` parameters. With this form of the function, access to Amazon S3 is provided by an IAM role on the PostgreSQL DB instance.

```
aws_s3.table_import_from_s3 (
   table_name text,
   column_list text,
   options text,
   bucket text,
   file_path text,
   region text
)
```

- Instead of using the `credentials` parameter to specify Amazon S3 access, use the combination of the `access_key`, `session_key`, and `session_token` parameters.

```
aws_s3.table_import_from_s3 (
   table_name text,
   column_list text,
   options text,
   bucket text,
   file_path text,
   region text,
   access_key text,
   secret_key text,
   session_token text
)
```

Alternate parameters

bucket

A text string containing the name of the Amazon S3 bucket that contains the file.

file_path

A text string containing the Amazon S3 file name including the path of the file.

region

A text string identifying the AWS Region location of the file. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

access_key

A text string containing the access key to use for the import operation. The default is NULL.

secret_key

A text string containing the secret key to use for the import operation. The default is NULL.

session_token

(Optional) A text string containing the session key to use for the import operation. The default is NULL.

aws_commons.create_s3_uri

Creates an `aws_commons._s3_uri_1` structure to hold Amazon S3 file information. Use the results of the `aws_commons.create_s3_uri` function in the `s3_info` parameter of the aws_s3.table_import_from_s3 function.

Syntax

```
aws_commons.create_s3_uri(
   bucket text,
   file_path text,
   region text
)
```

Parameters

bucket

A required text string containing the Amazon S3 bucket name for the file.

file_path

A required text string containing the Amazon S3 file name including the path of the file.

region

A required text string containing the AWS Region that the file is in. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

aws_commons.create_aws_credentials

Sets an access key and secret key in an `aws_commons._aws_credentials_1` structure. Use the results of the `aws_commons.create_aws_credentials` function in the `credentials` parameter of the aws_s3.table_import_from_s3 function.

Syntax

```
aws_commons.create_aws_credentials(
   access_key text,
   secret_key text,
   session_token text
)
```

Parameters

access_key

A required text string containing the access key to use for importing an Amazon S3 file. The default is NULL.

secret_key

A required text string containing the secret key to use for importing an Amazon S3 file. The default is NULL.

session_token

An optional text string containing the session token to use for importing an Amazon S3 file. The default is NULL. If you provide an optional `session_token`, you can use temporary credentials.

Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3

You can query data from an Aurora PostgreSQL DB cluster and export it directly into files stored in an Amazon S3 bucket. To do this, you first install the Aurora PostgreSQL `aws_s3` extension. This extension provides you with the functions that you use to export the results of queries to Amazon S3. Following, you can find out how to install the extension and how to export data to Amazon S3.

You can export from a provisioned or an Aurora Serverless v2 DB instance. These steps aren't supported for Aurora Serverless v1.

> **Note**
>
> Cross-account export to Amazon S3 isn't supported.

All currently available versions of Aurora PostgreSQL support exporting data to Amazon Simple Storage Service. For detailed version information, see Amazon Aurora PostgreSQL updates in the *Release Notes for Aurora PostgreSQL*.

If you don't have a bucket set up for your export, see the following topics the *Amazon Simple Storage Service User Guide*.

- Setting up Amazon S3
- Create a bucket

By default, the data exported from Aurora PostgreSQL to Amazon S3 uses server-side encryption with AWS managed key. You can alternatively use customer managed key that you have already created. If you are using bucket encryption, the Amazon S3 bucket must be encrypted with AWS Key Management Service (AWS KMS) key (SSE-KMS). Currently, buckets encrypted with Amazon S3 managed keys (SSE-S3) are not supported.

Note

You can save DB and DB cluster snapshot data to Amazon S3 using the AWS Management Console, AWS CLI, or Amazon RDS API. For more information, see Exporting DB cluster snapshot data to Amazon S3.

Topics

Installing the aws_s3 extension

Before you can use Amazon Simple Storage Service with your Aurora PostgreSQL DB cluster, you need to install the `aws_s3` extension. This extension provides functions for exporting data from the writer instance of an Aurora PostgreSQL DB cluster to an Amazon S3 bucket. It also provides functions for importing data from an Amazon S3. For more information, see Importing data from Amazon S3 into an Aurora PostgreSQL DB cluster. The `aws_s3` extension depends on some of the helper functions in the `aws_commons` extension, which is installed automatically when needed.

To install the aws_s3 extension

1. Use psql (or pgAdmin) to connect to the writer instance of your Aurora PostgreSQL DB cluster as a user that has `rds_superuser` privileges. If you kept the default name during the setup process, you connect as `postgres`.

```
psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password
```

2. To install the extension, run the following command.

```
postgres=> CREATE EXTENSION aws_s3 CASCADE;
```

```
NOTICE: installing required extension "aws_commons"
CREATE EXTENSION
```

3. To verify that the extension is installed, you can use the psql `\dx` metacommand.

```
postgres=> \dx
       List of installed extensions
    Name     | Version |   Schema   |                Description
-------------+---------+------------+---------------------------------------------
 aws_commons | 1.2     | public     | Common data types across AWS services
 aws_s3      | 1.1     | public     | AWS S3 extension for importing data from S3
 plpgsql     | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

The functions for importing data from Amazon S3 and exporting data to Amazon S3 are now available to use.

Verify that your Aurora PostgreSQL version supports exports to Amazon S3

You can verify that your Aurora PostgreSQL version supports export to Amazon S3 by using the `describe-db-engine-versions` command. The following example checks to see if version 10.14 can export to Amazon S3.

```
aws rds describe-db-engine-versions --region us-east-1 \
--engine aurora-postgresql --engine-version 10.14 | grep s3Export
```

If the output includes the string `"s3Export"`, then the engine supports Amazon S3 exports. Otherwise, the engine doesn't support them.

Overview of exporting data to Amazon S3

To export data stored in an Aurora PostgreSQL database to an Amazon S3 bucket, use the following procedure.

To export Aurora PostgreSQL data to S3

1. Identify an Amazon S3 file path to use for exporting data. For details about this process, see Specifying the Amazon S3 file path to export to.
2. Provide permission to access the Amazon S3 bucket.

To export data to an Amazon S3 file, give the Aurora PostgreSQL DB cluster permission to access the Amazon S3 bucket that the export will use for storage. Doing this includes the following steps:

1. Create an IAM policy that provides access to an Amazon S3 bucket that you want to export to.
2. Create an IAM role.
3. Attach the policy you created to the role you created.
4. Add this IAM role to your DB cluster .

For details about this process, see Setting up access to an Amazon S3 bucket.

3. Identify a database query to get the data. Export the query data by calling the `aws_s3.query_export_to_s3` function.

   After you complete the preceding preparation tasks, use the aws_s3.query_export_to_s3 function to export query results to Amazon S3. For details about this process, see Exporting query data using the aws_s3.query_export_to_s3 function.

Specifying the Amazon S3 file path to export to

Specify the following information to identify the location in Amazon S3 where you want to export data to:

- Bucket name – A *bucket* is a container for Amazon S3 objects or files.

  For more information on storing data with Amazon S3, see Create a bucket and View an object in the *Amazon Simple Storage Service User Guide*.
- File path – The file path identifies where the export is stored in the Amazon S3 bucket. The file path consists of the following:
  - An optional path prefix that identifies a virtual folder path.
  - A file prefix that identifies one or more files to be stored. Larger exports are stored in multiple files, each with a maximum size of approximately 6 GB. The additional file names have the same file prefix but with `_part`*XX* appended. The *XX* represents 2, then 3, and so on.

  For example, a file path with an `exports` folder and a `query-1-export` file prefix is `/exports/query-1-export`.

- AWS Region (optional) – The AWS Region where the Amazon S3 bucket is located. If you don't specify an AWS Region value, then Aurora saves your files into Amazon S3 in the same AWS Region as the exporting DB cluster.

  > **Note**
  >
  > Currently, the AWS Region must be the same as the region of the exporting DB cluster.

  For a listing of AWS Region names and associated values, see Regions and Availability Zones.

To hold the Amazon S3 file information about where the export is to be stored, you can use the aws_commons.create_s3_uri function to create an `aws_commons._s3_uri_1` composite structure as follows.

```
psql=> SELECT aws_commons.create_s3_uri(
   'DOC-EXAMPLE-BUCKET',
   'sample-filepath',
   'us-west-2'
) AS s3_uri_1 \gset
```

You later provide this `s3_uri_1` value as a parameter in the call to the aws_s3.query_export_to_s3 function. For examples, see Exporting query data using the aws_s3.query_export_to_s3 function.

Setting up access to an Amazon S3 bucket

To export data to Amazon S3, give your PostgreSQL DB cluster permission to access the Amazon S3 bucket that the files are to go in.

To do this, use the following procedure.

To give a PostgreSQL DB cluster access to Amazon S3 through an IAM role

1. Create an IAM policy.

   This policy provides the bucket and object permissions that allow your PostgreSQL DB cluster to access Amazon S3.

   As part of creating this policy, take the following steps:

a. Include in the policy the following required actions to allow the transfer of files from your PostgreSQL DB cluster to an Amazon S3 bucket:

   - `s3:PutObject`
   - `s3:AbortMultipartUpload`

b. Include the Amazon Resource Name (ARN) that identifies the Amazon S3 bucket and objects in the bucket. The ARN format for accessing Amazon S3 is: `arn:aws:s3:::`*`DOC-EXAMPLE-BUCKET`*`/*`

For more information on creating an IAM policy for Aurora PostgreSQL, see Creating and using an IAM policy for IAM database access. See also Tutorial: Create and attach your first customer managed policy in the *IAM User Guide*.

The following AWS CLI command creates an IAM policy named `rds-s3-export-policy` with these options. It grants access to a bucket named *`DOC-EXAMPLE-BUCKET`*.

> **Warning**
>
> We recommend that you set up your database within a private VPC that has endpoint policies configured for accessing specific buckets. For more information, see Using endpoint policies for Amazon S3 in the Amazon VPC User Guide.
> We strongly recommend that you do not create a policy with all-resource access. This access can pose a threat for data security. If you create a policy that gives `S3:PutObject` access to all resources using `"Resource":"*"`, then a user with export privileges can export data to all buckets in your account. In addition, the user can export data to *any publicly writable bucket within your AWS Region*.

After you create the policy, note the Amazon Resource Name (ARN) of the policy. You need the ARN for a subsequent step when you attach the policy to an IAM role.

```
aws iam create-policy  --policy-name rds-s3-export-policy  --policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "s3export",
         "Action": [
           "s3:PutObject*",
           "s3:ListBucket",
           "s3:GetObject*",
           "s3:DeleteObject*",
           "s3:GetBucketLocation",
           "s3:AbortMultipartUpload"
         ],
         "Effect": "Allow",
         "Resource": [
           "arn:aws:s3:::DOC-EXAMPLE-BUCKET/*"
         ]
       }
     ]
   }'
```

2. Create an IAM role.

   You do this so Aurora PostgreSQL can assume this IAM role on your behalf to access your Amazon S3 buckets. For more information, see Creating a role to delegate permissions to an IAM user in the *IAM User Guide*.

   We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource-based policies to limit the service's permissions to a specific resource. This is the most effective way to protect against the confused deputy problem.

   If you use both global condition context keys and the `aws:SourceArn` value contains the account ID, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

   - Use `aws:SourceArn` if you want cross-service access for a single resource.
   - Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

   In the policy, be sure to use the `aws:SourceArn` global condition context key with the full ARN of the resource. The following example shows how to do so using the AWS CLI command to create a role named `rds-s3-export-role`.

   Example

   For Linux, macOS, or Unix:

```
aws iam create-role  \
    --role-name rds-s3-export-role  \
    --assume-role-policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole",
         "Condition": {
             "StringEquals": {
                "aws:SourceAccount": "111122223333",
                "aws:SourceArn": "arn:aws:rds:us-east-1:111122223333:db:dbname"
                }
             }
       }
     ]
   }'
```

For Windows:

```
aws iam create-role  ^
    --role-name rds-s3-export-role  ^
    --assume-role-policy-document '{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
            "Service": "rds.amazonaws.com"
          },
         "Action": "sts:AssumeRole",
         "Condition": {
             "StringEquals": {
                "aws:SourceAccount": "111122223333",
                "aws:SourceArn": "arn:aws:rds:us-east-1:111122223333:db:dbname"
                }
             }
       }
     ]
    }'
```

3. Attach the IAM policy that you created to the IAM role that you created.

   The following AWS CLI command attaches the policy created earlier to the role named `rds-s3-export-role.` Replace *`your-policy-arn`* with the policy ARN that you noted in an earlier step.

   ```
   aws iam attach-role-policy  --policy-arn your-policy-arn  --role-name rds-s3-export-role
   ```

4. Add the IAM role to the DB cluster. You do so by using the AWS Management Console or AWS CLI, as described following.

Console

To add an IAM role for a PostgreSQL DB cluster using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose the PostgreSQL DB cluster name to display its details.
3. On the **Connectivity & security** tab, in the **Manage IAM roles** section, choose the role to add under **Add IAM roles to this instance**.
4. Under **Feature**, choose **s3Export**.
5. Choose **Add role**.

AWS CLI

To add an IAM role for a PostgreSQL DB cluster using the CLI

- Use the following command to add the role to the PostgreSQL DB cluster named `my-db-cluster`. Replace *`your-role-arn`* with the role ARN that you noted in a previous step. Use `s3Export` for the value of the `--feature-name` option.

  Example

  For Linux, macOS, or Unix:

  ```
  aws rds add-role-to-db-cluster \
  ```

```
    --db-cluster-identifier my-db-cluster \
    --feature-name s3Export \
    --role-arn your-role-arn   \
    --region your-region
```

For Windows:

```
aws rds add-role-to-db-cluster ^
    --db-cluster-identifier my-db-cluster ^
    --feature-name s3Export ^
    --role-arn your-role-arn ^
    --region your-region
```

Exporting query data using the aws_s3.query_export_to_s3 function

Export your PostgreSQL data to Amazon S3 by calling the aws_s3.query_export_to_s3 function.

Topics

Prerequisites

Before you use the `aws_s3.query_export_to_s3` function, be sure to complete the following prerequisites:

- Install the required PostgreSQL extensions as described in Overview of exporting data to Amazon S3.
- Determine where to export your data to Amazon S3 as described in Specifying the Amazon S3 file path to export to.
- Make sure that the DB cluster has export access to Amazon S3 as described in Setting up access to an Amazon S3 bucket.

The examples following use a database table called `sample_table`. These examples export the data into a bucket called *DOC-EXAMPLE-BUCKET*. The example table and data are created with the following SQL statements in psql.

```
psql=> CREATE TABLE sample_table (bid bigint PRIMARY KEY, name varchar(80));
psql=> INSERT INTO sample_table (bid,name) VALUES (1, 'Monday'), (2,'Tuesday'), (3,
 'Wednesday');
```

Calling aws_s3.query_export_to_s3

The following shows the basic ways of calling the aws_s3.query_export_to_s3 function.

These examples use the variable `s3_uri_1` to identify a structure that contains the information identifying the Amazon S3 file. Use the aws_commons.create_s3_uri function to create the structure.

```
psql=> SELECT aws_commons.create_s3_uri(
   'DOC-EXAMPLE-BUCKET',
   'sample-filepath',
   'us-west-2'
) AS s3_uri_1 \gset
```

Although the parameters vary for the following two `aws_s3.query_export_to_s3` function calls, the results are the same for these examples. All rows of the `sample_table` table are exported into a bucket called *DOC-EXAMPLE-BUCKET*.

```
psql=> SELECT * FROM aws_s3.query_export_to_s3('SELECT * FROM
 sample_table', :'s3_uri_1');

psql=> SELECT * FROM aws_s3.query_export_to_s3('SELECT * FROM
 sample_table', :'s3_uri_1', options :='format text');
```

The parameters are described as follows:

- `'SELECT * FROM sample_table'` – The first parameter is a required text string containing an SQL query. The PostgreSQL engine runs this query. The results of the query are copied to the S3 bucket identified in other parameters.
- `:'s3_uri_1'` – This parameter is a structure that identifies the Amazon S3 file. This example uses a variable to identify the previously created structure. You can instead create the

structure by including the aws_commons.create_s3_uri function call inline within the aws_s3.query_export_to_s3 function call as follows.

```
SELECT * from aws_s3.query_export_to_s3('select * from sample_table', 
   aws_commons.create_s3_uri('DOC-EXAMPLE-BUCKET', 'sample-filepath', 'us-west-2')
);
```

- options :='format text' – The options parameter is an optional text string containing PostgreSQL COPY arguments. The copy process uses the arguments and format of the PostgreSQL COPY command.

If the file specified doesn't exist in the Amazon S3 bucket, it's created. If the file already exists, it's overwritten. The syntax for accessing the exported data in Amazon S3 is the following.

```
s3-region://bucket-name[/path-prefix]/file-prefix
```

Larger exports are stored in multiple files, each with a maximum size of approximately 6 GB. The additional file names have the same file prefix but with _partXX appended. The *XX* represents 2, then 3, and so on. For example, suppose that you specify the path where you store data files as the following.

```
s3-us-west-2://DOC-EXAMPLE-BUCKET/my-prefix
```

If the export has to create three data files, the Amazon S3 bucket contains the following data files.

```
s3-us-west-2://DOC-EXAMPLE-BUCKET/my-prefix
s3-us-west-2://DOC-EXAMPLE-BUCKET/my-prefix_part2
s3-us-west-2://DOC-EXAMPLE-BUCKET/my-prefix_part3
```

For the full reference for this function and additional ways to call it, see aws_s3.query_export_to_s3. For more about accessing files in Amazon S3, see View an object in the *Amazon Simple Storage Service User Guide*.

Exporting to a CSV file that uses a custom delimiter

The following example shows how to call the aws_s3.query_export_to_s3 function to export data to a file that uses a custom delimiter. The example uses arguments of the PostgreSQL COPY command to specify the comma-separated value (CSV) format and a colon (:) delimiter.

```
SELECT * from aws_s3.query_export_to_s3('select * from basic_test', :'s3_uri_1',
 options :='format csv, delimiter $$:$$');
```

Exporting to a binary file with encoding

The following example shows how to call the aws_s3.query_export_to_s3 function to export data to a binary file that has Windows-1253 encoding.

```
SELECT * from aws_s3.query_export_to_s3('select * from basic_test', :'s3_uri_1',
 options :='format binary, encoding WIN1253');
```

Troubleshooting access to Amazon S3

If you encounter connection problems when attempting to export data to Amazon S3, first confirm that the outbound access rules for the VPC security group associated with your DB instance permit network connectivity. Specifically, the security group must have a rule that allows the DB instance to send TCP traffic to port 443 and to any IPv4 addresses (0.0.0.0/0). For more information, see Provide access to the DB cluster in the VPC by creating a security group.

See also the following for recommendations:

- Troubleshooting Amazon Aurora identity and access
- Troubleshooting Amazon S3 in the *Amazon Simple Storage Service User Guide*
- Troubleshooting Amazon S3 and IAM in the *IAM User Guide*

Function reference

Functions

- aws_s3.query_export_to_s3
- aws_commons.create_s3_uri

aws_s3.query_export_to_s3

Exports a PostgreSQL query result to an Amazon S3 bucket. The `aws_s3` extension provides the `aws_s3.query_export_to_s3` function.

The two required parameters are `query` and `s3_info`. These define the query to be exported and identify the Amazon S3 bucket to export to. An optional parameter called `options` provides for defining various export parameters. For examples of using the `aws_s3.query_export_to_s3` function, see Exporting query data using the aws_s3.query_export_to_s3 function.

Syntax

```
aws_s3.query_export_to_s3(
    query text,
    s3_info aws_commons._s3_uri_1,
    options text,
    kms_key text
)
```

Input parameters

query

A required text string containing an SQL query that the PostgreSQL engine runs. The results of this query are copied to an S3 bucket identified in the `s3_info` parameter.

s3_info

An `aws_commons._s3_uri_1` composite type containing the following information about the S3 object:

- `bucket` – The name of the Amazon S3 bucket to contain the file.
- `file_path` – The Amazon S3 file name and path.
- `region` – The AWS Region that the bucket is in. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

  Currently, this value must be the same AWS Region as that of the exporting DB cluster . The default is the AWS Region of the exporting DB cluster .

To create an `aws_commons._s3_uri_1` composite structure, see the aws_commons.create_s3_uri function.

options

An optional text string containing arguments for the PostgreSQL `COPY` command. These arguments specify how the data is to be copied when exported. For more details, see the PostgreSQL COPY documentation.

kms_key text

An optional text string containing the customer managed KMS key of the S3 bucket to export the data to.

Alternate input parameters

To help with testing, you can use an expanded set of parameters instead of the `s3_info` parameter. Following are additional syntax variations for the `aws_s3.query_export_to_s3` function.

Instead of using the `s3_info` parameter to identify an Amazon S3 file, use the combination of the `bucket`, `file_path`, and `region` parameters.

```
aws_s3.query_export_to_s3(
    query text,
    bucket text,
    file_path text,
    region text,
    options text,
    kms_key text
)
```

query

A required text string containing an SQL query that the PostgreSQL engine runs. The results of this query are copied to an S3 bucket identified in the `s3_info` parameter.

bucket

A required text string containing the name of the Amazon S3 bucket that contains the file.

file_path

A required text string containing the Amazon S3 file name including the path of the file.

region

An optional text string containing the AWS Region that the bucket is in. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

Currently, this value must be the same AWS Region as that of the exporting DB cluster . The default is the AWS Region of the exporting DB cluster .

options

An optional text string containing arguments for the PostgreSQL COPY command. These arguments specify how the data is to be copied when exported. For more details, see the PostgreSQL COPY documentation.

kms_key text

An optional text string containing the customer managed KMS key of the S3 bucket to export the data to.

Output parameters

```
aws_s3.query_export_to_s3(
    OUT rows_uploaded bigint,
    OUT files_uploaded bigint,
    OUT bytes_uploaded bigint
)
```

rows_uploaded

The number of table rows that were successfully uploaded to Amazon S3 for the given query.

files_uploaded

The number of files uploaded to Amazon S3. Files are created in sizes of approximately 6 GB. Each additional file created has `_partXX` appended to the name. The *XX* represents 2, then 3, and so on as needed.

bytes_uploaded

The total number of bytes uploaded to Amazon S3.

Examples

```
psql=> SELECT * from aws_s3.query_export_to_s3('select * from sample_table', 'DOC-EXAMPLE-BUCKET', 'sample-filepath');
psql=> SELECT * from aws_s3.query_export_to_s3('select * from sample_table', 'DOC-EXAMPLE-BUCKET', 'sample-filepath','us-west-2');
psql=> SELECT * from aws_s3.query_export_to_s3('select * from sample_table', 'DOC-EXAMPLE-BUCKET', 'sample-filepath','us-west-2','format text');
```

aws_commons.create_s3_uri

Creates an `aws_commons._s3_uri_1` structure to hold Amazon S3 file information. You use the results of the `aws_commons.create_s3_uri` function in the `s3_info` parameter of the aws_s3.query_export_to_s3 function. For an example of using the `aws_commons.create_s3_uri` function, see Specifying the Amazon S3 file path to export to.

Syntax

```
aws_commons.create_s3_uri(
   bucket text,
   file_path text,
   region text
)
```

Input parameters

bucket

A required text string containing the Amazon S3 bucket name for the file.

file_path

A required text string containing the Amazon S3 file name including the path of the file.

region

A required text string containing the AWS Region that the file is in. For a listing of AWS Region names and associated values, see Regions and Availability Zones.

Invoking an AWS Lambda function from an Aurora PostgreSQL DB cluster

AWS Lambda is an event-driven compute service that lets you run code without provisioning or managing servers. It's available for use with many AWS services, including Aurora PostgreSQL. For example, you can use Lambda functions to process event notifications from a database, or to load data from files whenever a new file is uploaded to Amazon S3. To learn more about Lambda, see What is AWS Lambda? in the *AWS Lambda Developer Guide.*

> **Note**
>
> Invoking AWS Lambda functions is supported in Aurora PostgreSQL 11.9 and higher (including Aurora Serverless v2).

Setting up Aurora PostgreSQL to work with Lambda functions is a multi-step process involving AWS Lambda, IAM, your VPC, and your Aurora PostgreSQL DB cluster. Following, you can find summaries of the necessary steps.

For more information about Lambda functions, see Getting started with Lambda and AWS Lambda foundations in the *AWS Lambda Developer Guide*.

Topics

Step 1: Configure your Aurora PostgreSQL DB cluster for outbound connections to AWS Lambda

Lambda functions always run inside an Amazon VPC that's owned by the AWS Lambda service. Lambda applies network access and security rules to this VPC and it maintains and monitors the VPC automatically. Your Aurora PostgreSQL DB cluster sends network traffic to the Lambda service's VPC. How you configure this depends on whether your Aurora DB cluster's primary DB instance is public or private.

- **Public Aurora PostgreSQL DB cluster** – A DB cluster's primary DB instance is public if it's located in a public subnet on your VPC, and if the instance's "PubliclyAccessible" property is `true`. To find the value of this property, you can use the describe-db-instances AWS CLI command. Or, you can use the AWS Management Console to open the **Connectivity & security** tab and check that **Publicly accessible** is **Yes**. To verify that the instance is in the public subnet of your VPC, you can use the AWS Management Console or the AWS CLI.

To set up access to Lambda, you use the AWS Management Console or the AWS CLI to create an outbound rule on your VPC's security group. The outbound rule specifies that TCP can use port 443 to send packets to any IPv4 addresses (0.0.0.0/0).

- **Private Aurora PostgreSQL DB cluster –** In this case, the instance's "PubliclyAccessible" property is `false` or it's in a private subnet. To allow the instance to work with Lambda, you can use a Network Address Translation) NAT gateway. For more information, see NAT gateways. Or, you can configure your VPC with a VPC endpoint for Lambda. For more information, see VPC endpoints in the *Amazon VPC User Guide*. The endpoint responds to calls made by your Aurora PostgreSQL DB cluster to your Lambda functions.

Your VPC can now interact with the AWS Lambda VPC at the network level. Next, you configure the permissions using IAM.

Step 2: Configure IAM for your Aurora PostgreSQL DB cluster and AWS Lambda

Invoking Lambda functions from your Aurora PostgreSQL DB cluster requires certain privileges. To configure the necessary privileges, we recommend that you create an IAM policy that allows invoking Lambda functions, assign that policy to a role, and then apply the role to your DB cluster. This approach gives the DB cluster privileges to invoke the specified Lambda function on your behalf. The following steps show you how to do this using the AWS CLI.

To configure IAM permissions for using your cluster with Lambda

1. Use the create-policy AWS CLI command to create an IAM policy that allows your Aurora PostgreSQL DB cluster to invoke the specified Lambda function. (The statement ID (Sid) is an optional description for your policy statement and has no effect on usage.) This policy gives your Aurora DB cluster the minimum permissions needed to invoke the specified Lambda function.

```
aws iam create-policy  --policy-name rds-lambda-policy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Sid": "AllowAccessToExampleFunction",
        "Effect": "Allow",
        "Action": "lambda:InvokeFunction",
        "Resource": "arn:aws:lambda:aws-region:444455556666:function:my-function"
        }
    ]
}'
```

Alternatively, you can use the predefined `AWSLambdaRole` policy that allows you to invoke any of your Lambda functions. For more information, see Identity-based IAM policies for Lambda

2. Use the create-role AWS CLI command to create an IAM role that the policy can assume at runtime.

```
aws iam create-role  --role-name rds-lambda-role --assume-role-policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "rds.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}'
```

3. Apply the policy to the role by using the attach-role-policy AWS CLI command.

```
aws iam attach-role-policy \
    --policy-arn arn:aws:iam::444455556666:policy/rds-lambda-policy \
    --role-name rds-lambda-role --region aws-region
```

4. Apply the role to your Aurora PostgreSQL DB cluster by using the add-role-to-db-cluster AWS CLI command. This last step allows your DB cluster's database users to invoke Lambda functions.

```
aws rds add-role-to-db-cluster \
       --db-cluster-identifier my-cluster-name \
       --feature-name Lambda \
       --role-arn  arn:aws:iam::444455556666:role/rds-lambda-role   \
       --region aws-region
```

With the VPC and the IAM configurations complete, you can now install the aws_lambda extension. (Note that you can install the extension at any time, but until you set up the correct VPC support and IAM privileges, the aws_lambda extension adds nothing to your Aurora PostgreSQL DB cluster's capabilities.)

Step 3: Install the aws_lambda extension for an Aurora PostgreSQL DB cluster

To use AWS Lambda with your Aurora PostgreSQL DB cluster, add the aws_lambda PostgreSQL extension to your Aurora PostgreSQL DB cluster. This extension provides your Aurora PostgreSQL DB cluster with the ability to call Lambda functions from PostgreSQL.

To install the aws_lambda extension in your Aurora PostgreSQL DB cluster

Use the PostgreSQL psql command-line or the pgAdmin tool to connect to your Aurora PostgreSQL DB cluster .

1. Connect to your Aurora PostgreSQL DB cluster instance as a user with rds_superuser privileges. The default postgres user is shown in the example.

   ```
   psql -h cluster-instance.444455556666.aws-region.rds.amazonaws.com -U postgres -p
    5432
   ```

2. Install the aws_lambda extension. The aws_commons extension is also required. It provides helper functions to aws_lambda and many other Aurora extensions for PostgreSQL. If it's not already on your Aurora PostgreSQLDB cluster , it's installed with aws_lambda as shown following.

   ```
   CREATE EXTENSION IF NOT EXISTS aws_lambda CASCADE;
   NOTICE:  installing required extension "aws_commons"
   CREATE EXTENSION
   ```

The aws_lambda extension is installed in your Aurora PostgreSQL DB cluster's primary DB instance. You can now create convenience structures for invoking your Lambda functions.

Step 4: Use Lambda helper functions with your Aurora PostgreSQL DB cluster (Optional)

You can use the helper functions in the `aws_commons` extension to prepare entities that you can more easily invoke from PostgreSQL. To do this, you need to have the following information about your Lambda functions:

- **Function name –** The name, Amazon Resource Name (ARN), version, or alias of the Lambda function. The IAM policy created in Step 2: Configure IAM for your cluster and Lambda requires the ARN, so we recommend that you use your function's ARN.
- **AWS Region –** (Optional) The AWS Region where the Lambda function is located if it's not in the same Region as your Aurora PostgreSQL DB cluster.

To hold the Lambda function name information, you use the aws_commons.create_lambda_function_arn function. This helper function creates an `aws_commons._lambda_function_arn_1` composite structure with the details needed by the invoke function. Following, you can find three alternative approaches to setting up this composite structure.

```
SELECT aws_commons.create_lambda_function_arn(
   'my-function',
   'aws-region'
) AS aws_lambda_arn_1 \gset
```

```
SELECT aws_commons.create_lambda_function_arn(
   '111122223333:function:my-function',
   'aws-region'
) AS lambda_partial_arn_1 \gset
```

```
SELECT aws_commons.create_lambda_function_arn(
   'arn:aws:lambda:aws-region:111122223333:function:my-function'
) AS lambda_arn_1 \gset
```

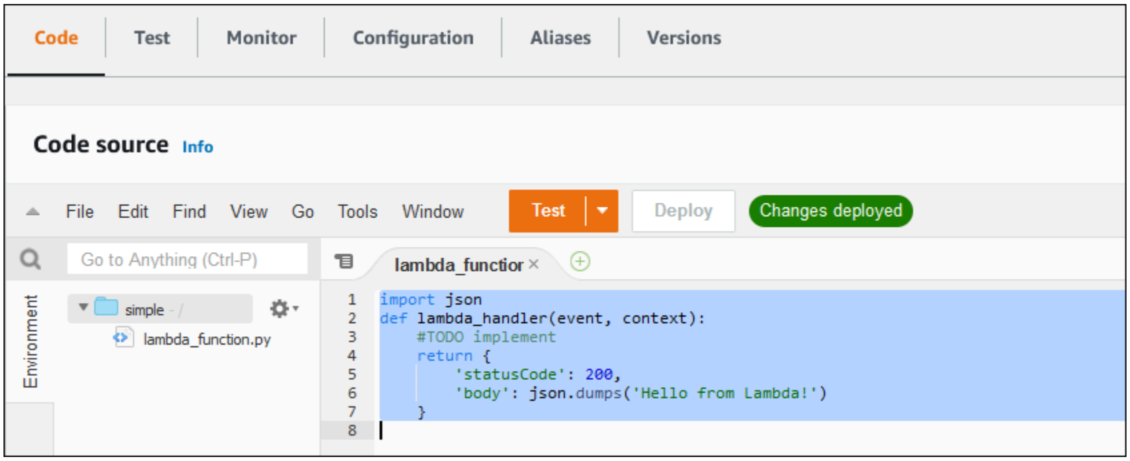
Any of these values can be used in calls to the aws_lambda.invoke function. For examples, see Step 5: Invoke a Lambda function from your Aurora PostgreSQL DB cluster.

Step 5: Invoke a Lambda function from your Aurora PostgreSQL DB cluster

The `aws_lambda.invoke` function behaves synchronously or asynchronously, depending on the `invocation_type`. The two alternatives for this parameter are `RequestResponse` (the default) and `Event`, as follows.

- **`RequestResponse`** – This invocation type is *synchronous*. It's the default behavior when the call is made without specifying an invocation type. The response payload includes the results of the `aws_lambda.invoke` function. Use this invocation type when your workflow requires receiving results from the Lambda function before proceeding.
- **`Event`** – This invocation type is *asynchronous*. The response doesn't include a payload containing results. Use this invocation type when your workflow doesn't need a result from the Lambda function to continue processing.

As a simple test of your setup, you can connect to your DB instance using `psql` and invoke an example function from the command line. Suppose that you have one of the basic functions set up on your Lambda service, such as the simple Python function shown in the following screenshot.



```
import json
def lambda_handler(event, context):
    #TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }
```

To invoke an example function

1. Connect to your primary DB instance using `psql` or pgAdmin.

```
psql -h cluster.444455556666.aws-region.rds.amazonaws.com -U postgres -p 5432
```

2. Invoke the function using its ARN.

```
SELECT * from
 aws_lambda.invoke(aws_commons.create_lambda_function_arn('arn:aws:lambda:aws-
region:444455556666:function:simple', 'us-west-1'), '{"body": "Hello from
 Postgres!"}'::json );
```

The response looks as follows.

```
status_code |                        payload                        |
 executed_version | log_result
-------------+-------------------------------------------------------
+------------------+------------
         200 | {"statusCode": 200, "body": "\"Hello from Lambda!\""} | $LATEST
     |
(1 row)
```

If your invocation attempt doesn't succeed, see Lambda function error messages .

Step 6: Grant other users permission to invoke Lambda functions

At this point in the procedures, only you as `rds_superuser` can invoke your Lambda functions. To allow other users to invoke any functions that you create, you need to grant them permission.

To grant others permission to invoke Lambda functions

1. Connect to your primary DB instance using `psql` or pgAdmin.

```
psql -h cluster.444455556666.aws-region.rds.amazonaws.com -U postgres -p 5432
```

2. Run the following SQL commands:

```
postgres=>  GRANT USAGE ON SCHEMA aws_lambda TO db_username;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA aws_lambda TO db_username;
```

Examples: Invoking Lambda functions from your Aurora PostgreSQL DB cluster

Following, you can find several examples of calling the aws_lambda.invoke function. Most of the examples use the composite structure `aws_lambda_arn_1` that you create in Step 4: Use Lambda helper functions with your Aurora PostgreSQL DB cluster (Optional) to simplify passing the function details. For an example of asynchronous invocation, see Example: Asynchronous (Event) invocation of Lambda functions. All the other examples listed use synchronous invocation.

To learn more about Lambda invocation types, see Invoking Lambda functions in the *AWS Lambda Developer Guide*. For more information about `aws_lambda_arn_1`, see aws_commons.create_lambda_function_arn.

Examples list

- Example: Synchronous (RequestResponse) invocation of Lambda functions
- Example: Asynchronous (Event) invocation of Lambda functions
- Example: Capturing the Lambda execution log in a function response
- Example: Including client context in a Lambda function
- Example: Invoking a specific version of a Lambda function

Example: Synchronous (RequestResponse) invocation of Lambda functions

Following are two examples of a synchronous Lambda function invocation. The results of these `aws_lambda.invoke` function calls are the same.

```
SELECT * FROM aws_lambda.invoke('aws_lambda_arn_1', '{"body": "Hello from
 Postgres!"}'::json);
```

```
SELECT * FROM aws_lambda.invoke('aws_lambda_arn_1', '{"body": "Hello from
 Postgres!"}'::json, 'RequestResponse');
```

The parameters are described as follows:

- `:'aws_lambda_arn_1'` – This parameter identifies the composite structure created in Step 4: Use Lambda helper functions with your Aurora PostgreSQL DB cluster (Optional), with the `aws_commons.create_lambda_function_arn` helper function. You can also create this structure inline within your `aws_lambda.invoke` call as follows.

  ```
  SELECT * FROM aws_lambda.invoke(aws_commons.create_lambda_function_arn('my-function',
   'aws-region'),
  '{"body": "Hello from Postgres!"}'::json
  );
  ```

- `'{"body": "Hello from PostgreSQL!"}'::json` – The JSON payload to pass to the Lambda function.
- `'RequestResponse'` – The Lambda invocation type.

Example: Asynchronous (Event) invocation of Lambda functions

Following is an example of an asynchronous Lambda function invocation. The `Event` invocation type schedules the Lambda function invocation with the specified input payload and returns immediately. Use the `Event` invocation type in certain workflows that don't depend on the results of the Lambda function.

```
SELECT * FROM aws_lambda.invoke('aws_lambda_arn_1', '{"body": "Hello from
 Postgres!"}'::json, 'Event');
```

Example: Capturing the Lambda execution log in a function response

You can include the last 4 KB of the execution log in the function response by using the `log_type` parameter in your `aws_lambda.invoke` function call. By default, this parameter is set to None, but you can specify `Tail` to capture the results of the Lambda execution log in the response, as shown following.

```
SELECT *, select convert_from(decode(log_result, 'base64'), 'utf-8') as log FROM
 aws_lambda.invoke(:'aws_lambda_arn_1', '{"body": "Hello from Postgres!"}'::json,
 'RequestResponse', 'Tail');
```

Set the aws_lambda.invoke function's `log_type` parameter to `Tail` to include the execution log in the response. The default value for the `log_type` parameter is None.

The `log_result` that's returned is a `base64` encoded string. You can decode the contents using a combination of the `decode` and `convert_from` PostgreSQL functions.

For more information about `log_type`, see aws_lambda.invoke.

Example: Including client context in a Lambda function

The `aws_lambda.invoke` function has a `context` parameter that you can use to pass information separate from the payload, as shown following.

```
SELECT *, convert_from(decode(log_result, 'base64'), 'utf-8') as log FROM
 aws_lambda.invoke(:'aws_lambda_arn_1', '{"body": "Hello from Postgres!"}'::json,
 'RequestResponse', 'Tail');
```

To include client context, use a JSON object for the aws_lambda.invoke function's `context` parameter.

For more information about the `context` parameter, see the aws_lambda.invoke reference.

Example: Invoking a specific version of a Lambda function

You can specify a particular version of a Lambda function by including the `qualifier` parameter with the `aws_lambda.invoke` call. Following, you can find an example that does this using `'custom_version'` as an alias for the version.

```
SELECT * FROM aws_lambda.invoke('aws_lambda_arn_1', '{"body": "Hello from
 Postgres!"}'::json, 'RequestResponse', 'None', NULL, 'custom_version');
```

You can also supply a Lambda function qualifier with the function name details instead, as follows.

```
SELECT * FROM aws_lambda.invoke(aws_commons.create_lambda_function_arn('my-
function:custom_version', 'us-west-2'),
'{"body": "Hello from Postgres!"}'::json);
```

For more information about `qualifier` and other parameters, see the aws_lambda.invoke reference.

Lambda function error messages

In the following list you can find information about error messages, with possible causes and solutions.

- **VPC configuration issues**

  VPC configuration issues can raise the following error messages when trying to connect:

  ```
  ERROR:  invoke API failed
  DETAIL: AWS Lambda client returned 'Unable to connect to endpoint'.
  CONTEXT:  SQL function "invoke" statement 1
  ```

  A common cause for this error is improperly configured VPC security group. Make sure you have an outbound rule for TCP open on port 443 of your VPC security group so that your VPC can connect to the Lambda VPC.
- **Lack of permissions needed to invoke Lambda functions**

  If you see either of the following error messages, the user (role) invoking the function doesn't have proper permissions.

```
ERROR:  permission denied for schema aws_lambda
```

```
ERROR:  permission denied for function invoke
```

A user (role) must be given specific grants to invoke Lambda functions. For more information, see Step 6: Grant other users permission to invoke Lambda functions.

- **Improper handling of errors in your Lambda functions**

  If a Lambda function throws an exception during request processing, `aws_lambda.invoke` fails with a PostgreSQL error such as the following.

  ```
  SELECT * FROM aws_lambda.invoke('aws_lambda_arn_1', '{"body": "Hello from
   Postgres!"}'::json);
  ERROR:  lambda invocation failed
  DETAIL:  "arn:aws:lambda:us-west-2:555555555555:function:my-function" returned error
   "Unhandled", details: "<Error details string>".
  ```

  Be sure to handle errors in your Lambda functions or in your PostgreSQL application.

AWS Lambda function and parameter reference

Following is the reference for the functions and parameters to use for invoking Lambda with Aurora PostgreSQL .

Functions and parameters

- aws_lambda.invoke
- aws_commons.create_lambda_function_arn
- aws_lambda parameters

aws_lambda.invoke

Runs a Lambda function for an Aurora PostgreSQL DB cluster .

For more details about invoking Lambda functions, see also Invoke in the *AWS Lambda Developer Guide.*

Syntax

JSON

```
aws_lambda.invoke(
IN function_name TEXT,
IN payload JSON,
IN region TEXT DEFAULT NULL,
IN invocation_type TEXT DEFAULT 'RequestResponse',
IN log_type TEXT DEFAULT 'None',
IN context JSON DEFAULT NULL,
IN qualifier VARCHAR(128) DEFAULT NULL,
OUT status_code INT,
OUT payload JSON,
OUT executed_version TEXT,
OUT log_result TEXT)
```

```
aws_lambda.invoke(
IN function_name aws_commons._lambda_function_arn_1,
IN payload JSON,
IN invocation_type TEXT DEFAULT 'RequestResponse',
IN log_type TEXT DEFAULT 'None',
IN context JSON DEFAULT NULL,
IN qualifier VARCHAR(128) DEFAULT NULL,
OUT status_code INT,
OUT payload JSON,
OUT executed_version TEXT,
OUT log_result TEXT)
```

JSONB

```
aws_lambda.invoke(
IN function_name TEXT,
IN payload JSONB,
IN region TEXT DEFAULT NULL,
IN invocation_type TEXT DEFAULT 'RequestResponse',
IN log_type TEXT DEFAULT 'None',
IN context JSONB DEFAULT NULL,
IN qualifier VARCHAR(128) DEFAULT NULL,
OUT status_code INT,
OUT payload JSONB,
OUT executed_version TEXT,
OUT log_result TEXT)
```

```
aws_lambda.invoke(
IN function_name aws_commons._lambda_function_arn_1,
IN payload JSONB,
IN invocation_type TEXT DEFAULT 'RequestResponse',
IN log_type TEXT DEFAULT 'None',
IN context JSONB DEFAULT NULL,
IN qualifier VARCHAR(128) DEFAULT NULL,
OUT status_code INT,
OUT payload JSONB,
OUT executed_version TEXT,
OUT log_result TEXT
)
```

Input parameters

function_name

The identifying name of the Lambda function. The value can be the function name, an ARN, or a partial ARN. For a listing of possible formats, see Lambda function name formats in the *AWS Lambda Developer Guide.*

payload

The input for the Lambda function. The format can be JSON or JSONB. For more information, see JSON Types in the PostgreSQL documentation.

region

(Optional) The Lambda Region for the function. By default, Aurora resolves the AWS Region from the full ARN in the `function_name` or it uses the Aurora PostgreSQL DB instance Region. If this Region value conflicts with the one provided in the `function_name` ARN, an error is raised.

invocation_type

The invocation type of the Lambda function. The value is case-sensitive. Possible values include the following:

- `RequestResponse` – The default. This type of invocation for a Lambda function is synchronous and returns a response payload in the result. Use the `RequestResponse` invocation type when your workflow depends on receiving the Lambda function result immediately.

- `Event` – This type of invocation for a Lambda function is asynchronous and returns immediately without a returned payload. Use the `Event` invocation type when you don't need results of the Lambda function before your workflow moves on.
- `DryRun` – This type of invocation tests access without running the Lambda function.

log_type

The type of Lambda log to return in the `log_result` output parameter. The value is case-sensitive. Possible values include the following:

- Tail – The returned `log_result` output parameter will include the last 4 KB of the execution log.
- None – No Lambda log information is returned.

context

Client context in JSON or JSONB format. Fields to use include than `custom` and `env`.

qualifier

A qualifier that identifies a Lambda function's version to be invoked. If this value conflicts with one provided in the `function_name` ARN, an error is raised.

Output parameters

status_code

An HTTP status response code. For more information, see Lambda Invoke response elements in the *AWS Lambda Developer Guide.*

payload

The information returned from the Lambda function that ran. The format is in JSON or JSONB.

executed_version

The version of the Lambda function that ran.

log_result

The execution log information returned if the `log_type` value is `Tail` when the Lambda function was invoked. The result contains the last 4 KB of the execution log encoded in Base64.

aws_commons.create_lambda_function_arn

Creates an `aws_commons._lambda_function_arn_1` structure to hold Lambda function name information. You can use the results of the `aws_commons.create_lambda_function_arn` function in the `function_name` parameter of the aws_lambda.invoke [aws_lambda.invoke](#) function.

Syntax

```
aws_commons.create_lambda_function_arn(
    function_name TEXT,
    region TEXT DEFAULT NULL
    )
    RETURNS aws_commons._lambda_function_arn_1
```

Input parameters

function_name

A required text string containing the Lambda function name. The value can be a function name, a partial ARN, or a full ARN.

region

An optional text string containing the AWS Region that the Lambda function is in. For a listing of Region names and associated values, see [Regions and Availability Zones](#).

aws_lambda parameters

In this table, you can find parameters associated with the `aws_lambda` function.

Parameter	Description
`aws_lambda.connect_timeout_ms`	This is a dynamic parameter and it sets the maximum wait time while connecting to AWS Lambda. The default values is `1000`. Allowed values for this parameter are 1 - 900000.
`aws_lambda.request_timeout_ms`	This is a dynamic parameter and it sets the maximum wait time while waiting for response from AWS Lambda. The default values is `3000`. Allowed values for this parameter are 1 - 900000.

Parameter	Description
`aws_lambda.endpoint_override`	Specifies the endpoint that can be used to connect to AWS Lambda. An empty string selects the default AWS Lambda endpoint for the region. You must restart the database for this static parameter change to take effect.

Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs

You can configure your Aurora PostgreSQL DB cluster to export log data to Amazon CloudWatch Logs on a regular basis. When you do so, events from your Aurora PostgreSQL DB cluster's PostgreSQL log are automatically *published* to Amazon CloudWatch, as Amazon CloudWatch Logs. In CloudWatch, you can find the exported log data in a *Log group* for your Aurora PostgreSQL DB cluster. The log group contains one or more *log streams* that contain the events from the PostgreSQL log from each instance in the cluster.

Publishing the logs to CloudWatch Logs allows you to keep your cluster's PostgreSQL log records in highly durable storage. With the log data available in CloudWatch Logs, you can evaluate and improve your cluster's operations. You can also use CloudWatch to create alarms and view metrics. To learn more, see Monitoring log events in Amazon CloudWatch.

> **Note**
>
> Publishing your PostgreSQL logs to CloudWatch Logs consumes storage, and you incur charges for that storage. Be sure to delete any CloudWatch Logs that you no longer need.

Turning the export log option off for an existing Aurora PostgreSQL DB cluster doesn't affect any data that's already held in CloudWatch Logs. Existing logs remain available in CloudWatch Logs based on your log retention settings. To learn more about CloudWatch Logs, see What is Amazon CloudWatch Logs?

Aurora PostgreSQL supports publishing logs to CloudWatch Logs for the following versions.

- 14.3 and higher 14 versions
- 13.3 and higher 13 versions
- 12.8 and higher 12 versions

- 11.12 and higher 11 versions

Turning on the option to publish logs to Amazon CloudWatch

To publish your Aurora PostgreSQL DB cluster's PostgreSQL log to CloudWatch Logs, choose the **Log export** option for the cluster. You can choose the Log export setting when you create your Aurora PostgreSQL DB cluster. Or, you can modify the cluster later on. When you modify an existing cluster, its PostgreSQL logs from each instance are published to CloudWatch cluster from that point on. For Aurora PostgreSQL, the PostgreSQL log (`postgresql.log`) is the only log that gets published to Amazon CloudWatch.

You can use the AWS Management Console, the AWS CLI, or the RDS API to turn on the Log export feature for your Aurora PostgreSQL DB cluster.

Console

You choose the Log exports option to start publishing the PostgreSQL logs from your Aurora PostgreSQL DB cluster to CloudWatch Logs.

To turn on the Log export feature from the console

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora PostgreSQL DB cluster whose log data you want to publish to CloudWatch Logs.
4. Choose **Modify**.
5. In the **Log exports** section, choose **PostgreSQL log**.
6. Choose **Continue**, and then choose **Modify cluster** on the summary page.

AWS CLI

You can turn on the log export option to start publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs with the AWS CLI. To do so, run the modify-db-cluster AWS CLI command with the following options:

- `--db-cluster-identifier`—The DB cluster identifier.
- `--cloudwatch-logs-export-configuration`—The configuration setting for the log types to be set for export to CloudWatch Logs for the DB cluster.

You can also publish Aurora PostgreSQL logs by running one of the following AWS CLI commands:

- create-db-cluster
- restore-db-cluster-from-s3
- restore-db-cluster-from-snapshot
- restore-db-cluster-to-point-in-time

Run one of these AWS CLI commands with the following options:

- `--db-cluster-identifier`—The DB cluster identifier.
- `--engine`—The database engine.
- `--enable-cloudwatch-logs-exports`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

Other options might be required depending on the AWS CLI command that you run.

Example

The following command creates an Aurora PostgreSQL DB cluster to publish log files to CloudWatch Logs.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
    --db-cluster-identifier my-db-cluster \
    --engine aurora-postgresql \
    --enable-cloudwatch-logs-exports postgresql
```

For Windows:

```
aws rds create-db-cluster ^
    --db-cluster-identifier my-db-cluster ^
    --engine aurora-postgresql ^
    --enable-cloudwatch-logs-exports postgresql
```

Example

The following command modifies an existing Aurora PostgreSQL DB cluster to publish log files to CloudWatch Logs. The `--cloudwatch-logs-export-configuration` value is a JSON object. The key for this object is `EnableLogTypes`, and its value is `postgresql`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier my-db-cluster \
    --cloudwatch-logs-export-configuration '{"EnableLogTypes":["postgresql"]}'
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier my-db-cluster ^
    --cloudwatch-logs-export-configuration '{\"EnableLogTypes\":[\"postgresql\"]}'
```

Note

When using the Windows command prompt, make sure to escape double quotation marks (") in JSON code by prefixing them with a backslash (\).

Example

The following example modifies an existing Aurora PostgreSQL DB cluster to disable publishing log files to CloudWatch Logs. The `--cloudwatch-logs-export-configuration` value is a JSON object. The key for this object is `DisableLogTypes`, and its value is `postgresql`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbinstance \
    --cloudwatch-logs-export-configuration '{"DisableLogTypes":["postgresql"]}'
```

For Windows:

```
aws rds modify-db-cluster ^
```

```
    --db-cluster-identifier mydbinstance ^
    --cloudwatch-logs-export-configuration "{\"DisableLogTypes\":[\"postgresql\"]}"
```

> **Note**
>
> When using the Windows command prompt, you must escape double quotes (") in JSON code by prefixing them with a backslash (\).

RDS API

You can turn on the log export option to start publishing Aurora PostgreSQL logs with the RDS API. To do so, run the ModifyDBCluster operation with the following options:

- `DBClusterIdentifier` – The DB cluster identifier.
- `CloudwatchLogsExportConfiguration` – The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

You can also publish Aurora PostgreSQL logs with the RDS API by running one of the following RDS API operations:

- CreateDBCluster
- RestoreDBClusterFromS3
- RestoreDBClusterFromSnapshot
- RestoreDBClusterToPointInTime

Run the RDS API action with the following parameters:

- `DBClusterIdentifier`—The DB cluster identifier.
- `Engine`—The database engine.
- `EnableCloudwatchLogsExports`—The configuration setting for the log types to be enabled for export to CloudWatch Logs for the DB cluster.

Other parameters might be required depending on the AWS CLI command that you run.

Monitoring log events in Amazon CloudWatch

With Aurora PostgreSQL log events published and available as Amazon CloudWatch Logs, you can view and monitor events using Amazon CloudWatch. For more information about monitoring, see View log data sent to CloudWatch Logs.

When you turn on Log exports, a new log group is automatically created using the prefix `/aws/rds/cluster/` with the name of your Aurora PostgreSQL and the log type, as in the following pattern.

```
/aws/rds/cluster/your-cluster-name/postgresql
```

As an example, suppose that an Aurora PostgreSQL DB cluster named `docs-lab-apg-small` exports its log to Amazon CloudWatch Logs. Its log group name in Amazon CloudWatch is shown following.

```
/aws/rds/cluster/docs-lab-apg-small/postgresql
```

If a log group with the specified name exists, Aurora uses that log group to export log data for the Aurora DB cluster. Each DB instance in the Aurora PostgreSQL DB cluster uploads its PostgreSQL log to the log group as a distinct log stream. You can examine the log group and its log streams using the various graphical and analytical tools available in Amazon CloudWatch.

For example, you can search for information within the log events from your Aurora PostgreSQL DB cluster, and filter events by using the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API. For more information, Searching and filtering log data in the *Amazon CloudWatch Logs User Guide*.

By default, new log groups are created using **Never expire** for their retention period. You can use the CloudWatch Logs console, the AWS CLI, or the CloudWatch Logs API to change the log retention period. To learn more, see Change log data retention in CloudWatch Logs in the *Amazon CloudWatch Logs User Guide*.

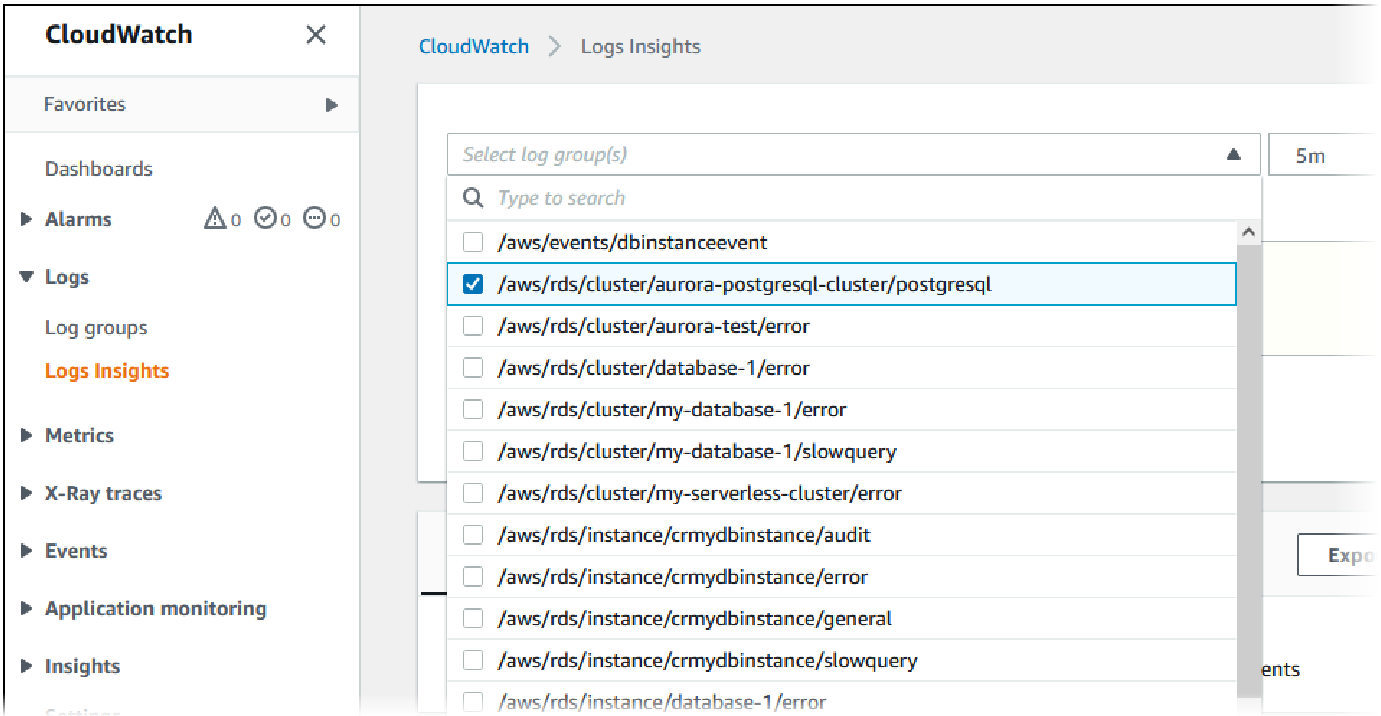
> **Tip**
>
> You can use automated configuration, such as AWS CloudFormation, to create log groups with predefined log retention periods, metric filters, and access permissions.

Analyzing PostgreSQL logs using CloudWatch Logs Insights

With the PostgreSQL logs from your Aurora PostgreSQL DB cluster published as CloudWatch Logs, you can use CloudWatch Logs Insights to interactively search and analyze your log data in Amazon CloudWatch Logs. CloudWatch Logs Insights includes a query language, sample queries, and other tools for analyzing your log data so that you can identify potential issues and verify fixes. To learn more, see Analyzing log data with CloudWatch Logs Insights in the *Amazon CloudWatch Logs User Guide*. Amazon CloudWatch Logs

To analyze PostgreSQL logs with CloudWatch Logs Insights

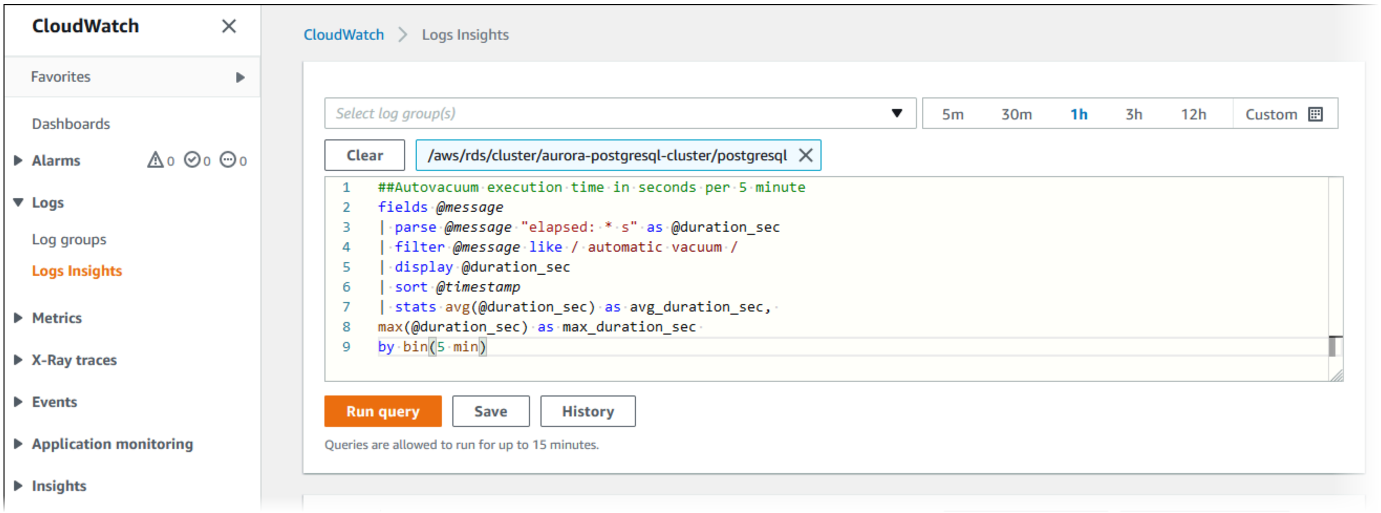
1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, open **Logs** and choose **Log insights**.
3. In **Select log group(s)**, select the log group for your Aurora PostgreSQL DB cluster.



□ /aws/events/dbinstanceevent
☑ /aws/rds/cluster/aurora-postgresql-cluster/postgresql
□ /aws/rds/cluster/aurora-test/error
□ /aws/rds/cluster/database-1/error
□ /aws/rds/cluster/my-database-1/error
□ /aws/rds/cluster/my-database-1/slowquery
□ /aws/rds/cluster/my-serverless-cluster/error
□ /aws/rds/instance/crmydbinstance/audit
□ /aws/rds/instance/crmydbinstance/error
□ /aws/rds/instance/crmydbinstance/general
□ /aws/rds/instance/crmydbinstance/slowquery
□ /aws/rds/instance/database-1/error

4. In the query editor, delete the query that is currently shown, enter the following, and then choose **Run query**.

```
##Autovacuum execution time in seconds per 5 minute
fields @message
| parse @message "elapsed: * s" as @duration_sec
| filter @message like / automatic vacuum /
| display @duration_sec
| sort @timestamp
| stats avg(@duration_sec) as avg_duration_sec,
max(@duration_sec) as max_duration_sec
by bin(5 min)
```

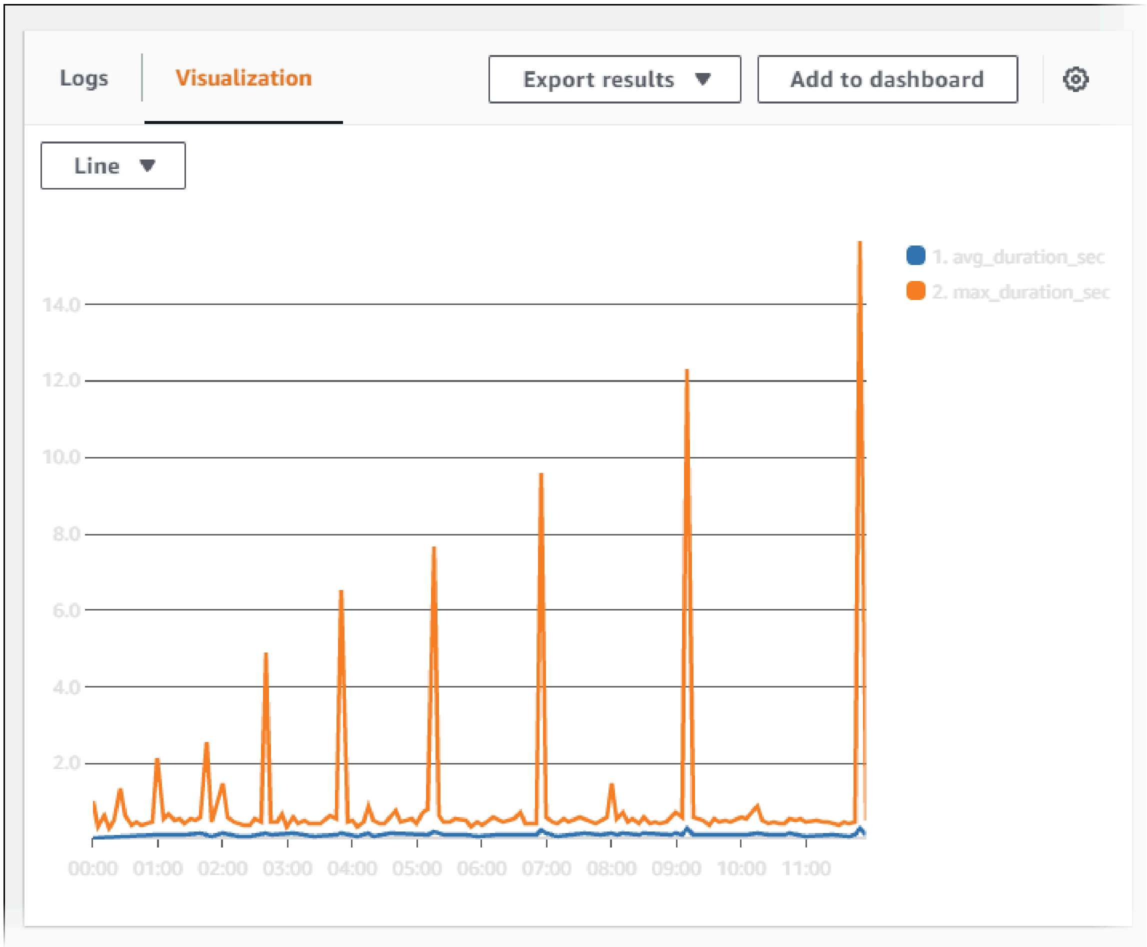


/aws/rds/cluster/aurora-postgresql-cluster/postgresql

```
##Autovacuum execution time in seconds per 5 minute
fields @message
| parse @message "elapsed: * s" as @duration_sec
| filter @message like / automatic vacuum /
| display @duration_sec
| sort @timestamp
| stats avg(@duration_sec) as avg_duration_sec,
max(@duration_sec) as max_duration_sec
by bin(5 min)
```

Queries are allowed to run for up to 15 minutes.

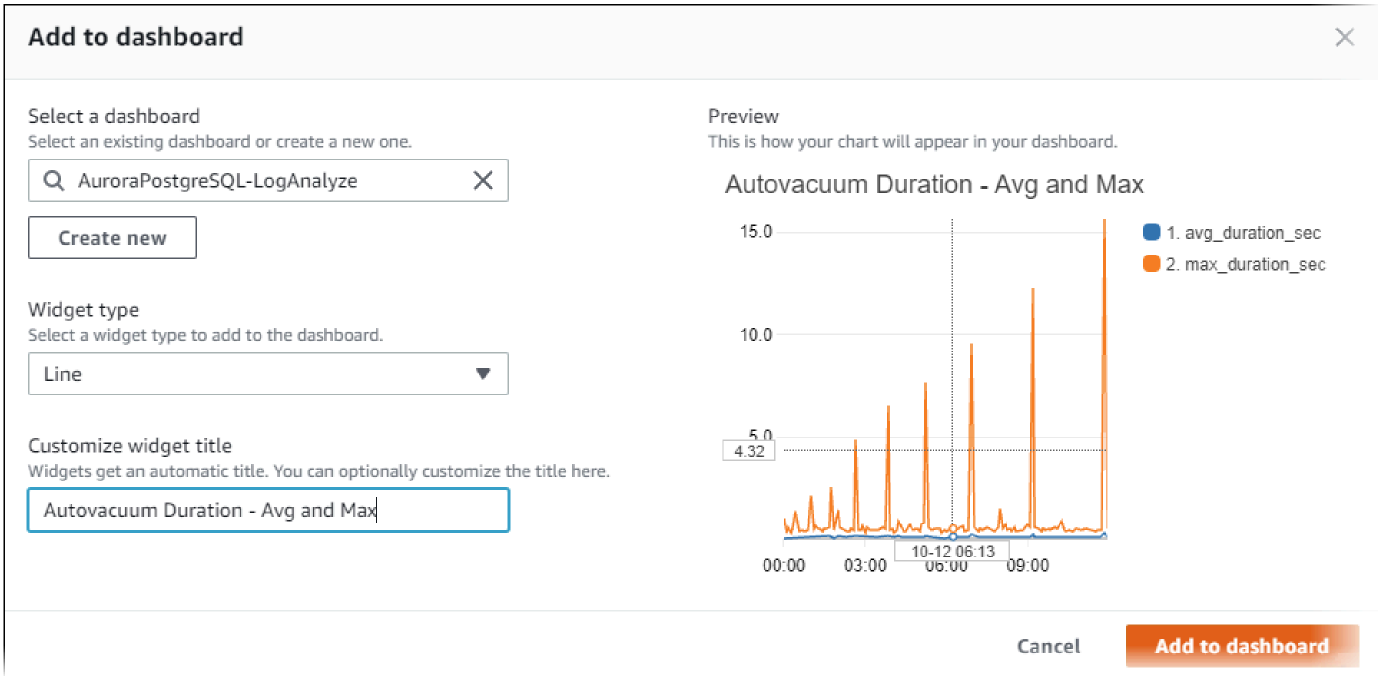
5. Choose the **Visualization** tab.



6. Choose **Add to dashboard**.

7. In **Select a dashboard**, either select a dashboard or enter a name to create a new dashboard.

8. In **Widget type**, choose a widget type for your visualization.



Add to dashboard

Select a dashboard

Select an existing dashboard or create a new one.

AuroraPostgreSQL-LogAnalyze

Create new

Widget type

Select a widget type to add to the dashboard.

Line

Customize widget title

Widgets get an automatic title. You can optionally customize the title here.

Autovacuum Duration - Avg and Max

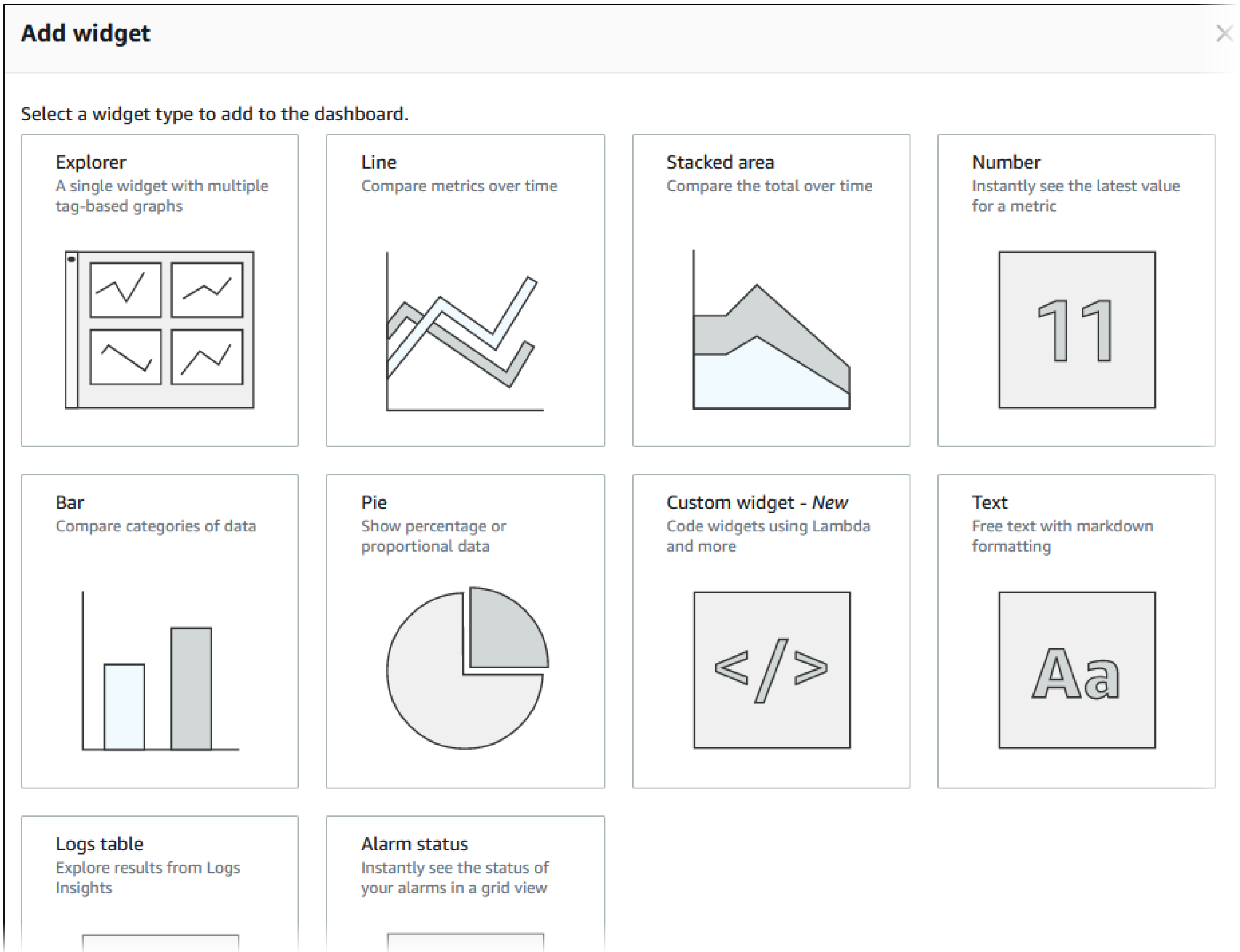
Preview

This is how your chart will appear in your dashboard.

9. (Optional) Add more widgets based on your log query results.

   a. Choose **Add widget**.

   b. Choose a widget type, such as **Line**.



Add widget

Select a widget type to add to the dashboard.

Explorer
A single widget with multiple tag-based graphs

Line
Compare metrics over time

Stacked area
Compare the total over time

Number
Instantly see the latest value for a metric

Bar
Compare categories of data

Pie
Show percentage or proportional data

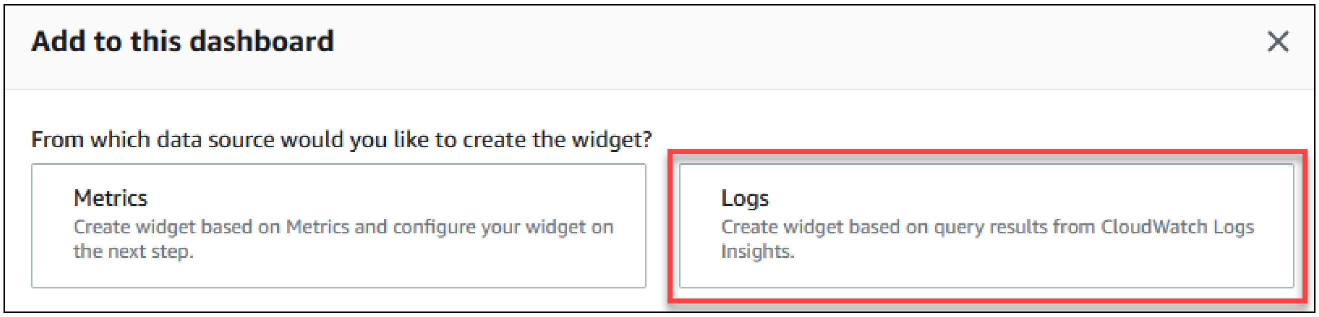
Custom widget - *New*
Code widgets using Lambda and more

Text
Free text with markdown formatting

Logs table
Explore results from Logs Insights

Alarm status
Instantly see the status of your alarms in a grid view

c. In the **Add to this dashboard** window, choose **Logs**.



Add to this dashboard

From which data source would you like to create the widget?

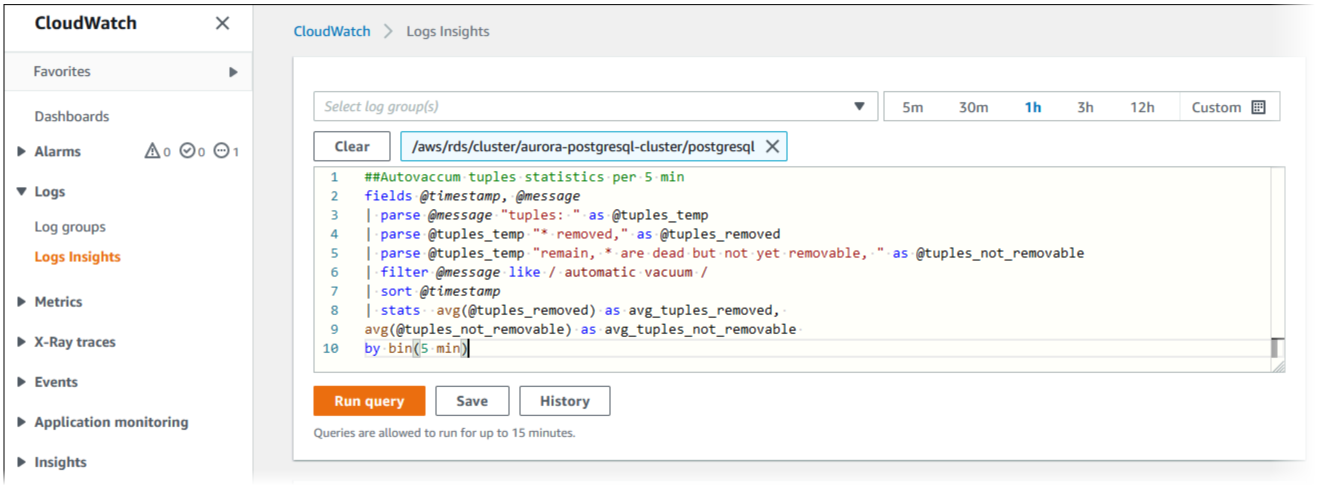
Metrics
Create widget based on Metrics and configure your widget on the next step.

Logs
Create widget based on query results from CloudWatch Logs Insights.

d. In **Select log group(s)**, select the log group for your DB cluster.

e. In the query editor, delete the query that is currently shown, enter the following, and then choose **Run query**.

```
##Autovacuum tuples statistics per 5 min
fields @timestamp, @message
| parse @message "tuples: " as @tuples_temp
| parse @tuples_temp "* removed," as @tuples_removed
| parse @tuples_temp "remain, * are dead but not yet removable, " as
 @tuples_not_removable
| filter @message like / automatic vacuum /
| sort @timestamp
| stats  avg(@tuples_removed) as avg_tuples_removed,
avg(@tuples_not_removable) as avg_tuples_not_removable
by bin(5 min)
```



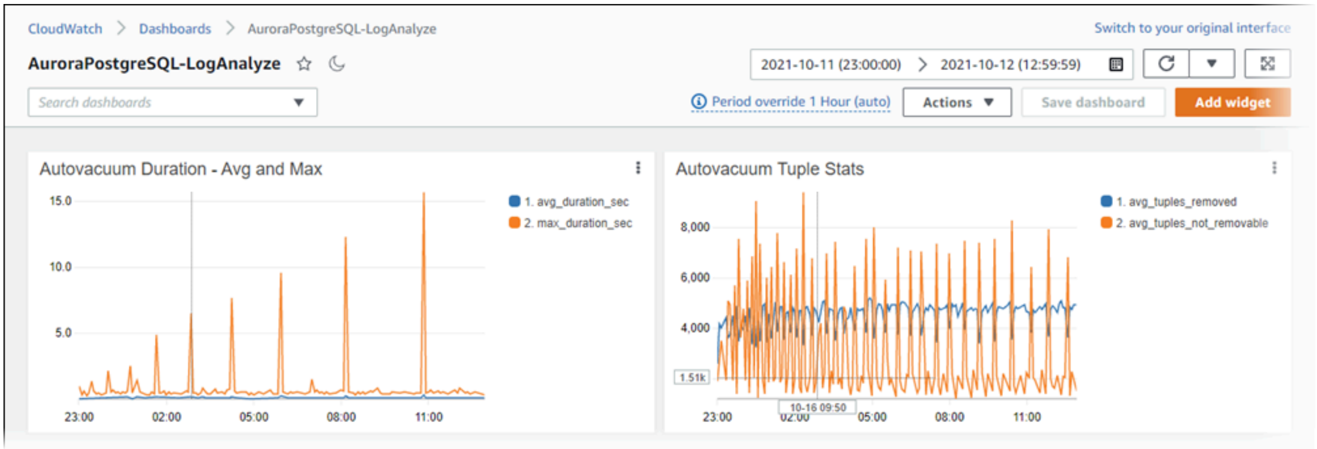
/aws/rds/cluster/aurora-postgresql-cluster/postgresql

```
##Autovaccum tuples statistics per 5 min
fields @timestamp, @message
| parse @message "tuples: " as @tuples_temp
| parse @tuples_temp "* removed," as @tuples_removed
| parse @tuples_temp "remain, * are dead but not yet removable, " as @tuples_not_removable
| filter @message like / automatic vacuum /
| sort @timestamp
| stats  avg(@tuples_removed) as avg_tuples_removed,
avg(@tuples_not_removable) as avg_tuples_not_removable
by bin(5 min)
```

Queries are allowed to run for up to 15 minutes.

f. Choose **Create widget**.

Your dashboard should look similar to the following image.



Monitoring query execution plans for Aurora PostgreSQL

You can monitor query execution plans in your Aurora PostgreSQL DB instance to detect the execution plans contributing to current database load and to track performance statistics of execution plans over time using `aurora_compute_plan_id` parameter. Whenever a query executes, the execution plan used by the query is assigned an identifier and the same identifier is used by subsequent executions of the same plan.

The `aurora_compute_plan_id` is turned on by default in DB parameter group from Aurora PostgreSQL versions 14.10, 15.5, and higher versions. Assignment of a plan identifier is default behavior and can be turned off by setting `aurora_compute_plan_id` to OFF in the parameter group.

This plan identifier is used in several utilities that serve a different purpose.

Topics

- Accessing query execution plans using Aurora functions
- Parameter reference for Aurora PostgreSQL query execution plans

Accessing query execution plans using Aurora functions

With `aurora_compute_plan_id`, you can access the execution plans using the following functions:

- aurora_stat_activity
- aurora_stat_plans

For more information on these functions, see Aurora PostgreSQL functions reference.

Parameter reference for Aurora PostgreSQL query execution plans

You can monitor the query execution plans using the below parameters in a DB parameter group.

Parameters

- aurora_compute_plan_id
- aurora_stat_plans.minutes_until_recapture

- aurora_stat_plans.calls_until_recapture
- aurora_stat_plans.with_costs
- aurora_stat_plans.with_analyze
- aurora_stat_plans.with_timing
- aurora_stat_plans.with_buffers
- aurora_stat_plans.with_wal
- aurora_stat_plans.with_triggers

> **Note**
>
> The configuration for `aurora_stat_plans.with_*` parameters takes effect only for newly captured plans.

aurora_compute_plan_id

Set to `off` to prevent a plan identifier from being assigned.

Default	Allowed values	Description
on	0(off)	Set to `off` to prevent a plan identifier from being assigned.
	1(on)	Set to on to assign a plan identifier.

aurora_stat_plans.minutes_until_recapture

The number of minutes to pass before a plan is recaptured. Default is 0 which will disable recapturing a plan. When the `aurora_stat_plans.calls_until_recapture` threshold is passed, the plan will be recaptured.

Default	Allowed values	Description
0	0-1073741823	Set the number of minutes to pass before a plan is recaptured.

aurora_stat_plans.calls_until_recapture

The number of calls to a plan before it is recaptured. Default is 0 which will disable recapturing a plan after a number of calls. When the `aurora_stat_plans.minutes_until_recapture` threshold is passed, the plan will be recaptured.

Default	Allowed values	Description
0	0-1073741823	Set the number of calls before a plan is recaptured.

aurora_stat_plans.with_costs

Captures an EXPLAIN plan with estimated costs. The allowed values are on and `off`. The default is on.

Default	Allowed values	Description
on	0(off)	Doesn't show estimated cost and rows for each plan node.
	1(on)	Shows estimated cost and rows for each plan node.

aurora_stat_plans.with_analyze

Controls the EXPLAIN plan with ANALYZE. This mode is only used the first time a plan is captured. The allowed values are on and `off`. The default is `off`.

Default	Allowed values	Description
off	0(off)	Doesn't include actual run time statistics for the plan.
	1(on)	Includes actual run time statistics for the plan.

aurora_stat_plans.with_timing

Plan timing will be captured in the explain when ANALYZE is used. The default is on.

Default	Allowed values	Description
on	0(off)	Doesn't include actual start up time and time spent in each plan node.
	1(on)	Includes actual start up time and time spent in each plan node.

aurora_stat_plans.with_buffers

Plan buffer usage statistics will be captured in the explain when ANALYZE is used. The default is `off`.

Default	Allowed values	Description
off	0(off)	Doesn't include information on buffer usage.
	1(on)	Includes information on buffer usage.

aurora_stat_plans.with_wal

Plan wal usage statistics will be captured in the explain when ANALYZE is used. The default is `off`.

Default	Allowed values	Description
off	0(off)	Doesn't include information on WAL record generation.
	1(on)	Includes information on WAL record generation.

aurora_stat_plans.with_triggers

Plan trigger execution statistics will be captured in the explain when `ANALYZE` is used. The default is `off`.

Default	Allowed values	Description
off	0(off)	Doesn't include triggers execution statistics.

Default	Allowed values	Description
	1(on)	Includes triggers execution statistics.

Managing query execution plans for Aurora PostgreSQL

Aurora PostgreSQL query plan management is an optional feature that you can use with your Amazon Aurora PostgreSQL-Compatible Edition DB cluster. This feature is packaged as the `apg_plan_mgmt` extension that you can install in your Aurora PostgreSQL DB cluster. Query plan management allows you to manage the query execution plans generated by the optimizer for your SQL applications. The `apg_plan_mgmt` AWS extension builds on the native query processing functionality of the PostgreSQL database engine.

Following, you can find information about Aurora PostgreSQL query plan management features, how to set it up, and how to use it with your Aurora PostgreSQL DB cluster. Before you get started, we recommend that you review any release notes for the specific version of the `apg_plan_mgmt` extension available for your Aurora PostgreSQL version. For more information, see Aurora PostgreSQL apg_plan_mgmt extension versions in the *Release Notes for Aurora PostgreSQL*.

Topics

Overview of Aurora PostgreSQL query plan management

Aurora PostgreSQL query plan management is designed to ensure plan stability regardless of changes to the database that might cause query plan regression. *Query plan regression* occurs when the optimizer chooses a sub-optimal plan for a given SQL statement after system or database changes. Changes to statistics, constraints, environment settings, query parameter bindings, and upgrades to the PostgreSQL database engine can all cause plan regression.

With Aurora PostgreSQL query plan management, you can control how and when query execution plans change. The benefits of Aurora PostgreSQL query plan management include the following.

- Improve plan stability by forcing the optimizer to choose from a small number of known, good plans.
- Optimize plans centrally and then distribute the best plans globally.
- Identify indexes that aren't used and assess the impact of creating or dropping an index.
- Automatically detect a new minimum-cost plan discovered by the optimizer.
- Try new optimizer features with less risk, because you can choose to approve only the plan changes that improve performance.

You can use the tools provided by query plan management proactively, to specify the best plan for certain queries. Or you can use query plan management to react to changing circumstances and avoid plan regressions. For more information, see Best practices for Aurora PostgreSQL query plan management.

Topics

Supported SQL statements

Query plan management supports the following types of SQL statements.

- Any SELECT, INSERT, UPDATE, or DELETE statement, regardless of complexity.
- Prepared statements. For more information, see PREPARE in the PostgreSQL documentation.
- Dynamic statements, including those run in immediate-mode. For more information, see Dynamic SQL and EXECUTE IMMEDIATE in PostgreSQL documentation.
- Embedded SQL commands and statements. For more information, see Embedded SQL Commands in the PostgreSQL documentation.
- Statements inside named functions. For more information, see CREATE FUNCTION in the PostgreSQL documentation.

- Statements containing temp tables.
- Statements inside procedures and DO-blocks.

You can use query plan management with EXPLAIN in manual mode to capture a plan without actually running it. For more information, see Analyzing the optimizer's chosen plan. To learn more about query plan management's modes (manual, automatic), see Capturing Aurora PostgreSQL execution plans.

Aurora PostgreSQL query plan management supports all PostgreSQL language features, including partitioned tables, inheritance, row-level security, and recursive common table expressions (CTEs). To learn more about these PostgreSQL language features, see Table Partitioning, Row Security Policies, and WITH Queries (Common Table Expressions) and other topics in the PostgreSQL documentation.

For information about different versions of the Aurora PostgreSQL query plan management feature, see Aurora PostgreSQL apg_plan_mgmt extension versions in the *Release Notes for Aurora PostgreSQL*.

Query plan management limitations

The current release of Aurora PostgreSQL query plan management has the following limitations.

- **Plans aren't captured for statements that reference system relations** – Statements that reference system relations, such as pg_class, aren't captured. This is by design, to prevent a large number of system-generated plans that are used internally from being captured. This also applies to system tables inside views.
- **Larger DB instance class might be needed for your Aurora PostgreSQL DB cluster** – Depending on the workload, query plan management might need a DB instance class that has more than 2 vCPUs. The number of max_worker_processes is limited by the DB instance class size. The number of max_worker_processes provided by a 2-vCPU DB instance class (db.t3.medium, for example) might not be sufficient for a given workload. We recommend that you choose a DB instance class with more than 2 vCPUs for your Aurora PostgreSQL DB cluster if you use query plan managment.

  When the DB instance class can't support the workload, query plan management raises an error message such as the following.

  ```
  WARNING: could not register plan insert background process
  ```

```
HINT: You may need to increase max_worker_processes.
```

In this case, you should scale up your Aurora PostgreSQL DB cluster to a DB instance class size with more memory. For more information, see Supported DB engines for DB instance classes.

- **Plans already stored in sessions aren't affected** – Query plan management provides a way to influence query plans without changing the application code. However, when a generic plan is already stored in an existing session and if you want to change its query plan, then you must first setplan_cache_mode to force_custom_plan in the DB cluster parameter group.
- queryid in apg_plan_mgmt.dba_plans and pg_stat_statements can diverge when:
  - Objects are dropped and recreated after storing in apg_plan_mgmt.dba_plans.
  - apg_plan_mgmt.plans table is imported from another cluster.

For information about different versions of the Aurora PostgreSQL query plan management feature, see Aurora PostgreSQL apg_plan_mgmt extension versions in the *Release Notes for Aurora PostgreSQL*.

Query plan management terminology

The following terms are used throughout this topic.

managed statement

A SQL statement captured by the optimizer under query plan management. A managed statement has one or more query execution plans stored in the apg_plan_mgmt.dba_plans view.

plan baseline

The set of approved plans for a given managed statement. That is, all the plans for the managed statement that have "Approved" for their status column in the dba_plan view.

plan history

The set of all captured plans for a given managed statement. The plan history contains all plans captured for the statement, regardless of status.

query plan regression

The case when the optimizer chooses a less optimal plan than it did before a given change to the database environment, such as a new PostgreSQL version or changes to statistics.

Aurora PostgreSQL query plan management versions

Query plan management is supported by all currently available Aurora PostgreSQL releases. For more information, see the list of Amazon Aurora PostgreSQL updates in the *Release Notes for Aurora PostgreSQL*.

Query plan management functionality is added to your Aurora PostgreSQL DB cluster when you install the `apg_plan_mgmt` extension. Different versions of Aurora PostgreSQL support different versions of the `apg_plan_mgmt` extension. We recommend that you upgrade the query plan management extension to the latest release for your version of Aurora PostgreSQL.

> **Note**
>
> For release notes for each `apg_plan_mgmt` extension versions, see Aurora PostgreSQL apg_plan_mgmt extension versions in the *Release Notes for Aurora PostgreSQL*.

You can identify the version running on your cluster by connecting to an instance using `psql` and using the metacommand \dx to list extensions as shown following.

```
labdb=> \dx
                       List of installed extensions
     Name      | Version |    Schema     |                            Description
---------------+---------+---------------
+-------------------------------------------------------------------
 apg_plan_mgmt | 1.0     | apg_plan_mgmt | Amazon Aurora with PostgreSQL compatibility
 Query Plan Management
 plpgsql       | 1.0     | pg_catalog    | PL/pgSQL procedural language
(2 rows)
```

The output shows that this cluster is using 1.0 version of the extension. Only certain `apg_plan_mgmt` versions are available for a given Aurora PostgreSQL version. In some cases, you might need to upgrade the Aurora PostgreSQL DB cluster to a new minor release or apply a patch so that you can upgrade to the most recent version of query plan management. The `apg_plan_mgmt` version 1.0 shown in the output is from an Aurora PostgreSQL version 10.17 DB cluster, which doesn't have a newer version of `apg_plan_mgmt` available. In this case, the Aurora PostgreSQL DB cluster should be upgraded to a more recent version of PostgreSQL.

For more information about upgrading your Aurora PostgreSQL DB cluster to a new version of PostgreSQL, see Amazon Aurora PostgreSQL updates.

To learn how to upgrade the `apg_plan_mgmt` extension, see Upgrading Aurora PostgreSQL query plan management.

Turning on Aurora PostgreSQL query plan management

Setting up query plan management for your Aurora PostgreSQL DB cluster involves installing an extension and changing several DB cluster parameter settings. You need `rds_superuser` permissions to install the `apg_plan_mgmt` extension and to turn on the feature for the Aurora PostgreSQL DB cluster.

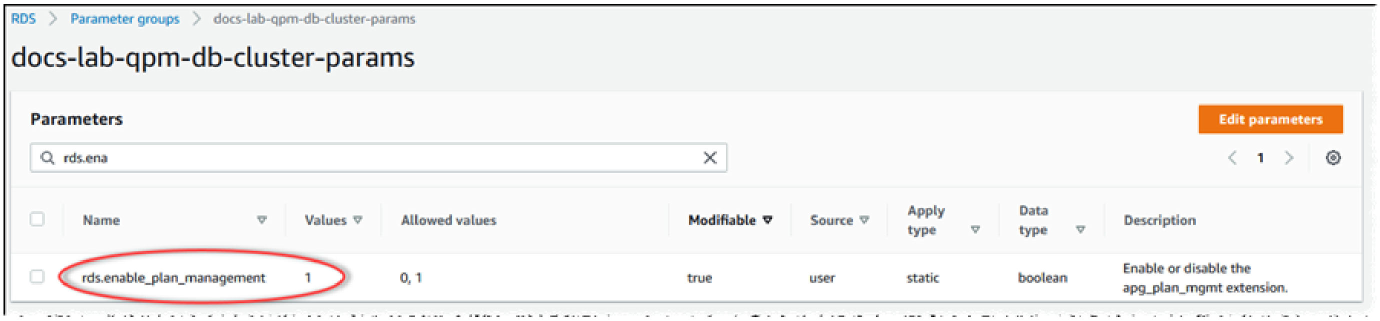
Installing the extension creates a new role, `apg_plan_mgmt`. This role allows database users to view, manage, and maintain query plans. As an administrator with `rds_superuser` privileges, be sure to grant the `apg_plan_mgmt` role to database users as needed.

Only users with the `rds_superuser` role can complete the following procedure. The `rds_superuser` is required for creating the `apg_plan_mgmt` extension and its `apg_plan_mgmt` role. Users must be granted the `apg_plan_mgmt` role to administer the `apg_plan_mgmt` extension.

To turn on query plan management for your Aurora PostgreSQL DB cluster

The following steps turn on query plan management for all SQL statements that get submitted to the Aurora PostgreSQL DB cluster. This is known as *automatic* mode. To learn more about the difference between modes, see Capturing Aurora PostgreSQL execution plans.

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Create a custom DB cluster parameter group for your Aurora PostgreSQL DB cluster. You need to change certain parameters to activate query plan management and to set its behavior. For more information, see Creating a DB parameter group.
3. Open the custom DB cluster parameter group and set the `rds.enable_plan_management` parameter to 1, as shown in the following image.



docs-lab-qpm-db-cluster-params

Parameters

rds.ena

Name	Values	Allowed values	Modifiable	Source	Apply type	Data type	Description
rds.enable_plan_management	1	0, 1	true	user	static	boolean	Enable or disable the apg_plan_mgmt extension.

For more information, see Modifying parameters in a DB cluster parameter group.

4. Create a custom DB parameter group that you can use to set query plan parameters at the instance level. For more information, see Creating a DB cluster parameter group.
5. Modify the writer instance of the Aurora PostgreSQL DB cluster to use the custom DB parameter group. For more information, see Modifying a DB instance in a DB cluster.
6. Modify the Aurora PostgreSQL DB cluster to use the custom DB cluster parameter group. For more information, see Modifying the DB cluster by using the console, CLI, and API.
7. Reboot your DB instance to enable the custom parameter group settings.
8. Connect to your Aurora PostgreSQL DB cluster's DB instance endpoint using `psql` or `pgAdmin`. The following example uses the default `postgres` account for the `rds_superuser` role.

```
psql --host=cluster-instance-1.111122223333.aws-region.rds.amazonaws.com --
port=5432 --username=postgres --password --dbname=my-db
```

9. Create the `apg_plan_mgmt` extension for your DB instance, as shown following.

```
labdb=> CREATE EXTENSION apg_plan_mgmt;
CREATE EXTENSION
```

> **Tip**
>
> Install the `apg_plan_mgmt` extension in the template database for your application. The default template database is named `template1`. To learn more, see Template Databases in the PostgreSQL documentation.

10. Change the `apg_plan_mgmt.capture_plan_baselines` parameter to `automatic`. This setting causes the optimizer to generate plans for every SQL statement that is either planned or executed two or more times.

> **Note**
>
> Query plan management also has a *manual* mode that you can use for specific SQL statements. To learn more, see Capturing Aurora PostgreSQL execution plans.

11. Change the value of `apg_plan_mgmt.use_plan_baselines` parameter to "on." This parameter causes the optimizer to choose a plan for the statement from its plan baseline. To learn more, see Using Aurora PostgreSQL managed plans.

> **Note**
>
> You can modify the value of either of these dynamic parameters for the session without needing to reboot the instance.

When your query plan management set up is complete, be sure to grant the `apg_plan_mgmt` role to any database users that need to view, manage, or maintain query plans.

Upgrading Aurora PostgreSQL query plan management

We recommend that you upgrade the query plan management extension to the latest release for your version of Aurora PostgreSQL.

1. Connect to the writer instance of your Aurora PostgreSQL DB cluster as a user that has `rds_superuser` privileges. If you kept the default name when you set up your instance, you connect as `postgres` This example shows how to use `psql`, but you can also use pgAdmin if you prefer.

```
psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password
```

2. Run the following query to upgrade the extension.

```
ALTER EXTENSION apg_plan_mgmt UPDATE TO '2.1';
```

3. Use the apg_plan_mgmt.validate_plans function to update the hashes of all plans. The optimizer validates all Approved, Unapproved, and Rejected plans to ensure that they's still viable plans for new version of the extension.

```
SELECT apg_plan_mgmt.validate_plans('update_plan_hash');
```

To learn more about using this function, see Validating plans.

4. Use the apg_plan_mgmt.reload function to refresh any plans in the shared memory with the validated plans from the dba_plans view.

```
SELECT apg_plan_mgmt.reload();
```

To learn more about all functions available for query plan management, see Function reference for Aurora PostgreSQL query plan management.

Turning off Aurora PostgreSQL query plan management

You can disable query plan management at any time by turning off the `apg_plan_mgmt.use_plan_baselines` and `apg_plan_mgmt.capture_plan_baselines`.

```
labdb=> SET apg_plan_mgmt.use_plan_baselines = off;

labdb=> SET apg_plan_mgmt.capture_plan_baselines = off;
```

Best practices for Aurora PostgreSQL query plan management

Query plan management lets you control how and when query execution plans change. As a DBA, your main goals when using QPM include preventing regressions when there are changes to your database, and controlling whether to allow the optimizer to use a new plan. In the following, you can find some recommended best practices for using query plan management. Proactive and reactive plan management approaches differ in how and when new plans get approved for use.

Contents

Proactive plan management to help prevent performance regression

To prevent plan performance regressions from occurring, you *evolve* plan baselines by running a procedure that compares the performance of newly discovered plans to the performance of the existing baseline of Approved plans, and then automatically approves the fastest set of plans as the new baseline. In this way, the baseline of plans improves over time as faster plans are discovered.

1. In a development environment, identify the SQL statements that have the greatest impact on performance or system throughput. Then capture the plans for these statements as described in Manually capturing plans for specific SQL statements and Automatically capturing plans.
2. Export the captured plans from the development environment and import them into the production environment. For more information, see Exporting and importing plans.
3. In production, run your application and enforce the use of approved managed plans. For more information, see Using Aurora PostgreSQL managed plans. While the application runs, also add new plans as the optimizer discovers them. For more information, see Automatically capturing plans.
4. Analyze the unapproved plans and approve those that perform well. For more information, see Evaluating plan performance.
5. While your application continues to run, the optimizer begins to use the new plans as appropriate.

Ensuring plan stability after a major version upgrade

Each major version of PostgreSQL includes enhancements and changes to the query optimizer that are designed to improve performance. However, query execution plans generated by the optimizer in earlier versions might cause performance regressions in newer upgraded versions. You can use query plan management to resolve these performance issues and to ensure plan stability after a major version upgrade.

The optimizer always uses a minimum-cost Approved plan, even if more than one Approved plan for the same statement exists. After an upgrade the optimizer might discover new plans but they will be saved as Unapproved plans. These plans are performed only if approved using the reactive style of plan management with the unapproved_plan_execution_threshold parameter. You can maximize plan stability using the proactive style of plan management with the evolve_plan_baselines parameter. This compares the performance of the new plans to the old plans and approves or rejects plans that are at least 10% faster than the next best plan.

After upgrading, you can use the `evolve_plan_baselines` function to compare plan performance before and after the upgrade using your query parameter bindings. The following steps assume that you have been using approved managed plans in your production environment, as detailed in Using Aurora PostgreSQL managed plans.

1. Before upgrading, run your application with the query plan manager running. While the application runs, add new plans as the optimizer discovers them. For more information, see Automatically capturing plans.
2. Evaluate each plan's performance. For more information, see Evaluating plan performance.
3. After upgrading, analyze your approved plans again using the `evolve_plan_baselines` function. Compare performance before and after using your query parameter bindings. If the new plan is fast, you can add it to your approved plans. If it's faster than another plan for the same parameter bindings, then you can mark the slower plan as Rejected.

   For more information, see Approving better plans. For reference information about this function, see apg_plan_mgmt.evolve_plan_baselines.

For more information, see Ensuring consistent performance after major version upgrades with Amazon Aurora PostgreSQL-Compatible Edition Query Plan Management.

Note

When you perform a major version upgrade using logical replication or AWS DMS, make sure that you replicate the `apg_plan_mgmt` schema to ensure existing plans are copied to the upgraded instance. For more information on logical replication, see Using logical replication to perform a major version upgrade for Aurora PostgreSQL.

Reactive plan management to detect and repair performance regressions

By monitoring your application as it runs, you can detect plans that cause performance regressions. When you detect regressions, you manually reject or fix the bad plans by following these steps:

1. While your application runs, enforce the use of managed plans and automatically add newly discovered plans as unapproved. For more information, see Using Aurora PostgreSQL managed plans and Automatically capturing plans.
2. Monitor your running application for performance regressions.
3. When you discover a plan regression, set the plan's status to `rejected`. The next time the optimizer runs the SQL statement, it automatically ignores the rejected plan and uses a different approved plan instead. For more information, see Rejecting or disabling slower plans.

In some cases, you might prefer to fix a bad plan rather than reject, disable, or delete it. Use the pg_hint_plan extension to experiment with improving a plan. With pg_hint_plan, you use special comments to tell the optimizer to override how it normally creates a plan. For more information, see Fixing plans using pg_hint_plan.

Understanding Aurora PostgreSQL query plan management

With query plan management turned on for your Aurora PostgreSQL DB cluster, the optimizer generates and stores query execution plans for any SQL statement that it processes more than once. The optimizer always sets the status of a managed statement's first generated plan to Approved, and stores it in the dba_plans view.

The set of approved plans saved for a managed statement is known as its *plan baseline*. As your application runs, the optimizer might generate additional plans for managed statements. The optimizer sets additional captured plans to a status of Unapproved.

Later, you can decide if the Unapproved plans perform well and change them to Approved, Rejected, or Preferred. To do so, you use the apg_plan_mgmt.evolve_plan_baselines function or the apg_plan_mgmt.set_plan_status function.

When the optimizer generates a plan for a SQL statement, query plan management saves the plan in the apg_plan_mgmt.plans table. Database users that have been granted the apg_plan_mgmt role can see the plan details by querying the apg_plan_mgmt.dba_plans view. For example, the following query lists details for plans currently in the view for a non-production Aurora PostgreSQL DB cluster.

- sql_hash – An identifier for the SQL statement that's the hash value for the normalized text of the SQL statement.
- plan_hash – A unique identifier for the plan that's a combination of the sql_hash and a hash of the plan.
- status – The status of the plan. The optimizer can run an approved plan.
- enabled – Indicates whether the plan is ready to use (true) or not (false).
- plan_outline – A representation of the plan that's used to recreate the actual execution plan. Operators in the tree structure map to operators in EXPLAIN output.

The apg_plan_mgmt.dba_plans view has many more columns that contain all details of the plan, such as when the plan was last used. For complete details, see Reference for the apg_plan_mgmt.dba_plans view.

Normalization and the SQL hash

In the apg_plan_mgmt.dba_plans view, you can identify a managed statement by its SQL hash value. The SQL hash is calculated on a normalized representation of the SQL statement that removes some differences, such as literal values.

The *normalization* process for each SQL statement preserves space and case, so that you can still read and understand the gist of the SQL statement. Normalization removes or replaces the following items.

- Leading block comments
- The EXPLAIN keyword and EXPLAIN options, and EXPLAIN ANALYZE
- Trailing spaces
- All literals

As an example, take the following statement.

```
/*Leading comment*/ EXPLAIN SELECT /* Query 1 */ * FROM t WHERE x > 7 AND y = 1;
```

The optimizer normalizes this statement as shown following.

```
SELECT /* Query 1 */ * FROM t WHERE x > CONST AND y = CONST;
```

Normalization allows the same SQL hash to be used for similar SQL statements that might differ only in their literal or parameter values. In other words, multiple plans for the same SQL hash can exist, with a different plan that's optimal under different conditions.

> **Note**
>
> A single SQL statement that's used with different schemas has different plans because it's bound to the specific schema at runtime. The planner uses the statistics for schema binding to choose the optimal plan.

To learn more about how the optimizer chooses a plan, see Using Aurora PostgreSQL managed plans. In that section, you can learn how to use EXPLAIN and EXPLAIN ANALYZE to preview a plan before it's actually used. For details, see Analyzing the optimizer's chosen plan. For an image that outlines the process for choosing a plan, see How the optimizer chooses which plan to run.

Capturing Aurora PostgreSQL execution plans

Aurora PostgreSQL query plan management offers two different modes for capturing query execution plans, automatic or manual. You choose the mode by setting the value of the `apg_plan_mgmt.capture_plans_baselines` to `automatic` or to `manual`. You can capture execution plans for specific SQL statements by using manual plan capture. Alternatively, you can capture all (or the slowest) plans that are executed two or more times as your application runs by using automatic plan capture.

When capturing plans, the optimizer sets the status of a managed statement's first captured plan to `approved`. The optimizer sets the status of any additional plans captured for a managed statement to `unapproved`. However, more than one plan might occasionally be saved with the `approved` status. This can happen when multiple plans are created for a statement in parallel and before the first plan for the statement is committed.

To control the maximum number of plans that can be captured and stored in the `dba_plans` view, set the `apg_plan_mgmt.max_plans` parameter in your DB instance-level parameter group. A change to the `apg_plan_mgmt.max_plans` parameter requires a DB instance reboot for a new value to take effect. For more information, see the apg_plan_mgmt.max_plans parameter.

Manually capturing plans for specific SQL statements

If you have a known set of SQL statements to manage, put the statements into a SQL script file and then manually capture plans. The following shows a psql example of how to capture query plans manually for a set of SQL statements.

```
psql> SET apg_plan_mgmt.capture_plan_baselines = manual;
psql> \i my-statements.sql
psql> SET apg_plan_mgmt.capture_plan_baselines = off;
```

After capturing a plan for each SQL statement, the optimizer adds a new row to the `apg_plan_mgmt.dba_plans` view.

We recommend that you use either EXPLAIN or EXPLAIN EXECUTE statements in the SQL script file. Make sure that you include enough variations in parameter values to capture all the plans of interest.

If you know of a better plan than the optimizer's minimum cost plan, you might be able to force the optimizer to use the better plan. To do so, specify one or more optimizer hints. For more information, see Fixing plans using pg_hint_plan. To compare the performance of the `unapproved` and `approved` plans and approve, reject, or delete them, see Evaluating plan performance.

Automatically capturing plans

Use automatic plan capture for situations such as the following:

- You don't know the specific SQL statements that you want to manage.
- You have hundreds or thousands of SQL statements to manage.
- Your application uses a client API. For example, JDBC uses unnamed prepared statements or bulk-mode statements that can't be expressed in psql.

To capture plans automatically

1. Turn on automatic plan capture by setting `apg_plan_mgmt.capture_plan_baselines` to `automatic` in the DB instance-level parameter group. For more information, see Modifying parameters in a DB parameter group.
2. Reboot your DB instance.
3. As the application runs, the optimizer captures plans for each SQL statement that runs at least twice.

   As the application runs with default query plan management parameter settings, the optimizer captures plans for each SQL statement that runs at least twice. Capturing all plans while using the defaults has very little run-time overhead and can be enabled in production.

To turn off automatic plan capture

- Set the `apg_plan_mgmt.capture_plan_baselines` parameter to `off` from the DB instance-level parameter group.

To measure the performance of the unapproved plans and approve, reject, or delete them, see Evaluating plan performance.

Using Aurora PostgreSQL managed plans

To get the optimizer to use captured plans for your managed statements, set the parameter `apg_plan_mgmt.use_plan_baselines` to `true`. The following is a local instance example.

```
SET apg_plan_mgmt.use_plan_baselines = true;
```

While the application runs, this setting causes the optimizer to use the minimum-cost, preferred, or approved plan that is valid and enabled for each managed statement.

Analyzing the optimizer's chosen plan

When the `apg_plan_mgmt.use_plan_baselines` parameter is set to `true`, you can use EXPLAIN ANALYZE SQL statements to cause the optimizer to show the plan it would use if it were to run the statement. The following is an example.

```
EXPLAIN ANALYZE EXECUTE rangeQuery (1,10000);
```

```

                                                    QUERY PLAN
--------------------------------------------------------------------------
 Aggregate  (cost=393.29..393.30 rows=1 width=8) (actual time=7.251..7.251 rows=1
 loops=1)
   ->  Index Only Scan using t1_pkey on t1 t  (cost=0.29..368.29 rows=10000 width=0)
 (actual time=0.061..4.859 rows=10000 loops=1)
Index Cond: ((id >= 1) AND (id <= 10000))
         Heap Fetches: 10000
 Planning time: 1.408 ms
 Execution time: 7.291 ms
 Note: An Approved plan was used instead of the minimum cost plan.
 SQL Hash: 1984047223, Plan Hash: 512153379
```

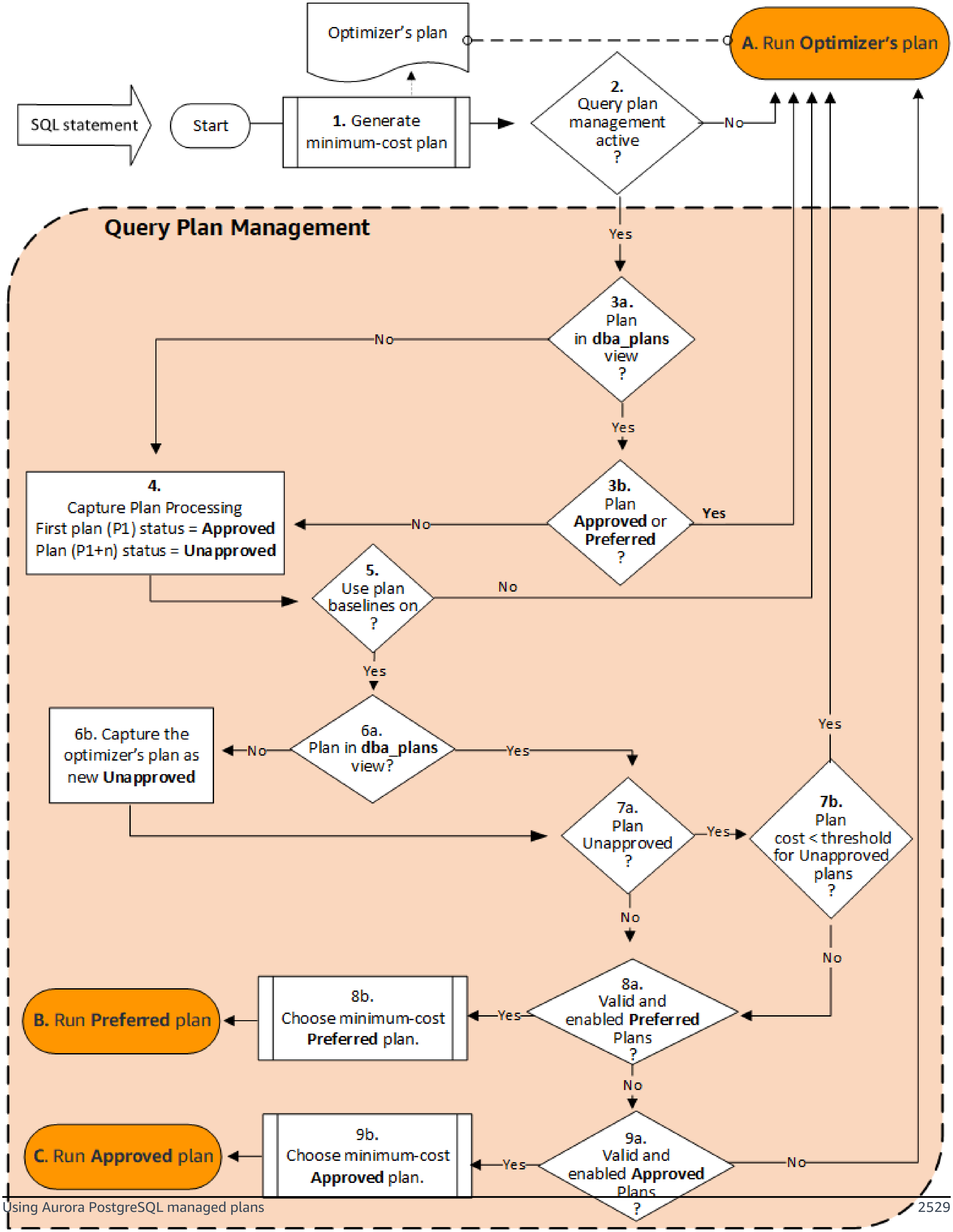
The output shows the Approved plan from the baseline that would run. However, the output also shows that it found a lower-cost plan. In this case, you capture this new minimum cost plan by turning on automatic plan capture as described in Automatically capturing plans.

New plans are always captured by the optimizer as `Unapproved`. Use the `apg_plan_mgmt.evolve_plan_baselines` function to compare plans and change them to approved, rejected, or disabled. For more information, see Evaluating plan performance.

How the optimizer chooses which plan to run

The cost of an execution plan is an estimate that the optimizer makes to compare different plans. When calculating a plan's cost, the optimizer includes factors such as CPU and I/O operations required by that plan. To learn more about PostgreSQL query planner cost estimates, see Query Planning in the PostgreSQL documentation.

The following image shows how a plan is chosen for a given SQL statement when query plan management is active, and when it's not.


Optimizer's plan
A. Run Optimizer's plan
SQL statement
Start
1. Generate minimum-cost plan
2. Query plan management active ?
No
Yes
Query Plan Management
3a. Plan in dba_plans view ?
No
Yes
3b. Plan Approved or Preferred ?
Yes
No
4. Capture Plan Processing
First plan (P1) status = Approved
Plan (P1+n) status = Unapproved
5. Use plan baselines on ?
No
Yes
6b. Capture the optimizer's plan as new Unapproved
No
6a. Plan in dba_plans view?
Yes
Yes
7a. Plan Unapproved ?
Yes
7b. Plan cost < threshold for Unapproved plans ?
No
No
B. Run Preferred plan
8b. Choose minimum-cost Preferred plan.
Yes
8a. Valid and enabled Preferred Plans ?
No
C. Run Approved plan
9b. Choose minimum-cost Approved plan.
Yes
9a. Valid and enabled Approved Plans ?
No

The flow is as follows:

1. The optimizer generates a minimum-cost plan for the SQL statement.
2. If query plan management isn't active, the optimizer's plan is run immediately (A. Run Optimizer's plan). Query plan management is inactive when the `apg_plan_mgmt.capture_plan_baselines` and the `apg_plan_mgmt.use_plan_baselines` parameters are both at their default settings ("off" and "false," respectively).

   Otherwise, query plan management is active. In this case, the SQL statement and the optimizer's plan for it are further assessed before a plan is chosen.

   > **Tip**
   >
   > Database users with the `apg_plan_mgmt` role can pro-actively compare plans, change status of plans, and force the use of specific plans as needed. For more information, see Maintaining Aurora PostgreSQL execution plans.

3. The SQL statement might already have plans that were stored by query plan management in the past. Plans are stored in the `apg_plan_mgmt.dba_plans`, along with information about the SQL statements that were used to create them. Information about a plan includes its status. A plan's status can determine whether it's used or not, as follows.
   a. If the plan isn't among the stored plans for the SQL statement, it means that it's the first time this particular plan was generated by the optimizer for the given SQL statement. The plan is sent to Capture Plan Processing (4).
   b. If the plan is among the stored plans and its status is Approved or Preferred, the plan is run (A. Run Optimizer's plan).

      If the plan is among the stored plans but it's neither Approved nor Preferred, the plan is sent to Capture Plan Processing (4).
4. When a plan is captured for the first time for a given SQL statement, the plan's status is always set to Approved (P1). If the optimizer subsequently generates the same plan for the same SQL statement, the status of that plan is changed to Unapproved (P1+n).

   With the plan captured and its status updated, the evaluation continues at the next step (5).

5. A plan's *baseline* consists of the history of the SQL statement and its plans at various states. Query plan management can take the baseline into account when choosing a plan, depending on whether the use plan baselines option is turned on or not, as follows.

   - Use plan baselines is "off" when the `apg_plan_mgmt.use_plan_baselines` parameter is set to its default value (`false`). The plan isn't compared to the baseline before it's run (A. Run Optimizer's plan).
   - Use plan baselines is "on" when the `apg_plan_mgmt.use_plan_baselines` parameter is set to `true`. The plan is further assessed using the baseline (6).

6. The plan is compared to other plans for the statement in the baseline.

   a. If the optimizer's plan is among the plans in the baseline, its status is checked (7a).

   b. If the optimizer's plan isn't among plans in the baseline, the plan is added to the plans for the statement as a new `Unapproved` plan.

7. The plan's status is checked to determine only if it's Unapproved.

   a. If the plan's status is Unapproved, the plan's estimated cost is compared to the cost estimate specified for the unapproved execution plan threshold.

      - If the plan's estimated cost is below the threshold, the optimizer uses it even though it's an Unapproved plan (A. Run Optimizer's plan). Generally, the optimizer won't run an Unapproved plan. However, when the `apg_plan_mgmt.unapproved_plan_execution_threshold` parameter specifies a cost threshold value, the optimizer compares the Unapproved plan's cost to the threshold. If the estimated cost is less than the threshold, the optimizer runs the plan. For more information, see apg_plan_mgmt.unapproved_plan_execution_threshold.
      - If the plan's estimated cost isn't below the threshold, the plan's other attributes are checked (8a).

   b. If the plan's status is anything other than Unapproved, its other attributes are checked (8a).

8. The optimizer won't use a plan that's disabled. That is, the plan that has its `enable` attribute set to 'f' (false). The optimizer also won't use a plan that has a status of Rejected.

   The optimizer can't use any plans that aren't valid. Plans can become invalid over time when the objects that they depend on, such as indexes and table partitions, are removed or deleted.

   a. If the statement has any enabled and valid Preferred plans, the optimizer chooses the minimum-cost plan from among the Preferred plans stored for this SQL statement. The optimizer then runs the minimum-cost Preferred plan.

   b. If the statement doesn't have any enabled and valid Preferred plans, it's assessed in the next step (9).

9. If the statement has any enabled and valid Approved plans, the optimizer chooses the minimum-cost plan from among the Approved plans stored for this SQL statement. The optimizer then runs the minimum-cost Approved plan.

   If the statement doesn't have any valid and enabled Approved plans, the optimizer uses the minimum cost plan (A. Run Optimizer's plan).

Examining Aurora PostgreSQL query plans in the dba_plans view

Database users and administrators that have been granted the `apg_plan_mgmt` role can view and manage the plans stored in the `apg_plan_mgmt.dba_plans`. An Aurora PostgreSQL DB cluster's administrator (someone with `rds_superuser` permissions) must explicitly grant this role to the database users who need to work with query plan management.

The `apg_plan_mgmt` view contains the plan history for all managed SQL statements for every database on the writer instance of the Aurora PostgreSQL DB cluster. This view lets you examine plans, their state, when last used, and all other relevant details.

As discussed in Normalization and the SQL hash, each managed plan is identified by the combined SQL hash value and a plan hash value. With these identifiers, you can use tools such as Amazon RDS Performance Insights to track individual plan performance. For more information about Performance Insights, see Using Amazon RDS performance insights.

Listing managed plans

To list the managed plans, use a SELECT statement on the `apg_plan_mgmt.dba_plans` view. The following example displays some columns in the `dba_plans` view such as the `status`, which identifies the approved and unapproved plans.

```
SELECT sql_hash, plan_hash, status, enabled, stmt_name
FROM apg_plan_mgmt.dba_plans;

 sql_hash   | plan_hash |   status   | enabled | stmt_name
------------+-----------+------------+---------+------------
 1984047223 | 512153379 | Approved   | t       | rangequery
 1984047223 | 512284451 | Unapproved | t       | rangequery
 (2 rows)
```

For readability, the query and the output shown list just a few of the columns from the dba_plans view. For complete information, see Reference for the apg_plan_mgmt.dba_plans view.

Maintaining Aurora PostgreSQL execution plans

Query plan management provides techniques and functions to add, maintain, and improve execution plans.

Evaluating plan performance

After the optimizer captures plans as unapproved, use the apg_plan_mgmt.evolve_plan_baselines function to compare plans based on their actual performance. Depending on the outcome of your performance experiments, you can change a plan's status from unapproved to either approved or rejected. You can instead decide to use the apg_plan_mgmt.evolve_plan_baselines function to temporarily disable a plan if it does not meet your requirements.

Approving better plans

The following example demonstrates how to change the status of managed plans to approved using the apg_plan_mgmt.evolve_plan_baselines function.

```
SELECT apg_plan_mgmt.evolve_plan_baselines (
   sql_hash, 
   plan_hash, 
   min_speedup_factor := 1.0, 
   action := 'approve'
) 
FROM apg_plan_mgmt.dba_plans WHERE status = 'Unapproved';
```

```
NOTICE:     rangequery (1,10000)
NOTICE:     Baseline   [ Planning time 0.761 ms, Execution time 13.261 ms]
NOTICE:     Baseline+1 [ Planning time 0.204 ms, Execution time 8.956 ms]
NOTICE:     Total time benefit: 4.862 ms, Execution time benefit: 4.305 ms
NOTICE:     Unapproved -> Approved
evolve_plan_baselines 
-----------------------
0
(1 row)
```

The output shows a performance report for the rangequery statement with parameter bindings of 1 and 10,000. The new unapproved plan (Baseline+1) is better than the best previously approved plan (Baseline). To confirm that the new plan is now Approved, check the apg_plan_mgmt.dba_plans view.

```
SELECT sql_hash, plan_hash, status, enabled, stmt_name
FROM apg_plan_mgmt.dba_plans;
```

```
sql_hash  | plan_hash |  status  | enabled | stmt_name
------------+-----------+----------+---------+------------
1984047223 | 512153379 | Approved | t       | rangequery
1984047223 | 512284451 | Approved | t       | rangequery
(2 rows)
```

The managed plan now includes two approved plans that are the statement's plan baseline. You can also call the apg_plan_mgmt.set_plan_status function to directly set a plan's status field to 'Approved', 'Rejected', 'Unapproved', or 'Preferred'.

Rejecting or disabling slower plans

To reject or disable plans, pass 'reject' or 'disable' as the action parameter to the apg_plan_mgmt.evolve_plan_baselines function. This example disables any captured Unapproved plan that is slower by at least 10 percent than the best Approved plan for the statement.

```
SELECT apg_plan_mgmt.evolve_plan_baselines(
sql_hash,  -- The managed statement ID
plan_hash, -- The plan ID
1.1,       -- number of times faster the plan must be
'disable'  -- The action to take. This sets the enabled field to false.
)
FROM apg_plan_mgmt.dba_plans
WHERE status = 'Unapproved' AND   -- plan is Unapproved
origin = 'Automatic';       -- plan was auto-captured
```

You can also directly set a plan to rejected or disabled. To directly set a plan's enabled field to true or false, call the apg_plan_mgmt.set_plan_enabled function. To directly set a plan's status field to 'Approved', 'Rejected', 'Unapproved', or 'Preferred', call the apg_plan_mgmt.set_plan_status function.

Validating plans

Use the apg_plan_mgmt.validate_plans function to delete or disable plans that are invalid.

Plans can become invalid or stale when objects that they depend on are removed, such as an index or a table. However, a plan might be invalid only temporarily if the removed object gets recreated. If an invalid plan can become valid later, you might prefer to disable an invalid plan or do nothing rather than delete it.

To find and delete all plans that are invalid and haven't been used in the past week, use the apg_plan_mgmt.validate_plans function as follows.

```
SELECT apg_plan_mgmt.validate_plans(sql_hash, plan_hash, 'delete') 
FROM apg_plan_mgmt.dba_plans
WHERE last_used < (current_date - interval '7 days');
```

To enable or disabled a plan directly, use the apg_plan_mgmt.set_plan_enabled function.

Fixing plans using pg_hint_plan

The query optimizer is well-designed to find an optimal plan for all statements, and in most cases the optimizer finds a good plan. However, occasionally you might know that a much better plan exists than that generated by the optimizer. Two recommended ways to get the optimizer to generate a desired plan include using the pg_hint_plan extension or setting Grand Unified Configuration (GUC) variables in PostgreSQL:

- pg_hint_plan extension – Specify a "hint" to modify how the planner works by using PostgreSQL's pg_hint_plan extension. To install and learn more about how to use the pg_hint_plan extension, see the pg_hint_plan documentation.
- GUC variables – Override one or more cost model parameters or other optimizer parameters, such as the from_collapse_limit or GEQO_threshold.

When you use one of these techniques to force the query optimizer to use a plan, you can also use query plan management to capture and enforce use of the new plan.

You can use the pg_hint_plan extension to change the join order, the join methods, or the access paths for a SQL statement. You use a SQL comment with special pg_hint_plan syntax to modify how the optimizer creates a plan. For example, assume the problem SQL statement has a two-way join.

```
SELECT *
FROM t1, t2
WHERE t1.id = t2.id;
```

Then suppose that the optimizer chooses the join order (t1, t2), but you know that the join order (t2, t1) is faster. The following hint forces the optimizer to use the faster join order, (t2, t1). Include EXPLAIN so that the optimizer generates a plan for the SQL statement but without running the statement. (Output not shown.)

```
/*+ Leading ((t2 t1)) */ EXPLAIN SELECT *
FROM t1, t2
WHERE t1.id = t2.id;
```

The following steps show how to use `pg_hint_plan`.

To modify the optimizer's generated plan and capture the plan using pg_hint_plan

1. Turn on the manual capture mode.

   ```
   SET apg_plan_mgmt.capture_plan_baselines = manual;
   ```

2. Specify a hint for the SQL statement of interest.

   ```
   /*+ Leading ((t2 t1)) */ EXPLAIN SELECT *
   FROM t1, t2
   WHERE t1.id = t2.id;
   ```

   After this runs, the optimizer captures the plan in the `apg_plan_mgmt.dba_plans` view. The captured plan doesn't include the special `pg_hint_plan` comment syntax because query plan management normalizes the statement by removing leading comments.

3. View the managed plans by using the `apg_plan_mgmt.dba_plans` view.

   ```
   SELECT sql_hash, plan_hash, status, sql_text, plan_outline
   FROM apg_plan_mgmt.dba_plans;
   ```

4. Set the status of the plan to `Preferred`. Doing so makes sure that the optimizer chooses to run it, instead of selecting from the set of approved plans, when the minimum-cost plan isn't already `Approved` or `Preferred`.

```
SELECT apg_plan_mgmt.set_plan_status(sql-hash, plan-hash, 'preferred' );
```

5. Turn off manual plan capture and enforce the use of managed plans.

   ```
   SET apg_plan_mgmt.capture_plan_baselines = false;
   SET apg_plan_mgmt.use_plan_baselines = true;
   ```

   Now, when the original SQL statement runs, the optimizer chooses either an `Approved` or `Preferred` plan. If the minimum-cost plan isn't `Approved` or `Preferred`, then the optimizer chooses the `Preferred` plan.

Deleting plans

Plans are automatically deleted if they haven't been used in over a month, specifically, 32 days. That's the default setting for the `apg_plan_mgmt.plan_retention_period` parameter. You can change the plan retention period to a longer period of time, or to a shorter period of time starting from the value of 1. Determining the number of days since a plan was last used is calculated by subtracting the `last_used` date from the current date. The `last_used` date is the most recent date that the optimizer chose the plan as the minimum cost plan or that the plan was run. The date is stored for the plan in the `apg_plan_mgmt.dba_plans` view.

We recommend that you delete plans that haven't been used for a long time or that aren't useful. Every plan has a `last_used` date that the optimizer updates each time it executes a plan or chooses the plan as the minimum-cost plan for a statement. Check the last `last_used` dates to identify the plans that you can safely delete.

The following query returns a three column table with the count on the total number of plans, plans failed to delete, and the plans successfully deleted. It has a nested query that is an example of how to use the `apg_plan_mgmt.delete_plan` function to delete all plans that haven't been chosen as the minimum-cost plan in the last 31 days and its status is not `Rejected`.

```
SELECT (SELECT COUNT(*) from apg_plan_mgmt.dba_plans) total_plans,
       COUNT(*) FILTER (WHERE result = -1) failed_to_delete,
       COUNT(*) FILTER (WHERE result = 0) successfully_deleted
       FROM (
            SELECT apg_plan_mgmt.delete_plan(sql_hash, plan_hash) as result
            FROM apg_plan_mgmt.dba_plans
            WHERE last_used < (current_date - interval '31 days')
```

```
             AND status <> 'Rejected'
             ) as dba_plans ;
```

```
 total_plans | failed_to_delete | successfully_deleted
-------------+------------------+----------------------
           3 |                0 |                    2
```

For more information, see apg_plan_mgmt.delete_plan.

To delete plans that aren't valid and that you expect to remain invalid, use the `apg_plan_mgmt.validate_plans` function. This function lets you delete or disable invalid plans. For more information, see Validating plans.

Important

If you don't delete extraneous plans, you might eventually run out of shared memory that's set aside for query plan management. To control how much memory is available for managed plans, use the `apg_plan_mgmt.max_plans` parameter. Set this parameter in your custom DB parameter group and reboot your DB instance for changes to take effect. For more information, see the apg_plan_mgmt.max_plans parameter.

Exporting and importing plans

You can export your managed plans and import them into another DB instance.

To export managed plans

An authorized user can copy any subset of the `apg_plan_mgmt.plans` table to another table, and then save it using the pg_dump command. The following is an example.

```
CREATE TABLE plans_copy AS SELECT *
FROM apg_plan_mgmt.plans [ WHERE predicates ] ;
```

```
% pg_dump --table apg_plan_mgmt.plans_copy -Ft mysourcedatabase > plans_copy.tar
```

```
DROP TABLE apg_plan_mgmt.plans_copy;
```

To import managed plans

1. Copy the .tar file of the exported managed plans to the system where the plans are to be restored.
2. Use the `pg_restore` command to copy the tar file into a new table.

```
% pg_restore --dbname mytargetdatabase -Ft plans_copy.tar
```

3. Merge the `plans_copy` table with the `apg_plan_mgmt.plans` table, as shown in the following example.

Note

In some cases, you might dump from one version of the `apg_plan_mgmt` extension and restore into a different version. In these cases, the columns in the plans table might be different. If so, name the columns explicitly instead of using SELECT *.

```
INSERT INTO apg_plan_mgmt.plans SELECT * FROM plans_copy
 ON CONFLICT ON CONSTRAINT plans_pkey
 DO UPDATE SET
 status = EXCLUDED.status,
 enabled = EXCLUDED.enabled,
 -- Save the most recent last_used date
 --
 last_used = CASE WHEN EXCLUDED.last_used > plans.last_used
 THEN EXCLUDED.last_used ELSE plans.last_used END,
 -- Save statistics gathered by evolve_plan_baselines, if it ran:
 --
 estimated_startup_cost = EXCLUDED.estimated_startup_cost,
 estimated_total_cost = EXCLUDED.estimated_total_cost,
 planning_time_ms = EXCLUDED.planning_time_ms,
 execution_time_ms = EXCLUDED.execution_time_ms,
 total_time_benefit_ms = EXCLUDED.total_time_benefit_ms,
 execution_time_benefit_ms = EXCLUDED.execution_time_benefit_ms;
```

4. Reload the managed plans into shared memory and remove the temporary plans table.

```
SELECT apg_plan_mgmt.reload(); -- refresh shared memory
DROP TABLE plans_copy;
```

Reference for Aurora PostgreSQL query plan management

Following, you can find reference information for several Aurora PostgreSQL query plan management features and functionality.

Topics

Parameter reference for Aurora PostgreSQL query plan management

You can set your preferences for the `apg_plan_mgmt` extension by using the parameters listed in this section. These are available in the custom DB cluster parameter and the DB parameter group associated with your Aurora PostgreSQL DB cluster. These parameters control the behavior of the query plan management feature and how it affects the optimizer. For information about setting up query plan management, see Turning on Aurora PostgreSQL query plan management. Changing the parameters following has no effect if the `apg_plan_mgmt` extension isn't set up as detailed in that section. For information about modifying parameters, see Modifying parameters in a DB cluster parameter group and Working with DB parameter groups in a DB instance.

Parameters

apg_plan_mgmt.capture_plan_baselines

Captures query execution plans generated by the optimizer for each SQL statement and stores them in the `dba_plans` view. By default, the maximum number of plans that can be stored is 10,000 as specified by the `apg_plan_mgmt.max_plans` parameter. For reference information, see apg_plan_mgmt.max_plans.

You can set this parameter in the custom DB cluster parameter group or in the custom DB parameter group. Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
off	automatic	Turns on plan capture for all databases on the DB instance. Collects a plan for each SQL statement that runs two or more times. Use this setting for large or evolving workloads to provide plan stability.
	manual	Turns on plan capture for subsequent statements only, until you turn it off again. Using this setting lets you capture query execution plans for specific critical SQL statements only or for known problematic queries.
	off	Turns off plan capture.

For more information, see Capturing Aurora PostgreSQL execution plans.

apg_plan_mgmt.plan_capture_threshold

Specifies a threshold so that if the total cost of the query execution plan is below the threshold, the plan won't be captured in the `apg_plan_mgmt.dba_plans` view.

Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
0	0 - 1.79769e+308	Sets the threshold of the `apg_plan_mgmt` query plan total execution cost for capturing plans.

For more information, see Examining Aurora PostgreSQL query plans in the dba_plans view.

apg_plan_mgmt.explain_hashes

Specifies if the `EXPLAIN [ANALYZE]` shows `sql_hash` and `plan_hash` at the end of its output. Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
0	0 (off)	EXPLAIN does not show sql_hash and plan_hash without hashes true option.
	1 (on)	EXPLAIN shows sql_hash and plan_hash without hashes true option.

apg_plan_mgmt.log_plan_enforcement_result

Specifies if the results has to be recorded to see if the QPM managed plans are used properly. When a stored generic plan is used, there will be no records written in the log files. Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
none	none	Does not show any plan enforcement result in log files.
	on_error	Only shows plan enforcement result in log files when QPM fails to use managed plans.
	all	Shows all plan enforcement results in log files including both successes and failures.

apg_plan_mgmt.max_databases

Specifies the maximum number of databases on your Aurora PostgreSQL DB cluster's Writer instance that can use query plan management. By default, up to 10 databases can use query plan management. If you have more than 10 databases on the instance, you can change the value of this setting. To find out how many databases are on a given instance, connect to the instance using `psql`. Then, use the psql metacommand, `\l`, to list the databases.

Changing the value of this parameter requires that you reboot the instance for the setting to take effect.

Default	Allowed values	Description
10	10-2147483647	Maximum number of databases that can use query plan management on the instance.

You can set this parameter in the custom DB cluster parameter group or in the custom DB parameter group.

apg_plan_mgmt.max_plans

Sets the maximum number of SQL statements that the query plan manager can maintain in the `apg_plan_mgmt.dba_plans` view. We recommend setting this parameter to `10000` or higher for all Aurora PostgreSQL versions.

You can set this parameter in the custom DB cluster parameter group or in the custom DB parameter group. Changing the value of this parameter requires that you reboot the instance for the setting to take effect.

Default	Allowed values	Description
10000	10-2147483647	Maximum number of plans that can be stored in the `apg_plan_mgmt.dba_plans` view. Default for Aurora PostgreSQL version 10 and older versions is 1000.

For more information, see Examining Aurora PostgreSQL query plans in the dba_plans view.

apg_plan_mgmt.plan_hash_version

Specifies the use cases that the plan_hash calculation is designed to cover. A higher version of `apg_plan_mgmt.plan_hash_version` covers all the functionality of the lower version. For example, version 3 covers the use cases supported by version 2.

Changing the value of this parameter must be followed by a call to `apg_plan_mgmt.validate_plans('update_plan_hash')`. It updates the plan_hash values in each database with apg_plan_mgmt installed and entries in the plans table. For more information, see Validating plans

Default	Allowed values	Description
1	1	Default plan_hash calculation.
	2	plan_hash calculation modified for multi-schema support.
	3	plan_hash calculation modified for multi-schema support and partitioned table support.
	4	plan_hash calculation modified for parallel operators and to support materialize nodes.

apg_plan_mgmt.plan_retention_period

Specifies the number of days to keep plans in the `apg_plan_mgmt.dba_plans` view, after which they're automatically deleted. By default, a plan is deleted when 32 days have elapsed since the plan was last used (the `last_used` column in the `apg_plan_mgmt.dba_plans` view). You can change this setting to any number, 1 and over.

Changing the value of this parameter requires that you reboot the instance for the setting to take effect.

Default	Allowed values	Description
32	1-2147483647	Maximum number of days since a plan was last used before it's deleted.

For more information, see Examining Aurora PostgreSQL query plans in the dba_plans view.

apg_plan_mgmt.unapproved_plan_execution_threshold

Specifies a cost threshold below which an Unapproved plan can be used by the optimizer. By default the threshold is 0, so the optimizer doesn't run Unapproved plans. Setting this parameter to a trivially low cost threshold such as 100 avoids plan enforcement overhead on trivial plans.

You can also set this parameter to an extremely large value like 10000000 using the reactive style of plan management. This allows the optimizer to use all chosen plans with no plan enforcement overhead. But, when a bad plan is found, you can manually mark it as "rejected" so that it is not used next time.

The value of this parameter represents a cost estimate for running a given plan. If an Unapproved plan is below that estimated cost, the optimizer uses it for the SQL statement. You can see captured plans and their status (Approved, Unapproved) in the `dba_plans` view. To learn more, see Examining Aurora PostgreSQL query plans in the dba_plans view.

Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
0	0-2147483647	Estimated plan cost below which an Unapproved plan is used.

For more information, see Using Aurora PostgreSQL managed plans.

apg_plan_mgmt.use_plan_baselines

Specifies that the optimizer should use one of the Approved plans captured and stored in the `apg_plan_mgmt.dba_plans` view. By default, this parameter is off (false), causing the optimizer to use the minimum-cost plan that it generates without any further assessment. Turning this parameter on (setting it to true) forces the optimizer to choose a query execution plan for the statement from its plan baseline. For more information, see Using Aurora PostgreSQL managed plans. To find an image detailing this process, see How the optimizer chooses which plan to run.

You can set this parameter in the custom DB cluster parameter group or in the custom DB parameter group. Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
false	true	Use an Approved, Preferred, or Unapproved plan from the `apg_plan_mgmt.dba_plans` . If none of those meet all evaluation criterion for the optimizer, it can then use its own generated minimum-cost plan. For more information, see How the optimizer chooses which plan to run.

Default	Allowed values	Description
	false	Use the minimum cost plan generated by the optimizer.

You can evaluate response times of different captured plans and change plan status, as needed. For more information, see Maintaining Aurora PostgreSQL execution plans.

auto_explain.hashes

Specifies if the auto_explain output shows sql_hash and plan_hash. Changing the value of this parameter doesn't require a reboot.

Default	Allowed values	Description
0(off)	0(off)	`auto_explain` result does not show `sql_hash` and `plan_hash` .
	1(on)	`auto_explain` result shows `sql_hash` and `plan_hash` .

Function reference for Aurora PostgreSQL query plan management

The `apg_plan_mgmt` extension provides the following functions.

Functions

- apg_plan_mgmt.copy_outline
- apg_plan_mgmt.delete_plan
- apg_plan_mgmt.evolve_plan_baselines
- apg_plan_mgmt.get_explain_plan
- apg_plan_mgmt.plan_last_used
- apg_plan_mgmt.reload
- apg_plan_mgmt.set_plan_enabled
- apg_plan_mgmt.set_plan_status
- apg_plan_mgmt.update_plans_last_used
- apg_plan_mgmt.validate_plans

apg_plan_mgmt.copy_outline

Copy a given SQL plan hash and plan outline to a target SQL plan hash and outline, thereby overwriting the target's plan hash and outline. This function is available in `apg_plan_mgmt` 2.3 and higher releases.

Syntax

```
apg_plan_mgmt.copy_outline(
    source_sql_hash,
    source_plan_hash,
    target_sql_hash,
    target_plan_hash,
    force_update_target_plan_hash
)
```

Return value

Returns 0 when the copy is successful. Raises exceptions for invalid inputs.

Parameters

Parameter	Description
`source_sql_hash`	The `sql_hash` ID associated with the `plan_hash` to copy to the target query.
`source_plan_hash`	The `plan_hash` ID to copy to the target query.
`target_sql_hash`	The `sql_hash` ID of the query to update with the source plan hash and outline.
`target_plan_hash`	The `plan_hash` ID of the query to update with the source plan hash and outline.
`force_update_target_plan_hash`	(Optional) The `target_plan_hash` ID of the query is updated even if the source plan isn't reproducible for the `target_sql_hash` . When set to true, the function can

Parameter	Description
	be used to copy plans across schemas where relation names and columns are consistent.

Usage notes

This function allows you to copy a plan hash and plan outline that uses hints to other, similar statements, and thus saves you from needing to use in-line hint statements at every occurrence in the target statements. If the updated target query results in an invalid plan, this function raises an error and rolls back the attempted update.

apg_plan_mgmt.delete_plan

Delete a managed plan.

Syntax

```
apg_plan_mgmt.delete_plan(
    sql_hash,
    plan_hash
)
```

Return value

Returns 0 if the delete was successful or -1 if the delete failed.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID.

apg_plan_mgmt.evolve_plan_baselines

Verifies whether an already approved plan is faster or whether a plan identified by the query optimizer as a minimum cost plan is faster.

Syntax

```
apg_plan_mgmt.evolve_plan_baselines(
    sql_hash,
    plan_hash,
    min_speedup_factor,
    action
)
```

Return value

The number of plans that were not faster than the best approved plan.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID. Use NULL to mean all plans that have the same `sql_hash` ID value.
`min_speedup_factor`	The *minimum speedup factor* can be the number of times faster that a plan must be than the best of the already approved plans to approve it. Alternatively, this factor can be the number of times slower that a plan must be to reject or disable it. This is a positive float value.
`action`	The action the function is to perform. Valid values include the following . Case does not matter. • `'disable'` – Disable each matching plan that does not meet the minimum speedup factor. • `'approve'` – Enable each matching plan that meets the minimum speedup factor and set its status to `approved`.

Parameter	Description
	- `'reject'` – For each matching plan that does not meet the minimum speedup factor, set its status to `rejected`. - NULL – The function simply returns the number of plans that have no performance benefit because they do not meet the minimum speedup factor.

Usage notes

Set specified plans to approved, rejected, or disabled based on whether the planning plus execution time is faster than the best approved plan by a factor that you can set. The action parameter might be set to `'approve'` or `'reject'` to automatically approve or reject a plan that meets the performance criteria. Alternatively, it might be set to '' (empty string) to do the performance experiment and produce a report, but take no action.

You can avoid pointlessly rerunning of the `apg_plan_mgmt.evolve_plan_baselines` function for a plan on which it was recently run. To do so, restrict the plans to just the recently created unapproved plans. Alternatively, you can avoid running the `apg_plan_mgmt.evolve_plan_baselines` function on any approved plan that has a recent `last_verified` timestamp.

Conduct a performance experiment to compare the planning plus execution time of each plan relative to the other plans in the baseline. In some cases, there is only one plan for a statement and the plan is approved. In such a case, compare the planning plus execution time of the plan to the planning plus execution time of using no plan.

The incremental benefit (or disadvantage) of each plan is recorded in the `apg_plan_mgmt.dba_plans` view in the `total_time_benefit_ms` column. When this value is positive, there is a measurable performance advantage to including this plan in the baseline.

In addition to collecting the planning and execution time of each candidate plan, the `last_verified` column of the `apg_plan_mgmt.dba_plans` view is updated with the `current_timestamp`. The `last_verified` timestamp might be used to avoid running this function again on a plan that recently had its performance verified.

apg_plan_mgmt.get_explain_plan

Generates the text of an `EXPLAIN` statement for the specified SQL statement.

Syntax

```
apg_plan_mgmt.get_explain_plan(
    sql_hash,
    plan_hash,
    [explainOptionList]
)
```

Return value

Returns runtime statistics for the specified SQL statements. Use without `explainOptionList` to return a simple `EXPLAIN` plan.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID.
`explainOptionList`	A comma-separated list of explain options. Valid values include `'analyze'` , `'verbose'` , `'buffers'` , `'hashes'`, and `'format json'`. If the `explainOptionList` is NULL or an empty string (''), this function generates an `EXPLAIN` statement, without any statistics.

Usage notes

For the `explainOptionList`, you can use any of the same options that you would use with an `EXPLAIN` statement. The Aurora PostgreSQL optimizer concatenates the list of options that you provide to the `EXPLAIN` statement.

apg_plan_mgmt.plan_last_used

Returns the `last_used` date of the specified plan from shared memory.

Note

The value in shared memory is always current on the primary DB instance in the DB cluster. The value is only periodically flushed to the `last_used` column of the `apg_plan_mgmt.dba_plans` view.

Syntax

```
apg_plan_mgmt.plan_last_used(
    sql_hash,
    plan_hash
)
```

Return value

Returns the `last_used` date.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID.

apg_plan_mgmt.reload

Reload plans into shared memory from the `apg_plan_mgmt.dba_plans` view.

Syntax

```
apg_plan_mgmt.reload()
```

Return value

None.

Parameters

None.

Usage notes

Call `reload` for the following situations:

- Use it to refresh the shared memory of a read-only replica immediately, rather than wait for new plans to propagate to the replica.
- Use it after importing managed plans.

apg_plan_mgmt.set_plan_enabled

Enable or disable a managed plan.

Syntax

```
apg_plan_mgmt.set_plan_enabled(
    sql_hash,
    plan_hash,
    [true | false]
)
```

Return value

Returns 0 if the setting was successful or -1 if the setting failed.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID.
`enabled`	Boolean value of true or false: - A value of `true` enables the plan.

Parameter	Description
	- A value of `false` disables the plan.

apg_plan_mgmt.set_plan_status

Set a managed plan's status to `Approved`, `Unapproved`, `Rejected`, or `Preferred`.

Syntax

```
apg_plan_mgmt.set_plan_status(
    sql_hash,
    plan_hash,
    status
)
```

Return value

Returns 0 if the setting was successful or -1 if the setting failed.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID.
`status`	A string with one of the following values: - `'Approved'` - `'Unapproved'` - `'Rejected'` - `'Preferred'` The case you use does not matter, however the status value is set to initial uppercase in the `apg_plan_mgmt.dba_plans` view. For more information

Parameter	Description
	about these values, see `status` in [Reference for the apg_plan_mgmt.dba_plans view]().

apg_plan_mgmt.update_plans_last_used

Immediately updates the plans table with the `last_used` date stored in shared memory.

Syntax

```
apg_plan_mgmt.update_plans_last_used()
```

Return value

None.

Parameters

None.

Usage notes

Call `update_plans_last_used` to make sure queries against the `dba_plans.last_used` column use the most current information. If the `last_used` date isn't updated immediately, a background process updates the plans table with the `last_used` date once every hour (by default).

For example, if a statement with a certain `sql_hash` begins to run slowly, you can determine which plans for that statement were executed since the performance regression began. To do that, first flush the data in shared memory to disk so that the `last_used` dates are current, and then query for all plans of the `sql_hash` of the statement with the performance regression. In the query, make sure the `last_used` date is greater than or equal to the date on which the performance regression began. The query identifies the plan or set of plans that might be responsible for the performance regression. You can use `apg_plan_mgmt.get_explain_plan` with `explainOptionList` set to `verbose, hashes`. You can also use `apg_plan_mgmt.evolve_plan_baselines` to analyze the plan and any alternative plans that might perform better.

The `update_plans_last_used` function has an effect only on the primary DB instance of the DB cluster.

apg_plan_mgmt.validate_plans

Validate that the optimizer can still recreate plans. The optimizer validates `Approved`, `Unapproved`, and `Preferred` plans, whether the plan is enabled or disabled. `Rejected` plans are not validated. Optionally, you can use the `apg_plan_mgmt.validate_plans` function to delete or disable invalid plans.

Syntax

```
apg_plan_mgmt.validate_plans(
    sql_hash,
    plan_hash,
    action)

apg_plan_mgmt.validate_plans(
    action)
```

Return value

The number of invalid plans.

Parameters

Parameter	Description
`sql_hash`	The `sql_hash` ID of the plan's managed SQL statement.
`plan_hash`	The managed plan's `plan_hash` ID. Use NULL to mean all plans for the same `sql_hash` ID value.
`action`	The action the function is to perform for invalid plans. Valid string values include the following. Case does not matter. • `'disable'` – Each invalid plan is disabled. • `'delete'` – Each invalid plan is deleted. • `'update_plan_hash'` – Updates the `plan_hash` ID for plans that can't be reproduced exactly. It also allows you to fix a plan by rewriting the SQL. You can then register the good plan as an `Approved` plan for the original SQL.

Parameter	Description
	- NULL – The function simply returns the number of invalid plans. No other action is performed. - " – An empty string produces a message indicating the number of both valid and invalid plans. Any other value is treated like the empty string.

Usage notes

Use the form `validate_plans(action)` to validate all the managed plans for all the managed statements in the entire `apg_plan_mgmt.dba_plans` view.

Use the form `validate_plans(sql_hash, plan_hash, action)` to validate a managed plan specified with `plan_hash`, for a managed statement specified with `sql_hash`.

Use the form `validate_plans(sql_hash, NULL, action)` to validate all the managed plans for the managed statement specified with `sql_hash`.

Reference for the apg_plan_mgmt.dba_plans view

The columns of plan information in the `apg_plan_mgmt.dba_plans` view include the following.

dba_plans column	Description
`cardinality_error`	A measure of the error between the estimated cardinality versus the actual cardinality. *Cardinality* is the number of table rows that the plan is to process. If the cardinality error is large, then it increases the likelihood that the plan isn't optimal. This column is populated by the apg_plan_mgmt.evolve_plan_baselines function.
`compatibility_level`	The feature level of the Aurora PostgreSQL optimizer.
`created_by`	The authenticated user (`session_user`) who created the plan.

dba_plans column	Description
`enabled`	An indicator of whether the plan is enabled or disabled. All plans are enabled by default. You can disable plans to prevent them from being used by the optimizer. To modify this value, use the apg_plan_mgmt.set_plan_enabled function.
`environment_variables`	The PostgreSQL Grand Unified Configuration (GUC) parameters and values that the optimizer has overridden at the time the plan was captured.
`estimated_startup_cost`	The estimated optimizer setup cost before the optimizer delivers rows of a table.
`estimated_total_cost`	The estimated optimizer cost to deliver the final table row.
`execution_time_benefit_ms`	The execution time benefit in milliseconds of enabling the plan. This column is populated by the apg_plan_mgmt.evolve_plan_baselines function.
`execution_time_ms`	The estimated time in milliseconds that the plan would run. This column is populated by the apg_plan_mgmt.evolve_plan_baselines function.
`has_side_effects`	A value that indicates that the SQL statement is a data manipulation language (DML) statement or a SELECT statement that contains a VOLATILE function.
`last_used`	This value is updated to the current date whenever the plan is either executed or when the plan is the query optimizer's minimum-cost plan. This value is stored in shared memory and periodically flushed to disk. To get the most up-to-date value, read the date from shared memory by calling the function `apg_plan_mgmt.plan_last_used(sql_hash, plan_hash)` instead of reading the `last_used` value. For additional information, see the apg_plan_mgmt.plan_retention_period parameter.

dba_plans column	Description
last_validated	The most recent date and time when it was verified that the plan could be recreated by either the apg_plan_mgmt.validate_plans function or the apg_plan_mgmt.evolve_plan_baselines function.
last_verified	The most recent date and time when a plan was verified to be the best-performing plan for the specified parameters by the apg_plan_mgmt.evolve_plan_baselines function.
origin	How the plan was captured with the apg_plan_mgmt.capture_plan_baselines parameter. Valid values include the following: M – The plan was captured with manual plan capture. A – The plan was captured with automatic plan capture.
param_list	The parameter values that were passed to the statement if this is a prepared statement.
plan_created	The date and time the plan that was created.
plan_hash	The plan identifier. The combination of plan_hash and sql_hash uniquely identifies a specific plan.
plan_outline	A representation of the plan that is used to recreate the actual execution plan, and that is database-independent. Operators in the tree correspond to operators that appear in the EXPLAIN output.
planning_time_ms	The actual time to run the planner, in milliseconds. This column is populated by the apg_plan_mgmt.evolve_plan_baselines function.

dba_plans column	Description
`queryId`	A statement hash, as calculated by the `pg_stat_statements` extension. This isn't a stable or database-independent identifier because it depends on object identifiers (OIDs). The value will be `0` if `compute_query_id` is `off` when capturing the query plan.
`sql_hash`	A hash value of the SQL statement text, normalized with literals removed.
`sql_text`	The full text of the SQL statement.
`status`	A plan's status, which determines how the optimizer uses a plan. Valid values include the following. • `Approved` – A usable plan that the optimizer can choose to run. The optimizer runs the least-cost plan from a managed statement's set of approved plans (baseline). To reset a plan to approved, use the apg_plan_mgmt.evolve_plan_baselines function. • `Unapproved` – A captured plan that you have not verified for use. For more information, see Evaluating plan performance. • `Rejected` – A plan that the optimizer won't use. For more information, see Rejecting or disabling slower plans. • `Preferred` – A plan that you have determined is a preferred plan to use for a managed statement. If the optimizer's minimum-cost plan isn't an approved or preferred plan, you can reduce plan enforcement overhead. To do so, make a subset of the approved plans `Preferred` . When the optimizer's minimum cost isn't an `Approved` plan, a `Preferred` plan is chosen before an `Approved` plan. To reset a plan to `Preferred` , use the apg_plan_mgmt.set_plan_status function.

dba_plans column	Description
`stmt_name`	The name of the SQL statement within a PREPARE statement . This value is an empty string for an unnamed prepared statement. This value is NULL for a nonprepared statement.
`total_time_benefit_ms`	The total time benefit in milliseconds of enabling this plan. This value considers both planning time and execution time. If this value is negative, there is a disadvantage to enabling this plan. This column is populated by the apg_plan_mgmt.evolve_plan_baselines function.

Advanced features in Query Plan Management

Following you can find information about the advanced Aurora PostgreSQL Query Plan Management (QPM) features:

Topics

- Capturing Aurora PostgreSQL execution plans in Replicas
- Supporting table partition

Capturing Aurora PostgreSQL execution plans in Replicas

QPM (Query Plan Management) allows you to capture the query plans generated by Aurora Replicas and stores them on the primary DB instance of the Aurora DB cluster. You can collect the query plans from all the Aurora Replicas, and maintain a set of optimal plans in a central persistent table on the primary instance. You can then apply these plans on other Replicas when needed. This helps you to maintain the stability of execution plans and improve performance of the queries across the DB clusters and engine versions.

Topics

- Prerequisites
- Managing plan capture for Aurora Replicas
- Troubleshooting

Prerequisites

Turn on `capture_plan_baselines parameter` in Aurora Replica - Set `capture_plan_baselines` parameter to automatic or manual to capture plans in Aurora Replicas. For more information, see apg_plan_mgmt.capture_plan_baselines.

Install postgres_fdw extension - You must install `postgres_fdw` foreign data wrapper extension to capture plans in Aurora Replicas. Run the following command in each database, to install the extension.

```
postgres=> CREATE EXTENSION IF NOT EXISTS postgres_fdw;
```

Managing plan capture for Aurora Replicas

Turn on plan capture for Aurora Replicas

You must have `rds_superuser` privileges to create or remove Plan Capture in Aurora Replicas. For more information on user roles and permissions, see Understanding PostgreSQL roles and permissions.

To capture plans, call the function apg_plan_mgmt.create_replica_plan_capture in the writer DB instance, as shown in the following:

```
postgres=> CALL
 apg_plan_mgmt.create_replica_plan_capture('cluster_endpoint', 'password');
```

- cluster_endpoint - cluster_endpoint (writer endpoint) provides failover support for Plan Capture in Aurora Replicas.
- password - We recommend you to follow the below guidelines while creating the password to enhance the security:
  - It must contain at least 8 characters.
  - It must contain at least one uppercase, one lowercase letter, and one number.
  - It must have at least one special character (?, !, #, <, >, *, and so on).

Note

If you change the cluster endpoint, password, or port number, you must run `apg_plan_mgmt.create_replica_plan_capture()` again with the cluster endpoint and password to re-initialize the plan capture. If not, capturing plans from Aurora Replicas will fail.

Turn off plan capture for Aurora Replicas

You can turn off `capture_plan_baselines` parameter in Aurora Replica by setting its value to `off` in the Parameter group.

Remove plan capture for Aurora Replicas

You can completely remove Plan Capture in Aurora Replicas but make sure before you do. To remove plan capture, call `apg_plan_mgmt.remove_replica_plan_capture` as shown:

```
postgres=> CALL apg_plan_mgmt.remove_replica_plan_capture();
```

You must call apg_plan_mgmt.create_replica_plan_capture() again to turn on plan capture in Aurora Replicas with the cluster endpoint and password.

Troubleshooting

Following, you can find troubleshooting ideas and workarounds if the plan is not captured in Aurora Replicas as expected.

- **Parameter settings** - Check if the `capture_plan_baselines` parameter is set to proper value to turn on plan capture.
- **`postgres_fdw` extension is installed** - Use the following query to check if `postgres_fdw` is installed.

  ```
  postgres=> SELECT * FROM pg_extension WHERE extname = 'postgres_fdw'
  ```

- **create_replica_plan_capture() is called** - Use the following command to check if the user mapping exits. Otherwise, call `create_replica_plan_capture()` to initialize the feature.

  ```
  postgres=> SELECT * FROM pg_foreign_server WHERE srvname =
   'apg_plan_mgmt_writer_foreign_server';
  ```

- **Cluster endpoint and port number** - Check if the cluster endpoint and port number if appropriate. There won't be any error message displayed if these values are incorrect.

Use the following command to verify if the endpoint used in the create() and to check which database it resides in:

```
postgres=> SELECT srvoptions FROM pg_foreign_server WHERE srvname =
 'apg_plan_mgmt_writer_foreign_server';
```

- **reload()** - You must call apg_plan_mgmt.reload() after calling apg_plan_mgmt.delete_plan() in Aurora Replicas to make the delete function effective. This ensures that the change has been successfully implemented.
- **Password** - You must enter password in create_replica_plan_capture() as per the guidelines mentioned. Otherwise, you will receive an error message. For more information,see Managing plan capture for Aurora Replicas. Use another password that aligns with the requirements.
- **Cross-Region connection** - Plan capture in Aurora Replicas is also supported in Aurora global database, where writer instance and Aurora Replicas can be in different regions. The writer instance and cross-Region Replica must be able to communicate, using VPC Peering. For more information, see VPC peering. If a cross-Region failover happens, you must reconfigure the endpoint to new primary DB cluster endpoint.

Supporting table partition

Aurora PostgreSQL Query Plan Management (QPM) supports declarative table partitioning in the following versions:

- 15.3 and higher 15 versions
- 14.8 and higher 14 versions
- 13.11 and higher 13 versions

For more information, see Table Partitioning.

Topics

Setting up table partition

To set up table partition in Aurora PostgreSQL QPM, do as follows:

1. Set apg_plan_mgmt.plan_hash_version to 3 or more in the DB cluster parameter group.
2. Navigate to a database that uses Query Plan Management and has entries in apg_plan_mgmt.dba_plans view.
3. Call apg_plan_mgmt.validate_plans('update_plan_hash') to update the plan_hash value in the plans table.
4. Repeat steps 2-3 for all databases with Query Plan Management enabled that have entries in apg_plan_mgmt.dba_plans view.

For more information on these parameters, see Parameter reference for Aurora PostgreSQL query plan management.

Capturing plans for table partition

In QPM, different plans are distinguished by their plan_hash value. To understand how the plan_hash changes, you must first understand similar kind of plans.

The combination of access methods, digit-stripped index names and digit-stripped partition names, accumulated at the Append node level must be constant for plans to be considered the same. The specific partitions accessed in the plans are not significant. In the following example, a table tbl_a is created with 4 partitions.

```
postgres=>create table tbl_a(i int, j int, k int, l int, m int) partition by range(i);
CREATE TABLE
postgres=>create table tbl_a1 partition of tbl_a for values from (0) to (1000);
CREATE TABLE
postgres=>create table tbl_a2 partition of tbl_a for values from (1001) to (2000);
CREATE TABLE
postgres=>create table tbl_a3 partition of tbl_a for values from (2001) to (3000);
CREATE TABLE
postgres=>create table tbl_a4 partition of tbl_a for values from (3001) to (4000);
CREATE TABLE
postgres=>create index t_i on tbl_a using btree (i);
CREATE INDEX
postgres=>create index t_j on tbl_a using btree (j);
CREATE INDEX
```

```
postgres=>create index t_k on tbl_a using btree (k);
CREATE INDEX
```

The following plans are considered the same because a single scan method is being used to scan `tbl_a` irrespective of the number of partitions that the query looks up.

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 999 and j < 9910 and k > 50;

                        QUERY PLAN
-------------------------------------------------------------------
Seq Scan on tbl_a1 tbl_a
    Filter: ((i >= 990) AND (i <= 999) AND (j < 9910) AND (k > 50))
SQL Hash: 1553185667, Plan Hash: -694232056
(3 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 1100 and j < 9910 and k > 50;

                        QUERY PLAN
-------------------------------------------------------------------
Append
    ->  Seq Scan on tbl_a1 tbl_a_1
            Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
    ->  Seq Scan on tbl_a2 tbl_a_2
            Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
    SQL Hash: 1553185667, Plan Hash: -694232056
    (6 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 2100 and j < 9910 and k > 50;

                QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a3 tbl_a_3
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -694232056
```

```
(8 rows)
```

The following 3 plans are also considered the same because at the parent level, the access methods, digit-stripped index names and digit-stripped partition names are `SeqScan tbl_a`, `IndexScan (i_idx) tbl_a`.

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 1100 and j < 9910 and k > 50;

                                QUERY PLAN
---------------------------------------------------------------------------
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a2_i_idx on tbl_a2 tbl_a_2
         Index Cond: ((i >= 990) AND (i <= 1100))
         Filter: ((j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -993736942
(7 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 2100 and j < 9910 and k > 50;

                                QUERY PLAN
---------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a3_i_idx on tbl_a3 tbl_a_3
         Index Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -993736942
(10 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 3100 and j < 9910 and k > 50;

                                QUERY PLAN
---------------------------------------------------------------------------
```

```
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a3 tbl_a_3
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a4_i_idx on tbl_a4 tbl_a_4
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -993736942
(11 rows)
```

Irrespective of different order and number of occurrences in child partitions, the access methods, digit-stripped index names and digit-stripped partition names are constant at the parent level for each of the above plans.

However, the plans would be considered different if any of the following conditions are met:

- Any additional access methods are used in the plan.

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between
 990 and 2100 and j < 9910 and k > 50;

                                QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Bitmap Heap Scan on tbl_a3 tbl_a_3
         Recheck Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
         ->  Bitmap Index Scan on tbl_a3_i_idx
               Index Cond: ((i >= 990) AND (i <= 2100))
 SQL Hash: 1553185667, Plan Hash: 1134525070
(11 rows)
```

- Any of the access methods in the plan are not used anymore.

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between
 990 and 1100 and j < 9910 and k > 50;
```

```
                               QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -694232056
(6 rows)
```

- The index associated with an index method is changed.

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between
 990 and 1100 and j < 9910 and k > 50;

                             QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Seq Scan on tbl_a1 tbl_a_1
         Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a2_j_idx on tbl_a2 tbl_a_2
         Index Cond: (j < 9910)
         Filter: ((i >= 990) AND (i <= 1100) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -993343726
(7 rows)
```

Enforcing a table partition plan

Approved plans for partitioned tables are enforced with positional correspondence. The plans are not specific to the partitions, and can be enforced on partitions other than the plans referenced in the original query. Plans also have the capability of being enforced for queries accessing a different number of partitions than the original approved outline.

For example, if the approved outline is for the following plan:

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 2100 and j < 9910 and k > 50;

                                QUERY PLAN
--------------------------------------------------------------------------
 Append
```

```
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a3_i_idx on tbl_a3 tbl_a_3
         Index Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
 SQL Hash: 1553185667, Plan Hash: -993736942
(10 rows)
```

Then, this plan can be enforced on SQL queries referencing 2, 4, or more partitions as well. The possible plans that could arise from these scenarios for 2 and 4 partition access are:

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 1100 and j < 9910 and k > 50;

                                QUERY PLAN
----------------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 1100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 1100) AND (j < 9910) AND (k > 50))
 Note: An Approved plan was used instead of the minimum cost plan.
 SQL Hash: 1553185667, Plan Hash: -993736942, Minimum Cost Plan Hash: -1873216041
(8 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 3100 and j < 9910 and k > 50;

                                QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a3_i_idx on tbl_a3 tbl_a_3
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a4 tbl_a_4
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
 Note: An Approved plan was used instead of the minimum cost plan.
 SQL Hash: 1553185667, Plan Hash: -993736942, Minimum Cost Plan Hash: -1873216041
(12 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 3100 and j < 9910 and k > 50;

                                QUERY PLAN
----------------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 3100) AND (j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a3_i_idx on tbl_a3 tbl_a_3
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Index Scan using tbl_a4_i_idx on tbl_a4 tbl_a_4
         Index Cond: ((i >= 990) AND (i <= 3100))
         Filter: ((j < 9910) AND (k > 50))
 Note: An Approved plan was used instead of the minimum cost plan.
 SQL Hash: 1553185667, Plan Hash: -993736942, Minimum Cost Plan Hash: -1873216041
(14 rows)
```

Consider another approved plan with different access methods for each partition:

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 2100 and j < 9910 and k > 50;

                                QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 2100) AND (j < 9910) AND (k > 50))
   ->  Bitmap Heap Scan on tbl_a3 tbl_a_3
         Recheck Cond: ((i >= 990) AND (i <= 2100))
         Filter: ((j < 9910) AND (k > 50))
         ->  Bitmap Index Scan on tbl_a3_i_idx
               Index Cond: ((i >= 990) AND (i <= 2100))
 SQL Hash: 1553185667, Plan Hash: 2032136998
(12 rows)
```

In this case, any plan that reads from two partitions would fail to be enforced. Unless all of the (access method, index name) combinations from the approved plan are usable, the plan cannot be enforced. For example, the following plans have different plan hashes and the approved plan can't be enforced in these cases:

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 1900 and j < 9910 and k > 50;

                              QUERY PLAN
-------------------------------------------------------------------------
 Append
   ->  Bitmap Heap Scan on tbl_a1 tbl_a_1
         Recheck Cond: ((i >= 990) AND (i <= 1900))
         Filter: ((j < 9910) AND (k > 50))
         ->  Bitmap Index Scan on tbl_a1_i_idx
               Index Cond: ((i >= 990) AND (i <= 1900))
   ->  Bitmap Heap Scan on tbl_a2 tbl_a_2
         Recheck Cond: ((i >= 990) AND (i <= 1900))
         Filter: ((j < 9910) AND (k > 50))
         ->  Bitmap Index Scan on tbl_a2_i_idx
               Index Cond: ((i >= 990) AND (i <= 1900))
  Note: This is not an Approved plan.  No usable Approved plan was found.
  SQL Hash: 1553185667, Plan Hash: -568647260
(13 rows)
```

```
postgres=>explain (hashes true, costs false) select j, k from tbl_a where i between 990
 and 1900 and j < 9910 and k > 50;

                              QUERY PLAN
--------------------------------------------------------------------------
 Append
   ->  Index Scan using tbl_a1_i_idx on tbl_a1 tbl_a_1
         Index Cond: ((i >= 990) AND (i <= 1900))
         Filter: ((j < 9910) AND (k > 50))
   ->  Seq Scan on tbl_a2 tbl_a_2
         Filter: ((i >= 990) AND (i <= 1900) AND (j < 9910) AND (k > 50))
 Note: This is not an Approved plan.  No usable Approved plan was found.
 SQL Hash: 1553185667, Plan Hash: -496793743
(8 rows)
```

Naming Convention

For QPM to enforce a plan with declarative partitioned tables, you must follow specific naming rules for parent tables, table partitions, and indexes:

- **Parent table names –** These names must differ by alphabets or special characters, and not by just digits. For example, tA, tB, and tC are acceptable names for separate parent tables while t1, t2, and t3 are not.
- **Individual partition table names –** Partitions of the same parent should differ from one another by digits only. For example, acceptable partition names of tA could be tA1, tA2 or t1A, t2A or even multiple digits.

  Any other differences (letters, special characters) will not guarantee plan enforcement.
- **Index names –** In partition table hierarchy, make sure that all indexes have unique names. This means that the non-numeric parts of the names must be different. For example, if you have a partitioned table named `tA` with an index named `tA_col1_idx1`, you can't have another index named `tA_col1_idx2`. However, you can have an index called `tA_a_col1_idx2` because the non-numeric part of the name is unique. This rule applies to indexes created on both the parent table and individual partition tables.

Failure to adhere to the above naming conventions may result in failure of approved plans enforcement. The following example illustrates such a failed enforcement:

```
postgres=>create table t1(i int, j int, k int, l int, m int) partition by range(i);
CREATE TABLE
postgres=>create table t1a partition of t1 for values from (0) to (1000);
CREATE TABLE
postgres=>create table t1b partition of t1 for values from (1001) to (2000);
CREATE TABLE
postgres=>SET apg_plan_mgmt.capture_plan_baselines TO 'manual';
SET
postgres=>explain (hashes true, costs false) select count(*) from t1 where i > 0;

                            QUERY PLAN
--------------------------------------------------------------------------
 Aggregate
   ->  Append
```

```
         ->  Seq Scan on t1a t1_1
               Filter: (i > 0)
         ->  Seq Scan on t1b t1_2
               Filter: (i > 0)
 SQL Hash: -1720232281, Plan Hash: -1010664377
(7 rows)
```

```
postgres=>SET apg_plan_mgmt.use_plan_baselines TO 'on';
SET
postgres=>explain (hashes true, costs false) select count(*) from t1 where i > 1000;

                            QUERY PLAN
-------------------------------------------------------------------------
 Aggregate
   ->  Seq Scan on t1b t1
         Filter: (i > 1000)
 Note: This is not an Approved plan. No usable Approved plan was found.
 SQL Hash: -1720232281, Plan Hash: 335531806
(5 rows)
```

Even though the two plans might appear identical, their `Plan Hash` values are different due to the names of the child tables. The table names vary by alpha characters instead of just digits leading to an enforcement failure.

Working with extensions and foreign data wrappers

To extend the functionality to your Aurora PostgreSQL-Compatible Edition DB cluster, you can install and use various PostgreSQL *extensions*. For example, if your use case calls for intensive data entry across very large tables, you can install the `pg_partman` extension to partition your data and thus spread the workload.

> **Note**
>
> As of Aurora PostgreSQL 14.5, Aurora PostgreSQL supports Trusted Language Extensions for PostgreSQL. This feature is implemented as the extension `pg_tle`, which you can add to your Aurora PostgreSQL. By using this extension, developers can create their own PostgreSQL extensions in a safe environment that simplifies the setup and configuration requirements, as well as much of the preliminary testing for new extensions. For more information, see Working with Trusted Language Extensions for PostgreSQL.

In some cases, rather than installing an extension, you might add a specific *module* to the list of `shared_preload_libraries` in your Aurora PostgreSQL DB cluster's custom DB cluster parameter group. Typically, the default DB cluster parameter group loads only the `pg_stat_statements`, but several other modules are available to add to the list. For example, you can add scheduling capability by adding the `pg_cron` module, as detailed in Scheduling maintenance with the PostgreSQL pg_cron extension. As another example, you can log query execution plans by loading the `auto_explain` module. To learn more, see Logging execution plans of queries in the AWS knowledge center.

An extension that provides access to external data is more specifically known as a *foreign data wrapper* (FDW). As one example, the `oracle_fdw` extension allows your Aurora PostgreSQL DB cluster to work with Oracle databases.

You can also specify precisely which extensions can be installed on your Aurora PostgreSQL DB instance, by listing them in the `rds.allowed_extensions` parameter. For more information, see Restricting installation of PostgreSQL extensions.

Following, you can find information about setting up and using some of the extensions, modules, and FDWs available for Aurora PostgreSQL. For simplicity's sake, these are all referred to as "extensions." You can find listings of extensions that you can use with the currently available

Aurora PostgreSQL versions, see Extension versions for Amazon Aurora PostgreSQL in the *Release Notes for Aurora PostgreSQL*.

- Managing large objects with the lo module
- Managing spatial data with the PostGIS extension
- Managing PostgreSQL partitions with the pg_partman extension
- Scheduling maintenance with the PostgreSQL pg_cron extension
- Using pgAudit to log database activity
- Using pglogical to synchronize data across instances
- Working with Oracle databases by using the oracle_fdw extension
- Working with SQL Server databases by using the tds_fdw extension

Using Amazon Aurora delegated extension support for PostgreSQL

Using Amazon Aurora delegated extension support for PostgreSQL, you can delegate the extension management to a user who need not be an `rds_superuser`. With this delegated extension support, a new role called `rds_extension` is created and you must assign this to a user to manage other extensions. This role can create, update, and drop extensions.

You can specify the extensions that can be installed on your Aurora PostgreSQL DB instance, by listing them in the `rds.allowed_extensions` parameter. For more information, see Using PostgreSQL extensions with Amazon RDS for PostgreSQL.

You can restrict the list of extensions available that can be managed by the user with the `rds_extension` role using `rds.allowed_delegated_extensions` parameter.

The delegated extension support is available in the following versions:

- All higher versions
- 15.5 and higher 15 versions
- 14.10 and higher 14 versions
- 13.13 and higher 13 versions
- 12.17 and higher 12 versions

Topics

Turning on delegate extension support to a user

You must perform the following to enable delegate extension support to a user:

1. **Grant `rds_extension` role to a user** – Connect to the database as `rds_superuser` and execute the following command:

   ```
   Postgres => grant rds_extension to user_name;
   ```

2. **Set the list of extensions available for delegated users to manage** – The `rds.allowed_delegated_extensions` allows you to specify a subset of the available extensions using `rds.allowed_extensions` in the DB cluster parameter. You can perform this at one of the following levels:

   - In the cluster or the instance parameter group, through the AWS Management Console or API. For more information, see Working with parameter groups.
   - Use the following command at the database level:

     ```
     alter database database_name set rds.allowed_delegated_extensions =
      'extension_name_1,
                         extension_name_2,...extension_name_n';
     ```

   - Use the following command at the user level:

     ```
     alter user user_name set rds.allowed_delegated_extensions = 'extension_name_1,
                         extension_name_2,...extension_name_n';
     ```

> **Note**
>
> You need not restart the database after changing the `rds.allowed_delegated_extensions` dynamic parameter.

3. **Allow access to the delegated user to objects created during the extension creation process** – Certain extensions create objects that require additional permissions to be granted before the user with `rds_extension` role can access them. The `rds_superuser` must grant the delegated user access to those objects. One of the options is to use an event trigger to automatically grant permission to the delegated user. For more information, refer to the event trigger example in Turning off the support for the delegated extension.

Configuration used in Aurora delegated extension support for PostgreSQL

Config tion Name	Description	Default Value	Notes	Who can modify or grant permission
`rds.a ed_de ted_e sions`	This parameter limits the extensions a rds_extension role can manage in a database. It must be a subset of rds.allowed_extensions.	empty string	• By default, this parameter is empty string, which means that no extensions have been delegated to users with `rds_extension` . • Any supported extension can be added if the user has permission to do so. To do this, set the `rds.allow`	rds_superuser

Configuration Name	Description	Default Value	Notes	Who can modify or grant permission
			`ed_delegated_extensions` parameter to a string of comma-separated extension names. By adding a list of extensions to this parameter, you explicitly identify the extensions that the user with the `rds_extension` role can install. • When set to `*`, it means that all extensions listed in `rds_allowed_extensions` are delegated to users with `rds_extension` role. To learn more about setting up this parameter, see Turning on	

Config tion Name	Description	Default Value	Notes	Who can modify or grant permission
			delegate extension support to a user.	
`rds.aed_exions`	This parameter lets the customer limit the extensions that can be installed in the Aurora PostgreSQL DB instance. For more information, see Restricting installation of PostgreSQL extensions	"*"	By default, this parameter is set to "*", which means that all extensions supported on RDS for PostgreSQL and Aurora PostgreSQL are allowed to be created by users with necessary privileges. Empty means no extensions can be installed in the Aurora PostgreSQL DB instance.	administrator

Configuration Name	Description	Default Value	Notes	Who can modify or grant permission
`rds-delegated_extension_allow_drop_cascade`	This parameter controls the ability for user with `rds_extension` to drop the extension using a cascade option.	off	By default, `rds-delegated_extension_allow_drop_cascade` is set to `off`. This means that users with `rds_extension` are not allowed to drop an extension using the cascade option. To grant that ability, the `rds.delegated_extension_allow_drop_cascade` parameter should be set to on.	rds_superuser

Turning off the support for the delegated extension

Turning off partially

The delegated users can’t create new extensions but can still update existing extensions.

- Reset `rds.allowed_delegated_extensions` to the default value in the DB cluster parameter group.
- Use the following command at the database level:

```
alter database database_name reset rds.allowed_delegated_extensions;
```

- Use the following command at the user level:

```
alter user user_name reset rds.allowed_delegated_extensions;
```

Turning off fully

Revoking `rds_extension` role from a user will revert the user to standard permissions. The user can no longer create, update, or drop extensions.

```
postgres => revoke rds_extension from user_name;
```

Example of event trigger

If you want to allow a delegated user with `rds_extension` to use extensions that require setting permissions on their objects created by the extension creation, you can customize the below example of an event trigger and add only the extensions for which you want the delegated users to have access to the full functionality. This event trigger can be created on template1 (the default template), therefore all database created from template1 will have that event trigger. When a delegated user installs the extension, this trigger will automatically grant ownership on the objects created by the extension.

```

CREATE OR REPLACE FUNCTION create_ext()

  RETURNS event_trigger AS $$

DECLARE

  schemaname TEXT;
  databaseowner TEXT;

  r RECORD;

BEGIN

  IF tg_tag = 'CREATE EXTENSION' and current_user != 'rds_superuser' THEN
    RAISE NOTICE 'SECURITY INVOKER';
```

```
   RAISE NOTICE 'user: %', current_user;
   FOR r IN SELECT * FROM pg_event_trigger_ddl_commands()
   LOOP
       CONTINUE WHEN r.command_tag != 'CREATE EXTENSION' OR r.object_type != 
'extension';

       schemaname = (
           SELECT n.nspname
           FROM pg_catalog.pg_extension AS e
           INNER JOIN pg_catalog.pg_namespace AS n
           ON e.extnamespace = n.oid
           WHERE e.oid = r.objid
       );

       databaseowner = (
           SELECT pg_catalog.pg_get_userbyid(d.datdba)
           FROM pg_catalog.pg_database d
           WHERE d.datname = current_database()
       );
       RAISE NOTICE 'Record for event trigger %, objid: %,tag: %, current_user: %, 
schema: %, database_owenr: %', r.object_identity, r.objid, tg_tag, current_user, 
schemaname, databaseowner;
       IF r.object_identity = 'address_standardizer_data_us' THEN
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE %I.us_gaz TO 
%I WITH GRANT OPTION;', schemaname, databaseowner);
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE %I.us_lex TO 
%I WITH GRANT OPTION;', schemaname, databaseowner);
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE %I.us_rules 
TO %I WITH GRANT OPTION;', schemaname, databaseowner);
       ELSIF r.object_identity = 'dict_int' THEN
           EXECUTE format('ALTER TEXT SEARCH DICTIONARY %I.intdict OWNER TO %I;', 
schemaname, databaseowner);
       ELSIF r.object_identity = 'pg_partman' THEN
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE 
%I.part_config TO %I WITH GRANT OPTION;', schemaname, databaseowner);
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE 
%I.part_config_sub TO %I WITH GRANT OPTION;', schemaname, databaseowner);
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON TABLE 
%I.custom_time_partitions TO %I WITH GRANT OPTION;', schemaname, databaseowner);
       ELSIF r.object_identity = 'postgis_topology' THEN
           EXECUTE format('GRANT SELECT, UPDATE, INSERT, DELETE ON ALL TABLES IN 
SCHEMA topology TO %I WITH GRANT OPTION;', databaseowner);
           EXECUTE format('GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA topology TO 
%I WITH GRANT OPTION;', databaseowner);
```

```
            EXECUTE format('GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA topology TO %I
 WITH GRANT OPTION;', databaseowner);
            EXECUTE format('GRANT USAGE ON SCHEMA topology TO %I WITH GRANT OPTION;',
 databaseowner);
        END IF;
    END LOOP;
  END IF;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

CREATE EVENT TRIGGER log_create_ext ON ddl_command_end EXECUTE PROCEDURE create_ext();
```

Benefits of using Amazon Aurora delegated extension support

By using Amazon Aurora delegated extension support for PostgreSQL, you securely delegate the extension management to users who do not have the `rds_superuser` role. This feature provides the following benefits:

- You can easily delegate extension management to users of your choice.
- This doesn’t require `rds_superuser` role.
- Provides ability to support different set of extensions for different databases in the same DB cluster.

Limitation of Aurora delegated extension support for PostgreSQL

- Objects created during the extension creation process may require additional privileges for the extension to function properly.

Permissions required for certain extensions

In order to create, use, or update the following extensions, the delegated user should have the necessary privileges on the following functions, tables, and schema.

Exte s that need own or perm ns	Function	Tables	Schema	Text Search Dictionary	Comment
addr tand er_da s		us_gaz, us_lex, us_lex, I.us_rules			
amch	bt_index_check, bt_index_parent_check				
dict_				intdict	
pg_p n		custom_time_partitions, part_config, part_config_sub			
pg_s tater					
Post	st_tileenvelope	spatial_ref_sys			
postg aster					
postg opol		topology, layer	topology		the delegated user Must be

Exter s that need own or perm ns	Function	Tables	Schema	Text Search Dictionary	Comment
					the database owner
log_f	create_foreign_table_for_log_file				
rds_t	role_password_encryption_type				
postg iger_ oder		geocode_settings_default, geocode_settings	tiger		
pg_fr acem	pg_freespace				
pg_v lity	pg_visibility				

Security Considerations

Keep in mind that a user with `rds_extension` role will be able to manage extensions on all databases they have the connect privilege on. If the intention is to have a delegated user manage extension on a single database, a good practice is to revoke all privileges from public on each database, then explicitly grant the connect privilege for that specific database to the delegate user.

There are several extensions that can allow a user to access information from multiple database. Ensure the users you grant `rds_extension` has cross database capabilities before adding these extensions to `rds.allowed_delegated_extensions`. For example, `postgres_fdw` and `dblink` provide functionality to query across databases on the same instance or remote instances. `log_fdw` reads the postgres engine log files, which are for all databases in the instance, potentially containing slow queries or error messages from multiple databases. `pg_cron` enables running scheduled background jobs on the DB instance and can configure jobs to run in a different database.

Drop extension cascade disabled

The ability to drop the extension with cascade option by a user with the `rds_extension` role is controlled by `rds.delegated_extension_allow_drop_cascade` parameter. By default, `rds-delegated_extension_allow_drop_cascade` is set to `off`. This means that users with the `rds_extension` role are not allowed to drop an extension using the cascade option as shown in the below query.

```
DROP EXTENSION CASCADE;
```

As this will automatically drop objects that depend on the extension, and in turn all objects that depend on those objects. Attempting to use the cascade option will result in an error.

To grant that ability, the `rds.delegated_extension_allow_drop_cascade` parameter should be set to on.

Changing the `rds.delegated_extension_allow_drop_cascade` dynamic parameter doesn't require a database restart. You can do this at one of the following levels:

- In the cluster or the instance parameter group, through the AWS Management Console or API.
- Using the following command at the database level:

  ```
  alter database database_name set rds.delegated_extension_allow_drop_cascade = 'on';
  ```

- Using the following command at the user level:

  ```
  alter role tenant_user set rds.delegated_extension_allow_drop_cascade = 'on';
  ```

Example extensions that can be added using delegated extension support

- `rds_tools`

```
extension_test_db=> create extension rds_tools;
CREATE EXTENSION
extension_test_db=> SELECT * from rds_tools.role_password_encryption_type() where
 rolname = 'pg_read_server_files';
ERROR: permission denied for function role_password_encryption_type
```

- `amcheck`

```
extension_test_db=> CREATE TABLE amcheck_test (id int);
CREATE TABLE
extension_test_db=> INSERT INTO amcheck_test VALUES (generate_series(1,100000));
INSERT 0 100000
extension_test_db=> CREATE INDEX amcheck_test_btree_idx ON amcheck_test USING btree
 (id);
CREATE INDEX
extension_test_db=> create extension amcheck;
CREATE EXTENSION
extension_test_db=> SELECT bt_index_check('amcheck_test_btree_idx'::regclass);
ERROR: permission denied for function bt_index_check
extension_test_db=> SELECT bt_index_parent_check('amcheck_test_btree_idx'::regclass);
ERROR: permission denied for function bt_index_parent_check
```

- `pg_freespacemap`

```
extension_test_db=> create extension pg_freespacemap;
CREATE EXTENSION
extension_test_db=> SELECT * FROM pg_freespace('pg_authid');
ERROR: permission denied for function pg_freespace
extension_test_db=> SELECT * FROM pg_freespace('pg_authid',0);
ERROR: permission denied for function pg_freespace
```

- `pg_visibility`

```
extension_test_db=> create extension pg_visibility;
CREATE EXTENSION
extension_test_db=> select * from pg_visibility('pg_database'::regclass);
ERROR: permission denied for function pg_visibility
```

- `postgres_fdw`

```
extension_test_db=> create extension postgres_fdw;
CREATE EXTENSION
extension_test_db=> create server myserver foreign data wrapper postgres_fdw options
 (host 'foo', dbname 'foodb', port '5432');
ERROR: permission denied for foreign-data wrapper postgres_fdw
```

Managing large objects with the lo module

The lo module (extension) is for database users and developers working with PostgreSQL databases through JDBC or ODBC drivers. Both JDBC and ODBC expect the database to handle deletion of large objects when references to them change. However, PostgreSQL doesn't work that way. PostgreSQL doesn't assume that an object should be deleted when its reference changes. The result is that objects remain on disk, unreferenced. The lo extension includes a function that you use to trigger on reference changes to delete objects if needed.

> **Tip**
>
> To determine if your database can benefit from the lo extension, use the vacuumlo utility to check for orphaned large objects. To get counts of orphaned large objects without taking any action, run the utility with the `-n` option (no-op). To learn how, see vacuumlo utility following.

The lo module is available for Aurora PostgreSQL 13.7, 12.11, 11.16, 10.21 and higher minor versions.

To install the module (extension), you need `rds_superuser` privileges. Installing the lo extension adds the following to your database:

- `lo` – This is a large object (lo) data type that you can use for binary large objects (BLOBs) and other large objects. The `lo` data type is a domain of the `oid` data type. In other words, it's an object identifier with optional constraints. For more information, see Object identifiers in the PostgreSQL documentation. In simple terms, you can use the `lo` data type to distinguish your database columns that hold large object references from other object identifiers (OIDs).
- `lo_manage` – This is a function that you can use in triggers on table columns that contain large object references. Whenever you delete or modify a value that references a large object, the

trigger unlinks the object (lo_unlink) from its reference. Use the trigger on a column only if the column is the sole database reference to the large object.

For more information about the large objects module, see lo in the PostgreSQL documentation.

Installing the lo extension

Before installing the lo extension, make sure that you have rds_superuser privileges.

To install the lo extension

1. Use psql to connect to the primary DB instance of your Aurora PostgreSQL DB cluster.

   ```
   psql --host=your-cluster-instance-1.666666666666.aws-region.rds.amazonaws.com --
   port=5432 --username=postgres --password
   ```

   When prompted, enter your password. The psql client connects and displays the default administrative connection database, postgres=>, as the prompt.

2. Install the extension as follows.

   ```
   postgres=> CREATE EXTENSION lo;
   CREATE EXTENSION
   ```

You can now use the lo data type to define columns in your tables. For example, you can create a table (images) that contains raster image data. You can use the lo data type for a column raster, as shown in the following example, which creates a table.

```
postgres=> CREATE TABLE images (image_name text, raster lo);
```

Using the lo_manage trigger function to delete objects

You can use the lo_manage function in a trigger on a lo or other large object columns to clean up (and prevent orphaned objects) when the lo is updated or deleted.

To set up triggers on columns that reference large objects

- Do one of the following:

- Create a BEFORE UPDATE OR DELETE trigger on each column to contain unique references to large objects, using the column name for the argument.

  ```
  postgres=> CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON images
      FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
  ```

- Apply a trigger only when the column is being updated.

  ```
  postgres=> CREATE TRIGGER t_raster BEFORE UPDATE OF images
      FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
  ```

The `lo_manage` trigger function works only in the context of inserting or deleting column data, depending on how you define the trigger. It has no effect when you perform a DROP or TRUNCATE operation on a database. That means that you should delete object columns from any tables before dropping, to prevent creating orphaned objects.

For example, suppose that you want to drop the database containing the `images` table. You delete the column as follows.

```
postgres=> DELETE FROM images COLUMN raster
```

Assuming that the `lo_manage` function is defined on that column to handle deletes, you can now safely drop the table.

Using the vacuumlo utility

The vacuumlo utility identifies and can remove orphaned large objects from databases. This utility has been available since PostgreSQL 9.1.24. If your database users routinely work with large objects, we recommend that you run vacuumlo occasionally to clean up orphaned large objects.

Before installing the lo extension, you can use vacuumlo to assess whether your Aurora PostgreSQL DB cluster can benefit. To do so, run `vacuumlo` with the `-n` option (no-op) to show what would be removed, as shown in the following:

```
$ vacuumlo -v -n -h your-cluster-instance-1.666666666666.aws-region.rds.amazonaws.com -
p 5433 -U postgres docs-lab-spatial-db
Password:*****
Connected to database "docs-lab-spatial-db"
Test run: no large objects will be removed!
```

```
Would remove 0 large objects from database "docs-lab-spatial-db".
```

As the output shows, orphaned large objects aren't a problem for this particular database.

For more information about this utility, see vacuumlo in the PostgreSQL documentation.

Managing spatial data with the PostGIS extension

PostGIS is an extension to PostgreSQL for storing and managing spatial information. To learn more about PostGIS, see PostGIS.net.

Starting with version 10.5, PostgreSQL supports the libprotobuf 1.3.0 library used by PostGIS for working with map box vector tile data.

Setting up the PostGIS extension requires `rds_superuser` privileges. We recommend that you create a user (role) to manage the PostGIS extension and your spatial data. The PostGIS extension and its related components add thousands of functions to PostgreSQL. Consider creating the PostGIS extension in its own schema if that makes sense for your use case. The following example shows how to install the extension in its own database, but this isn't required.

Topics

Step 1: Create a user (role) to manage the PostGIS extension

First, connect to your RDS for PostgreSQL DB instance as a user that has `rds_superuser` privileges. If you kept the default name when you set up your instance, you connect as `postgres`.

```
psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --username=postgres
 --password
```

Create a separate role (user) to administer the PostGIS extension.

```
postgres=>  CREATE ROLE gis_admin LOGIN PASSWORD 'change_me';
CREATE ROLE
```

Grant this role `rds_superuser` privileges, to allow the role to install the extension.

```
postgres=> GRANT rds_superuser TO gis_admin;
GRANT
```

Create a database to use for your PostGIS artifacts. This step is optional. Or you can create a schema in your user database for the PostGIS extensions, but this also isn't required.

```
postgres=> CREATE DATABASE lab_gis;
CREATE DATABASE
```

Give the `gis_admin` all privileges on the `lab_gis` database.

```
postgres=> GRANT ALL PRIVILEGES ON DATABASE lab_gis TO gis_admin;
GRANT
```

Exit the session and reconnect to your RDS for PostgreSQL DB instance as `gis_admin`.

```
postgres=> psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
username=gis_admin --password --dbname=lab_gis
Password for user gis_admin:...
lab_gis=>
```

Continue setting up the extension as detailed in the next steps.

Step 2: Load the PostGIS extensions

The PostGIS extension includes several related extensions that work together to provide geospatial functionality. Depending on your use case, you might not need all the extensions created in this step.

Use `CREATE EXTENSION` statements to load the PostGIS extensions.

```
CREATE EXTENSION postgis;
CREATE EXTENSION
CREATE EXTENSION postgis_raster;
CREATE EXTENSION
CREATE EXTENSION fuzzystrmatch;
CREATE EXTENSION
CREATE EXTENSION postgis_tiger_geocoder;
CREATE EXTENSION
CREATE EXTENSION postgis_topology;
CREATE EXTENSION
CREATE EXTENSION address_standardizer_data_us;
CREATE EXTENSION
```

You can verify the results by running the SQL query shown in the following example, which lists the extensions and their owners.

```
SELECT n.nspname AS "Name",
  pg_catalog.pg_get_userbyid(n.nspowner) AS "Owner"
  FROM pg_catalog.pg_namespace n
  WHERE n.nspname !~ '^pg_' AND n.nspname <> 'information_schema'
  ORDER BY 1;
List of schemas
     Name     |   Owner
--------------+-----------
 public       | postgres
 tiger        | rdsadmin
 tiger_data   | rdsadmin
 topology     | rdsadmin
(4 rows)
```

Step 3: Transfer ownership of the extensions

Use the ALTER SCHEMA statements to transfer ownership of the schemas to the `gis_admin` role.

```
ALTER SCHEMA tiger OWNER TO gis_admin;
ALTER SCHEMA
ALTER SCHEMA tiger_data OWNER TO gis_admin;
ALTER SCHEMA
ALTER SCHEMA topology OWNER TO gis_admin;
ALTER SCHEMA
```

You can confirm the ownership change by running the following SQL query. Or you can use the \dn metacommand from the psql command line.

```
SELECT n.nspname AS "Name",
  pg_catalog.pg_get_userbyid(n.nspowner) AS "Owner"
  FROM pg_catalog.pg_namespace n
  WHERE n.nspname !~ '^pg_' AND n.nspname <> 'information_schema'
  ORDER BY 1;

       List of schemas
     Name     |     Owner
--------------+---------------
 public       | postgres
 tiger        | gis_admin
 tiger_data   | gis_admin
 topology     | gis_admin
(4 rows)
```

Step 4: Transfer ownership of the PostGIS objects

Use the following function to transfer ownership of the PostGIS objects to the `gis_admin` role. Run the following statement from the psql prompt to create the function.

```
CREATE FUNCTION exec(text) returns text language plpgsql volatile AS $f$ BEGIN EXECUTE
 $1; RETURN $1; END; $f$;
CREATE FUNCTION
```

Next, run the following query to run the `exec` function that in turn runs the statements and alters the permissions.

```
SELECT exec('ALTER TABLE ' || quote_ident(s.nspname) || '.' || quote_ident(s.relname)
 || ' OWNER TO gis_admin;')
  FROM (
    SELECT nspname, relname
    FROM pg_class c JOIN pg_namespace n ON (c.relnamespace = n.oid)
    WHERE nspname in ('tiger','topology') AND
    relkind IN ('r','S','v') ORDER BY relkind = 'S')
s;
```

Step 5: Test the extensions

To avoid needing to specify the schema name, add the `tiger` schema to your search path using the following command.

```
SET search_path=public,tiger;
SET
```

Test the `tiger` schema by using the following SELECT statement.

```
SELECT address, streetname, streettypeabbrev, zip
 FROM normalize_address('1 Devonshire Place, Boston, MA 02109') AS na;
address | streetname | streettypeabbrev |  zip
---------+------------+------------------+-------
       1 | Devonshire | Pl               | 02109
(1 row)
```

To learn more about this extension, see Tiger Geocoder in the PostGIS documentation.

Test access to the `topology` schema by using the following `SELECT` statement. This calls the `createtopology` function to register a new topology object (my_new_topo) with the specified spatial reference identifier (26986) and default tolerance (0.5). To learn more, see CreateTopology in the PostGIS documentation.

```
SELECT topology.createtopology('my_new_topo',26986,0.5);
 createtopology
----------------
              1
(1 row)
```

Step 6: Upgrade the PostGIS extension

Each new release of PostgreSQL supports one or more versions of the PostGIS extension compatible with that release. Upgrading the PostgreSQL engine to a new version doesn't automatically upgrade the PostGIS extension. Before upgrading the PostgreSQL engine, you typically upgrade PostGIS to the newest available version for the current PostgreSQL version. For details, see PostGIS extension versions.

After the PostgreSQL engine upgrade, you then upgrade the PostGIS extension again, to the version supported for the newly upgraded PostgreSQL engine version. For more information about upgrading the PostgreSQL engine, see Testing an upgrade of your production DB cluster to a new major version.

You can check for available PostGIS extension version updates on your Aurora PostgreSQL DB cluster at any time. To do so, run the following command. This function is available with PostGIS 2.5.0 and higher versions.

```
SELECT postGIS_extensions_upgrade();
```

If your application doesn't support the latest PostGIS version, you can install an older version of PostGIS that's available in your major version as follows.

```
CREATE EXTENSION postgis VERSION "2.5.5";
```

If you want to upgrade to a specific PostGIS version from an older version, you can also use the following command.

```
ALTER EXTENSION postgis UPDATE TO "2.5.5";
```

Depending on the version that you're upgrading from, you might need to use this function again. The result of the first run of the function determines if an additional upgrade function is needed. For example, this is the case for upgrading from PostGIS 2 to PostGIS 3. For more information, see Upgrading PostGIS 2 to PostGIS 3.

If you upgraded this extension to prepare for a major version upgrade of the PostgreSQL engine, you can continue with other preliminary tasks. For more information, see Testing an upgrade of your production DB cluster to a new major version.

PostGIS extension versions

We recommend that you install the versions of all extensions such as PostGIS as listed in Extension versions for Aurora PostgreSQL-Compatible Edition in the *Release Notes for Aurora PostgreSQL*. To get a list of versions that are available in your release, use the following command.

```
SELECT * FROM pg_available_extension_versions WHERE name='postgis';
```

You can find version information in the following sections in the *Release Notes for Aurora PostgreSQL*:

- Extension versions for Aurora PostgreSQL 14
- Extension versions for Aurora PostgreSQL-Compatible Edition 13
- Extension versions for Aurora PostgreSQL-Compatible Edition 12

- Extension versions for Aurora PostgreSQL-Compatible Edition 11
- Extension versions for Aurora PostgreSQL-Compatible Edition 10
- Extension versions for Aurora PostgreSQL-Compatible Edition 9.6

Upgrading PostGIS 2 to PostGIS 3

Starting with version 3.0, the PostGIS raster functionality is now a separate extension, `postgis_raster`. This extension has its own installation and upgrade path. This removes dozens of functions, data types, and other artifacts required for raster image processing from the core `postgis` extension. That means that if your use case doesn't require raster processing, you don't need to install the `postgis_raster` extension.

In the following upgrade example, the first upgrade command extracts raster functionality into the `postgis_raster` extension. A second upgrade command is then required to upgrade `postgres_raster` to the new version.

To upgrade from PostGIS 2 to PostGIS 3

1. Identify the default version of PostGIS that's available to the PostgreSQL version on your Aurora PostgreSQL DB cluster. To do so, run the following query.

```
SELECT * FROM pg_available_extensions
    WHERE default_version > installed_version;
  name    | default_version | installed_version |                          comment
---------+-----------------+-------------------
+---------------------------------------------------------------
 postgis | 3.1.4           | 2.3.7             | PostGIS geometry and geography
 spatial types and functions
(1 row)
```

2. Identify the versions of PostGIS installed in each database on the writer instance of your Aurora PostgreSQL DB cluster. In other words, query each user database as follows.

```
SELECT
    e.extname AS "Name",
    e.extversion AS "Version",
    n.nspname AS "Schema",
    c.description AS "Description"
FROM
    pg_catalog.pg_extension e
    LEFT JOIN pg_catalog.pg_namespace n ON n.oid = e.extnamespace
    LEFT JOIN pg_catalog.pg_description c ON c.objoid = e.oid
    AND c.classoid = 'pg_catalog.pg_extension'::pg_catalog.regclass
WHERE
    e.extname LIKE '%postgis%'
ORDER BY
    1;
  Name   | Version | Schema |                             Description
---------+---------+--------
+-------------------------------------------------------------------
 postgis | 2.3.7   | public | PostGIS geometry, geography, and raster spatial types
 and functions
(1 row)
```

This mismatch between the default version (PostGIS 3.1.4) and the installed version (PostGIS 2.3.7) means that you need to upgrade the PostGIS extension.

```
ALTER EXTENSION postgis UPDATE;
ALTER EXTENSION
WARNING: unpackaging raster
WARNING: PostGIS Raster functionality has been unpackaged
```

3. Run the following query to verify that the raster functionality is now in its own package.

```
SELECT
    probin,
    count(*)
FROM
    pg_proc
WHERE
    probin LIKE '%postgis%'
GROUP BY
    probin;
          probin          | count
--------------------------+-------
 $libdir/rtpostgis-2.3    | 107
 $libdir/postgis-3        | 487
(2 rows)
```

The output shows that there's still a difference between versions. The PostGIS functions are version 3 (postgis-3), while the raster functions (rtpostgis) are version 2 (rtpostgis-2.3). To complete the upgrade, you run the upgrade command again, as follows.

```
postgres=> SELECT postgis_extensions_upgrade();
```

You can safely ignore the warning messages. Run the following query again to verify that the upgrade is complete. The upgrade is complete when PostGIS and all related extensions aren't marked as needing upgrade.

```
SELECT postgis_full_version();
```

4. Use the following query to see the completed upgrade process and the separately packaged extensions, and verify that their versions match.

```
SELECT
    e.extname AS "Name",
    e.extversion AS "Version",
    n.nspname AS "Schema",
    c.description AS "Description"
FROM
    pg_catalog.pg_extension e
    LEFT JOIN pg_catalog.pg_namespace n ON n.oid = e.extnamespace
    LEFT JOIN pg_catalog.pg_description c ON c.objoid = e.oid
        AND c.classoid = 'pg_catalog.pg_extension'::pg_catalog.regclass
WHERE
    e.extname LIKE '%postgis%'
ORDER BY
    1;
      Name      | Version | Schema |                             Description
----------------+---------+--------
+---------------------------------------------------------------------
 postgis        | 3.1.5   | public | PostGIS geometry, geography, and raster
 spatial types and functions
 postgis_raster | 3.1.5   | public | PostGIS raster types and functions
(2 rows)
```

The output shows that the PostGIS 2 extension was upgraded to PostGIS 3, and both `postgis` and the now separate `postgis_raster` extension are version 3.1.5.

After this upgrade completes, if you don't plan to use the raster functionality, you can drop the extension as follows.

```
DROP EXTENSION postgis_raster;
```

Managing PostgreSQL partitions with the pg_partman extension

PostgreSQL table partitioning provides a framework for high-performance handling of data input and reporting. Use partitioning for databases that require very fast input of large amounts of data. Partitioning also provides for faster queries of large tables. Partitioning helps maintain data without impacting the database instance because it requires less I/O resources.

By using partitioning, you can split data into custom-sized chunks for processing. For example, you can partition time-series data for ranges such as hourly, daily, weekly, monthly, quarterly, yearly, custom, or any combination of these. For a time-series data example, if you partition the table by hour, each partition contains one hour of data. If you partition the time-series table by day, the partitions holds one day's worth of data, and so on. The partition key controls the size of a partition.

When you use an `INSERT` or `UPDATE` SQL command on a partitioned table, the database engine routes the data to the appropriate partition. PostgreSQL table partitions that store the data are child tables of the main table.

During database query reads, the PostgreSQL optimizer examines the `WHERE` clause of the query and, if possible, directs the database scan to only the relevant partitions.

Starting with version 10, PostgreSQL uses declarative partitioning to implement table partitioning. This is also known as native PostgreSQL partitioning. Before PostgreSQL version 10, you used triggers to implement partitions.

PostgreSQL table partitioning provides the following features:

- Creation of new partitions at any time.
- Variable partition ranges.
- Detachable and reattachable partitions using data definition language (DDL) statements.

  For example, detachable partitions are useful for removing historical data from the main partition but keeping historical data for analysis.
- New partitions inherit the parent database table properties, including the following:
  - Indexes

  - Primary keys, which must include the partition key column
  - Foreign keys
  - Check constraints
  - References
- Creating indexes for the full table or each specific partition.

You can't alter the schema for an individual partition. However, you can alter the parent table (such as adding a new column), which propagates to partitions.

Topics

Overview of the PostgreSQL pg_partman extension

You can use the PostgreSQL `pg_partman` extension to automate the creation and maintenance of table partitions. For more general information, see PG Partition Manager in the `pg_partman` documentation.

> **Note**
>
> The `pg_partman` extension is supported on Aurora PostgreSQL versions 12.6 and higher.

Instead of having to manually create each partition, you configure `pg_partman` with the following settings:

- Table to be partitioned
- Partition type
- Partition key
- Partition granularity
- Partition precreation and management options

After you create a PostgreSQL partitioned table, you register it with `pg_partman` by calling the `create_parent` function. Doing this creates the necessary partitions based on the parameters you pass to the function.

The `pg_partman` extension also provides the `run_maintenance_proc` function, which you can call on a scheduled basis to automatically manage partitions. To ensure that the proper partitions are created as needed, schedule this function to run periodically (such as hourly). You can also ensure that partitions are automatically dropped.

Enabling the pg_partman extension

If you have multiple databases inside the same PostgreSQL DB instance for which you want to manage partitions, enable the `pg_partman` extension separately for each database. To enable the `pg_partman` extension for a specific database, create the partition maintenance schema and then create the `pg_partman` extension as follows.

```
CREATE SCHEMA partman;
CREATE EXTENSION pg_partman WITH SCHEMA partman;
```

> **Note**
>
> To create the `pg_partman` extension, make sure that you have `rds_superuser` privileges.

If you receive an error such as the following, grant the `rds_superuser` privileges to the account or use your superuser account.

```
ERROR: permission denied to create extension "pg_partman"
HINT: Must be superuser to create this extension.
```

To grant `rds_superuser` privileges, connect with your superuser account and run the following command.

```
GRANT rds_superuser TO user-or-role;
```

For the examples that show using the pg_partman extension, we use the following sample database table and partition. This database uses a partitioned table based on a timestamp. A schema data_mart contains a table named events with a column named created_at. The following settings are included in the events table:

- Primary keys event_id and created_at, which must have the column used to guide the partition.
- A check constraint ck_valid_operation to enforce values for an operation table column.
- Two foreign keys, where one (fk_orga_membership) points to the external table organization and the other (fk_parent_event_id) is a self-referenced foreign key.
- Two indexes, where one (idx_org_id) is for the foreign key and the other (idx_event_type) is for the event type.

The following DDL statements create these objects, which are automatically included on each partition.

```
CREATE SCHEMA data_mart;
CREATE TABLE data_mart.organization ( org_id BIGSERIAL,
        org_name TEXT,
        CONSTRAINT pk_organization PRIMARY KEY (org_id)
    );

CREATE TABLE data_mart.events(
        event_id        BIGSERIAL,
        operation       CHAR(1),
        value           FLOAT(24),
        parent_event_id BIGINT,
        event_type      VARCHAR(25),
        org_id          BIGSERIAL,
        created_at      timestamp,
        CONSTRAINT pk_data_mart_event PRIMARY KEY (event_id, created_at),
        CONSTRAINT ck_valid_operation CHECK (operation = 'C' OR operation = 'D'),
        CONSTRAINT fk_orga_membership
            FOREIGN KEY(org_id)
            REFERENCES data_mart.organization (org_id),
        CONSTRAINT fk_parent_event_id
            FOREIGN KEY(parent_event_id, created_at)
            REFERENCES data_mart.events (event_id,created_at)
    ) PARTITION BY RANGE (created_at);

CREATE INDEX idx_org_id     ON  data_mart.events(org_id);
CREATE INDEX idx_event_type ON  data_mart.events(event_type);
```

Configuring partitions using the create_parent function

After you enable the `pg_partman` extension, use the `create_parent` function to configure partitions inside the partition maintenance schema. The following example uses the `events` table example created in Enabling the pg_partman extension. Call the `create_parent` function as follows.

```
SELECT partman.create_parent( p_parent_table => 'data_mart.events',
 p_control => 'created_at',
 p_type => 'native',
 p_interval=> 'daily',
 p_premake => 30);
```

The parameters are as follows:

- `p_parent_table` – The parent partitioned table. This table must already exist and be fully qualified, including the schema.
- `p_control` – The column on which the partitioning is to be based. The data type must be an integer or time-based.
- `p_type` – The type is either `'native'` or `'partman'`. You typically use the `native` type for its performance improvements and flexibility. The `partman` type relies on inheritance.
- `p_interval` – The time interval or integer range for each partition. Example values include `daily`, hourly, and so on.
- `p_premake` – The number of partitions to create in advance to support new inserts.

For a complete description of the `create_parent` function, see Creation Functions in the `pg_partman` documentation.

Configuring partition maintenance using the run_maintenance_proc function

You can run partition maintenance operations to automatically create new partitions, detach partitions, or remove old partitions. Partition maintenance relies on the `run_maintenance_proc` function of the `pg_partman` extension and the `pg_cron` extension, which initiates an internal scheduler. The `pg_cron` scheduler automatically executes SQL statements, functions, and procedures defined in your databases.

The following example uses the `events` table example created in Enabling the pg_partman extension to set partition maintenance operations to run automatically. As a prerequisite, add `pg_cron` to the `shared_preload_libraries` parameter in the DB instance's parameter group.

```
CREATE EXTENSION pg_cron;

UPDATE partman.part_config
SET infinite_time_partitions = true,
    retention = '3 months',
    retention_keep_table=true
WHERE parent_table = 'data_mart.events';
SELECT cron.schedule('@hourly', $$CALL partman.run_maintenance_proc()$$);
```

Following, you can find a step-by-step explanation of the preceding example:

1. Modify the parameter group associated with your DB instance and add `pg_cron` to the `shared_preload_libraries` parameter value. This change requires a DB instance restart for it to take effect. For more information, see Modifying parameters in a DB parameter group.
2. Run the command `CREATE EXTENSION pg_cron;` using an account that has the `rds_superuser` permissions. Doing this enables the `pg_cron` extension. For more information, see Scheduling maintenance with the PostgreSQL pg_cron extension.
3. Run the command `UPDATE partman.part_config` to adjust the `pg_partman` settings for the `data_mart.events` table.
4. Run the command `SET ...` to configure the `data_mart.events` table, with these clauses:
    a. `infinite_time_partitions = true,` – Configures the table to be able to automatically create new partitions without any limit.
    b. `retention = '3 months',` – Configures the table to have a maximum retention of three months.
    c. `retention_keep_table=true` – Configures the table so that when the retention period is due, the table isn't deleted automatically. Instead, partitions that are older than the retention period are only detached from the parent table.
5. Run the command `SELECT cron.schedule ...` to make a `pg_cron` function call. This call defines how often the scheduler runs the `pg_partman` maintenance procedure, `partman.run_maintenance_proc`. For this example, the procedure runs every hour.

For a complete description of the `run_maintenance_proc` function, see Maintenance Functions in the `pg_partman` documentation.

Scheduling maintenance with the PostgreSQL pg_cron extension

You can use the PostgreSQL `pg_cron` extension to schedule maintenance commands within a PostgreSQL database. For more information about the extension, see What is pg_cron? in the pg_cron documentation.

The `pg_cron` extension is supported on Aurora PostgreSQL engine versions 12.6 and higher version

To learn more about using `pg_cron`, see Schedule jobs with pg_cron on your RDS for PostgreSQL or your Aurora PostgreSQL-Compatible Edition databases.

Topics

- Setting up the pg_cron extension
- Granting database users permissions to use pg_cron
- Scheduling pg_cron jobs
- Reference for the pg_cron extension

Setting up the pg_cron extension

Set up the `pg_cron` extension as follows:

1. Modify the custom parameter group associated with your PostgreSQL DB instance by adding `pg_cron` to the `shared_preload_libraries` parameter value.

   Restart the PostgreSQL DB instance to have changes to the parameter group take effect. To learn more about working with parameter groups, see Amazon Aurora PostgreSQL parameters.
2. After the PostgreSQL DB instance has restarted, run the following command using an account that has `rds_superuser` permissions. For example, if you used the default settings when you created your Aurora PostgreSQL DB cluster, connect as user `postgres` and create the extension.

   ```
   CREATE EXTENSION pg_cron;
   ```

The pg_cron scheduler is set in the default PostgreSQL database named postgres. The pg_cron objects are created in this postgres database and all scheduling actions run in this database.

3. You can use the default settings, or you can schedule jobs to run in other databases within your PostgreSQL DB instance. To schedule jobs for other databases within your PostgreSQL DB instance, see the example in Scheduling a cron job for a database other than the default database.

Granting database users permissions to use pg_cron

Installing the pg_cron extension requires the rds_superuser privileges. However, permissions to use the pg_cron can be granted (by a member of the rds_superuser group/role) to other database users, so that they can schedule their own jobs. We recommend that you grant permissions to the cron schema only as needed if it improves operations in your production environment.

To grant a database user permission in the cron schema, run the following command:

```
postgres=> GRANT USAGE ON SCHEMA cron TO db-user;
```

This gives *db-user* permission to access the cron schema to schedule cron jobs for the objects that they have permissions to access. If the database user doesn't have permissions, the job fails after posting the error message to the postgresql.log file, as shown in the following:

```
2020-12-08 16:41:00 UTC::@:[30647]:ERROR: permission denied for table table-name
2020-12-08 16:41:00 UTC::@:[27071]:LOG: background worker "pg_cron" (PID 30647) exited
 with exit code 1
```

In other words, make sure that database users that are granted permissions on the cron schema also have permissions on the objects (tables, schemas, and so on) that they plan to schedule.

The details of the cron job and its success or failure are also captured in the cron.job_run_details table. For more information, see Tables for scheduling jobs and capturing status .

Scheduling pg_cron jobs

The following sections show how you can schedule various management tasks using pg_cron jobs.

Note

When you create pg_cron jobs, check that the max_worker_processes setting is larger than the number of cron.max_running_jobs. A pg_cron job fails if it runs out of background worker processes. The default number of pg_cron jobs is 5. For more information, see Parameters for managing the pg_cron extension.

Topics

- Vacuuming a table
- Purging the pg_cron history table
- Logging errors to the postgresql.log file only
- Scheduling a cron job for a database other than the default database

Vacuuming a table

Autovacuum handles vacuum maintenance for most cases. However, you might want to schedule a vacuum of a specific table at a time of your choosing.

Following is an example of using the cron.schedule function to set up a job to use VACUUM FREEZE on a specific table every day at 22:00 (GMT).

```
SELECT cron.schedule('manual vacuum', '0 22 * * *', 'VACUUM FREEZE pgbench_accounts');
 schedule
----------
1
(1 row)
```

After the preceding example runs, you can check the history in the cron.job_run_details table as follows.

```
postgres=> SELECT * FROM cron.job_run_details;
jobid  | runid | job_pid | database | username | command                         |
 status    | return_message | start_time                    | end_time
-------+-------+---------+----------+----------+---------------------------------
+-----------+----------------+-------------------------------
+-------------------------------
```

```
 1     | 1     | 3395    | postgres | adminuser| vacuum freeze pgbench_accounts
 | succeeded | VACUUM         | 2020-12-04 21:10:00.050386+00 | 2020-12-04
 21:10:00.072028+00
(1 row)
```

Following is a query of the `cron.job_run_details` table to see the failed jobs.

```
postgres=> SELECT * FROM cron.job_run_details WHERE status = 'failed';
jobid | runid | job_pid | database | username | command                      | status
 | return_message                                   | start_time                    |
 end_time
------+-------+---------+----------+----------+------------------------------+--------
+---------------------------------------------------+-------------------------------
+------------------------------
 5    | 4     | 30339   | postgres | adminuser| vacuum freeze pgbench_account | failed
 | ERROR: relation "pgbench_account" does not exist | 2020-12-04 21:48:00.015145+00 |
 2020-12-04 21:48:00.029567+00
(1 row)
```

For more information, see Tables for scheduling jobs and capturing status .

Purging the pg_cron history table

The `cron.job_run_details` table contains a history of cron jobs that can become very large over time. We recommend that you schedule a job that purges this table. For example, keeping a week's worth of entries might be sufficient for troubleshooting purposes.

The following example uses the cron.schedule function to schedule a job that runs every day at midnight to purge the `cron.job_run_details` table. The job keeps only the last seven days. Use your `rds_superuser` account to schedule the job such as the following.

```
SELECT cron.schedule('0 0 * * *', $$DELETE
    FROM cron.job_run_details
    WHERE end_time < now() - interval '7 days'$$);
```

For more information, see Tables for scheduling jobs and capturing status .

Logging errors to the postgresql.log file only

To prevent writing to the `cron.job_run_details` table, modify the parameter group associated with the PostgreSQL DB instance and set the `cron.log_run` parameter to off. The `pg_cron` extension no longer writes to the table and captures errors to the `postgresql.log` file only. For more information, see [Modifying parameters in a DB parameter group]().

Use the following command to check the value of the `cron.log_run` parameter.

```
postgres=> SHOW cron.log_run;
```

For more information, see [Parameters for managing the pg_cron extension]().

Scheduling a cron job for a database other than the default database

The metadata for `pg_cron` is all held in the PostgreSQL default database named `postgres`. Because background workers are used for running the maintenance cron jobs, you can schedule a job in any of your databases within the PostgreSQL DB instance:

1. In the cron database, schedule the job as you normally do using the [cron.schedule]().

   ```
   postgres=> SELECT cron.schedule('database1 manual vacuum', '29 03 * * *', 'vacuum
    freeze test_table');
   ```

2. As a user with the `rds_superuser` role, update the database column for the job that you just created so that it runs in another database within your PostgreSQL DB instance.

   ```
   postgres=> UPDATE cron.job SET database = 'database1' WHERE jobid = 106;
   ```

3. Verify by querying the `cron.job` table.

   ```
   postgres=> SELECT * FROM cron.job;
   jobid | schedule    | command                        | nodename  | nodeport |
    database | username  | active | jobname
   ------+-------------+--------------------------------+-----------+----------
   +----------+-----------+--------+-------------------------
   106   | 29 03 * * * | vacuum freeze test_table       | localhost | 8192     |
    database1| adminuser | t      | database1 manual vacuum
     1   | 59 23 * * * | vacuum freeze pgbench_accounts | localhost | 8192     |
    postgres | adminuser | t      | manual vacuum
   (2 rows)
   ```

Note

In some situations, you might add a cron job that you intend to run on a different database. In such cases, the job might try to run in the default database (`postgres`) before you update the correct database column. If the user name has permissions, the job successfully runs in the default database.

Reference for the pg_cron extension

You can use the following parameters, functions, and tables with the `pg_cron` extension. For more information, see What is pg_cron? in the pg_cron documentation.

Topics

Parameters for managing the pg_cron extension

Following is a list of parameters that control the `pg_cron` extension behavior.

Parameter	Description
cron.database_name	The database in which `pg_cron` metadata is kept.
`cron.host`	The hostname to connect to PostgreSQL. You can't modify this value.
`cron.log_run`	Log every job that runs in the `job_run_details` table. Values are `on` or `off`. For more information, see Tables for scheduling jobs and capturing status .

Parameter	Description
`cron.log_statement`	Log all cron statements before running them. Values are on or `off`.
`cron.max_running_jobs`	The maximum number of jobs that can run concurrently.
`cron.use_background_workers`	Use background workers instead of client sessions. You can't modify this value.

Use the following SQL command to display these parameters and their values.

```
postgres=> SELECT name, setting, short_desc FROM pg_settings WHERE name LIKE 'cron.%'
 ORDER BY name;
```

Function reference: cron.schedule

This function schedules a cron job. The job is initially scheduled in the default `postgres` database. The function returns a `bigint` value representing the job identifier. To schedule jobs to run in other databases within your PostgreSQL DB instance, see the example in Scheduling a cron job for a database other than the default database.

The function has two syntax formats.

Syntax

```
cron.schedule (job_name,
    schedule,
    command
);

cron.schedule (schedule,
    command
);
```

Parameters

Parameter	Description
`job_name`	The name of the cron job.
`schedule`	Text indicating the schedule for the cron job. The format is the standard cron format.
`command`	Text of the command to run.

Examples

```
postgres=> SELECT cron.schedule ('test','0 10 * * *', 'VACUUM pgbench_history');
 schedule
----------
      145
(1 row)

postgres=> SELECT cron.schedule ('0 15 * * *', 'VACUUM pgbench_accounts');
 schedule
----------
      146
(1 row)
```

Function reference: cron.unschedule

This function deletes a cron job. You can specify either the `job_name` or the `job_id`. A policy makes sure that you are the owner to remove the schedule for the job. The function returns a Boolean indicating success or failure.

The function has the following syntax formats.

Syntax

```
cron.unschedule (job_id);

cron.unschedule (job_name);
```

Parameters

Parameter	Description
`job_id`	A job identifier that was returned from the `cron.schedule` function when the cron job was scheduled.
`job_name`	The name of a cron job that was scheduled with the `cron.schedule` function.

Examples

```
postgres=> SELECT cron.unschedule(108);
 unschedule
------------
 t
(1 row)

postgres=> SELECT cron.unschedule('test');
 unschedule
------------
 t
(1 row)
```

Tables for scheduling jobs and capturing status

The following tables are used to schedule the cron jobs and record how the jobs completed.

Table	Description
`cron.job`	Contains the metadata about each scheduled job. Most interactions with this table should be done by using the `cron.schedule` and `cron.unschedule` functions.

Table	Description
	Important We recommend that you don't give update or insert privileges directly to this table. Doing so would allow the user to update the `username` column to run as `rds-superuser` .
`cron.job_run_details`	Contains historic information about past scheduled jobs that ran. This is useful to investigate the status, return messages, and start and end time from the job that ran. **Note** To prevent this table from growing indefinitely, purge it on a regular basis. For an example, see Purging the pg_cron history table.

Using pgAudit to log database activity

Financial institutions, government agencies, and many industries need to keep *audit logs* to meet regulatory requirements. By using the PostgreSQL Audit extension (pgAudit) with your Aurora PostgreSQL DB cluster, you can capture the detailed records that are typically needed by auditors or to meet regulatory requirements. For example, you can set up the pgAudit extension to track changes made to specific databases and tables, to record the user who made the change, and many other details.

The pgAudit extension builds on the functionality of the native PostgreSQL logging infrastructure by extending the log messages with more detail. In other words, you use the same approach to view your audit log as you do to view any log messages. For more information about PostgreSQL logging, see Aurora PostgreSQL database log files.

The pgAudit extension redacts sensitive data such as cleartext passwords from the logs. If your Aurora PostgreSQL DB cluster is configured to log data manipulation language (DML) statements as detailed in Turning on query logging for your Aurora PostgreSQL DB cluster, you can avoid the cleartext password issue by using the PostgreSQL Audit extension.

You can configure auditing on your database instances with a great degree of specificity. You can audit all databases and all users. Or, you can choose to audit only certain databases, users, and other objects. You can also explicitly exclude certain users and databases from being audited. For more information, see Excluding users or databases from audit logging.

Given the amount of detail that can be captured, we recommend that if you do use pgAudit, you monitor your storage consumption.

The pgAudit extension is supported on all available Aurora PostgreSQL versions. For a list of pgAudit versions supported by Aurora PostgreSQL version, see Extension versions for Amazon Aurora PostgreSQL in the *Release Notes for Aurora PostgreSQL*.

Topics

Setting up the pgAudit extension

To set up the pgAudit extension on your Aurora PostgreSQL DB cluster, you first add pgAudit to the shared libraries on the custom DB cluster parameter group for your Aurora PostgreSQL DB cluster. For information about creating a custom DB cluster parameter group, see Working with parameter groups. Next, you install the pgAudit extension. Finally, you specify the databases and objects that you want to audit. The procedures in this section show you how. You can use the AWS Management Console or the AWS CLI.

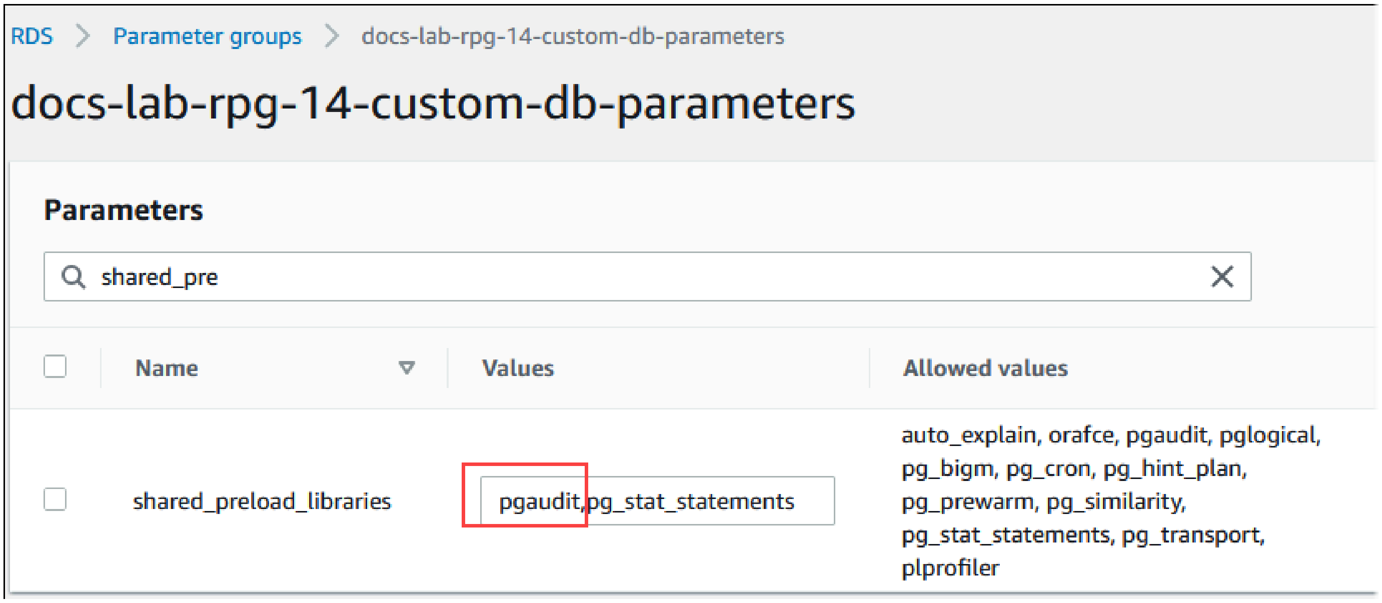
You must have permissions as the `rds_superuser` role to perform all these tasks.

The steps following assume that your Aurora PostgreSQL DB cluster is associated with a custom DB cluster parameter group.

Console

To set up the pgAudit extension

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose your Aurora PostgreSQL DB cluster's Writer instance .
3. Open the **Configuration** tab for your Aurora PostgreSQL DB cluster writer instance. Among the Instance details, find the **Parameter group** link.
4. Choose the link to open the custom parameters associated with your Aurora PostgreSQL DB cluster.
5. In the **Parameters** search field, type `shared_pre` to find the `shared_preload_libraries` parameter.
6. Choose **Edit parameters** to access the property values.
7. Add `pgaudit` to the list in the **Values** field. Use a comma to separate items in the list of values.



RDS > Parameter groups > docs-lab-rpg-14-custom-db-parameters

docs-lab-rpg-14-custom-db-parameters

Parameters

shared_pre

□	Name	Values	Allowed values
□	shared_preload_libraries	pgaudit,pg_stat_statements	auto_explain, orafce, pgaudit, pglogical, pg_bigm, pg_cron, pg_hint_plan, pg_prewarm, pg_similarity, pg_stat_statements, pg_transport, plprofiler

8. Reboot the writer instance of your Aurora PostgreSQL DB cluster so that your change to the `shared_preload_libraries` parameter takes effect.
9. When the instance is available, verify that pgAudit has been initialized. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

```
SHOW shared_preload_libraries;
```

```
shared_preload_libraries
--------------------------
rdsutils,pgaudit
(1 row)
```

10. With pgAudit initialized, you can now create the extension. You need to create the extension after initializing the library because the `pgaudit` extension installs event triggers for auditing data definition language (DDL) statements.

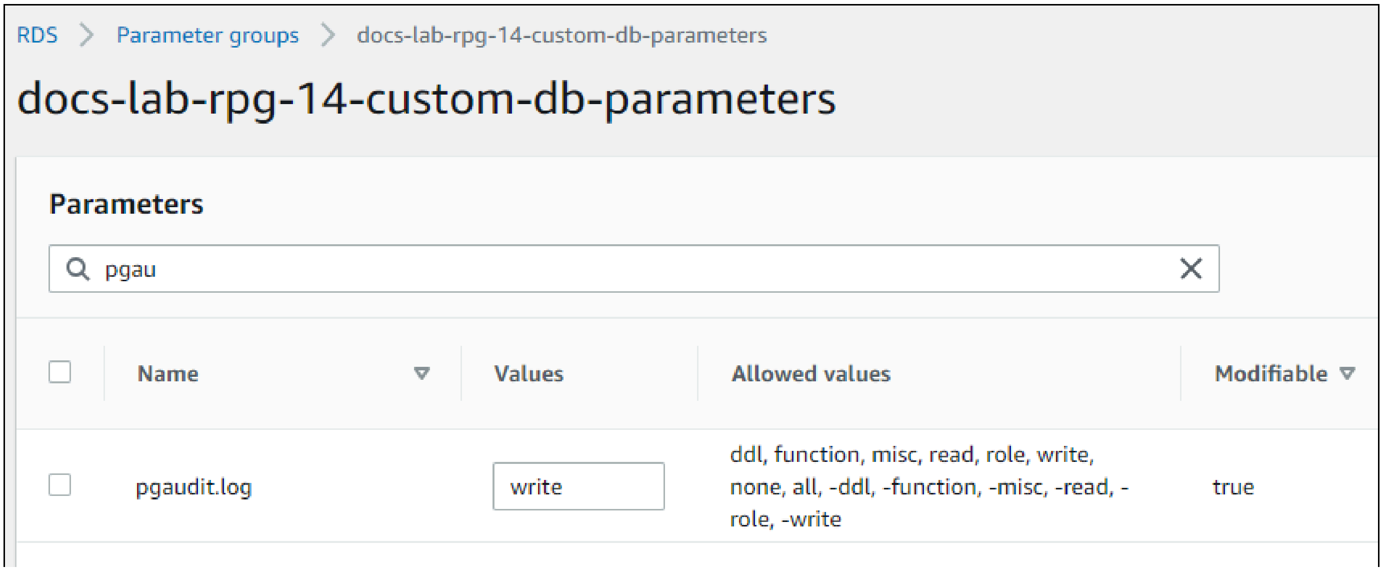
```
CREATE EXTENSION pgaudit;
```

11. Close the `psql` session.

```
labdb=> \q
```

12. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

13. Find the `pgaudit.log` parameter in the list and set to the appropriate value for your use case. For example, setting the `pgaudit.log` parameter to `write` as shown in the following image captures inserts, updates, deletes, and some other types changes to the log.



RDS > Parameter groups > docs-lab-rpg-14-custom-db-parameters

docs-lab-rpg-14-custom-db-parameters

Parameters

pgau

□	Name	Values	Allowed values	Modifiable
□	pgaudit.log	write	ddl, function, misc, read, role, write, none, all, -ddl, -function, -misc, -read, -role, -write	true

You can also choose one of the following values for the `pgaudit.log` parameter.

- none – This is the default. No database changes are logged.
- all – Logs everything (read, write, function, role, ddl, misc).

- ddl – Logs all data definition language (DDL) statements that aren't included in the ROLE class.
- function – Logs function calls and DO blocks.
- misc – Logs miscellaneous commands, such as DISCARD, FETCH, CHECKPOINT, VACUUM, and SET.
- read – Logs SELECT and COPY when the source is a relation (such as a table) or a query.
- role – Logs statements related to roles and privileges, such as GRANT, REVOKE, CREATE ROLE, ALTER ROLE, and DROP ROLE.
- write – Logs INSERT, UPDATE, DELETE, TRUNCATE, and COPY when the destination is a relation (table).

14. Choose **Save changes**.
15. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
16. Choose your Aurora PostgreSQL DB cluster's writer instance from the Databases list to select it, and then choose **Reboot** from the Actions menu.

AWS CLI

To setup pgAudit

To setup pgAudit using the AWS CLI, you call the modify-db-parameter-group operation to modify the audit log parameters in your custom parameter group, as shown in the following procedure.

1. Use the following AWS CLI command to add pgaudit to the shared_preload_libraries parameter.

```
aws rds modify-db-parameter-group \
   --db-parameter-group-name custom-param-group-name \
   --parameters
 "ParameterName=shared_preload_libraries,ParameterValue=pgaudit,ApplyMethod=pending-
reboot" \
   --region aws-region
```

2. Use the following AWS CLI command to reboot the writer instance of your Aurora PostgreSQL DB cluster so that the pgaudit library is initialized.

```
aws rds reboot-db-instance \
    --db-instance-identifier writer-instance \
    --region aws-region
```

3. When the instance is available, you can verify that pgaudit has been initialized. Use psql to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

   ```
   SHOW shared_preload_libraries;
   shared_preload_libraries
   --------------------------
   rdsutils,pgaudit
   (1 row)
   ```

   With pgAudit initialized, you can now create the extension.

   ```
   CREATE EXTENSION pgaudit;
   ```

4. Close the psql session so that you can use the AWS CLI.

   ```
   labdb=> \q
   ```

5. Use the following AWS CLI command to specify the classes of statement that want logged by session audit logging. The example sets the pgaudit.log parameter to write, which captures inserts, updates, and deletes to the log.

   ```
   aws rds modify-db-parameter-group \
      --db-parameter-group-name custom-param-group-name \
      --parameters
    "ParameterName=pgaudit.log,ParameterValue=write,ApplyMethod=pending-reboot" \
      --region aws-region
   ```

   You can also choose one of the following values for the pgaudit.log parameter.

   - none – This is the default. No database changes are logged.
   - all – Logs everything (read, write, function, role, ddl, misc).
   - ddl – Logs all data definition language (DDL) statements that aren't included in the ROLE class.
   - function – Logs function calls and DO blocks.
   - misc – Logs miscellaneous commands, such as DISCARD, FETCH, CHECKPOINT, VACUUM, and SET.

- read – Logs `SELECT` and `COPY` when the source is a relation (such as a table) or a query.
- role – Logs statements related to roles and privileges, such as `GRANT`, `REVOKE`, `CREATE ROLE`, `ALTER ROLE`, and `DROP ROLE`.
- write – Logs `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY` when the destination is a relation (table).

Reboot the writer instance of your Aurora PostgreSQL DB cluster using the following AWS CLI command.

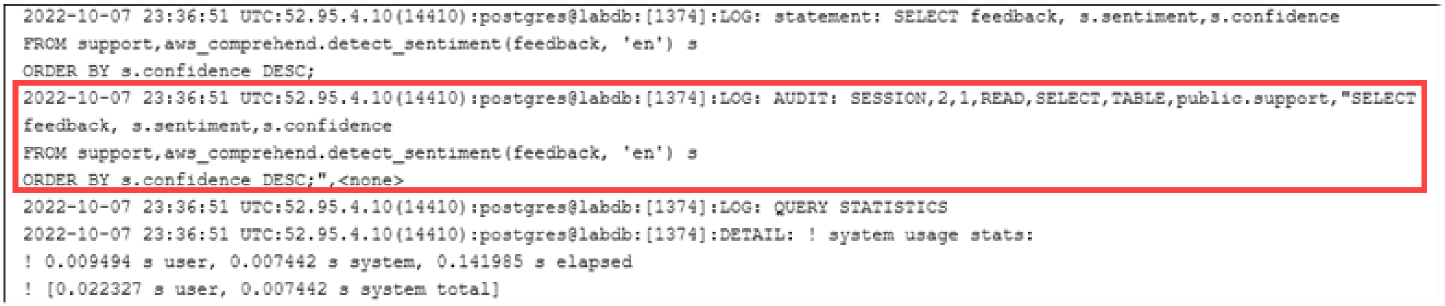
```
aws rds reboot-db-instance \
    --db-instance-identifier writer-instance \
    --region aws-region
```

Auditing database objects

With pgAudit set up on your Aurora PostgreSQL DB cluster and configured for your requirements, more detailed information is captured in the PostgreSQL log. For example, while the default PostgreSQL logging configuration identifies the date and time that a change was made in a database table, with the pgAudit extension the log entry can include the schema, user who made the change, and other details depending on how the extension parameters are configured. You can set up auditing to track changes in the following ways.

- For each session, by user. For the session level, you can capture the fully qualified command text.
- For each object, by user and by database.

The object auditing capability is activated when you create the `rds_pgaudit` role on your system and then add this role to the `pgaudit.role` parameter in your custom parameter parameter group. By default, the `pgaudit.role` parameter is unset and the only allowable value is `rds_pgaudit`. The following steps assume that `pgaudit` has been initialized and that you have created the `pgaudit` extension by following the procedure in Setting up the pgAudit extension.



```
2022-10-07 23:36:51 UTC:52.95.4.10(14410):postgres@labdb:[1374]:LOG: statement: SELECT feedback, s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;
2022-10-07 23:36:51 UTC:52.95.4.10(14410):postgres@labdb:[1374]:LOG: AUDIT: SESSION,2,1,READ,SELECT,TABLE,public.support,"SELECT
feedback, s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;",<none>
2022-10-07 23:36:51 UTC:52.95.4.10(14410):postgres@labdb:[1374]:LOG: QUERY STATISTICS
2022-10-07 23:36:51 UTC:52.95.4.10(14410):postgres@labdb:[1374]:DETAIL: ! system usage stats:
! 0.009494 s user, 0.007442 s system, 0.141985 s elapsed
! [0.022327 s user, 0.007442 s system total]
```

As shown in this example, the "LOG: AUDIT: SESSION" line provides information about the table and its schema, among other details.

To set up object auditing

1. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster.

   ```
   psql --host=your-instance-name.aws-region.rds.amazonaws.com --port=5432 --
   username=postgrespostgres --password --dbname=labdb
   ```

2. Create a database role named `rds_pgaudit` using the following command.

   ```
   labdb=> CREATE ROLE rds_pgaudit;
   CREATE ROLE
   labdb=>
   ```

3. Close the `psql` session.

   ```
   labdb=> \q
   ```

   In the next few steps, use the AWS CLI to modify the audit log parameters in your custom parameter group.

4. Use the following AWS CLI command to set the `pgaudit.role` parameter to `rds_pgaudit`. By default, this parameter is empty, and `rds_pgaudit` is the only allowable value.

   ```
   aws rds modify-db-parameter-group \
      --db-parameter-group-name custom-param-group-name \
      --parameters
    "ParameterName=pgaudit.role,ParameterValue=rds_pgaudit,ApplyMethod=pending-reboot"
    \
      --region aws-region
   ```

5. Use the following AWS CLI command to reboot the writer instance of your Aurora PostgreSQL DB cluster so that your changes to the parameters take effect.

   ```
   aws rds reboot-db-instance \
       --db-instance-identifier writer-instance \
       --region aws-region
   ```

6. Run the following command to confirm that the `pgaudit.role` is set to `rds_pgaudit`.

   ```
   SHOW pgaudit.role;
   pgaudit.role
   ------------------
   rds_pgaudit
   ```

To test pgAudit logging, you can run several example commands that you want to audit. For example, you might run the following commands.

```
CREATE TABLE t1 (id int);
GRANT SELECT ON t1 TO rds_pgaudit;
SELECT * FROM t1;
id
----
(0 rows)
```

The database logs should contain an entry similar to the following.

```
...
2017-06-12 19:09:49 UTC:...:rds_test@postgres:[11701]:LOG: AUDIT:
OBJECT,1,1,READ,SELECT,TABLE,public.t1,select * from t1;
...
```

For information on viewing the logs, see Monitoring Amazon Aurora log files.

To learn more about the pgAudit extension, see pgAudit on GitHub.

Excluding users or databases from audit logging

As discussed in Aurora PostgreSQL database log files, PostgreSQL logs consume storage space. Using the pgAudit extension adds to the volume of data gathered in your logs to varying degrees, depending on the changes that you track. You might not need to audit every user or database in your Aurora PostgreSQL DB cluster.

To minimize impacts to your storage and to avoid needlessly capturing audit records, you can exclude users and databases from being audited. You can also change logging within a given session. The following examples show you how.

> **Note**
>
> Parameter settings at the session level take precedence over the settings in the custom DB cluster parameter group for the Aurora PostgreSQL DB cluster's writer instance. If you don't want database users to bypass your audit logging configuration settings, be sure to change their permissions.

Suppose that your Aurora PostgreSQL DB cluster is configured to audit the same level of activity for all users and databases. You then decide that you don't want to audit the user `myuser`. You can turn off auditing for `myuser` with the following SQL command.

```
ALTER USER myuser SET pgaudit.log TO 'NONE';
```

Then, you can use the following query to check the `user_specific_settings` column for `pgaudit.log` to confirm that the parameter is set to `NONE`.

```
SELECT
    usename AS user_name,
    useconfig AS user_specific_settings
FROM
    pg_user
WHERE
    usename = 'myuser';
```

You see output such as the following.

```
 user_name | user_specific_settings
-----------+------------------------
 myuser    | {pgaudit.log=NONE}
(1 row)
```

You can turn off logging for a given user in the midst of their session with the database with the following command.

```
ALTER USER myuser IN DATABASE mydatabase SET pgaudit.log TO 'none';
```

Use the following query to check the settings column for pgaudit.log for a specific user and database combination.

```
SELECT
    usename AS "user_name",
    datname AS "database_name",
    pg_catalog.array_to_string(setconfig, E'\n') AS "settings"
FROM
    pg_catalog.pg_db_role_setting s
    LEFT JOIN pg_catalog.pg_database d ON d.oid = setdatabase
    LEFT JOIN pg_catalog.pg_user r ON r.usesysid = setrole
WHERE
    usename = 'myuser'
    AND datname = 'mydatabase'
ORDER BY
    1,
    2;
```

You see output similar to the following.

```
  user_name | database_name |     settings
-----------+---------------+------------------
 myuser    | mydatabase    | pgaudit.log=none
(1 row)
```

After turning off auditing for `myuser`, you decide that you don't want to track changes to `mydatabase`. You turn off auditing for that specific database by using the following command.

```
ALTER DATABASE mydatabase SET pgaudit.log to 'NONE';
```

Then, use the following query to check the database_specific_settings column to confirm that pgaudit.log is set to NONE.

```
SELECT
a.datname AS database_name,
b.setconfig AS database_specific_settings
FROM
pg_database a
FULL JOIN pg_db_role_setting b ON a.oid = b.setdatabase
WHERE
a.datname = 'mydatabase';
```

You see output such as the following.

```
 database_name | database_specific_settings
---------------+----------------------------
 mydatabase    | {pgaudit.log=NONE}
(1 row)
```

To return settings to the default setting for myuser, use the following command:

```
ALTER USER myuser RESET pgaudit.log;
```

To return settings to their default setting for a database, use the following command.

```
ALTER DATABASE mydatabase RESET pgaudit.log;
```

To reset user and database to the default setting, use the following command.

```
ALTER USER myuser IN DATABASE mydatabase RESET pgaudit.log;
```

You can also capture specific events to the log by setting the `pgaudit.log` to one of the other allowed values for the `pgaudit.log` parameter. For more information, see List of allowable settings for the `pgaudit.log` parameter.

```
ALTER USER myuser SET pgaudit.log TO 'read';
ALTER DATABASE mydatabase SET pgaudit.log TO 'function';
ALTER USER myuser IN DATABASE mydatabase SET pgaudit.log TO 'read,function'
```

Reference for the pgAudit extension

You can specify the level of detail that you want for your audit log by changing one or more of the parameters listed in this section.

Controlling pgAudit behavior

You can control the audit logging by changing one or more of the parameters listed in the following table.

Parameter	Description
`pgaudit.log`	Specifies the statement classes that will be logged by session audit logging. Allowable values include ddl, function, misc, read, role, write, none, all. For more information, see List of allowable settings for the `pgaudit.log` parameter.
`pgaudit.log_catalog`	When turned on (set to 1), adds statements to audit trail if all relations in a statement are in pg_catalog.
`pgaudit.log_level`	Specifies the log level to use for log entries. Allowed values: debug5, debug4, debug3, debug2, debug1, info, notice, warning, log
`pgaudit.log_parameter`	When turned on (set to 1), parameters passed with the statement are captured in the audit log.
`pgaudit.log_relation`	When turned on (set to 1), the audit log for the session creates a separate log entry for each relation (TABLE, VIEW, and so on) referenced in a SELECT or DML statement.
`pgaudit.log_statement_once`	Specifies whether logging will include the statement text and parameters with the first log entry for a statement/substatement combination or with every entry.
`pgaudit.role`	Specifies the master role to use for object audit logging. The only allowable entry is `rds_pgaudit`.

List of allowable settings for the `pgaudit.log` parameter

Value	Description
none	This is the default. No database changes are logged.
all	Logs everything (read, write, function, role, ddl, misc).
ddl	Logs all data definition language (DDL) statements that aren't included in the ROLE class.
function	Logs function calls and DO blocks.
misc	Logs miscellaneous commands, such as DISCARD, FETCH, CHECKPOINT , VACUUM, and SET.
read	Logs SELECT and COPY when the source is a relation (such as a table) or a query.
role	Logs statements related to roles and privileges, such as GRANT, REVOKE, CREATE ROLE, ALTER ROLE, and DROP ROLE.
write	Logs INSERT, UPDATE, DELETE, TRUNCATE, and COPY when the destination is a relation (table).

To log multiple event types with session auditing, use a comma-separated list. To log all event types, set pgaudit.log to ALL. Reboot your DB instance to apply the changes.

With object auditing, you can refine audit logging to work with specific relations. For example, you can specify that you want audit logging for READ operations on one or more tables.

Using pglogical to synchronize data across instances

All currently available Aurora PostgreSQL versions support the pglogical extension. The pglogical extension predates the functionally similar logical replication feature that was introduced by PostgreSQL in version 10. For more information, see Using PostgreSQL logical replication with Aurora.

The pglogical extension supports logical replication between two or more Aurora PostgreSQL DB clusters. It also supports replication between different PostgreSQL versions, and between databases running on RDS for PostgreSQL DB instances and Aurora PostgreSQL DB clusters. The `pglogical` extension uses a publish-subscribe model to replicate changes to tables and other objects, such as sequences, from a publisher to a subscriber. It relies on a replication slot to ensure that changes are synchronized from a publisher node to a subscriber node, defined as follows.

- The *publisher node* is the Aurora PostgreSQL DB cluster that's the source of data to be replicated to other nodes. The publisher node defines the tables to be replicated in a publication set.
- The *subscriber node* is the Aurora PostgreSQL DB cluster that receives WAL updates from the publisher. The subscriber creates a subscription to connect to the publisher and get the decoded WAL data. When the subscriber creates the subscription, the replication slot is created on the publisher node.

Following, you can find information about setting up the `pglogical` extension.

Topics

Requirements and limitations for the pglogical extension

All currently available releases of Aurora PostgreSQL support the `pglogical` extension.

Both the publisher node and the subscriber node must be set up for logical replication.

The tables that you want to replicate from subscriber to publisher must have the same names and the same schema. These tables must also contain the same columns, and the columns must use the same data types. Both publisher and subscriber tables must have the same primary keys. We recommend that you use only the PRIMARY KEY as the unique constraint.

The tables on the subscriber node can have more permissive constraints than those on the publisher node for CHECK constraints and NOT NULL constraints.

The pglogical extension provides features such as two-way replication that aren't supported by the logical replication feature built into PostgreSQL (version 10 and higher). For more information, see PostgreSQL bi-directional replication using pglogical.

Setting up the pglogical extension

To set up the pglogical extension on your Aurora PostgreSQL DB cluster, you add pglogical to the shared libraries on the custom DB cluster parameter group for your Aurora PostgreSQL DB cluster. You also need to set the value of the rds.logical_replication parameter to 1, to turn on logical decoding. Finally, you create the extension in the database. You can use the AWS Management Console or the AWS CLI for these tasks.

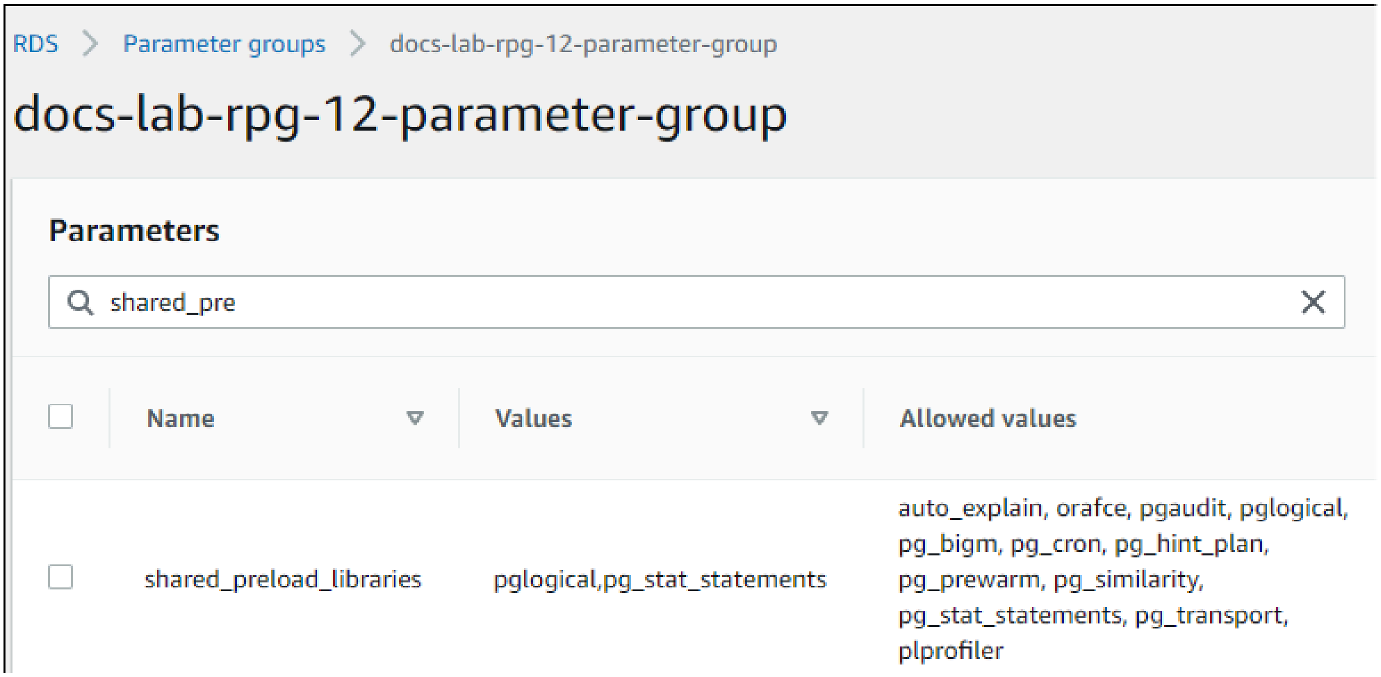
You must have permissions as the rds_superuser role to perform these tasks.

The steps following assume that your Aurora PostgreSQL DB cluster is associated with a custom DB cluster parameter group. For information about creating a custom DB cluster parameter group, see Working with parameter groups.

Console

To set up the pglogical extension

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose your Aurora PostgreSQL DB cluster's Writer instance .
3. Open the **Configuration** tab for your Aurora PostgreSQL DB cluster writer instance. Among the Instance details, find the **Parameter group** link.
4. Choose the link to open the custom parameters associated with your Aurora PostgreSQL DB cluster.
5. In the **Parameters** search field, type shared_pre to find the shared_preload_libraries parameter.
6. Choose **Edit parameters** to access the property values.
7. Add pglogical to the list in the **Values** field. Use a comma to separate items in the list of values.



RDS > Parameter groups > docs-lab-rpg-12-parameter-group

docs-lab-rpg-12-parameter-group

Parameters

shared_pre

□	Name	Values	Allowed values
□	shared_preload_libraries	pglogical,pg_stat_statements	auto_explain, orafce, pgaudit, pglogical, pg_bigm, pg_cron, pg_hint_plan, pg_prewarm, pg_similarity, pg_stat_statements, pg_transport, plprofiler

8. Find the `rds.logical_replication` parameter and set it to 1, to turn on logical replication.
9. Reboot the writer instance of your Aurora PostgreSQL DB cluster so that your changes take effect.
10. When the instance is available, you can use `psql` (or pgAdmin) to connect to the writer instance of your Aurora PostgreSQL DB cluster.

```
psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password --dbname=labdb
```

11. To verify that pglogical is initialized, run the following command.

```
SHOW shared_preload_libraries;
shared_preload_libraries
--------------------------
rdsutils,pglogical
(1 row)
```

12. Verify the setting that enables logical decoding, as follows.

```
SHOW wal_level;
wal_level
-----------
 logical
(1 row)
```

13. Create the extension, as follows.

    ```
    CREATE EXTENSION pglogical;
    EXTENSION CREATED
    ```

14. Choose **Save changes**.

15. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

16. Choose your Aurora PostgreSQL DB cluster's writer instance from the Databases list to select it, and then choose **Reboot** from the Actions menu.

AWS CLI

To setup the pglogical extension

To setup pglogical using the AWS CLI, you call the modify-db-parameter-group operation to modify certain parameters in your custom parameter group as shown in the following procedure.

1. Use the following AWS CLI command to add `pglogical` to the `shared_preload_libraries` parameter.

   ```
   aws rds modify-db-parameter-group \
      --db-parameter-group-name custom-param-group-name \
      --parameters
    "ParameterName=shared_preload_libraries,ParameterValue=pglogical,ApplyMethod=pending-
   reboot" \
      --region aws-region
   ```

2. Use the following AWS CLI command to set `rds.logical_replication` to 1 to turn on the logical decoding capability for the writer instance of the Aurora PostgreSQL DB cluster.

   ```
   aws rds modify-db-parameter-group \
      --db-parameter-group-name custom-param-group-name \
      --parameters
    "ParameterName=rds.logical_replication,ParameterValue=1,ApplyMethod=pending-
   reboot" \
      --region aws-region
   ```

3. Use the following AWS CLI command to reboot the writer instance of your Aurora PostgreSQL DB cluster so that the pglogical library is initialized.

```
aws rds reboot-db-instance \
    --db-instance-identifier writer-instance \
    --region aws-region
```

4. When the instance is available, use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster.

   ```
   psql --host=111122223333.aws-region.rds.amazonaws.com --port=5432 --
   username=postgres --password --dbname=labdb
   ```

5. Create the extension, as follows.

   ```
   CREATE EXTENSION pglogical;
   EXTENSION CREATED
   ```

6. Reboot the writer instance of your Aurora PostgreSQL DB cluster using the following AWS CLI command.

   ```
   aws rds reboot-db-instance \
       --db-instance-identifier writer-instance \
       --region aws-region
   ```

Setting up logical replication for Aurora PostgreSQL DB cluster

The following procedure shows you how to start logical replication between two Aurora PostgreSQL DB clusters. The steps assume that both the source (publisher) and the target (subscriber) have the `pglogical` extension set up as detailed in Setting up the pglogical extension.

To create the publisher node and define the tables to replicate

These steps assume that your Aurora PostgreSQL DB cluster has a writer instance with a database that has one or more tables that you want to replicate to another node. You need to recreate the table structure from the publisher on the subscriber, so first, get the table structure if necessary. You can do that by using the `psq1` metacommand `\d` *`tablename`* and then creating the same table on the subscriber instance. The following procedure creates an example table on the publisher (source) for demonstration purposes.

1. Use `psql` to connect to the instance that has the table you want to use as a source for subscribers.

   ```
   psql --host=source-instance.aws-region.rds.amazonaws.com --port=5432 --
   username=postgres --password --dbname=labdb
   ```

   If you don't have an existing table that you want to replicate, you can create a sample table as follows.

   a. Create an example table using the following SQL statement.

      ```
      CREATE TABLE docs_lab_table (a int PRIMARY KEY);
      ```

   b. Populate the table with generated data by using the following SQL statement.

      ```
      INSERT INTO docs_lab_table VALUES (generate_series(1,5000));
      INSERT 0 5000
      ```

   c. Verify that data exists in the table by using the following SQL statement.

      ```
      SELECT count(*) FROM docs_lab_table;
      ```

2. Identify this Aurora PostgreSQL DB cluster as the publisher node, as follows.

   ```
   SELECT pglogical.create_node(
       node_name := 'docs_lab_provider',
       dsn := 'host=source-instance.aws-region.rds.amazonaws.com port=5432
    dbname=labdb');
    create_node
   -------------
      3410995529
   (1 row)
   ```

3. Add the table that you want to replicate to the default replication set. For more information about replication sets, see Replication sets in the pglogical documentation.

   ```
   SELECT pglogical.replication_set_add_table('default', 'docs_lab_table', 'true',
    NULL, NULL);
    replication_set_add_table
     ---------------------------
     t
   ```

```
  (1 row)
```

The publisher node setup is complete. You can now set up the subscriber node to receive the updates from the publisher.

To set up the subscriber node and create a subscription to receive updates

These steps assume that the Aurora PostgreSQL DB cluster has been set up with the `pglogical` extension. For more information, see Setting up the pglogical extension.

1. Use `psql` to connect to the instance that you want to receive updates from the publisher.

   ```
   psql --host=target-instance.aws-region.rds.amazonaws.com --port=5432 --
   username=postgres --password --dbname=labdb
   ```

2. On the subscriber Aurora PostgreSQL DB cluster, ,create the same table that exists on the publisher. For this example, the table is `docs_lab_table`. You can create the table as follows.

   ```
   CREATE TABLE docs_lab_table (a int PRIMARY KEY);
   ```

3. Verify that this table is empty.

   ```
   SELECT count(*) FROM docs_lab_table;
    count
   -------
     0
   (1 row)
   ```

4. Identify this Aurora PostgreSQL DB cluster as the subscriber node, as follows.

   ```
   SELECT pglogical.create_node(
       node_name := 'docs_lab_target',
       dsn := 'host=target-instance.aws-region.rds.amazonaws.com port=5432
    sslmode=require dbname=labdb user=postgres password=********');
    create_node
   -------------
      2182738256
   (1 row)
   ```

5. Create the subscription.

```
SELECT pglogical.create_subscription(
   subscription_name := 'docs_lab_subscription',
   provider_dsn := 'host=source-instance.aws-region.rds.amazonaws.com port=5432
 sslmode=require dbname=labdb user=postgres password=*******',
   replication_sets := ARRAY['default'],
   synchronize_data := true,
   forward_origins := '{}' );
 create_subscription
---------------------
1038357190
(1 row)
```

When you complete this step, the data from the table on the publisher is created in the table on the subscriber. You can verify that this has occurred by using the following SQL query.

```
SELECT count(*) FROM docs_lab_table;
 count
-------
  5000
(1 row)
```

From this point forward, changes made to the table on the publisher are replicated to the table on the subscriber.

Reestablishing logical replication after a major upgrade

Before you can perform a major version upgrade of an Aurora PostgreSQL DB cluster that's set up as a publisher node for logical replication, you must drop all replication slots, even those that aren't active. We recommend that you temporarily divert database transactions from the publisher node, drop the replication slots, upgrade the Aurora PostgreSQL DB cluster, and then re-establish and restart replication.

The replication slots are hosted on the publisher node only. The Aurora PostgreSQL subscriber node in a logical replication scenario has no slots to drop. The Aurora PostgreSQL major version upgrade process supports upgrading the subscriber to a new major version of PostgreSQL independent of the publisher node. However, the upgrade process does disrupt the replication process and interferes with the synchronization of WAL data between publisher node and subscriber node. You need to re-establish logical replication between publisher and subscriber after upgrading the publisher, the subscriber, or both. The following procedure shows you how to determine that replication has been disrupted and how to resolve the issue.

Determining that logical replication has been disrupted

You can determine that the replication process has been disrupted by querying either the publisher node or the subscriber node, as follows.

To check the publisher node

- Use `psql` to connect to the publisher node, and then query the `pg_replication_slots` function. Note the value in the active column. Normally, this will return `t` (true), showing that replication is active. If the query returns `f` (false), it's an indication that replication to the subscriber has stopped.

```
SELECT slot_name,plugin,slot_type,active FROM pg_replication_slots;
                    slot_name              |      plugin      | slot_type | active
-------------------------------------------+------------------+-----------+--------
 pgl_labdb_docs_labcb4fa94_docs_lab3de412c | pglogical_output | logical   | f
(1 row)
```

To check the subscriber node

On the subscriber node, you can check the status of replication in three different ways.

- Look through the PostgreSQL logs on the subscriber node to find failure messages. The log identifies failure with messages that include exit code 1, as shown following.

```
2022-07-06 16:17:03 UTC::@:[7361]:LOG: background worker "pglogical apply
 16404:2880255011" (PID 14610) exited with exit code 1
2022-07-06 16:19:44 UTC::@:[7361]:LOG: background worker "pglogical apply
 16404:2880255011" (PID 21783) exited with exit code 1
```

- Query the `pg_replication_origin` function. Connect to the database on the subscriber node using `psql` and query the `pg_replication_origin` function, as follows.

```
SELECT * FROM pg_replication_origin;
 roident | roname
---------+--------
(0 rows)
```

The empty result set means that replication has been disrupted. Normally, you see output such as the following.

```
 roident |                         roname
---------+---------------------------------------------------
       1 | pgl_labdb_docs_labcb4fa94_docs_lab3de412c
(1 row)
```

- Query the pglogical.show_subscription_status function as shown in the following example.

```
SELECT subscription_name,status,slot_name FROM pglogical.show_subscription_status();
     subscription_name | status |              slot_name
---====----------------+--------+-------------------------------------
 docs_lab_subscription | down   | pgl_labdb_docs_labcb4fa94_docs_lab3de412c
(1 row)
```

This output shows that replication has been disrupted. Its status is down. Normally, the output shows the status as replicating.

If your logical replication process has been disrupted, you can re-establish replication by following these steps.

To reestablish logical replication between publisher and subscriber nodes

To re-establish replication, you first disconnect the subscriber from the publisher node and then re-establish the subscription, as outlined in these steps.

1. Connect to the subscriber node using psql as follows.

```
psql --host=222222222222.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password --dbname=labdb
```

2. Deactivate the subscription by using the pglogical.alter_subscription_disable function.

```
SELECT pglogical.alter_subscription_disable('docs_lab_subscription',true);
 alter_subscription_disable
----------------------------
 t
```

```
(1 row)
```

3. Get the publisher node's identifier by querying the `pg_replication_origin`, as follows.

   ```
   SELECT * FROM pg_replication_origin;
    roident |               roname
   ---------+-------------------------------------
          1 | pgl_labdb_docs_labcb4fa94_docs_lab3de412c
   (1 row)
   ```

4. Use the response from the previous step with the `pg_replication_origin_create` command to assign the identifier that can be used by the subscription when re-established.

   ```
   SELECT pg_replication_origin_create('pgl_labdb_docs_labcb4fa94_docs_lab3de412c');
     pg_replication_origin_create
   ------------------------------
                               1
   (1 row)
   ```

5. Turn on the subscription by passing its name with a status of `true`, as shown in the following example.

   ```
   SELECT pglogical.alter_subscription_enable('docs_lab_subscription',true);
     alter_subscription_enable
   ---------------------------
    t
   (1 row)
   ```

Check the status of the node. Its status should be `replicating` as shown in this example.

```
SELECT subscription_name,status,slot_name
  FROM pglogical.show_subscription_status();
             subscription_name |   status    |              slot_name
-------------------------------+-------------+-------------------------------------
 docs_lab_subscription         | replicating |
 pgl_labdb_docs_lab98f517b_docs_lab3de412c
(1 row)
```

Check the status of the subscriber's replication slot on the publisher node. The slot's `active` column should return `t` (true), indicating that replication has been re-established.

```
SELECT slot_name,plugin,slot_type,active
  FROM pg_replication_slots;
                    slot_name              |      plugin      | slot_type | active
-------------------------------------------+------------------+-----------+--------
 pgl_labdb_docs_lab98f517b_docs_lab3de412c | pglogical_output | logical   | t
(1 row)
```

Managing logical replication slots for Aurora PostgreSQL

Before you can perform a major version upgrade on an Aurora PostgreSQL DB cluster's writer instance that's serving as a publisher node in a logical replication scenario, you must drop the replication slots on the instance. The major version upgrade pre-check process notifies you that the upgrade can't proceed until the slots are dropped.

To identify replication slots that were created using the `pglogical` extension, log in to each database and get the name of the nodes. When you query the subscriber node, you get both the publisher and the subscriber nodes in the output, as shown in this example.

```
SELECT * FROM pglogical.node;
node_id   |     node_name
------------+-------------------
 2182738256 | docs_lab_target
 3410995529 | docs_lab_provider
(2 rows)
```

You can get the details about the subscription with the following query.

```
SELECT sub_name,sub_slot_name,sub_target
  FROM pglogical.subscription;
 sub_name |         sub_slot_name          | sub_target
----------+--------------------------------+------------
  docs_lab_subscription     | pgl_labdb_docs_labcb4fa94_docs_lab3de412c | 2182738256
(1 row)
```

You can now drop the subscription, as follows.

```
SELECT pglogical.drop_subscription(subscription_name := 'docs_lab_subscription');
 drop_subscription
-------------------
                 1
```

```
(1 row)
```

After dropping the subscription, you can delete the node.

```
SELECT pglogical.drop_node(node_name := 'docs-lab-subscriber');
 drop_node
-----------
 t
(1 row)
```

You can verify that the node no longer exists, as follows.

```
SELECT * FROM pglogical.node;
 node_id | node_name
---------+-----------
(0 rows)
```

Parameter reference for the pglogical extension

In the table you can find parameters associated with the pglogical extension. Parameters such as pglogical.conflict_log_level and pglogical.conflict_resolution are used to handle update conflicts. Conflicts can emerge when changes are made locally to the same tables that are subscribed to changes from the publisher. Conflicts can also occur during various scenarios, such as two-way replication or when multiple subscribers are replicating from the same publisher. For more information, see PostgreSQL bi-directional replication using pglogical.

Parameter	Description
pglogical.batch_inserts	Batch inserts if possible. Not set by default. Change to '1' to turn on, '0' to turn off.
pglogical.conflict_log_level	Sets the log level to use for logging resolved conflicts. Supported string values are debug5, debug4, debug3, debug2, debug1, info, notice, warning, error, log, fatal, panic.
pglogical.conflict_resolution	Sets method to use to resolve conflicts when conflicts are resolvable. Supported string values are error, apply_remote, keep_local, last_update_wins, first_update_wins.

Parameter	Description
pglogical.extra_connection_options	Connection options to add to all peer node connections.
pglogical.synchronous_commit	pglogical specific synchronous commit value
pglogical.use_spi	Use SPI (server programming interface) instead of low-level API to apply changes. Set to '1' to turn on, '0' to turn off. For more information about SPI, see Server Programming Interface in the PostgreSQL documentation.

Working with the supported foreign data wrappers for Amazon Aurora PostgreSQL

A foreign data wrapper (FDW) is a specific type of extension that provides access to external data. For example, the `oracle_fdw` extension allows your Aurora PostgreSQL DB instance to work with Oracle databases.

Following, you can find information about several supported PostgreSQL foreign data wrappers.

Topics

Using the log_fdw extension to access the DB log using SQL

Aurora PostgreSQL DB cluster supports the `log_fdw` extension, which you can use to access your database engine log using a SQL interface. The `log_fdw` extension provides two functions that make it easy to create foreign tables for database logs:

- `list_postgres_log_files` – Lists the files in the database log directory and the file size in bytes.
- `create_foreign_table_for_log_file(table_name text, server_name text, log_file_name text)` – Builds a foreign table for the specified file in the current database.

All functions created by `log_fdw` are owned by `rds_superuser`. Members of the `rds_superuser` role can grant access to these functions to other database users.

By default, the log files are generated by Amazon Aurora in `stderr` (standard error) format, as specified in `log_destination` parameter. There are only two options for this parameter, `stderr` and `csvlog` (comma-separated values, CSV). If you add the `csvlog` option to the parameter, Amazon Aurora generates both `stderr` and `csvlog` logs. This can affect the storage capacity on your DB cluster, so you need to be aware of the other parameters that affect log handling. For more information, see Setting the log destination (`stderr`, `csvlog`).

One benefit of generating `csvlog` logs is that the `log_fdw` extension lets you build foreign tables with the data neatly split into several columns. To do this, your instance needs to be associated with a custom DB parameter group so that you can change the setting for `log_destination`. For more information about how to do so, see Working with parameter groups.

The following example assumes that the `log_destination` parameter includes `cvslog`.

To use the log_fdw extension

1. Install the `log_fdw` extension.

   ```
   postgres=> CREATE EXTENSION log_fdw;
   CREATE EXTENSION
   ```

2. Create the log server as a foreign data wrapper.

   ```
   postgres=> CREATE SERVER log_server FOREIGN DATA WRAPPER log_fdw;
   CREATE SERVER
   ```

3. Select all from a list of log files.

   ```
   postgres=> SELECT * FROM list_postgres_log_files() ORDER BY 1;
   ```

   A sample response is as follows.

```
          file_name            | file_size_bytes
------------------------------+----------------
 postgresql.log.2023-08-09-22.csv |            1111
 postgresql.log.2023-08-09-23.csv |            1172
 postgresql.log.2023-08-10-00.csv |            1744
 postgresql.log.2023-08-10-01.csv |            1102
(4 rows)
```

4. Create a table with a single 'log_entry' column for the selected file.

```
postgres=> SELECT create_foreign_table_for_log_file('my_postgres_error_log',
     'log_server', 'postgresql.log.2023-08-09-22.csv');
```

The response provides no detail other than that the table now exists.

```
-----------------------------------
(1 row)
```

5. Select a sample of the log file. The following code retrieves the log time and error message description.

```
postgres=> SELECT log_time, message FROM my_postgres_error_log ORDER BY 1;
```

A sample response is as follows.

```
             log_time             |                                  message
----------------------------------
+----------------------------------------------------------------------------
Tue Aug 09 15:45:18.172 2023 PDT | ending log output to stderr
Tue Aug 09 15:45:18.175 2023 PDT | database system was interrupted; last known up
 at 2023-08-09 22:43:34 UTC
Tue Aug 09 15:45:18.223 2023 PDT | checkpoint record is at 0/90002E0
Tue Aug 09 15:45:18.223 2023 PDT | redo record is at 0/90002A8; shutdown FALSE
Tue Aug 09 15:45:18.223 2023 PDT | next transaction ID: 0/1879; next OID: 24578
Tue Aug 09 15:45:18.223 2023 PDT | next MultiXactId: 1; next MultiXactOffset: 0
Tue Aug 09 15:45:18.223 2023 PDT | oldest unfrozen transaction ID: 1822, in
 database 1
(7 rows)
```

Using the postgres_fdw extension to access external data

You can access data in a table on a remote database server with the [postgres_fdw](https://www.postgresql.org/docs/current/static/postgres-fdw.html) extension. If you set up a remote connection from your PostgreSQL DB instance, access is also available to your read replica.

To use postgres_fdw to access a remote database server

1. Install the postgres_fdw extension.

    ```
    CREATE EXTENSION postgres_fdw;
    ```

2. Create a foreign data server using CREATE SERVER.

    ```
    CREATE SERVER foreign_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host 'xxx.xx.xxx.xx', port '5432', dbname 'foreign_db');
    ```

3. Create a user mapping to identify the role to be used on the remote server.

    ```
    CREATE USER MAPPING FOR local_user
    SERVER foreign_server
    OPTIONS (user 'foreign_user', password 'password');
    ```

4. Create a table that maps to the table on the remote server.

    ```
    CREATE FOREIGN TABLE foreign_table (
            id integer NOT NULL,
            data text)
    SERVER foreign_server
    OPTIONS (schema_name 'some_schema', table_name 'some_table');
    ```

Working with MySQL databases by using the mysql_fdw extension

To access a MySQL-compatible database from your Aurora PostgreSQL DB cluster, you can install and use the `mysql_fdw` extension. This foreign data wrapper lets you work with RDS for MySQL, Aurora MySQL, MariaDB, and other MySQL-compatible databases. The connection from Aurora PostgreSQL DB cluster to the MySQL database is encrypted on a best-effort basis, depending on the client and server configurations. However, you can enforce encryption if you like. For more information, see Using encryption in transit with the extension.

The mysql_fdw extension is supported on Amazon Aurora PostgreSQL version 15.4, 14.9, 13.12, 12.16, and higher releases. It supports selects, inserts, updates, and deletes from an RDS for PostgreSQL DB to tables on a MySQL-compatible database instance.

Topics

- Setting up your Aurora PostgreSQL DB to use the mysql_fdw extension
- Example: Working with an Aurora MySQL database from Aurora PostgreSQL
- Using encryption in transit with the extension

Setting up your Aurora PostgreSQL DB to use the mysql_fdw extension

Setting up the mysql_fdw extension on your Aurora PostgreSQL DB cluster involves loading the extension in your DB cluster and then creating the connection point to the MySQL DB instance. For that task, you need to have the following details about the MySQL DB instance:

- Hostname or endpoint. For an Aurora MySQL DB cluster, you can find the endpoint by using the Console. Choose the Connectivity & security tab and look in the "Endpoint and port" section.
- Port number. The default port number for MySQL is 3306.
- Name of the database. The DB identifier.

You also need to provide access on the security group or the access control list (ACL) for the MySQL port, 3306. Both the Aurora PostgreSQL DB cluster and the Aurora MySQL DB cluster need access to port 3306. If access isn't configured correctly, when you try to connect to MySQL-compatible table you see an error message similar to the following:

```
ERROR: failed to connect to MySQL: Can't connect to MySQL server on 'hostname.aws-region.rds.amazonaws.com:3306' (110)
```

In the following procedure, you (as the rds_superuser account) create the foreign server. You then grant access to the foreign server to specific users. These users then create their own mappings to the appropriate MySQL user accounts to work with the MySQL DB instance.

To use mysql_fdw to access a MySQL database server

1. Connect to your PostgreSQL DB instance using an account that has the rds_superuser role. If you accepted the defaults when you created your Aurora PostgreSQL DB cluster , the user name is postgres, and you can connect using the psql command line tool as follows:

```
psql --host=your-DB-instance.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password
```

2. Install the `mysql_fdw` extension as follows:

```
postgres=> CREATE EXTENSION mysql_fdw;
CREATE EXTENSION
```

After the extension is installed on your Aurora PostgreSQL DB cluster , you set up the foreign server that provides the connection to a MySQL database.

To create the foreign server

Perform these tasks on the Aurora PostgreSQL DB cluster . The steps assume that you're connected as a user with `rds_superuser` privileges, such as `postgres`.

1. Create a foreign server in the Aurora PostgreSQL DB cluster :

```
postgres=> CREATE SERVER mysql-db FOREIGN DATA WRAPPER mysql_fdw OPTIONS (host 'db-
name.111122223333.aws-region.rds.amazonaws.com', port '3306');
CREATE SERVER
```

2. Grant the appropriate users access to the foreign server. These should be non-administrator users, that is, users without the `rds_superuser` role.

```
postgres=> GRANT USAGE ON FOREIGN SERVER mysql-db to user1;
GRANT
```

PostgreSQL users create and manage their own connections to the MySQL database through the foreign server.

Example: Working with an Aurora MySQL database from Aurora PostgreSQL

Suppose that you have a simple table on an Aurora PostgreSQL DB instance . Your Aurora PostgreSQL users want to query (`SELECT`), `INSERT`, `UPDATE`, and `DELETE` items on that table. Assume that the `mysql_fdw` extension was created on your RDS for PostgreSQL DB instance, as detailed in the preceding procedure. After you connect to the RDS for PostgreSQL DB instance as a user that has `rds_superuser` privileges, you can proceed with the following steps.

1. On the Aurora PostgreSQL DB instance, create a foreign server:

   ```
   test=> CREATE SERVER mysqldb FOREIGN DATA WRAPPER mysql_fdw OPTIONS (host 'your-DB.aws-region.rds.amazonaws.com', port '3306');
   CREATE SERVER
   ```

2. Grant usage to a user who doesn't have `rds_superuser` permissions, for example, `user1`:

   ```
   test=> GRANT USAGE ON FOREIGN SERVER mysqldb TO user1;
   GRANT
   ```

3. Connect as *`user1`*, and then create a mapping to the MySQL user:

   ```
   test=> CREATE USER MAPPING FOR user1 SERVER mysqldb OPTIONS (username 'myuser',
    password 'mypassword');
   CREATE USER MAPPING
   ```

4. Create a foreign table linked to the MySQL table:

   ```
   test=> CREATE FOREIGN TABLE mytab (a int, b text) SERVER mysqldb OPTIONS (dbname
    'test', table_name '');
   CREATE FOREIGN TABLE
   ```

5. Run a simple query against the foreign table:

   ```
   test=> SELECT * FROM mytab;
   a |   b
   ---+-------
   1 | apple
   (1 row)
   ```

6. You can add, change, and remove data from the MySQL table. For example:

   ```
   test=> INSERT INTO mytab values (2, 'mango');
   INSERT 0 1
   ```

   Run the SELECT query again to see the results:

   ```
   test=> SELECT * FROM mytab ORDER BY 1;
    a |   b
   ---+-------
   1 | apple
   ```

```
2 | mango
(2 rows)
```

Using encryption in transit with the extension

The connection to MySQL from Aurora PostgreSQL uses encryption in transit (TLS/SSL) by default. However, the connection falls back to non-encrypted when the client and server configuration differ. You can enforce encryption for all outgoing connections by specifying the `REQUIRE SSL` option on the RDS for MySQL user accounts. This same approach also works for MariaDB and Aurora MySQL user accounts.

For MySQL user accounts configured to `REQUIRE SSL`, the connection attempt fails if a secure connection can't be established.

To enforce encryption for existing MySQL database user accounts, you can use the `ALTER USER` command. The syntax varies, depending on the MySQL version, as shown in the following table. For more information, see ALTER USER in *MySQL Reference Manual*.

MySQL 5.7, MySQL 8.0	MySQL 5.6
`ALTER USER 'user'@'%' REQUIRE SSL;`	`GRANT USAGE ON *.* to 'user'@'%' REQUIRE SSL;`

For more information about the `mysql_fdw` extension, see the mysql_fdw documentation.

Working with Oracle databases by using the oracle_fdw extension

To access an Oracle database from your Aurora PostgreSQL DB cluster you can install and use the `oracle_fdw` extension. This extension is a foreign data wrapper for Oracle databases. To learn more about this extension, see the oracle_fdw documentation.

The `oracle_fdw` extension is supported on Aurora PostgreSQL 12.7 (Amazon Aurora release 4.2) and higher versions.

Topics

- Understanding the pg_user_mappings view and permissions

Turning on the oracle_fdw extension

To use the oracle_fdw extension, perform the following procedure.

To turn on the oracle_fdw extension

- Run the following command using an account that has `rds_superuser` permissions.

  ```
  CREATE EXTENSION oracle_fdw;
  ```

Example: Using a foreign server linked to an Amazon RDS for Oracle database

The following example shows the use of a foreign server linked to an Amazon RDS for Oracle database.

To create a foreign server linked to an RDS for Oracle database

1. Note the following on the RDS for Oracle DB instance:
   - Endpoint
   - Port
   - Database name
2. Create a foreign server.

   ```
   test=> CREATE SERVER oradb FOREIGN DATA WRAPPER oracle_fdw OPTIONS (dbserver
    '//endpoint:port/DB_name');
   CREATE SERVER
   ```

3. Grant usage to a user who doesn't have `rds_superuser` privileges, for example `user1`.

   ```
   test=> GRANT USAGE ON FOREIGN SERVER oradb TO user1;
   GRANT
   ```

4. Connect as `user1`, and create a mapping to an Oracle user.

   ```
   test=> CREATE USER MAPPING FOR user1 SERVER oradb OPTIONS (user 'oracleuser',
    password 'mypassword');
   ```

```
CREATE USER MAPPING
```

5. Create a foreign table linked to an Oracle table.

   ```
   test=> CREATE FOREIGN TABLE mytab (a int) SERVER oradb OPTIONS (table 'MYTABLE');
   CREATE FOREIGN TABLE
   ```

6. Query the foreign table.

   ```
   test=>  SELECT * FROM mytab;
   a
   ---
   1
   (1 row)
   ```

If the query reports the following error, check your security group and access control list (ACL) to make sure that both instances can communicate.

```
ERROR: connection for foreign table "mytab" cannot be established
DETAIL: ORA-12170: TNS:Connect timeout occurred
```

Working with encryption in transit

PostgreSQL-to-Oracle encryption in transit is based on a combination of client and server configuration parameters. For an example using Oracle 21c, see About the Values for Negotiating Encryption and Integrity in the Oracle documentation. The client used for oracle_fdw on Amazon RDS is configured with ACCEPTED, meaning that the encryption depends on the Oracle database server configuration.

If your database is on RDS for Oracle, see Oracle native network encryption to configure the encryption.

Understanding the pg_user_mappings view and permissions

The PostgreSQL catalog pg_user_mapping stores the mapping from an Aurora PostgreSQL user to the user on a foreign data (remote) server. Access to the catalog is restricted, but you use the pg_user_mappings view to see the mappings. In the following, you can find an example that shows how permissions apply with an example Oracle database, but this information applies more generally to any foreign data wrapper.

In the following output, you can find roles and permissions mapped to three different example users. Users `rdssu1` and `rdssu2` are members of the `rds_superuser` role, and `user1` isn't. The example uses the `psql` metacommand `\du` to list existing roles.

```
test=>  \du
                                                               List of roles
    Role name    |                         Attributes                         |
                  Member of
-----------------+------------------------------------------------------------
+-------------------------------------------------------------
 rdssu1          |                                                            |
 {rds_superuser}
 rdssu2          |                                                            |
 {rds_superuser}
 user1           |                                                            | {}
```

All users, including users that have `rds_superuser` privileges, are allowed to view their own user mappings (`umoptions`) in the `pg_user_mappings` table. As shown in the following example, when `rdssu1` tries to obtain all user mappings, an error is raised even though `rdssu1rds_superuser` privileges:

```
test=> SELECT * FROM pg_user_mapping;
ERROR: permission denied for table pg_user_mapping
```

Following are some examples.

```
test=> SET SESSION AUTHORIZATION rdssu1;
SET
test=> SELECT * FROM pg_user_mappings;
 umid  | srvid | srvname | umuser | usename    |            umoptions
-------+-------+---------+--------+------------+----------------------------------
 16414 | 16411 | oradb   |  16412 | user1      |
 16423 | 16411 | oradb   |  16421 | rdssu1     | {user=oracleuser,password=mypwd}
 16424 | 16411 | oradb   |  16422 | rdssu2     |
 (3 rows)

test=> SET SESSION AUTHORIZATION rdssu2;
SET
test=> SELECT * FROM pg_user_mappings;
 umid  | srvid | srvname | umuser | usename    |            umoptions
-------+-------+---------+--------+------------+----------------------------------
```

```
 16414 | 16411 | oradb   |  16412 | user1      |
 16423 | 16411 | oradb   |  16421 | rdssu1     |
 16424 | 16411 | oradb   |  16422 | rdssu2     | {user=oracleuser,password=mypwd}
 (3 rows)

test=> SET SESSION AUTHORIZATION user1;
SET
test=> SELECT * FROM pg_user_mappings;
 umid  | srvid | srvname | umuser | usename    |            umoptions
-------+-------+---------+--------+------------+---------------------------------
 16414 | 16411 | oradb   |  16412 | user1      | {user=oracleuser,password=mypwd}
 16423 | 16411 | oradb   |  16421 | rdssu1     |
 16424 | 16411 | oradb   |  16422 | rdssu2     |
 (3 rows)
```

Because of implementation differences between `information_schema._pg_user_mappings` and `pg_catalog.pg_user_mappings`, a manually created `rds_superuser` requires additional permissions to view passwords in `pg_catalog.pg_user_mappings`.

No additional permissions are required for an `rds_superuser` to view passwords in `information_schema._pg_user_mappings`.

Users who don't have the `rds_superuser` role can view passwords in `pg_user_mappings` only under the following conditions:

- The current user is the user being mapped and owns the server or holds the `USAGE` privilege on it.
- The current user is the server owner and the mapping is for `PUBLIC`.

Working with SQL Server databases by using the tds_fdw extension

You can use the PostgreSQL `tds_fdw` extension to access databases that support the tabular data stream (TDS) protocol, such as Sybase and Microsoft SQL Server databases. This foreign data wrapper lets you connect from your Aurora PostgreSQL DB cluster to databases that use the TDS protocol, including Amazon RDS for Microsoft SQL Server. For more information, see tds-fdw/tds_fdw documentation on GitHub.

The `tds_fdw` extension is supported on Amazon Aurora PostgreSQL version 13.6 and higher releases.

Setting up your Aurora PostgreSQL DB to use the tds_fdw extension

In the following procedures, you can find an example of setting up and using the `tds_fdw` with an Aurora PostgreSQL DB cluster. Before you can connect to a SQL Server database using `tds_fdw`, you need to get the following details for the instance:

- Hostname or endpoint. For an RDS for SQL Server DB instance, you can find the endpoint by using the Console. Choose the Connectivity & security tab and look in the "Endpoint and port" section.
- Port number. The default port number for Microsoft SQL Server is 1433.
- Name of the database. The DB identifier.

You also need to provide access on the security group or the access control list (ACL) for the SQL Server port, 1433. Both the Aurora PostgreSQL DB cluster and the RDS for SQL Server DB instance need access to port 1433. If access isn't configured correctly, when you try to query the Microsoft SQL Server you see the following error message:

```
ERROR: DB-Library error: DB #: 20009, DB Msg: Unable to connect:
Adaptive Server is unavailable or does not exist (mssql2019.aws-
region.rds.amazonaws.com), OS #: 0, OS Msg: Success, Level: 9
```

To use tds_fdw to connect to a SQL Server database

1. Connect to your Aurora PostgreSQL DB cluster's primary instance using an account that has the `rds_superuser` role:

```
psql --host=your-cluster-name-instance-1.aws-region.rds.amazonaws.com --port=5432
 --username=test --password
```

2. Install the `tds_fdw` extension:

```
test=> CREATE EXTENSION tds_fdw;
CREATE EXTENSION
```

After the extension is installed on your Aurora PostgreSQL DB cluster , you set up the foreign server.

To create the foreign server

Perform these tasks on the Aurora PostgreSQL DB cluster using an account that has `rds_superuser` privileges.

1. Create a foreign server in the Aurora PostgreSQL DB cluster:

   ```
   test=> CREATE SERVER sqlserverdb FOREIGN DATA WRAPPER tds_fdw OPTIONS
    (servername 'mssql2019.aws-region.rds.amazonaws.com', port '1433', database
    'tds_fdw_testing');
   CREATE SERVER
   ```

   To access non-ASCII data on the SQLServer side, create a server link with the character_set option in the Aurora PostgreSQL DB cluster:

   ```
   test=> CREATE SERVER sqlserverdb FOREIGN DATA WRAPPER tds_fdw OPTIONS (servername
    'mssql2019.aws-region.rds.amazonaws.com', port '1433', database 'tds_fdw_testing',
    character_set 'UTF-8');
   CREATE SERVER
   ```

2. Grant permissions to a user who doesn't have `rds_superuser` role privileges, for example, `user1`:

   ```
   test=> GRANT USAGE ON FOREIGN SERVER sqlserverdb TO user1;
   ```

3. Connect as user1 and create a mapping to a SQL Server user:

   ```
   test=> CREATE USER MAPPING FOR user1 SERVER sqlserverdb OPTIONS (username
    'sqlserveruser', password 'password');
   CREATE USER MAPPING
   ```

4. Create a foreign table linked to a SQL Server table:

   ```
   test=> CREATE FOREIGN TABLE mytab (a int) SERVER sqlserverdb OPTIONS (table
    'MYTABLE');
   CREATE FOREIGN TABLE
   ```

5. Query the foreign table:

   ```
   test=> SELECT * FROM mytab;
    a
   ```

```
---
 1
(1 row)
```

Using encryption in transit for the connection

The connection from Aurora PostgreSQL to SQL Server uses encryption in transit (TLS/SSL) depending on the SQL Server database configuration. If the SQL Server isn't configured for encryption, the RDS for PostgreSQL client making the request to the SQL Server database falls back to unencrypted.

You can enforce encryption for the connection to RDS for SQL Server DB instances by setting the `rds.force_ssl` parameter. To learn how, see Forcing connections to your DB instance to use SSL. For more information about SSL/TLS configuration for RDS for SQL Server, see Using SSL with a Microsoft SQL Server DB instance.

Working with Trusted Language Extensions for PostgreSQL

Trusted Language Extensions for PostgreSQL is an open source development kit for building PostgreSQL extensions. It allows you to build high performance PostgreSQL extensions and safely run them on your Aurora PostgreSQL DB cluster. By using Trusted Language Extensions (TLE) for PostgreSQL, you can create PostgreSQL extensions that follow the documented approach for extending PostgreSQL functionality. For more information, see Packaging Related Objects into an Extension in the PostgreSQL documentation.

One key benefit of TLE is that you can use it in environments that don't provide access to the file system underlying the PostgreSQL instance. Previously, installing a new extension required access to the file system. TLE removes this constraint. It provides a development environment for creating new extensions for any PostgreSQL database, including those running on your Aurora PostgreSQL DB clusters.

TLE is designed to prevent access to unsafe resources for the extensions that you create using TLE. Its runtime environment limits the impact of any extension defect to a single database connection. TLE also gives database administrators fine-grained control over who can install extensions, and it provides a permissions model for running them.

TLE is supported on Aurora PostgreSQL version 14.5 and higher versions.

The Trusted Language Extensions development environment and runtime are packaged as the `pg_tle` PostgreSQL extension, version 1.0.1. It supports creating extensions in JavaScript, Perl, Tcl, PL/pgSQL, and SQL. You install the `pg_tle` extension in your Aurora PostgreSQL DB cluster in the same way that you install other PostgreSQL extensions. After the `pg_tle` is set up, developers can use it to create new PostgreSQL extensions, known as *TLE extensions*.

In the following topics, you can find information about how to set up Trusted Language Extensions and how to get started creating your own TLE extensions.

Topics

- Creating TLE extensions for Aurora PostgreSQL
- Dropping your TLE extensions from a database
- Uninstalling Trusted Language Extensions for PostgreSQL
- Using PostgreSQL hooks with your TLE extensions
- Functions reference for Trusted Language Extensions for PostgreSQL
- Hooks reference for Trusted Language Extensions for PostgreSQL

Terminology

To help you better understand Trusted Language Extensions, view the following glossary for terms used in this topic.

Trusted Language Extensions for PostgreSQL

> *Trusted Language Extensions for PostgreSQL* is the official name of the open source development kit that's packaged as the `pg_tle` extension. It's available for use on any PostgreSQL system. For more information, see aws/pg_tle on GitHub.

Trusted Language Extensions

> *Trusted Language Extensions* is the short name for Trusted Language Extensions for PostgreSQL. This shortened name and its abbreviation (TLE) are also used in this documentation.

trusted language

> A *trusted language* is a programming or scripting language that has specific security attributes. For example, trusted languages typically restrict access to the file system, and they limit use of specified networking properties. The TLE development kit is designed to support trusted languages. PostgreSQL supports several different languages that are used to create trusted or untrusted extensions. For an example, see Trusted and Untrusted PL/Perl in the PostgreSQL documentation. When you create an extension using Trusted Language Extensions, the extension inherently uses trusted language mechanisms.

TLE extension

> A *TLE extension* is a PostgreSQL extension that's been created by using the Trusted Language Extensions (TLE) development kit.

Requirements for using Trusted Language Extensions for PostgreSQL

The following are requirements for setting up and using the TLE development kit.

- **Aurora PostgreSQL versions** – Trusted Language Extensions is supported on Aurora PostgreSQL version 14.5 and higher versions only.
  - If you need to upgrade your Aurora PostgreSQL DB cluster, see Upgrading Amazon Aurora PostgreSQL DB clusters.
  - If you don't yet have an Aurora DB cluster running PostgreSQL, you can create one. For more information, see Creating and connecting to an Aurora PostgreSQL DB cluster.
- **Requires `rds_superuser` privileges** – To set up and configure the `pg_tle` extension, your database user role must have the permissions of the `rds_superuser` role. By default, this role is granted to the `postgres` user that creates the Aurora PostgreSQL DB cluster.
- **Requires a custom DB parameter group** – Your Aurora PostgreSQL DB cluster must be configured with a custom DB parameter group. You use the custom DB parameter group for the writer instance of your Aurora PostgreSQL DB cluster.
  - If your Aurora PostgreSQL DB cluster isn't configured with a custom DB parameter group, you should create one and associate it with the writer instance of your Aurora PostgreSQL DB cluster. For a short summary of steps, see Creating and applying a custom DB parameter group.
  - If your Aurora PostgreSQL DB cluster is already configured using a custom DB parameter group, you can set up Trusted Language Extensions. For details, see Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

Creating and applying a custom DB parameter group

Use the following steps to create a custom DB parameter group and configure your Aurora PostgreSQL DB cluster to use it.

Console

To create a custom DB parameter group and use it with your Aurora PostgreSQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose Parameter groups from the Amazon RDS menu.

3. Choose **Create parameter group**.
4. In the **Parameter group details** page, enter the following information.

   - For **Parameter group family**, choose aurora-postgresql14.
   - For **Type**, choose DB Parameter Group.
   - For **Group name**, give your parameter group a meaningful name in the context of your operations.
   - For **Description**, enter a useful description so that others on your team can easily find it.
5. Choose **Create**. Your custom DB parameter group is created in your AWS Region. You can now modify your Aurora PostgreSQL DB cluster to use it by following the next steps.
6. Choose **Databases** from the Amazon RDS menu.
7. Choose the Aurora PostgreSQL DB cluster that you want to use with TLE from among those listed, and then choose **Modify.**
8. In the Modify DB cluster settings page, find **Database options** and use the selector to choose your custom DB parameter group.
9. Choose **Continue** to save the change.
10. Choose **Apply immediately** so that you can continue setting up the Aurora PostgreSQL DB cluster to use TLE.

To continue setting up your system for Trusted Language Extensions, see Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

For more information working with DB cluster and DB parameter groups, see Working with DB cluster parameter groups.

AWS CLI

You can avoid specifying the `--region` argument when you use CLI commands by configuring your AWS CLI with your default AWS Region. For more information, see Configuration basics in the *AWS Command Line Interface User Guide*.

To create a custom DB parameter group and use it with your Aurora PostgreSQL DB cluster

1. Use the create-db-parameter-group AWS CLI command to create a custom DB parameter group based on aurora-postgresql14 for your AWS Region. Note that in this step you create a DB parameter group to apply to the writer instance of your Aurora PostgreSQL DB cluster.

For Linux, macOS, or Unix:

```
aws rds create-db-parameter-group \
  --region aws-region \
  --db-parameter-group-name custom-params-for-pg-tle \
  --db-parameter-group-family aurora-postgresql14 \
  --description "My custom DB parameter group for Trusted Language Extensions"
```

For Windows:

```
aws rds create-db-parameter-group ^
  --region aws-region ^
  --db-parameter-group-name custom-params-for-pg-tle ^
  --db-parameter-group-family aurora-postgresql14 ^
  --description "My custom DB parameter group for Trusted Language Extensions"
```

Your custom DB parameter group is available in your AWS Region, so you can modify the writer instance of your Aurora PostgreSQL DB cluster to use it.

2. Use the modify-db-instance AWS CLI command to apply your custom DB parameter group to the writer instance of your Aurora PostgreSQL DB cluster. This command immediately reboots the active instance.

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
  --region aws-region \
  --db-instance-identifier your-writer-instance-name \
  --db-parameter-group-name custom-params-for-pg-tle \
  --apply-immediately
```

For Windows:

```
aws rds modify-db-instance ^
  --region aws-region ^
  --db-instance-identifier your-writer-instance-name ^
  --db-parameter-group-name custom-params-for-pg-tle ^
  --apply-immediately
```

To continue setting up your system for Trusted Language Extensions, see Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

For more information, see Working with DB parameter groups in a DB instance .

Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster

The following steps assume that your Aurora PostgreSQL DB cluster is associated with a custom DB cluster parameter group. You can use the AWS Management Console or the AWS CLI for these steps.

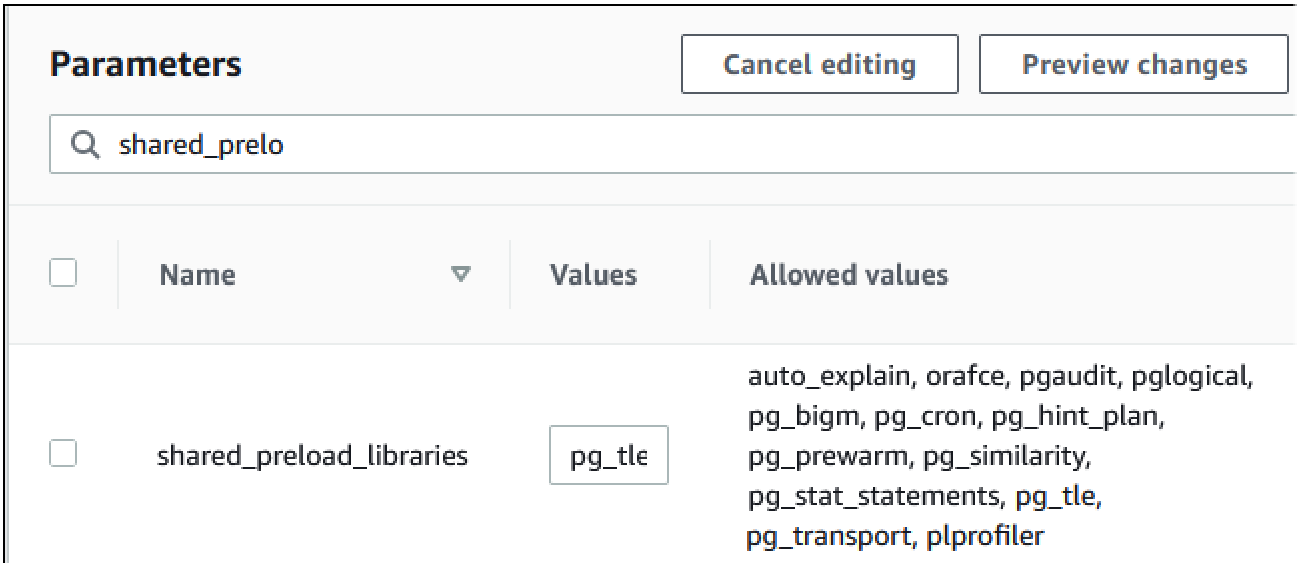
When you set up Trusted Language Extensions in your Aurora PostgreSQL DB cluster , you install it in a specific database for use by the database users who have permissions on that database.

Console

To set up Trusted Language Extensions

Perform the following steps using an account that's a member of the `rds_superuser` group (role).

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose your Aurora PostgreSQL DB cluster's Writer instance .
3. Open the **Configuration** tab for your Aurora PostgreSQL DB cluster writer instance. Among the Instance details, find the **Parameter group** link.
4. Choose the link to open the custom parameters associated with your Aurora PostgreSQL DB cluster.
5. In the **Parameters** search field, type `shared_pre` to find the `shared_preload_libraries` parameter.
6. Choose **Edit parameters** to access the property values.
7. Add `pg_tle` to the list in the **Values** field. Use a comma to separate items in the list of values.



□	Name	Values	Allowed values
□	shared_preload_libraries	pg_tle	auto_explain, orafce, pgaudit, pglogical, pg_bigm, pg_cron, pg_hint_plan, pg_prewarm, pg_similarity, pg_stat_statements, pg_tle, pg_transport, plprofiler

8. Reboot the writer instance of your Aurora PostgreSQL DB cluster so that your change to the `shared_preload_libraries` parameter takes effect.

9. When the instance is available, verify that `pg_tle` has been initialized. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

```
SHOW shared_preload_libraries;
shared_preload_libraries
--------------------------
rdsutils,pg_tle
(1 row)
```

10. With the `pg_tle` extension initialized, you can now create the extension.

```
CREATE EXTENSION pg_tle;
```

You can verify that the extension is installed by using the following `psql` metacommand.

```
labdb=> \dx
                         List of installed extensions
  Name   | Version |   Schema   |                Description
---------+---------+------------+--------------------------------------------
 pg_tle  | 1.0.1   | pgtle      | Trusted-Language Extensions for PostgreSQL
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
```

11. Grant the `pgtle_admin` role to the primary user name that you created for your Aurora PostgreSQL DB cluster when you set it up. If you accepted the default, it's `postgres`.

    ```
    labdb=> GRANT pgtle_admin TO postgres;
    GRANT ROLE
    ```

    You can verify that the grant has occurred by using the `psql` metacommand as shown in the following example. Only the `pgtle_admin` and `postgres` roles are shown in the output. For more information, see Understanding PostgreSQL roles and permissions.

    ```
    labdb=> \du
                                  List of roles
        Role name     |            Attributes             |               Member of
    -----------------+-----------------------------------
    +-----------------------------------
    pgtle_admin      | Cannot login                       | {}
    postgres         | Create role, Create DB            +| {rds_superuser,pgtle_admin}
                     | Password valid until infinity     |...
    ```

12. Close the `psql` session using the `\q` metacommand.

    ```
    \q
    ```

To get started creating TLE extensions, see Example: Creating a trusted language extension using SQL.

AWS CLI

You can avoid specifying the `--region` argument when you use CLI commands by configuring your AWS CLI with your default AWS Region. For more information, see Configuration basics in the *AWS Command Line Interface User Guide*.

To set up Trusted Language Extensions

1. Use the modify-db-parameter-group AWS CLI command to add `pg_tle` to the `shared_preload_libraries` parameter.

    ```
    aws rds modify-db-parameter-group \
       --db-parameter-group-name custom-param-group-name \
    ```

```
   --parameters
 "ParameterName=shared_preload_libraries,ParameterValue=pg_tle,ApplyMethod=pending-reboot" \
   --region aws-region
```

2. Use the reboot-db-instance AWS CLI command to reboot the writer instance of your Aurora PostgreSQL DB cluster and initialize the pg_tle library.

   ```
   aws rds reboot-db-instance \
       --db-instance-identifier writer-instance \
       --region aws-region
   ```

3. When the instance is available, you can verify that pg_tle has been initialized. Use psql to connect to the writer instance of your Aurora PostgreSQL DB cluster, and then run the following command.

   ```
   SHOW shared_preload_libraries;
   shared_preload_libraries
   --------------------------
   rdsutils,pg_tle
   (1 row)
   ```

   With pg_tle initialized, you can now create the extension.

   ```
   CREATE EXTENSION pg_tle;
   ```

4. Grant the pgtle_admin role to the primary user name that you created for your Aurora PostgreSQL DB cluster when you set it up. If you accepted the default, it's postgres.

   ```
   GRANT pgtle_admin TO postgres;
   GRANT ROLE
   ```

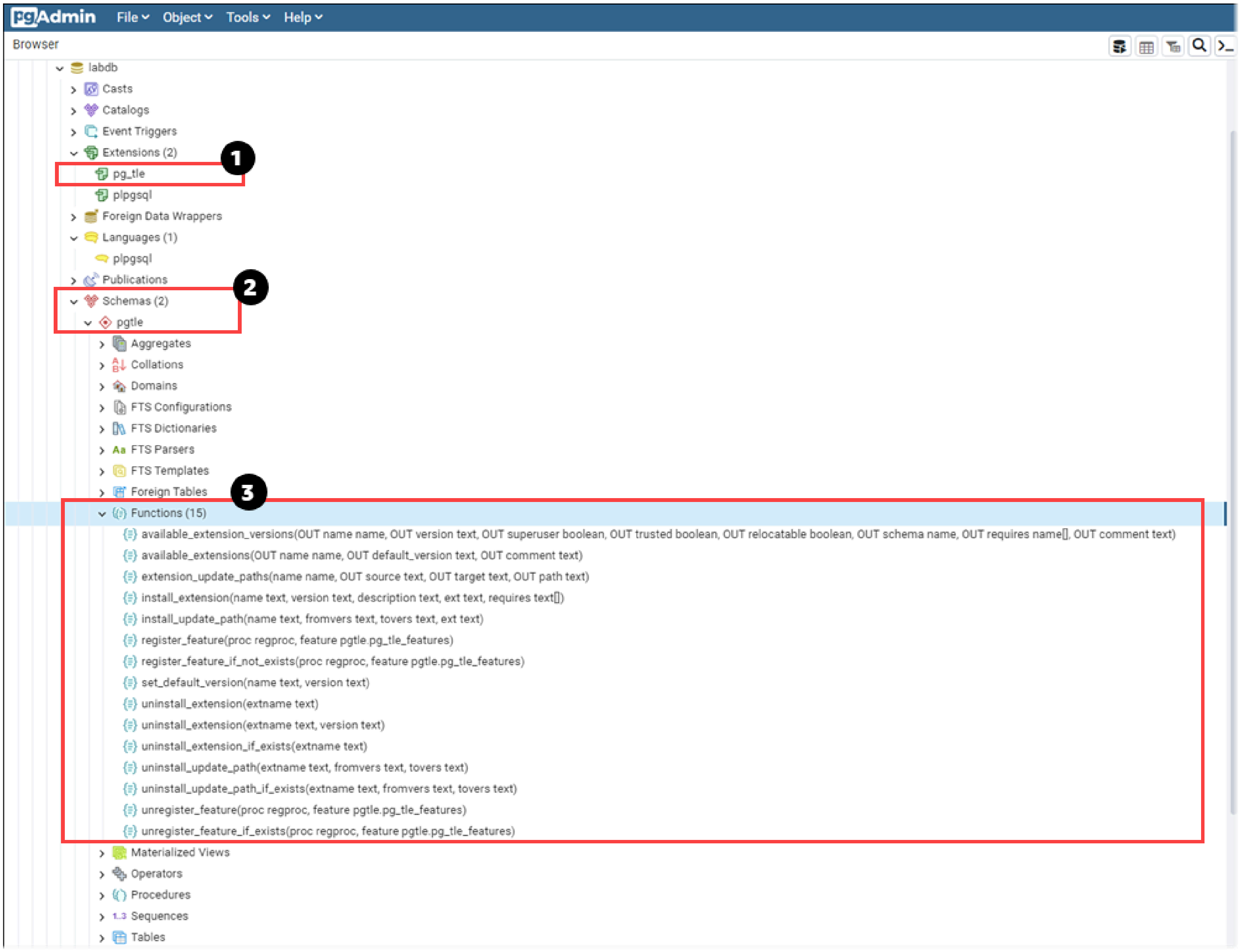
5. Close the psql session as follows.

   ```
   labdb=> \q
   ```

To get started creating TLE extensions, see Example: Creating a trusted language extension using SQL.

Overview of Trusted Language Extensions for PostgreSQL

Trusted Language Extensions for PostgreSQL is a PostgreSQL extension that you install in your Aurora PostgreSQL DB cluster in the same way that you set up other PostgreSQL extensions. In the following image of an example database in the pgAdmin client tool, you can view some of the components that comprise the `pg_tle` extension.



```
available_extension_versions(OUT name name, OUT version text, OUT superuser boolean, OUT trusted boolean, OUT relocatable boolean, OUT schema name, OUT requires name[], OUT comment text)
available_extensions(OUT name name, OUT default_version text, OUT comment text)
extension_update_paths(name name, OUT source text, OUT target text, OUT path text)
install_extension(name text, version text, description text, ext text, requires text[])
install_update_path(name text, fromvers text, tovers text, ext text)
register_feature(proc regproc, feature pgtle.pg_tle_features)
register_feature_if_not_exists(proc regproc, feature pgtle.pg_tle_features)
set_default_version(name text, version text)
uninstall_extension(extname text)
uninstall_extension(extname text, version text)
uninstall_extension_if_exists(extname text)
uninstall_update_path(extname text, fromvers text, tovers text)
uninstall_update_path_if_exists(extname text, fromvers text, tovers text)
unregister_feature(proc regproc, feature pgtle.pg_tle_features)
unregister_feature_if_exists(proc regproc, feature pgtle.pg_tle_features)
```

You can see the following details.

1. The Trusted Language Extensions (TLE) for PostgreSQL development kit is packaged as the `pg_tle` extension. As such, `pg_tle` is added to the available extensions for the database in which it's installed.
2. TLE has its own schema, `pgtle`. This schema contains helper functions (3) for installing and managing the extensions that you create.

3. TLE provides over a dozen helper functions for installing, registering, and managing your extensions. To learn more about these functions, see Functions reference for Trusted Language Extensions for PostgreSQL.

Other components of the pg_tle extension include the following:

- **The pgtle_admin role** – The pgtle_admin role is created when the pg_tle extension is installed. This role is privileged and should be treated as such. We strongly recommend that you follow the principle of *least privilege* when granting the pgtle_admin role to database users. In other words, grant the pgtle_admin role only to database users that are allowed to create, install, and manage new TLE extensions, such as postgres.
- **The pgtle.feature_info table** – The pgtle.feature_info table is a protected table that contains information about your TLEs, hooks, and the custom stored procedures and functions that they use. If you have pgtle_admin privileges, you use the following Trusted Language Extensions functions to add and update that information in the table.
  - pgtle.register_feature
  - pgtle.register_feature_if_not_exists
  - pgtle.unregister_feature
  - pgtle.unregister_feature_if_exists

Creating TLE extensions for Aurora PostgreSQL

You can install any extensions that you create with TLE in any Aurora PostgreSQL DB cluster that has the pg_tle extension installed. The pg_tle extension is scoped to the PostgreSQL database in which it's installed. The extensions that you create using TLE are scoped to the same database.

Use the various pgtle functions to install the code that makes up your TLE extension. The following Trusted Language Extensions functions all require the pgtle_admin role.

- pgtle.install_extension
- pgtle.install_update_path
- pgtle.register_feature
- pgtle.register_feature_if_not_exists
- pgtle.set_default_version
- pgtle.uninstall_extension(name)

- pgtle.uninstall_extension(name, version)
- pgtle.uninstall_extension_if_exists
- pgtle.uninstall_update_path
- pgtle.uninstall_update_path_if_exists
- pgtle.unregister_feature
- pgtle.unregister_feature_if_exists

Example: Creating a trusted language extension using SQL

The following example shows you how to create a TLE extension named `pg_distance` that contains a few SQL functions for calculating distances using different formulas. In the listing, you can find the function for calculating the Manhattan distance and the function for calculating the Euclidean distance. For more information about the difference between these formulas, see Taxicab geometry and Euclidean geometry in Wikipedia.

You can use this example in your own Aurora PostgreSQL DB cluster if you have the `pg_tle` extension set up as detailed in Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

> **Note**
>
> You need to have the privileges of the `pgtle_admin` role to follow this procedure.

To create the example TLE extension

The following steps use an example database named `labdb`. This database is owned by the `postgres` primary user. The `postgres` role also has the permissions of the `pgtle_admin` role.

1. Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster.

```
psql --host=db-instance-123456789012.aws-region.rds.amazonaws.com
--port=5432 --username=postgres --password --dbname=labdb
```

2. Create a TLE extension named `pg_distance` by copying the following code and pasting it into your `psql` session console.

```
SELECT pgtle.install_extension
```

```
(
 'pg_distance',
 '0.1',
  'Distance functions for two points',
$_pg_tle_$
    CREATE FUNCTION dist(x1 float8, y1 float8, x2 float8, y2 float8, norm int)
    RETURNS float8
    AS $$
      SELECT (abs(x2 - x1) ^ norm + abs(y2 - y1) ^ norm) ^ (1::float8 / norm);
    $$ LANGUAGE SQL;

    CREATE FUNCTION manhattan_dist(x1 float8, y1 float8, x2 float8, y2 float8)
    RETURNS float8
    AS $$
      SELECT dist(x1, y1, x2, y2, 1);
    $$ LANGUAGE SQL;

    CREATE FUNCTION euclidean_dist(x1 float8, y1 float8, x2 float8, y2 float8)
    RETURNS float8
    AS $$
      SELECT dist(x1, y1, x2, y2, 2);
    $$ LANGUAGE SQL;
$_pg_tle_$
);
```

You see the output, such as the following.

```
install_extension
---------------
 t
(1 row)
```

The artifacts that make up the `pg_distance` extension are now installed in your database. These artifacts include the control file and the code for the extension, which are items that need to be present so that the extension can be created using the `CREATE EXTENSION` command. In other words, you still need to create the extension to make its functions available to database users.

3. To create the extension, use the `CREATE EXTENSION` command as you do for any other extension. As with other extensions, the database user needs to have the `CREATE` permissions in the database.

```
CREATE EXTENSION pg_distance;
```

4. To test the pg_distance TLE extension, you can use it to calculate the Manhattan distance between four points.

   ```
   labdb=> SELECT manhattan_dist(1, 1, 5, 5);
   8
   ```

   To calculate the Euclidean distance between the same set of points, you can use the following.

   ```
   labdb=> SELECT euclidean_dist(1, 1, 5, 5);
   5.656854249492381
   ```

The pg_distance extension loads the functions in the database and makes them available to any users with permissions on the database.

Modifying your TLE extension

To improve query performance for the functions packaged in this TLE extension, add the following two PostgreSQL attributes to their specifications.

- IMMUTABLE – The IMMUTABLE attribute ensures that the query optimizer can use optimizations to improve query response times. For more information, see Function Volatility Categories in the PostgreSQL documentation.
- PARALLEL SAFE – The PARALLEL SAFE attribute is another attribute that allows PostgreSQL to run the function in parallel mode. For more information, see CREATE FUNCTION in the PostgreSQL documentation.

In the following example, you can see how the pgtle.install_update_path function is used to add these attributes to each function to create a version 0.2 of the pg_distance TLE extension. For more information about this function, see pgtle.install_update_path. You need to have the pgtle_admin role to perform this task.

To update an existing TLE extension and specify the default version

1. Connect to the writer instance of your Aurora PostgreSQL DB cluster using psql or another client tool, such as pgAdmin.

```
psql --host=db-instance-123456789012.aws-region.rds.amazonaws.com
--port=5432 --username=postgres --password --dbname=labdb
```

2. Modify the existing TLE extension by copying the following code and pasting it into your `psql` session console.

```
SELECT pgtle.install_update_path
(
 'pg_distance',
 '0.1',
 '0.2',
$_pg_tle_$
    CREATE OR REPLACE FUNCTION dist(x1 float8, y1 float8, x2 float8, y2 float8,
 norm int)
    RETURNS float8
    AS $$
      SELECT (abs(x2 - x1) ^ norm + abs(y2 - y1) ^ norm) ^ (1::float8 / norm);
    $$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;

    CREATE OR REPLACE FUNCTION manhattan_dist(x1 float8, y1 float8, x2 float8, y2
 float8)
    RETURNS float8
    AS $$
      SELECT dist(x1, y1, x2, y2, 1);
    $$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;

    CREATE OR REPLACE FUNCTION euclidean_dist(x1 float8, y1 float8, x2 float8, y2
 float8)
    RETURNS float8
    AS $$
      SELECT dist(x1, y1, x2, y2, 2);
    $$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;
$_pg_tle_$
);
```

You see a response similar to the following.

```
install_update_path
---------------------
 t
(1 row)
```

You can make this version of the extension the default version, so that database users don't have to specify a version when they create or update the extension in their database.

3. To specify that the modified version (version 0.2) of your TLE extension is the default version, use the `pgtle.set_default_version` function as shown in the following example.

   ```
   SELECT pgtle.set_default_version('pg_distance', '0.2');
   ```

   For more information about this function, see pgtle.set_default_version.

4. With the code in place, you can update the installed TLE extension in the usual way, by using `ALTER EXTENSION ... UPDATE` command, as shown here:

   ```
   ALTER EXTENSION pg_distance UPDATE;
   ```

Dropping your TLE extensions from a database

You can drop your TLE extensions by using the `DROP EXTENSION` command in the same way that you do for other PostgreSQL extensions. Dropping the extension doesn't remove the installation files that make up the extension, which allows users to re-create the extension. To remove the extension and its installation files, do the following two-step process.

To drop the TLE extension and remove its installation files

1. Use `psql` or another client tool to connect to the writer instance of your Aurora PostgreSQL DB cluster.

   ```
   psql --host=cluster-instance-1.111122223333.aws-region.rds.amazonaws.com --
   port=5432 --username=postgres --password --dbname=dbname
   ```

2. Drop the extension as you would any PostgreSQL extension.

   ```
   DROP EXTENSION your-TLE-extension
   ```

   For example, if you create the `pg_distance` extension as detailed in Example: Creating a trusted language extension using SQL, you can drop the extension as follows.

   ```
   DROP EXTENSION pg_distance;
   ```

You see output confirming that the extension has been dropped, as follows.

```
DROP EXTENSION
```

At this point, the extension is no longer active in the database. However, its installation files and control file are still available in the database, so database users can create the extension again if they like.

- If you want to leave the extension files intact so that database users can create your TLE extension, you can stop here.
- If you want to remove all files that make up the extension, continue to the next step.

3. To remove all installation files for your extension, use the `pgtle.uninstall_extension` function. This function removes all the code and control files for your extension.

```
SELECT pgtle.uninstall_extension('your-tle-extension-name');
```

For example, to remove all `pg_distance` installation files, use the following command.

```
SELECT pgtle.uninstall_extension('pg_distance');
 uninstall_extension
---------------------
 t
(1 row)
```

Uninstalling Trusted Language Extensions for PostgreSQL

If you no longer want to create your own TLE extensions using TLE, you can drop the `pg_tle` extension and remove all artifacts. This action includes dropping any TLE extensions in the database and dropping the `pgtle` schema.

To drop the `pg_tle` extension and its schema from a database

1. Use `psql` or another client tool to connect to the writer instance of your Aurora PostgreSQL DB cluster.

```
psql --host=cluster-instance-1.111122223333.aws-region.rds.amazonaws.com --
port=5432 --username=postgres --password --dbname=dbname
```

2. Drop the `pg_tle` extension from the database. If the database has your own TLE extensions still running in the database, you need to also drop those extensions. To do so, you can use the `CASCADE` keyword, as shown in the following.

   ```
   DROP EXTENSION pg_tle CASCADE;
   ```

   If the `pg_tle` extension isn't still active in the database, you don't need to use the `CASCADE` keyword.

3. Drop the `pgtle` schema. This action removes all the management functions from the database.

   ```
   DROP SCHEMA pgtle CASCADE;
   ```

   The command returns the following when the process completes.

   ```
   DROP SCHEMA
   ```

   The `pg_tle` extension, its schema and functions, and all artifacts are removed. To create new extensions using TLE, go through the setup process again. For more information, see Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

Using PostgreSQL hooks with your TLE extensions

A *hook* is a callback mechanism available in PostgreSQL that allows developers to call custom functions or other routines during regular database operations. The TLE development kit supports PostgreSQL hooks so that you can integrate custom functions with PostgreSQL behavior at runtime. For example, you can use a hook to associate the authentication process with your own custom code, or to modify the query planning and execution process for your specific needs.

Your TLE extensions can use hooks. If a hook is global in scope, it applies across all databases. Therefore, if your TLE extension uses a global hook, then you need to create your TLE extension in all databases that your users can access.

When you use the pg_tle extension to build your own Trusted Language Extensions, you can use the available hooks from a SQL API to build out the functions of your extension. You should register any hooks with pg_tle. For some hooks, you might also need to set various configuration parameters. For example, the passcode check hook can be set to on, off, or require. For more information about specific requirements for available pg_tle hooks, see Hooks reference for Trusted Language Extensions for PostgreSQL.

Example: Creating an extension that uses a PostgreSQL hook

The example discussed in this section uses a PostgreSQL hook to check the password provided during specific SQL operations and prevents database users from setting their passwords to any of those contained in the password_check.bad_passwords table. The table contains the top-ten most commonly used, but easily breakable choices for passwords.

To set up this example in your Aurora PostgreSQL DB cluster, you must have already installed Trusted Language Extensions. For details, see Setting up Trusted Language Extensions in your Aurora PostgreSQL DB cluster.

To set up the password-check hook example

1. Use psql to connect to the writer instance of your Aurora PostgreSQL DB cluster.

```
psql --host=db-instance-123456789012.aws-region.rds.amazonaws.com
--port=5432 --username=postgres --password --dbname=labdb
```

2. Copy the code from the Password-check hook code listing and paste it into your database.

```
SELECT pgtle.install_extension (
  'my_password_check_rules',
  '1.0',
  'Do not let users use the 10 most commonly used passwords',
$_pgtle_$
  CREATE SCHEMA password_check;
  REVOKE ALL ON SCHEMA password_check FROM PUBLIC;
  GRANT USAGE ON SCHEMA password_check TO PUBLIC;

  CREATE TABLE password_check.bad_passwords (plaintext) AS
  VALUES
    ('123456'),
    ('password'),
    ('12345678'),
    ('qwerty'),
    ('123456789'),
    ('12345'),
    ('1234'),
    ('111111'),
    ('1234567'),
    ('dragon');
  CREATE UNIQUE INDEX ON password_check.bad_passwords (plaintext);

  CREATE FUNCTION password_check.passcheck_hook(username text, password text,
 password_type pgtle.password_types, valid_until timestamptz, valid_null boolean)
  RETURNS void AS $$
    DECLARE
      invalid bool := false;
    BEGIN
      IF password_type = 'PASSWORD_TYPE_MD5' THEN
        SELECT EXISTS(
          SELECT 1
          FROM password_check.bad_passwords bp
          WHERE ('md5' || md5(bp.plaintext || username)) = password
        ) INTO invalid;
        IF invalid THEN
          RAISE EXCEPTION 'Cannot use passwords from the common password
 dictionary';
        END IF;
      ELSIF password_type = 'PASSWORD_TYPE_PLAINTEXT' THEN
        SELECT EXISTS(
          SELECT 1
          FROM password_check.bad_passwords bp
          WHERE bp.plaintext = password
        ) INTO invalid;
        IF invalid THEN
          RAISE EXCEPTION 'Cannot use passwords from the common common password
 dictionary';
        END IF;
      END IF;
    END
  $$ LANGUAGE plpgsql SECURITY DEFINER;

  GRANT EXECUTE ON FUNCTION password_check.passcheck_hook TO PUBLIC;

  SELECT pgtle.register_feature('password_check.passcheck_hook', 'passcheck');
$_pgtle_$
```

```
);
```

When the extension has been loaded into your database, you see the output such as the following.

```
 install_extension
-------------------
 t
(1 row)
```

3. While still connected to the database, you can now create the extension.

```
CREATE EXTENSION my_password_check_rules;
```

4. You can confirm that the extension has been created in the database by using the following psql metacommand.

```
\dx
                        List of installed extensions
          Name           | Version |   Schema   |
 Description
-------------------------+---------+------------
+-------------------------------------------------------------
 my_password_check_rules | 1.0     | public     | Prevent use of any of the top-ten
 most common bad passwords
 pg_tle                  | 1.0.1   | pgtle      | Trusted-Language Extensions for
 PostgreSQL
 plpgsql                 | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

5. Open another terminal session to work with the AWS CLI. You need to modify your custom DB parameter group to turn on the password-check hook. To do so, use the modify-db-parameter-group CLI command as shown in the following example.

```
aws rds modify-db-parameter-group \
    --region aws-region \
    --db-parameter-group-name your-custom-parameter-group \
    --parameters
 "ParameterName=pgtle.enable_password_check,ParameterValue=on,ApplyMethod=immediate"
```

It might take a few minutes for the change to the parameter group setting to take effect. This parameter is dynamic, however, so you don't need to restart the writer instance of the Aurora PostgreSQL DB cluster for the setting to take effect.

6. Open the `psql` session and query the database to verify that the password_check hook has been turned on.

```
labdb=> SHOW pgtle.enable_password_check;
pgtle.enable_password_check
-----------------------------
on
(1 row)
```

The password-check hook is now active. You can test it by creating a new role and using one of the bad passwords, as shown in the following example.

```
CREATE ROLE test_role PASSWORD 'password';
ERROR:  Cannot use passwords from the common password dictionary
CONTEXT:  PL/pgSQL function
 password_check.passcheck_hook(text,text,pgtle.password_types,timestamp with time
 zone,boolean) line 21 at RAISE
SQL statement "SELECT password_check.passcheck_hook(
    $1::pg_catalog.text,
    $2::pg_catalog.text,
    $3::pgtle.password_types,
    $4::pg_catalog.timestamptz,
    $5::pg_catalog.bool)"
```

The output has been formatted for readability.

The following example shows that `pgsql` interactive metacommand `\password` behavior is also affected by the password_check hook.

```
postgres=> SET password_encryption TO 'md5';
SET
postgres=> \password
Enter new password for user "postgres":*****
Enter it again:*****
ERROR:  Cannot use passwords from the common password dictionary
```

```
CONTEXT:  PL/pgSQL function
 password_check.passcheck_hook(text,text,pgtle.password_types,timestamp with time
 zone,boolean) line 12 at RAISE
SQL statement "SELECT password_check.passcheck_hook($1::pg_catalog.text,
 $2::pg_catalog.text, $3::pgtle.password_types, $4::pg_catalog.timestamptz,
 $5::pg_catalog.bool)"
```

You can drop this TLE extension and uninstall its source files if you want. For more information, see Dropping your TLE extensions from a database.

Password-check hook code listing

The example code shown here defines the specification for the `my_password_check_rules` TLE extension. When you copy this code and paste it into your database, the code for the `my_password_check_rules` extension is loaded into the database, and the `password_check` hook is registered for use by the extension.

```
SELECT pgtle.install_extension (
  'my_password_check_rules',
  '1.0',
  'Do not let users use the 10 most commonly used passwords',
$_pgtle_$
  CREATE SCHEMA password_check;
  REVOKE ALL ON SCHEMA password_check FROM PUBLIC;
  GRANT USAGE ON SCHEMA password_check TO PUBLIC;

  CREATE TABLE password_check.bad_passwords (plaintext) AS
  VALUES
    ('123456'),
    ('password'),
    ('12345678'),
    ('qwerty'),
    ('123456789'),
    ('12345'),
    ('1234'),
    ('111111'),
    ('1234567'),
    ('dragon');
  CREATE UNIQUE INDEX ON password_check.bad_passwords (plaintext);

  CREATE FUNCTION password_check.passcheck_hook(username text, password text,
 password_type pgtle.password_types, valid_until timestamptz, valid_null boolean)
  RETURNS void AS $$
```

```
    DECLARE
      invalid bool := false;
    BEGIN
      IF password_type = 'PASSWORD_TYPE_MD5' THEN
        SELECT EXISTS(
          SELECT 1
          FROM password_check.bad_passwords bp
          WHERE ('md5' || md5(bp.plaintext || username)) = password
        ) INTO invalid;
        IF invalid THEN
          RAISE EXCEPTION 'Cannot use passwords from the common password dictionary';
        END IF;
      ELSIF password_type = 'PASSWORD_TYPE_PLAINTEXT' THEN
        SELECT EXISTS(
          SELECT 1
          FROM password_check.bad_passwords bp
          WHERE bp.plaintext = password
        ) INTO invalid;
        IF invalid THEN
          RAISE EXCEPTION 'Cannot use passwords from the common common password
 dictionary';
        END IF;
      END IF;
    END
  $$ LANGUAGE plpgsql SECURITY DEFINER;

  GRANT EXECUTE ON FUNCTION password_check.passcheck_hook TO PUBLIC;

  SELECT pgtle.register_feature('password_check.passcheck_hook', 'passcheck');
$_pgtle_$
);
```

Functions reference for Trusted Language Extensions for PostgreSQL

View the following reference documentation about functions available in Trusted Language Extensions for PostgreSQL. Use these functions to install, register, update, and manage your *TLE extensions*, that is, the PostgreSQL extensions that you develop using the Trusted Language Extensions development kit.

Topics

- pgtle.available_extensions
- pgtle.available_extension_versions

- pgtle.extension_update_paths
- pgtle.install_extension
- pgtle.install_update_path
- pgtle.register_feature
- pgtle.register_feature_if_not_exists
- pgtle.set_default_version
- pgtle.uninstall_extension(name)
- pgtle.uninstall_extension(name, version)
- pgtle.uninstall_extension_if_exists
- pgtle.uninstall_update_path
- pgtle.uninstall_update_path_if_exists
- pgtle.unregister_feature
- pgtle.unregister_feature_if_exists

pgtle.available_extensions

The `pgtle.available_extensions` function is a set-returning function. It returns all available TLE extensions in the database. Each returned row contains information about a single TLE extension.

Function prototype

```
pgtle.available_extensions()
```

Role

None.

Arguments

None.

Output

- `name` – The name of the TLE extension.

- `default_version` – The version of the TLE extension to use when `CREATE EXTENSION` is called without a version specified.
- `description` – A more detailed description about the TLE extension.

Usage example

```
SELECT * FROM pgtle.available_extensions();
```

pgtle.available_extension_versions

The `available_extension_versions` function is a set-returning function. It returns a list of all available TLE extensions and their versions. Each row contains information about a specific version of the given TLE extension, including whether it requires a specific role.

Function prototype

```
pgtle.available_extension_versions()
```

Role

None.

Arguments

None.

Output

- `name` – The name of the TLE extension.
- `version` – The version of the TLE extension.
- `superuser` – This value is always `false` for your TLE extensions. The permissions needed to create the TLE extension or update it are the same as for creating other objects in the given database.
- `trusted` – This value is always `false` for a TLE extension.
- `relocatable` – This value is always `false` for a TLE extension.
- `schema` – Specifies the name of the schema in which the TLE extension is installed.
- `requires` – An array containing the names of other extensions needed by this TLE extension.
- `description` – A detailed description of the TLE extension.

For more information about output values, see Packaging Related Objects into an Extension > Extension Files in the PostgreSQL documentation.

Usage example

```
SELECT * FROM pgtle.available_extension_versions();
```

pgtle.extension_update_paths

The `extension_update_paths` function is a set-returning function. It returns a list of all the possible update paths for a TLE extension. Each row includes the available upgrades or downgrades for that TLE extension.

Function prototype

```
pgtle.extension_update_paths(name)
```

Role

None.

Arguments

`name` – The name of the TLE extension from which to get upgrade paths.

Output

- `source` – The source version for an update.
- `target` – The target version for an update.
- `path` – The upgrade path used to update a TLE extension from `source` version to `target` version, for example, `0.1--0.2`.

Usage example

```
SELECT * FROM pgtle.extension_update_paths('your-TLE');
```

pgtle.install_extension

The `install_extension` function lets you install the artifacts that make up your TLE extension in the database, after which it can be created using the `CREATE EXTENSION` command.

Function prototype

```
pgtle.install_extension(name text, version text, description text, ext text, requires
 text[] DEFAULT NULL::text[])
```

Role

None.

Arguments

- `name` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.
- `version` – The version of the TLE extension.
- `description` – A detailed description about the TLE extension. This description is displayed in the `comment` field in `pgtle.available_extensions()`.
- `ext` – The contents of the TLE extension. This value contains objects such as functions.
- `requires` – An optional parameter that specifies dependencies for this TLE extension. The `pg_tle` extension is automatically added as a dependency.

Many of these arguments are the same as those that are included in an extension control file for installing a PostgreSQL extension on the file system of a PostgreSQL instance. For more information, see the Extension Files in Packaging Related Objects into an Extension in the PostgreSQL documentation.

Output

This functions returns `OK` on success and `NULL` on error.

- `OK` – The TLE extension has been successfully installed in the database.
- `NULL` – The TLE extension hasn't been successfully installed in the database.

Usage example

```
SELECT pgtle.install_extension(
 'pg_tle_test',
 '0.1',
 'My first pg_tle extension',
$_pgtle_$
```

```
  CREATE FUNCTION my_test()
  RETURNS INT
  AS $$
    SELECT 42;
  $$ LANGUAGE SQL IMMUTABLE;
$_pgtle_$
);
```

pgtle.install_update_path

The `install_update_path` function provides an update path between two different versions of a TLE extension. This function allows users of your TLE extension to update its version by using the `ALTER EXTENSION ... UPDATE` syntax.

Function prototype

```
pgtle.install_update_path(name text, fromvers text, tovers text, ext text)
```

Role

`pgtle_admin`

Arguments

- `name` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.
- `fromvers` – The source version of the TLE extension for the upgrade.
- `tovers` – The destination version of the TLE extension for the upgrade.
- `ext` – The contents of the update. This value contains objects such as functions.

Output

None.

Usage example

```
SELECT pgtle.install_update_path('pg_tle_test', '0.1', '0.2',
  $_pgtle_$
    CREATE OR REPLACE FUNCTION my_test()
    RETURNS INT
    AS $$
      SELECT 21;
    $$ LANGUAGE SQL IMMUTABLE;
  $_pgtle_$
);
```

pgtle.register_feature

The register_feature function adds the specified internal PostgreSQL feature to the pgtle.feature_info table. PostgreSQL hooks are an example of an internal PostgreSQL feature. The Trusted Language Extensions development kit supports the use of PostgreSQL hooks. Currently, this function supports the following feature.

- passcheck – Registers the password-check hook with your procedure or function that customizes PostgreSQL's password-check behavior.

Function prototype

```
pgtle.register_feature(proc regproc, feature pg_tle_feature)
```

Role

pgtle_admin

Arguments

- proc – The name of a stored procedure or function to use for the feature.
- feature – The name of the pg_tle feature (such as passcheck) to register with the function.

Output

None.

Usage example

```
SELECT pgtle.register_feature('pw_hook', 'passcheck');
```

pgtle.register_feature_if_not_exists

The pgtle.register_feature_if_not_exists function adds the specified PostgreSQL feature to the pgtle.feature_info table and identifies the TLE extension or other procedure or function that uses the feature. For more information about hooks and Trusted Language Extensions, see Using PostgreSQL hooks with your TLE extensions.

Function prototype

```
pgtle.register_feature_if_not_exists(proc regproc, feature pg_tle_feature)
```

Role

`pgtle_admin`

Arguments

- `proc` – The name of a stored procedure or function that contains the logic (code) to use as a feature for your TLE extension. For example, the `pw_hook` code.
- `feature` – The name of the PostgreSQL feature to register for the TLE function. Currently, the only available feature is the `passcheck` hook. For more information, see Password-check hook (passcheck).

Output

Returns `true` after registering the feature for the specified extension. Returns `false` if the feature is already registered.

Usage example

```
SELECT pgtle.register_feature_if_not_exists('pw_hook', 'passcheck');
```

pgtle.set_default_version

The `set_default_version` function lets you specify a `default_version` for your TLE extension. You can use this function to define an upgrade path and designate the version as the default for your TLE extension. When database users specify your TLE extension in the `CREATE EXTENSION` and `ALTER EXTENSION ... UPDATE` commands, that version of your TLE extension is created in the database for that user.

This function returns `true` on success. If the TLE extension specified in the `name` argument doesn't exist, the function returns an error. Similarly, if the `version` of the TLE extension doesn't exist, it returns an error.

Function prototype

```
pgtle.set_default_version(name text, version text)
```

Role

`pgtle_admin`

Arguments

- `name` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.
- `version` – The version of the TLE extension to set the default.

Output

- `true` – When setting default version succeeds, the function returns `true`.
- `ERROR` – Returns an error message if a TLE extension with the specified name or version doesn't exist.

Usage example

```
SELECT * FROM pgtle.set_default_version('my-extension', '1.1');
```

pgtle.uninstall_extension(name)

The `uninstall_extension` function removes all versions of a TLE extension from a database. This function prevents future calls of `CREATE EXTENSION` from installing the TLE extension. If the TLE extension doesn't exist in the database, an error is raised.

The `uninstall_extension` function won't drop a TLE extension that's currently active in the database. To remove a TLE extension that's currently active, you need to explicitly call `DROP EXTENSION` to remove it.

Function prototype

```
pgtle.uninstall_extension(extname text)
```

Role

pgtle_admin

Arguments

- extname – The name of the TLE extension to uninstall. This name is the same as the one used with CREATE EXTENSION to load the TLE extension for use in a given database.

Output

None.

Usage example

```
SELECT * FROM pgtle.uninstall_extension('pg_tle_test');
```

pgtle.uninstall_extension(name, version)

The uninstall_extension(name, version) function removes the specified version of the TLE extension from the database. This function prevents CREATE EXTENSION and ALTER EXTENSION from installing or updating a TLE extension to the specified version. This function also removes all update paths for the specified version of the TLE extension. This function won't uninstall the TLE extension if it's currently active in the database. You must explicitly call DROP EXTENSION to remove the TLE extension. To uninstall all versions of a TLE extension, see pgtle.uninstall_extension(name).

Function prototype

```
pgtle.uninstall_extension(extname text, version text)
```

Role

pgtle_admin

Arguments

- extname – The name of the TLE extension. This value is used when calling CREATE EXTENSION.
- version – The version of the TLE extension to uninstall from the database.

Output

None.

Usage example

```
SELECT * FROM pgtle.uninstall_extension('pg_tle_test', '0.2');
```

pgtle.uninstall_extension_if_exists

The `uninstall_extension_if_exists` function removes all versions of a TLE extension from a given database. If the TLE extension doesn't exist, the function returns silently (no error message is raised). If the specified extension is currently active within a database, this function doesn't drop it. You must explicitly call `DROP EXTENSION` to remove the TLE extension before using this function to uninstall its artifacts.

Function prototype

```
pgtle.uninstall_extension_if_exists(extname text)
```

Role

`pgtle_admin`

Arguments

- `extname` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.

Output

The `uninstall_extension_if_exists` function returns `true` after uninstalling the specified extension. If the specified extension doesn't exist, the function returns `false`.

- `true` – Returns `true` after uninstalling the TLE extension.
- `false` – Returns `false` when the TLE extension doesn't exist in the database.

Usage example

```
SELECT * FROM pgtle.uninstall_extension_if_exists('pg_tle_test');
```

pgtle.uninstall_update_path

The `uninstall_update_path` function removes the specific update path from a TLE extension. This prevents `ALTER EXTENSION ... UPDATE TO` from using this as an update path.

If the TLE extension is currently being used by one of the versions on this update path, it remains in the database.

If the update path specified doesn't exist, this function raises an error.

Function prototype

```
pgtle.uninstall_update_path(extname text, fromvers text, tovers text)
```

Role

`pgtle_admin`

Arguments

- `extname` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.
- `fromvers` – The source version of the TLE extension used on the update path.
- `tovers` – The destination version of the TLE extension used on the update path.

Output

None.

Usage example

```
SELECT * FROM pgtle.uninstall_update_path('pg_tle_test', '0.1', '0.2');
```

pgtle.uninstall_update_path_if_exists

The `uninstall_update_path_if_exists` function is similar to `uninstall_update_path` in that it removes the specified update path from a TLE extension. However, if the update path doesn't exist, this function doesn't raise an error message. Instead, the function returns `false`.

Function prototype

```
pgtle.uninstall_update_path_if_exists(extname text, fromvers text, tovers text)
```

Role

`pgtle_admin`

Arguments

- `extname` – The name of the TLE extension. This value is used when calling `CREATE EXTENSION`.
- `fromvers` – The source version of the TLE extension used on the update path.
- `tovers` – The destination version of the TLE extension used on the update path.

Output

- `true` – The function has successfully updated the path for the TLE extension.
- `false` – The function wasn't able to update the path for the TLE extension.

Usage example

```
SELECT * FROM pgtle.uninstall_update_path_if_exists('pg_tle_test', '0.1', '0.2');
```

pgtle.unregister_feature

The `unregister_feature` function provides a way to remove functions that were registered to use `pg_tle` features, such as hooks. For information about registering a feature, see pgtle.register_feature.

Function prototype

```
pgtle.unregister_feature(proc regproc, feature pg_tle_features)
```

Role

`pgtle_admin`

Arguments

- `proc` – The name of a stored function to register with a `pg_tle` feature.
- `feature` – The name of the `pg_tle` feature to register with the function. For example, `passcheck` is a feature that can be registered for use by the trusted language extensions that you develop. For more information, see Password-check hook (passcheck).

Output

None.

Usage example

```
SELECT * FROM pgtle.unregister_feature('pw_hook', 'passcheck');
```

pgtle.unregister_feature_if_exists

The `unregister_feature` function provides a way to remove functions that were registered to use `pg_tle` features, such as hooks. For more information, see Using PostgreSQL hooks with your TLE extensions. Returns `true` after successfully unregistering the feature. Returns `false` if the feature wasn't registered.

For information about registering `pg_tle` features for your TLE extensions, see pgtle.register_feature.

Function prototype

```
pgtle.unregister_feature_if_exists('proc regproc', 'feature pg_tle_features')
```

Role

`pgtle_admin`

Arguments

- `proc` – The name of the stored function that was registered to include a `pg_tle` feature.
- `feature` – The name of the `pg_tle` feature that was registered with the trusted language extension.

Output

Returns `true` or `false`, as follows.

- `true` – The function has successfully unregistered the feature from extension.
- `false` – The function wasn't able to unregister the feature from the TLE extension.

Usage example

```
SELECT * FROM pgtle.unregister_feature_if_exists('pw_hook', 'passcheck');
```

Hooks reference for Trusted Language Extensions for PostgreSQL

Trusted Language Extensions for PostgreSQL supports PostgreSQL hooks. A *hook* is an internal callback mechanism available to developers for extending PostgreSQL's core functionality. By using hooks, developers can implement their own functions or procedures for use during various database operations, thereby modifying PostgreSQL's behavior in some way. For example, you can use a `passcheck` hook to customize how PostgreSQL handles the passwords supplied when creating or changing passwords for users (roles).

View the following documentation to learn about the hooks available for your TLE extensions.

Topics

- Password-check hook (passcheck)

Password-check hook (passcheck)

The `passcheck` hook is used to customize PostgreSQL behavior during the password-checking process for the following SQL commands and `psql` metacommand.

- `CREATE ROLE` *`username`* `...PASSWORD` – For more information, see CREATE ROLE in the PostgreSQL documentation.
- `ALTER ROLE` *`username`*`...PASSWORD` – For more information, see ALTER ROLE in the PostgreSQL documentation.
- `\password` *`username`* – This interactive `psql` metacommand securely changes the password for the specified user by hashing the password before transparently using the `ALTER ROLE ... PASSWORD` syntax. The metacommand is a secure wrapper for the `ALTER ROLE ... PASSWORD` command, thus the hook applies to the behavior of the `psql` metacommand.

For an example, see Password-check hook code listing.

Function prototype

```
passcheck_hook(username text, password text, password_type pgtle.password_types,
 valid_until timestamptz, valid_null boolean)
```

Arguments

A `passcheck` hook function takes the following arguments.

- `username` – The name (as text) of the role (username) that's setting a password.
- `password` – The plaintext or hashed password. The password entered should match the type specified in `password_type`.
- `password_type` – Specify the `pgtle.password_type` format of the password. This format can be one of the following options.
  - `PASSWORD_TYPE_PLAINTEXT` – A plaintext password.
  - `PASSWORD_TYPE_MD5` – A password that's been hashed using MD5 (message digest 5) algorithm.
  - `PASSWORD_TYPE_SCRAM_SHA_256` – A password that's been hashed using SCRAM-SHA-256 algorithm.
- `valid_until` – Specify the time when the password becomes invalid. This argument is optional. If you use this argument, specify the time as a `timestamptz` value.
- `valid_null` – If this Boolean is set to `true`, the `valid_until` option is set to `NULL`.

Configuration

The function `pgtle.enable_password_check` controls whether the passcheck hook is active. The passcheck hook has three possible settings.

- `off` – Turns off the `passcheck` password-check hook. This is the default value.
- on – Turns on the `passcode` password-check hook so that passwords are checked against the table.
- `require` – Requires a password check hook to be defined.

Usage notes

To turn the passcheck hook on or off, you need to modify the custom DB parameter group for the writer instance of your Aurora PostgreSQL DB cluster.

For Linux, macOS, or Unix:

```
aws rds modify-db-parameter-group \
    --region aws-region \
    --db-parameter-group-name your-custom-parameter-group \
    --parameters
 "ParameterName=pgtle.enable_password_check,ParameterValue=on,ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-parameter-group ^
    --region aws-region ^
    --db-parameter-group-name your-custom-parameter-group ^
    --parameters
 "ParameterName=pgtle.enable_password_check,ParameterValue=on,ApplyMethod=immediate"
```

Amazon Aurora PostgreSQL reference

Topics

Aurora PostgreSQL collations for EBCDIC and other mainframe migrations

Migrating mainframe applications to new platforms such as AWS ideally preserves application behavior. To preserve application behavior on a new platform exactly as it was on the mainframe requires that migrated data be collated using the same collation and sorting rules. For example, many Db2 migration solutions shift null values to u0180 (Unicode position 0180), so these collations sort u0180 first. This is one example of how collations can vary from their mainframe source and why it's necessary to choose a collation that better maps to the original EBCDIC collation.

Aurora PostgreSQL 14.3 and higher versions provide many ICU and EBCDIC collations to support such migration to AWS using the AWS Mainframe Modernization service. To learn more about this service, see What is AWS Mainframe Modernization?

In the following table, you can find Aurora PostgreSQL–provided collations. These collations follow EBCDIC rules and ensure that mainframe applications function the same on AWS as they did in the mainframe environment. The collation name includes the relevant code page, (cp*nnnn*), so that you can choose the appropriate collation for your mainframe source. For example, use `en-US-cp037-x-icu` for to achieve the collation behavior for EBCDIC data that originated from a mainframe application that used code page 037.

EBCDIC collations	AWS Blu Age collations	AWS Micro Focus collations
da-DK-cp1142-x-icu	da-DK-cp1142b-x-icu	da-DK-cp1142m-x-icu
da-DK-cp277-x-icu	da-DK-cp277b-x-icu	–

EBCDIC collations	AWS Blu Age collations	AWS Micro Focus collations
de-DE-cp1141-x-icu	de-DE-cp1141b-x-icu	de-DE-cp1141m-x-icu
de-DE-cp273-x-icu	de-DE-cp273b-x-icu	–
en-GB-cp1146-x-icu	en-GB-cp1146b-x-icu	en-GB-cp1146m-x-icu
en-GB-cp285-x-icu	en-GB-cp285b-x-icu	–
en-US-cp037-x-icu	en-US-cp037b-x-icu	–
en-US-cp1140-x-icu	en-US-cp1140b-x-icu	en-US-cp1140m-x-icu
es-ES-cp1145-x-icu	es-ES-cp1145b-x-icu	es-ES-cp1145m-x-icu
es-ES-cp284-x-icu	es-ES-cp284b-x-icu	–
fi-FI-cp1143-x-icu	fi-FI-cp1143b-x-icu	fi-FI-cp1143m-x-icu
fi-FI-cp278-x-icu	fi-FI-cp278b-x-icu	–
fr-FR-cp1147-x-icu	fr-FR-cp1147b-x-icu	fr-FR-cp1147m-x-icu
fr-FR-cp297-x-icu	fr-FR-cp297b-x-icu	–
it-IT-cp1144-x-icu	it-IT-cp1144b-x-icu	it-IT-cp1144m-x-icu
it-IT-cp280-x-icu	it-IT-cp280b-x-icu	–
nl-BE-cp1148-x-icu	nl-BE-cp1148b-x-icu	nl-BE-cp1148m-x-icu
nl-BE-cp500-x-icu	nl-BE-cp500b-x-icu	–

To learn more about AWS Blu Age, see Tutorial: Managed Runtime for AWS Blu Age in the *AWS Mainframe Modernization User Guide*.

For more information about working with AWS Micro Focus, see Tutorial: Managed Runtime for Micro Focus in the *AWS Mainframe Modernization User Guide*.

For more information about managing collations in PostgreSQL, see Collation Support in the PostgreSQL documentation.

Collations supported in Aurora PostgreSQL

Collations are set of rules that determine how character strings stored in the database are sorted and compared. Collations play a fundamental role in the computer system and are included as part of the operating system. Collations change over time when new characters are added to languages or when ordering rules change.

Collation libraries define specific rules and algorithms for a collation. The most popular collation libraries used within PostgreSQL are GNU C (glibc) and Internationalization components for Unicode (ICU). By default, Aurora PostgreSQL uses the glibc collation that includes unicode character sort orders for multi-byte character sequences.

When you create a new Aurora PostgreSQL DB cluster, it checks the operating system for the available collation. The PostgreSQL parameters of the `CREATE DATABASE` command `LC_COLLATE` and `LC_CTYPE` are used to specify a collation, which stands as the default collation in that database. Alternatively, you can also use the `LOCALE` parameter in `CREATE DATABASE` to set these parameters. This determines the default collation for character strings in the database and the rules for classifying characters as letters, numbers, or symbols. You can also choose a collation to use on a column, index, or on a query.

Aurora PostgreSQL depends on the glibc library in the operating system for collation support. Aurora PostgreSQL instance is periodically updated with the latest versions of the operating system. These updates sometimes include a newer version of the glibc library. Rarely, newer versions of glibc change the sort order or collation of some characters, which can cause the data to sort differently or produce invalid index entries. If you discover sort order issues for collation during an update, you might need to rebuild the indexes.

To reduce the possible impacts of the glibc updates, Aurora PostgreSQL now includes an independent default collation library. This collation library is available in Aurora PostgreSQL 14.6, 13.9, 12.13, 11.18 and newer minor version releases. It is compatible with glibc 2.26-59.amzn2, and provides sort order stability to prevent incorrect query results.

Aurora PostgreSQL functions reference

Following, you can find a list of Aurora PostgreSQL functions that are available for your Aurora DB clusters that run the Aurora PostgreSQL-Compatible Edition DB engine. These Aurora PostgreSQL functions are in addition to the standard PostgreSQL functions. For more information about standard PostgreSQL functions, see PostgreSQL–Functions and Operators.

Overview

You can use the following functions for Amazon RDS DB instances running Aurora PostgreSQL:

- aurora_db_instance_identifier
- aurora_ccm_status
- aurora_global_db_instance_status
- aurora_global_db_status
- aurora_list_builtins
- aurora_replica_status
- aurora_stat_activity
- aurora_stat_backend_waits
- aurora_stat_bgwriter
- aurora_stat_database
- aurora_stat_dml_activity
- aurora_stat_get_db_commit_latency
- aurora_stat_logical_wal_cache
- aurora_stat_memctx_usage
- aurora_stat_optimized_reads_cache
- aurora_stat_plans
- aurora_stat_reset_wal_cache
- aurora_stat_statements
- aurora_stat_system_waits
- aurora_stat_wait_event
- aurora_stat_wait_type
- aurora_version
- aurora_volume_logical_start_lsn
- aurora_wait_report

aurora_db_instance_identifier

Reports the name of the DB instance name to which you're connected.

Syntax

```
aurora_db_instance_identifier()
```

Arguments

None

Return type

VARCHAR string

Usage notes

This function displays the name of Aurora PostgreSQL-Compatible Edition cluster's DB instance for your database client or application connection.

This function is available starting with the release of Aurora PostgreSQL versions 13.7, 12.11, 11.16, 10.21 and for all other later versions.

Examples

The following example shows results of calling the `aurora_db_instance_identifier` function.

```
=> SELECT aurora_db_instance_identifier();
aurora_db_instance_identifier
-------------------------------
 test-my-instance-name
```

You can join the results of this function with the `aurora_replica_status` function to obtain details about the DB instance for your connection. The aurora_replica_status alone doesn't provide show you which DB instance you're using. The following example shows you how.

```
=> SELECT *
    FROM aurora_replica_status() rt,
         aurora_db_instance_identifier() di
    WHERE rt.server_id = di;
```

```
-[ RECORD 1 ]----------------------+-----------------------
server_id                          | test-my-instance-name
session_id                         | MASTER_SESSION_ID
durable_lsn                        | 88492069
highest_lsn_rcvd                   |
current_read_lsn                   |
cur_replay_latency_in_usec         |
active_txns                        |
is_current                         | t
last_transport_error               | 0
last_error_timestamp               |
last_update_timestamp              | 2022-06-03 11:18:25+00
feedback_xmin                      |
feedback_epoch                     |
replica_lag_in_msec                |
log_stream_speed_in_kib_per_second | 0
log_buffer_sequence_number         | 0
oldest_read_view_trx_id            |
oldest_read_view_lsn               |
pending_read_ios                   | 819
```

aurora_ccm_status

Displays the status of cluster cache manager.

Syntax

```
aurora_ccm_status()
```

Arguments

None.

Return type

SETOF record with the following columns:

- buffers_sent_last_minute – The number of buffers sent to the designated reader in the past minute.
- buffers_found_last_minute – The number of frequently accessed buffers identified during the past minute.

- `buffers_sent_last_scan` – The number of buffers sent to the designated reader during the last complete scan of the buffer cache.
- `buffers_found_last_scan` – The number of frequently accessed buffers sent during the last complete scan of the buffer cache. Buffers that are already cached on the designated reader aren't sent.
- `buffers_sent_current_scan` – The number of buffers sent during the current scan.
- `buffers_found_current_scan` – The number of frequently accessed buffers that were identified in the current scan.
- `current_scan_progress` – The number of buffers visited so far during the current scan.

Usage notes

You can use this function to check and monitor the cluster cache management (CCM) feature. This function works only if CCM is active on your Aurora PostgreSQL DB cluster. To use this function you connect to the Write DB instance on your Aurora PostgreSQL DB cluster.

You turn on CCM for an Aurora PostgreSQL DB cluster by setting the `apg_ccm_enabled` to 1 in the cluster's custom DB cluster parameter group. To learn how, see Configuring cluster cache management.

Cluster cache management is active on an Aurora PostgreSQL DB cluster when the cluster has an Aurora Reader instance configured as follows:

- The Aurora Reader instance uses same DB instance class type and size as the cluster's Writer instance.
- The Aurora Reader instance is configured as Tier-0 for the cluster. If the cluster has more than one Reader, this is its only Tier-0 Reader.

Setting more than one Reader to Tier-0 disables CCM. When CCM is disabled, calling this function returns the following error message:

```
ERROR: Cluster Cache Manager is disabled
```

You can also the PostgreSQL pg_buffercache extension to analyze the buffer cache. For more information, see pg_buffercache in the PostgreSQL documentation.

For more information, see Introduction to Aurora PostgreSQL cluster cache management.

Examples

The following example shows the results of calling the `aurora_ccm_status` function. This first example shows CCM statistics.

```
=> SELECT * FROM aurora_ccm_status();
 buffers_sent_last_minute | buffers_found_last_minute | buffers_sent_last_scan |
 buffers_found_last_scan | buffers_sent_current_scan | buffers_found_current_scan |
 current_scan_progress
--------------------------+---------------------------+------------------------
+--------------------------+---------------------------+----------------------------
+-----------------------
                  2242000 |                   2242003 |               17920442 |
         17923410 |                  14098000 |                   14100964 |
   15877443
```

For more complete detail, you can use expanded display, as shown following:

```
\x
Expanded display is on.
SELECT *  FROM aurora_ccm_status();
[ RECORD 1 ]-----------------------+---------
buffers_sent_last_minute           | 2242000
buffers_found_last_minute          | 2242003
buffers_sent_last_scan             | 17920442
buffers_found_last_scan            | 17923410
buffers_sent_current_scan          | 14098000
buffers_found_current_scan         | 14100964
current_scan_progress              | 15877443
```

This example shows how to check warm rate and warm percentage.

```
=> SELECT buffers_sent_last_minute * 8/60 AS warm_rate_kbps,
100 * (1.0-buffers_sent_last_scan/buffers_found_last_scan) AS warm_percent
FROM aurora_ccm_status ();
 warm_rate_kbps | warm_percent
----------------+--------------
 16523 |        100.0
```

aurora_global_db_instance_status

Displays the status of all Aurora instances, including replicas in an Aurora global DB cluster.

Syntax

```
aurora_global_db_instance_status()
```

Arguments

None

Return type

SETOF record with the following columns:

- `server_id` – The identifier of the DB instance.
- `session_id` – A unique identifier for the current session. A value of `MASTER_SESSION_ID` identifies the Writer (primary) DB instance.
- `aws_region` – The AWS Region in which this global DB instance runs. For a list of Regions, see Region availability.
- `durable_lsn` – The log sequence number (LSN) made durable in storage. A log sequence number (LSN) is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
- `highest_lsn_rcvd` – The highest LSN received by the DB instance from the writer DB instance.
- `feedback_epoch` – The epoch that the DB instance uses when it generates hot standby information. A *hot standby* is a DB instance that supports connections and queries while the primary DB is in recovery or standby mode. The hot standby information includes the epoch (point in time) and other details about the DB instance that's being used as a hot standby. For more information, see Hot Standby in the PostgreSQL documentation.
- `feedback_xmin` – The minimum (oldest) active transaction ID used by the DB instance.
- `oldest_read_view_lsn` – The oldest LSN used by the DB instance to read from storage.
- `visibility_lag_in_msec` – How far this DB instance is lagging behind the writer DB instance in milliseconds.

Usage notes

This function shows replication statistics for an Aurora DB cluster. For each Aurora PostgreSQL DB instance in the cluster, the function shows a row of data that includes any cross-Region replicas in a global database configuration.

You can run this function from any instance in an Aurora PostgreSQL DB cluster or an Aurora PostgreSQL global database. The function returns details about lag for all replica instances.

To learn more about monitoring lag using this function (`aurora_global_db_instance_status`) or by using using `aurora_global_db_status`, see Monitoring Aurora PostgreSQL-based global databases.

For more information about Aurora global databases, see Overview of Amazon Aurora global databases.

To get started with Aurora global databases, see Getting started with Amazon Aurora global databases or see Amazon Aurora FAQs.

Examples

This example shows cross-Region instance stats.

```
=> SELECT *
  FROM aurora_global_db_instance_status();
                server_id                 |              session_id               |
  aws_region   | durable_lsn | highest_lsn_rcvd | feedback_epoch | feedback_xmin |
 oldest_read_view_lsn | visibility_lag_in_msec
------------------------------------------+--------------------------------------
+--------------+-------------+------------------+----------------+---------------
+-----------------------+------------------------
 db-119-001-instance-01                   | MASTER_SESSION_ID                    | eu-
west-1     |  2534560273 |           [NULL] |         [NULL] |        [NULL] |
     [NULL] |                  [NULL]
 db-119-001-instance-02                   | 4ecff34d-d57c-409c-ba28-278b31d6fc40 | eu-
west-1     |  2534560266 |       2534560273 |              0 |      19669196 |
 2534560266 |                      6
 db-119-001-instance-03                   | 3e8a20fc-be86-43d5-95e5-bdf19d27ad6b | eu-
west-1     |  2534560266 |       2534560273 |              0 |      19669196 |
 2534560266 |                      6
 db-119-001-instance-04                   | fc1b0023-e8b4-4361-bede-2a7e926cead6 | eu-
west-1     |  2534560266 |       2534560273 |              0 |      19669196 |
 2534560254 |                     23
 db-119-001-instance-05                   | 30319b74-3f08-4e13-9728-e02aa1aa8649 | eu-
west-1     |  2534560266 |       2534560273 |              0 |      19669196 |
 2534560254 |                     23
 db-119-001-global-instance-1             | a331ffbb-d982-49ba-8973-527c96329c60 | eu-
central-1 |  2534560254 |       2534560266 |              0 |      19669196 |
 2534560247 |                    996
 db-119-001-global-instance-1             | e0955367-7082-43c4-b4db-70674064a9da | eu-
west-2    |  2534560254 |       2534560266 |              0 |      19669196 |
 2534560247 |                     14
 db-119-001-global-instance-1-eu-west-2a  | 1248dc12-d3a4-46f5-a9e2-85850491a897 | eu-
west-2    |  2534560254 |       2534560266 |              0 |      19669196 |
 2534560247 |                      0
```

This example shows how to check global replica lag in milliseconds.

```
=> SELECT CASE
          WHEN 'MASTER_SESSION_ID' = session_id THEN 'Primary'
         ELSE 'Secondary'
      END AS global_role,
      aws_region,
      server_id,
      visibility_lag_in_msec
  FROM aurora_global_db_instance_status()
  ORDER BY 1, 2, 3;
   global_role |  aws_region  |                server_id                 |
 visibility_lag_in_msec
-------------+--------------+-----------------------------------------
+------------------------
 Primary     | eu-west-1    | db-119-001-instance-01                   |
 [NULL]
 Secondary   | eu-central-1 | db-119-001-global-instance-1             |
     13
 Secondary   | eu-west-1    | db-119-001-instance-02                   |
     10
 Secondary   | eu-west-1    | db-119-001-instance-03                   |
      9
 Secondary   | eu-west-1    | db-119-001-instance-04                   |
      2
 Secondary   | eu-west-1    | db-119-001-instance-05                   |
     18
 Secondary   | eu-west-2    | db-119-001-global-instance-1             |
     14
 Secondary   | eu-west-2    | db-119-001-global-instance-1-eu-west-2a |
     13
```

This example shows how to check min, max and average lag per AWS Region from the global database configuration.

```
=> SELECT 'Secondary' global_role,
       aws_region,
       min(visibility_lag_in_msec) min_lag_in_msec,
       max(visibility_lag_in_msec) max_lag_in_msec,
       round(avg(visibility_lag_in_msec),0) avg_lag_in_msec
  FROM aurora_global_db_instance_status()
 WHERE aws_region NOT IN (SELECT   aws_region
                              FROM aurora_global_db_instance_status()
                             WHERE session_id='MASTER_SESSION_ID')
                          GROUP BY aws_region
UNION ALL
SELECT  'Primary' global_role,
        aws_region,
        NULL,
        NULL,
        NULL
    FROM aurora_global_db_instance_status()
   WHERE session_id='MASTER_SESSION_ID'
ORDER BY 1, 5;
 global_role |  aws_region  | min_lag_in_msec | max_lag_in_msec | avg_lag_in_msec
------------+--------------+-----------------+-----------------+-----------------
 Primary    | eu-west-1    |          [NULL] |          [NULL] |          [NULL]
 Secondary  | eu-central-1 |             133 |             133 |             133
 Secondary  | eu-west-2    |               0 |             495 |             248
```

aurora_global_db_status

Displays information about various aspects of Aurora global database lag, specifically, lag of the underlying Aurora storage (so called durability lag) and lag between the recovery point objective (RPO).

Syntax

```
aurora_global_db_status()
```

Arguments

None.

Return type

SETOF record with the following columns:

- `aws_region` – The AWS Region that this DB cluster is in. For a complete listing of AWS Regions by engine, see Regions and Availability Zones.
- `highest_lsn_written` – The highest log sequence number (LSN) that currently exists on this DB cluster. A log sequence number (LSN) is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
- `durability_lag_in_msec` – The difference in the timestamp values between the `highest_lsn_written` on a secondary DB cluster and the `highest_lsn_written` on the primary DB cluster. A value of -1 identifies the primary DB cluster of the Aurora global database.
- `rpo_lag_in_msec` – The recovery point objective (RPO) lag. The RPO lag is the time it takes for the most recent user transaction COMMIT to be stored on a secondary DB cluster after it's been stored on the primary DB cluster of the Aurora global database. A value of -1 denotes the primary DB cluster (and thus, lag isn't relevant).

  In simple terms, this metric calculates the recovery point objective for each Aurora PostgreSQL DB cluster in the Aurora global database, that is, how much data might be lost if there were an outage. As with lag, RPO is measured in time.
- `last_lag_calculation_time` – The timestamp that specifies when values were last calculated for `durability_lag_in_msec` and `rpo_lag_in_msec`. A time value such as `1970-01-01 00:00:00+00` means this is the primary DB cluster.
- `feedback_epoch` – The epoch that the secondary DB cluster uses when it generates hot standby information. A *hot standby* is a DB instance that supports connections and queries while the primary DB is in recovery or standby mode. The hot standby information includes the epoch (point in time) and other details about the DB instance that's being used as a hot standby. For more information, see Hot Standby in the PostgreSQL documentation.
- `feedback_xmin` – The minimum (oldest) active transaction ID used by a secondary DB cluster.

Usage notes

This function shows replication statistics for an Aurora global database. It shows one row for each DB cluster in an Aurora PostgreSQL global database. You can run this function from any instance in your Aurora PostgreSQL global database.

To evaluate Aurora global database replication lag, which is the visible data lag, see aurora_global_db_instance_status.

To learn more about using `aurora_global_db_status` and `aurora_global_db_instance_status` to monitor Aurora global database lag, see Monitoring Aurora PostgreSQL-based global databases. For more information about Aurora global databases, see Overview of Amazon Aurora global databases.

Examples

This example shows how to display cross-region storage statistics.

```
=> SELECT CASE
          WHEN '-1' = durability_lag_in_msec THEN 'Primary'
          ELSE 'Secondary'
       END AS global_role,
       *
  FROM aurora_global_db_status();
 global_role | aws_region | highest_lsn_written | durability_lag_in_msec |
 rpo_lag_in_msec | last_lag_calculation_time  | feedback_epoch | feedback_xmin
-------------+------------+---------------------+-----------------------
+-----------------+----------------------------+----------------+---------------
 Primary     | eu-west-1  |           131031557 |                     -1 |
 -1 | 1970-01-01 00:00:00+00     |              0 |             0
 Secondary   | eu-west-2  |           131031554 |                    410 |
  0 | 2021-06-01 18:59:36.124+00 |              0 |         12640
 Secondary   | eu-west-3  |           131031554 |                    410 |
  0 | 2021-06-01 18:59:36.124+00 |              0 |         12640
```

aurora_list_builtins

Lists all available Aurora PostgreSQL built-in functions, along with brief descriptions and function details.

Syntax

```
aurora_list_builtins()
```

Arguments

None

Return type

SETOF record

Examples

The following example shows results from calling the aurora_list_builtins function.

```
=> SELECT *
FROM aurora_list_builtins();

              Name                | Result data type |                   Argument data
 types                     | Type | Volatility |  Parallel  | Security |
             Description
-----------------------------------+------------------
+-----------------------------------------------------------+------+-----------
+------------+----------
+---------------------------------------------------------------------
 aurora_version                    | text             |
                           | func | stable     | safe       | invoker  | Amazon Aurora
 PostgreSQL-Compatible Edition version string
 aurora_stat_wait_type             | SETOF record     | OUT type_id smallint, OUT
 type_name text                 | func | volatile   | restricted | invoker  | Lists all
 supported wait types
 aurora_stat_wait_event            | SETOF record     | OUT type_id smallint, OUT
 event_id integer, OUT event_na.| func | volatile   | restricted | invoker  | Lists all
 supported wait events
                                   |                  |.me text
                           |      |            |            |          |
 aurora_list_builtins              | SETOF record     | OUT "Name" text, OUT "Result
 data type" text, OUT "Argum.| func | stable     | safe       | invoker  | Lists all
 Aurora built-in functions
                                   |                  |.ent data types" text, OUT
 "Type" text, OUT "Volatility" .|      |            |            |          |
                                   |                  |.text, OUT "Parallel" text, OUT
 "Security" text, OUT "Des.|      |            |            |          |
                                   |                  |.cription" text
                           |      |            |            |          |
 .
 .
 .
 aurora_stat_file                  | SETOF record     | OUT filename text, OUT
 allocated_bytes bigint, OUT used_.| func | stable     | safe       | invoker  | Lists
 all files present in Aurora storage
                                   |                  |.bytes bigint
                           |      |            |            |          |
 aurora_stat_get_db_commit_latency | bigint             | oid
                           | func | stable     | restricted | invoker   | Per DB commit
 latency in microsecs
```

aurora_replica_status

Displays the status of all Aurora PostgreSQL reader nodes.

Syntax

```
aurora_replica_status()
```

Arguments

None

Return type

SETOF record with the following columns:

- `server_id` – The DB instance ID (identifier).
- `session_id` – A unique identifier for the current session, returned for primary instance and reader instances as follows:
  - For the primary instance, `session_id` is always `` `MASTER_SESSION_ID’ ``.
  - For reader instances, `session_id` is always the `UUID` (universally unique identifier) of the reader instance.
- `durable_lsn` – The log sequence number (LSN) that's been saved in storage.
  - For the primary volume, the primary volume durable LSN (VDL) that's currently in effect.
  - For any secondary volumes, the VDL of the primary up to which the secondary has successfully been applied.

> **Note**
>
> A log sequence number (LSN) is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a transaction that's occurred later in the sequence.

- `highest_lsn_rcvd` – The highest (most recent) LSN received by the DB instance from the writer DB instance.
- `current_read_lsn` – The LSN of the most recent snapshot that's been applied to all readers.
- `cur_replay_latency_in_usec` – The number of microseconds that it's expected to take to replay the log on the secondary.
- `active_txns` – The number of currently active transactions.
- `is_current` – Not used.
- `last_transport_error` – Last replication error code.
- `last_error_timestamp` – Timestamp of last replication error.
- `last_update_timestamp` – Timestamp of last update to replica status. From Aurora PostgreSQL 13.9, the `last_update_timestamp` value for the DB instance that you are connected to is set to `NULL`.
- `feedback_xmin` – The hot standby feedback_xmin of the replica. The minimum (oldest) active transaction ID used by the DB instance.
- `feedback_epoch` – The epoch the DB instance uses when it generates hot standby information.
- `replica_lag_in_msec` – Time that reader instance lags behind writer writer instance, in milliseconds.
- `log_stream_speed_in_kib_per_second` – The log stream throughput in kilobytes per second.
- `log_buffer_sequence_number` – The log buffer sequence number.
- `oldest_read_view_trx_id` – Not used.
- `oldest_read_view_lsn` – The oldest LSN used by the DB instance to read from storage.
- `pending_read_ios` – The outstanding page reads that are still pending on replica.
- `read_ios` – The total number of page reads on replica.
- `iops` – Not used.
- `cpu` – CPU usage by the replica process. Note that this isn't CPU usage by the instance but rather the process. For information about CPU usage by the instance, see Instance-level metrics for Amazon Aurora.

Usage notes

The `aurora_replica_status` function returns values from an Aurora PostgreSQL DB cluster's replica status manager. You can use this function to obtain information about the status of replication on your Aurora PostgreSQL DB cluster, including metrics for all DB instances in your Aurora DB cluster. For example, you can do the following:

- **Get information about the type of instance (writer, reader) in the Aurora PostgreSQL DB cluster** – You can obtain this information by checking the values of the following columns:
  - `server_id` – Contains the name of the instance that you specified when you created the instance. In some cases, such as for the primary (writer) instance, the name is typically created for you by appending *-instance-1* to the name that you create for your Aurora PostgreSQL DB cluster.
  - `session_id` – The `session_id` field indicates whether the instance is a reader or a writer. For a writer instance, `session_id` is always set to `"MASTER_SESSION_ID"`. For a reader instance, `session_id` is set to the `UUID` of the specific reader.
- **Diagnose common replication issues, such as replica lag** – Replica lag is the time in milliseconds that the page cache of a reader instance is behind that of the writer instance. This lag occurs because Aurora clusters use asynchronous replication, as described in [Replication with Amazon Aurora](). It's shown in the `replica_lag_in_msec` column in the results returned by this function. Lag can also occur when a query is cancelled due to conflicts with recovery on a standby server. You can check `pg_stat_database_conflicts()` to verify that such a conflict is causing the replica lag (or not). For more information, see [The Statistics Collector]() in the *PostgreSQL documentation*. To learn more about high availability and replication, see [Amazon Aurora FAQs]().

  Amazon CloudWatch stores `replica_lag_in_msec` results over time, as the `AuroraReplicaLag` metric. For information about using CloudWatch metrics for Aurora, see [Monitoring Amazon Aurora metrics with Amazon CloudWatch]()

To learn more about troubleshooting Aurora read replicas and restarts, see [Why did my Amazon Aurora read replica fall behind and restart?]() in the [AWS Support Center]().

Examples

The following example shows how to get the replication status of all instances in an Aurora PostgreSQL DB cluster:

```
=> SELECT *
FROM aurora_replica_status();
```

The following example shows the writer instance in the `docs-lab-apg-main` Aurora PostgreSQL DB cluster:

```
=> SELECT server_id,
    CASE
        WHEN 'MASTER_SESSION_ID' = session_id THEN 'writer'
        ELSE 'reader'
    END AS instance_role
FROM aurora_replica_status()
WHERE session_id = 'MASTER_SESSION_ID';
        server_id       | instance_role
------------------------+---------------
 db-119-001-instance-01 | writer
```

The following example example lists all reader instances in a cluster:

```
=> SELECT server_id,
    CASE
        WHEN 'MASTER_SESSION_ID' = session_id THEN 'writer'
        ELSE 'reader'
    END AS instance_role
FROM aurora_replica_status()
WHERE session_id <> 'MASTER_SESSION_ID';
        server_id       | instance_role
------------------------+---------------
db-119-001-instance-02  | reader
db-119-001-instance-03  | reader
db-119-001-instance-04  | reader
db-119-001-instance-05  | reader
(4 rows)
```

The following example lists all instances, how far each instance is lagging behind the writer, and how long since the last update:

```
=> SELECT server_id,
    CASE
        WHEN 'MASTER_SESSION_ID' = session_id THEN 'writer'
        ELSE 'reader'
    END AS instance_role,
    replica_lag_in_msec AS replica_lag_ms,
    round(extract (epoch FROM (SELECT age(clock_timestamp(), last_update_timestamp))) *
 1000) AS last_update_age_ms
```

```
FROM aurora_replica_status()
ORDER BY replica_lag_in_msec NULLS FIRST;
       server_id        | instance_role | replica_lag_ms | last_update_age_ms
------------------------+---------------+----------------+--------------------
 db-124-001-instance-03 | writer        |         [NULL] |               1756
 db-124-001-instance-01 | reader        |             13 |               1756
 db-124-001-instance-02 | reader        |             13 |               1756
(3 rows)
```

aurora_stat_activity

Returns one row per server process, showing information related to the current activity of that process.

Syntax

```
aurora_stat_activity();
```

Arguments

None

Return type

Returns one row per server process. In additional to `pg_stat_activity` columns, the following field is added:

- planid – plan identifier

Usage notes

A supplementary view to `pg_stat_activity` returning the same columns with an additional `plan_id` column which shows the current query execution plan.

`aurora_compute_plan_id` must be enabled for the view to return a plan_id.

This function is available from Aurora PostgreSQL versions 14.10, 15.5, and for all other later versions.

Examples

The example query below aggregates the top load by query_id and plan_id.

```
db1=# select count(*), query_id, plan_id
db1-# from aurora_stat_activity() where state = 'active'
db1-# and pid <> pg_backend_pid()
db1-# group by query_id, plan_id
db1-# order by 1 desc;

count |  query_id             |  plan_id
-------+----------------------+-------------
 11    | -5471422286312252535 | -2054628807
 3     | -6907107586630739258 | -815866029
 1     | 5213711845501580017  |  300482084
(3 rows)
```

If the plan used for query_id changes, a new plan_id will be reported by aurora_stat_activity.

```
count  |  query_id            |  plan_id
-------+----------------------+-------------
 10    | -5471422286312252535 | 1602979607
 1     | -6907107586630739258 | -1809935983
 1     | -2446282393000597155 | -207532066
(3 rows)
```

aurora_stat_backend_waits

Displays statistics for wait activity for a specific backend process.

Syntax

```
aurora_stat_backend_waits(pid)
```

Arguments

pid – The ID for the backend process. You can obtain process IDs by using the pg_stat_activity view.

Return type

SETOF record with the following columns:

- `type_id` – A number that denotes the type of wait event, such as 1 for a lightweight lock (`LWLock`), 3 for a lock, or 6 for a client session, to name some examples. These values become meaningful when you join the results of this function with columns from `aurora_stat_wait_type` function, as shown in the Examples.
- `event_id` – An identifying number for the wait event. Join this value with the columns from `aurora_stat_wait_event` to obtain meaningful event names.
- `waits` – A count of the number of waits accumulated for the specified process ID.
- `wait_time` – Wait time in milliseconds.

Usage notes

You can use this function to analyze specific backend (session) wait events that occurred since a connection opened. To get more meaningful information about wait event names and types, you can combine this function `aurora_stat_wait_type` and `aurora_stat_wait_event`, by using JOIN as shown in the examples.

Examples

This example shows all waits, types, and event names for a backend process ID 3027.

```
=> SELECT type_name, event_name, waits, wait_time
        FROM aurora_stat_backend_waits(3027)
NATURAL JOIN aurora_stat_wait_type()
NATURAL JOIN aurora_stat_wait_event();
type_name |       event_name       | waits | wait_time
-----------+------------------------+-------+------------
 LWLock    | ProcArrayLock          |     3 |         27
 LWLock    | wal_insert             |   423 |      16336
 LWLock    | buffer_content         | 11840 |    1033634
 LWLock    | lock_manager           | 23821 |    5664506
 Lock      | tuple                  | 10258 |  152280165
 Lock      | transactionid          | 78340 | 1239808783
 Client    | ClientRead             | 34072 |   17616684
 IO        | ControlFileSyncUpdate  |     2 |          0
 IO        | ControlFileWriteUpdate |     4 |         32
 IO        | RelationMapRead        |     2 |        795
 IO        | WALWrite               | 36666 |      98623
 IO        | XactSync                |  4867 |    7331963
```

This example shows current and cumulative wait types and wait events for all active sessions (`pg_stat_activity state <> 'idle'`) (but without the current session that's invoking the function (`pid <> pg_backend_pid()`).

```
=> SELECT a.pid,
             a.usename,
             a.app_name,
             a.current_wait_type,
             a.current_wait_event,
             a.current_state,
             wt.type_name AS wait_type,
             we.event_name AS wait_event,
             a.waits,
             a.wait_time
        FROM (SELECT pid,
                     usename,
                     left(application_name,16) AS app_name,
                     coalesce(wait_event_type,'CPU') AS current_wait_type,
                     coalesce(wait_event,'CPU') AS current_wait_event,
                     state AS current_state,
                     (aurora_stat_backend_waits(pid)).*
                FROM pg_stat_activity
               WHERE pid <> pg_backend_pid()
                 AND state <> 'idle') a
NATURAL JOIN aurora_stat_wait_type() wt
NATURAL JOIN aurora_stat_wait_event() we;
  pid  | usename  | app_name | current_wait_type | current_wait_event | current_state |
 wait_type |       wait_event       | waits | wait_time
-------+----------+----------+-------------------+--------------------+---------------
+-----------+------------------------+-------+-----------
 30099 | postgres | pgbench  | Lock              | transactionid      | active        |
 LWLock    | wal_insert             |  1937 |     29975
 30099 | postgres | pgbench  | Lock              | transactionid      | active        |
 LWLock    | buffer_content         | 22903 |    760498
 30099 | postgres | pgbench  | Lock              | transactionid      | active        |
 LWLock    | lock_manager           | 10012 |    223207
 30099 | postgres | pgbench  | Lock              | transactionid      | active        |
 Lock      | tuple                  | 20315 |  63081529
 .
 .
 .
```

```
30099 | postgres | pgbench  | Lock              | transactionid      | active         |
IO        | WALWrite               | 93293 |    237440
30099 | postgres | pgbench  | Lock              | transactionid      | active         |
IO        | XactSync               | 13010 |  19525143
30100 | postgres | pgbench  | Lock              | transactionid      | active         |
LWLock    | ProcArrayLock          |     6 |        53
30100 | postgres | pgbench  | Lock              | transactionid      | active         |
LWLock    | wal_insert             |  1913 |     25450
30100 | postgres | pgbench  | Lock              | transactionid      | active         |
LWLock    | buffer_content         | 22874 |    778005
.
.
.
30109 | postgres | pgbench  | IO                | XactSync           | active         |
LWLock    | ProcArrayLock          |     3 |        71
30109 | postgres | pgbench  | IO                | XactSync           | active         |
LWLock    | wal_insert             |  1940 |     27741
30109 | postgres | pgbench  | IO                | XactSync           | active         |
LWLock    | buffer_content         | 22962 |    776352
30109 | postgres | pgbench  | IO                | XactSync           | active         |
LWLock    | lock_manager           |  9879 |    218826
30109 | postgres | pgbench  | IO                | XactSync           | active         |
Lock      | tuple                  | 20401 |  63581306
30109 | postgres | pgbench  | IO                | XactSync           | active         |
Lock      | transactionid          | 50769 | 211645008
30109 | postgres | pgbench  | IO                | XactSync           | active         |
Client    | ClientRead             | 89901 |  44192439
```

This example shows current and the top three (3) cumulative wait type and wait events for all active sessions (`pg_stat_activity state <> 'idle'`) excluding current session (`pid <>pg_backend_pid()`).

```
=> SELECT top3.*
       FROM (SELECT a.pid,
                    a.usename,
                    a.app_name,
                    a.current_wait_type,
                    a.current_wait_event,
                    a.current_state,
                    wt.type_name AS wait_type,
                    we.event_name AS wait_event,
                    a.waits,
                    a.wait_time,
                    RANK() OVER (PARTITION BY pid ORDER BY a.wait_time DESC)
               FROM (SELECT pid,
                            usename,
                            left(application_name,16) AS app_name,
                            coalesce(wait_event_type,'CPU') AS current_wait_type,
                            coalesce(wait_event,'CPU') AS current_wait_event,
                            state AS current_state,
                            (aurora_stat_backend_waits(pid)).*
                       FROM pg_stat_activity
                      WHERE pid <> pg_backend_pid()
                        AND state <> 'idle') a
       NATURAL JOIN aurora_stat_wait_type() wt
       NATURAL JOIN aurora_stat_wait_event() we) top3
 WHERE RANK <=3;
  pid  | usename  | app_name | current_wait_type | current_wait_event | current_state |
 wait_type |   wait_event    |  waits  | wait_time  | rank
-------+----------+----------+-------------------+--------------------+---------------
+-----------+-----------------+---------+------------+------
 20567 | postgres | psql     | CPU               | CPU                | active        |
 LWLock    | wal_insert      |   25000 |   67512003 |    1
 20567 | postgres | psql     | CPU               | CPU                | active        |
 IO        | WALWrite        | 3071758 |    1016961 |    2
 20567 | postgres | psql     | CPU               | CPU                | active        |
 IO        | BufFileWrite    |   20750 |     184559 |    3
 27743 | postgres | pgbench  | Lock              | transactionid      | active        |
 Lock      | transactionid   |  237350 | 1265580011 |    1
 27743 | postgres | pgbench  | Lock              | transactionid      | active        |
 Lock      | tuple           |   93641 |  341472318 |    2
 27743 | postgres | pgbench  | Lock              | transactionid      | active        |
 Client    | ClientRead      |  417556 |  204796837 |    3
 .
 .
 .
 27745 | postgres | pgbench  | IO                | XactSync           | active        |
 Lock      | transactionid   |  238068 | 1265816822 |    1
 27745 | postgres | pgbench  | IO                | XactSync           | active        |
 Lock      | tuple           |   93210 |  338312247 |    2
 27745 | postgres | pgbench  | IO                | XactSync           | active        |
 Client    | ClientRead      |  419157 |  207836533 |    3
 27746 | postgres | pgbench  | Lock              | transactionid      | active        |
 Lock      | transactionid   |  237621 | 1264528811 |    1
 27746 | postgres | pgbench  | Lock              | transactionid      | active        |
 Lock      | tuple           |   93563 |  339799310 |    2
```

```
  27746 | postgres | pgbench   | Lock                | transactionid      | active          |
  Client     | ClientRead       |  417304 |  208372727 |    3
```

aurora_stat_bgwriter

`aurora_stat_bgwriter` is a statistics view showing information about Optimized Reads cache writes.

Syntax

```
aurora_stat_bgwriter()
```

Arguments

None

Return type

SETOF record with all `pg_stat_bgwriter` columns and the following additional columns. For more information on `pg_stat_bgwriter` columns, see [`pg_stat_bgwriter`](#).

You can reset stats for this function using `pg_stat_reset_shared("bgwriter")`.

- `orcache_blks_written` – Total number of optimized reads cache data blocks written.
- `orcache_blk_write_time` – If `track_io_timing` is enabled, it tracks the total time spent writing optimized reads cache data file blocks, in milliseconds. For more information, see [track_io_timing](#).

Usage notes

This function is available in the following Aurora PostgreSQL versions:

- 15.4 and higher 15 versions
- 14.9 and higher 14 versions

Examples

```
=> select * from aurora_stat_bgwriter();
-[ RECORD 1 ]-----------------+-----------
orcache_blks_written          | 246522
```

```
orcache_blk_write_time         | 339276.404
```

aurora_stat_database

It carries all columns of pg_stat_database and adds new columns in the end.

Syntax

```
aurora_stat_database()
```

Arguments

None

Return type

SETOF record with all `pg_stat_database` columns and the following additional columns. For more information on `pg_stat_database` columns, see [pg_stat_database](#).

- `storage_blks_read` – Total number of shared blocks read from aurora storage in this database.
- `orcache_blks_hit` – Total number of optimized reads cache hits in this database.
- `local_blks_read` – Total number of local blocks read in this database.
- `storage_blk_read_time` – If `track_io_timing` is enabled, it tracks the total time spent reading data file blocks from aurora storage, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.
- `local_blk_read_time` – If `track_io_timing` is enabled, it tracks the total time spent reading local data file blocks, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.
- `orcache_blk_read_time` – If `track_io_timing` is enabled, it tracks the total time spent reading data file blocks from optimized reads cache, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.

Note

The value of `blks_read` is the sum of `storage_blks_read`, `orcache_blks_hit`, and `local_blks_read`.

The value of blk_read_time is the sum of storage_blk_read_time, orcache_blk_read_time, and local_blk_read_time.

Usage notes

This function is available in the following Aurora PostgreSQL versions:

- 15.4 and higher 15 versions
- 14.9 and higher 14 versions

Examples

The following example shows how it carries all the pg_stat_database columns and appends 6 new columns in the end:

```
=> select * from aurora_stat_database() where datid=14717;
-[ RECORD 1 ]------------+------------------------------
datid                    | 14717
datname                  | postgres
numbackends              | 1
xact_commit              | 223
xact_rollback            | 4
blks_read                | 1059
blks_hit                 | 11456
tup_returned             | 27746
tup_fetched              | 5220
tup_inserted             | 165
tup_updated              | 42
tup_deleted              | 91
conflicts                | 0
temp_files               | 0
temp_bytes               | 0
deadlocks                | 0
checksum_failures        |
checksum_last_failure    |
blk_read_time            | 3358.689
blk_write_time           | 0
session_time             | 1076007.997
active_time              | 3684.371
idle_in_transaction_time | 0
sessions                 | 10
```

```
sessions_abandoned        | 0
sessions_fatal            | 0
sessions_killed           | 0
stats_reset               | 2023-01-12 20:15:17.370601+00
orcache_blks_hit          | 425
orcache_blk_read_time     | 89.934
storage_blks_read         | 623
storage_blk_read_time     | 3254.914
local_blks_read           | 0
local_blk_read_time       | 0
```

aurora_stat_dml_activity

Reports cumulative activity for each type of data manipulation language (DML) operation on a database in an Aurora PostgreSQL cluster.

Syntax

```
aurora_stat_dml_activity(database_oid)
```

Arguments

database_oid

The object ID (OID) of the database in the Aurora PostgreSQL cluster.

Return type

SETOF record

Usage notes

The `aurora_stat_dml_activity` function is only available with Aurora PostgreSQL release 3.1 compatible with PostgreSQL engine 11.6 and later.

Use this function on Aurora PostgreSQL clusters with a large number of databases to identify which databases have more or slower DML activity, or both.

The `aurora_stat_dml_activity` function returns the number of times the operations ran and the cumulative latency in microseconds for SELECT, INSERT, UPDATE, and DELETE operations. The report includes only successful DML operations.

You can reset this statistic by using the PostgreSQL statistics access function `pg_stat_reset`. You can check the last time this statistic was reset by using the `pg_stat_get_db_stat_reset_time` function. For more information about PostgreSQL statistics access functions, see [The Statistics Collector](#) in the PostgreSQL documentation.

Examples

The following example shows how to report DML activity statistics for the connected database.

```
--Define the oid variable from connected database by using \gset
=> SELECT oid,
          datname
     FROM pg_database
    WHERE datname=(select current_database()) \gset
=> SELECT *
     FROM aurora_stat_dml_activity(:oid);
select_count | select_latency_microsecs | insert_count | insert_latency_microsecs |
 update_count | update_latency_microsecs | delete_count | delete_latency_microsecs
--------------+--------------------------+--------------+--------------------------
+--------------+--------------------------+--------------+--------------------------
       178957 |                 66684115 |       171065 |                 28876649 |
    519538 |            1454579206167 |            1 |                    53027


-- Showing the same results with expanded display on
=> SELECT *
     FROM aurora_stat_dml_activity(:oid);
-[ RECORD 1 ]------------+--------------
select_count             | 178957
select_latency_microsecs | 66684115
insert_count             | 171065
insert_latency_microsecs | 28876649
update_count             | 519538
update_latency_microsecs | 1454579206167
delete_count             | 1
delete_latency_microsecs | 53027
```

The following example shows DML activity statistics for all databases in the Aurora PostgreSQL cluster. This cluster has two databases, `postgres` and `mydb`. The comma-separated list corresponds with the `select_count`, `select_latency_microsecs`, `insert_count`, `insert_latency_microsecs`, `update_count`, `update_latency_microsecs`, `delete_count`, and `delete_latency_microsecs` fields.

Aurora PostgreSQL creates and uses a system database named `rdsadmin` to support administrative operations such as backups, restores, health checks, replication, and so on. These DML operations have no impact on your Aurora PostgreSQL cluster.

```
=> SELECT oid,
    datname,
    aurora_stat_dml_activity(oid)
    FROM pg_database;
oid  |    datname     |                    aurora_stat_dml_activity
-------+----------------
+-------------------------------------------------------------------
 14006 | template0      | (,,,,,,,)
 16384 | rdsadmin       | (2346623,1211703821,4297518,817184554,0,0,0,0)
     1 | template1      | (,,,,,,,)
 14007 | postgres       |
 (178961,66716329,171065,28876649,519538,1454579206167,1,53027)
 16401 | mydb           | (200246,64302436,200036,107101855,600000,83659417514,0,0)
```

The following example shows DML activity statistics for all databases, organized in columns for better readability.

```
SELECT db.datname,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 1), '()') AS select_count,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 2), '()') AS select_latency_microsecs,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 3), '()') AS insert_count,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 4), '()') AS insert_latency_microsecs,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 5), '()') AS update_count,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 6), '()') AS update_latency_microsecs,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 7), '()') AS delete_count,
       BTRIM(SPLIT_PART(db.asdmla::TEXT, ',', 8), '()') AS delete_latency_microsecs
FROM  (SELECT datname,
              aurora_stat_dml_activity(oid) AS asdmla
         FROM pg_database
      ) AS db;

    datname     | select_count | select_latency_microsecs | insert_count |
 insert_latency_microsecs | update_count | update_latency_microsecs | delete_count |
 delete_latency_microsecs
----------------+--------------+--------------------------+-------------
+--------------------------+--------------+--------------------------+-------------
+--------------------------
```

```
 template0      |              |                          |              |
             |              |                          |              |
 rdsadmin       | 4206523      | 2478812333               | 7009414      | 1338482258
             | 0            | 0                        | 0            | 0
 template1      |              |                          |              |
             |              |                          |              |
 fault_test     | 66           | 452099                   | 0            | 0
             | 0            | 0                        | 0            | 0
 db_access_test | 1            | 5982                     | 0            | 0
             | 0            | 0                        | 0            | 0
 postgres       | 42035        | 95179203                 | 5752         | 2678832898
             | 21157        | 441883182488             | 2            | 1520
 mydb           | 71           | 453514                   | 0            | 0
             | 1            | 190                      | 1            | 152
```

The following example shows the average cumulative latency (cumulative latency divided by count) for each DML operation for the database with the OID 16401.

```
=> SELECT select_count,
          select_latency_microsecs,
          select_latency_microsecs/NULLIF(select_count,0) select_latency_per_exec,
          insert_count,
          insert_latency_microsecs,
          insert_latency_microsecs/NULLIF(insert_count,0) insert_latency_per_exec,
          update_count,
          update_latency_microsecs,
          update_latency_microsecs/NULLIF(update_count,0) update_latency_per_exec,
          delete_count,
          delete_latency_microsecs,
          delete_latency_microsecs/NULLIF(delete_count,0) delete_latency_per_exec
     FROM aurora_stat_dml_activity(16401);
-[ RECORD 1 ]------------+-------------
select_count             | 451312
select_latency_microsecs | 80205857
select_latency_per_exec  | 177
insert_count             | 451001
insert_latency_microsecs | 123667646
insert_latency_per_exec  | 274
update_count             | 1353067
update_latency_microsecs | 200900695615
update_latency_per_exec  | 148478
delete_count             | 12
delete_latency_microsecs | 448
```

```
delete_latency_per_exec  | 37
```

aurora_stat_get_db_commit_latency

Gets the cumulative commit latency in microseconds for Aurora PostgreSQL databases. *Commit latency* is measured as the time between when a client submits a commit request and when it receives the commit acknowledgement.

Syntax

```
aurora_stat_get_db_commit_latency(database_oid)
```

Arguments

database_oid

The object ID (OID) of the Aurora PostgreSQL database.

Return type

SETOF record

Usage notes

Amazon CloudWatch uses this function to calculate the average commit latency. For more information about Amazon CloudWatch metrics and how to troubleshoot high commit latency, see Viewing metrics in the Amazon RDS console and Making better decisions about Amazon RDS with Amazon CloudWatch metrics.

You can reset this statistic by using the PostgreSQL statistics access function `pg_stat_reset`. You can check the last time this statistic was reset by using the `pg_stat_get_db_stat_reset_time` function. For more information about PostgreSQL statistics access functions, see The Statistics Collector in the PostgreSQL documentation.

Examples

The following example gets the cumulative commit latency for each database in the `pg_database` cluster.

```
=> SELECT oid,
    datname,
    aurora_stat_get_db_commit_latency(oid)
    FROM pg_database;

  oid  |    datname     | aurora_stat_get_db_commit_latency
-------+----------------+-----------------------------------
 14006 | template0      |                                 0
 16384 | rdsadmin       |                         654387789
     1 | template1      |                                 0
 16401 | mydb           |                            229556
 69768 | postgres       |                             22011
```

The following example gets the cumulative commit latency for the currently connected database. Before calling the `aurora_stat_get_db_commit_latency` function, the example first uses `\gset` to define a variable for the `oid` argument and sets its value from the connected database.

```
--Get the oid value from the connected database before calling
 aurora_stat_get_db_commit_latency
=> SELECT oid
     FROM pg_database
    WHERE datname=(SELECT current_database()) \gset
=> SELECT *
     FROM aurora_stat_get_db_commit_latency(:oid);

 aurora_stat_get_db_commit_latency
-----------------------------------
                        1424279160
```

The following example gets the cumulative commit latency for the `mydb` database in the `pg_database` cluster. Then, it resets this statistic by using the `pg_stat_reset` function and shows the results. Finally, it uses the `pg_stat_get_db_stat_reset_time` function to check the last time this statistic was reset.

```
=> SELECT oid,
    datname,
    aurora_stat_get_db_commit_latency(oid)
    FROM pg_database
    WHERE datname = 'mydb';

  oid  |  datname  | aurora_stat_get_db_commit_latency
-------+-----------+-----------------------------------
 16427 | mydb      |                           3320370
```

```
=> SELECT pg_stat_reset();
 pg_stat_reset
---------------



=> SELECT oid,
          datname,
          aurora_stat_get_db_commit_latency(oid)
     FROM pg_database
    WHERE datname = 'mydb';
  oid  |  datname  | aurora_stat_get_db_commit_latency
-------+-----------+-----------------------------------
 16427 | mydb      |                                 6


=> SELECT *
     FROM pg_stat_get_db_stat_reset_time(16427);

 pg_stat_get_db_stat_reset_time
----------------------------------
 2021-04-29 21:36:15.707399+00
```

aurora_stat_logical_wal_cache

Shows logical write-ahead log (WAL) cache usage per slot.

Syntax

```
SELECT * FROM aurora_stat_logical_wal_cache()
```

Arguments

None

Return type

SETOF record with the following columns:

- `name` – The name of the replication slot.
- `active_pid` – ID of the walsender process.
- `cache_hit` – The total number of wal cache hits since last reset.

- `cache_miss` – The total number of wal cache misses since last reset.
- `blks_read` – The total number of wal cache read requests.
- `hit_rate` – The WAL cache hit rate (cache_hit / blks_read).
- `last_reset_timestamp` – Last time that the counter was reset.

Usage notes

This function is available for the following versions.

- Aurora PostgreSQL 14.7
- Aurora PostgreSQL version 13.8 and higher
- Aurora PostgreSQL version 12.12 and higher
- Aurora PostgreSQLversion 11.17 and higher

Examples

The following example shows two replication slots with only one active. `aurora_stat_logical_wal_cache` function.

```
=> SELECT *
     FROM aurora_stat_logical_wal_cache();
    name    | active_pid | cache_hit | cache_miss | blks_read | hit_rate |
 last_reset_timestamp
------------+------------+-----------+------------+-----------+----------
+-------------------------------
 test_slot1 |      79183 |        24 |          0 |        24 | 100.00%  | 2022-08-05
 17:39:56.830635+00
 test_slot2 |            |         1 |          0 |         1 | 100.00%  | 2022-08-05
 17:34:04.036795+00
(2 rows)
```

aurora_stat_memctx_usage

Reports the memory context usage for each PostgreSQL process.

Syntax

```
aurora_stat_memctx_usage()
```

Arguments

None

Return type

SETOF record with the following columns:

- `pid` – The ID of the process.
- `name` – The name of the memory context.
- `allocated` – The number of bytes obtained from the underlying memory subsystem by the memory context.
- `used` – The number of bytes committed to clients of the memory context.
- `instances` – The count of currently existing contexts of this type.

Usage notes

This function displays the memory context usage for each PostgreSQL process. Some processes are labeled `anonymous`. The processes aren't exposed because they contain restricted keywords.

This function is available starting with the following Aurora PostgreSQL versions:

- 15.3 and higher 15 versions
- 14.8 and higher 14 versions
- 13.11 and higher 13 versions
- 12.15 and higher 12 versions
- 11.20 and higher 11 versions

Examples

The following example shows the results of calling the `aurora_stat_memctx_usage` function.

```
=> SELECT *
     FROM aurora_stat_memctx_usage();

    pid| name                            | allocated |   used  | instances
-------+---------------------------------+-----------+---------+-----------
123864 | Miscellaneous                   |     19520 |   15064 |         3
```

```
123864 | Aurora File Context             |      8192 |     616 |          1
123864 | Aurora WAL Context              |      8192 |     296 |          1
123864 | CacheMemoryContext              |    524288 |  422600 |          1
123864 | Catalog tuple context           |     16384 |   13736 |          1
123864 | ExecutorState                   |     32832 |   28304 |          1
123864 | ExprContext                     |      8192 |    1720 |          1
123864 | GWAL record construction        |      1024 |     832 |          1
123864 | MdSmgr                          |      8192 |     296 |          1
123864 | MessageContext                  |    532480 |  353832 |          1
123864 | PortalHeapMemory                |      1024 |     488 |          1
123864 | PortalMemory                    |      8192 |     576 |          1
123864 | printtup                        |      8192 |     296 |          1
123864 | RelCache hash table entries     |      8192 |    8152 |          1
123864 | RowDescriptionContext           |      8192 |    1344 |          1
123864 | smgr relation context           |      8192 |     296 |          1
123864 | Table function arguments        |      8192 |     352 |          1
123864 | TopTransactionContext           |      8192 |     632 |          1
123864 | TransactionAbortContext         |     32768 |     296 |          1
123864 | WAL record construction         |     50216 |   43904 |          1
123864 | hash table                      |     65536 |   52744 |          6
123864 | Relation metadata               |    191488 |  124240 |         87
104992 | Miscellaneous                   |      9280 |    7728 |          3
104992 | Aurora File Context             |      8192 |     376 |          1
104992 | Aurora WAL Context              |      8192 |     296 |          1
104992 ||Autovacuum Launcher             |      8192 |     296 |          1
104992 | Autovacuum database list        |     16384 |     744 |          2
104992 | CacheMemoryContext              |    262144 |  140288 |          1
104992 | Catalog tuple context           |      8192 |     296 |          1
104992 | GWAL record construction        |      1024 |     832 |          1
104992 | MdSmgr                          |      8192 |     296 |          1
104992 | PortalMemory                    |      8192 |     296 |          1
104992 | RelCache hash table entries     |      8192 |     296 |          1
104992 | smgr relation context           |      8192 |     296 |          1
104992 | Autovacuum start worker (tmp)   |      8192 |     296 |          1
104992 | TopTransactionContext           |     16384 |     592 |          2
104992 | TransactionAbortContext         |     32768 |     296 |          1
104992 | WAL record construction         |     50216 |   43904 |          1
104992 | hash table                      |     49152 |   34024 |          4
(39 rows)
```

Some restricted keywords will be hidden and the output will look as follows:

```
postgres=>SELECT *
```

```
     FROM aurora_stat_memctx_usage();

    pid| name                            | allocated |   used  | instances
-------+---------------------------------+-----------+---------+-----------
  5482 | anonymous                       |      8192 |     456 |         1
  5482 | anonymous                       |      8192 |     296 |         1
```

aurora_stat_optimized_reads_cache

This function shows tiered cache stats.

Syntax

```
aurora_stat_optimized_reads_cache()
```

Arguments

None

Return type

SETOF record with the following columns:

- `total_size` – Total optimized reads cache size.
- `used_size` – Used page size in optimized reads cache.

Usage notes

This function is available in the following Aurora PostgreSQL versions:

- 15.4 and higher 15 versions
- 14.9 and higher 14 versions

Examples

The following example shows the output on a r6gd.8xlarge instance :

```
=> select pg_size_pretty(total_size) as total_size, pg_size_pretty(used_size)
```

```
                 as used_size from aurora_stat_optimized_reads_cache();
total_size | used_size
-----------+-----------
1054 GB    | 975 GB
```

aurora_stat_plans

Returns a row for every tracked execution plan.

Syntax

```
aurora_stat_plans(
    showtext
)
```

Arguments

- showtext – Show the query and plan text. Valid values are NULL, true or false. True will show the query and plan text.

Return type

Returns a row for each tracked plan that contains all the columns from `aurora_stat_statements` and the following additional columns.

- planid – plan identifier
- explain_plan – explain plan text
- plan_type:
  - `no plan` - no plan was captured
  - `estimate` - plan captured with estimated costs
  - `actual` - plan captured with EXPLAIN ANALYZE
- plan_captured_time – last time a plan was captured

Usage notes

`aurora_compute_plan_id` must be enabled and `pg_stat_statements` must be in `shared_preload_libraries` for the plans to be tracked.

The number of plans available is controlled by the value set in the `pg_stat_statements.max` parameter. When `compute_plan_id` is enabled, you can track the plans up to this specified value in `aurora_stat_plans`.

This function is available from Aurora PostgreSQL versions 14.10, 15.5, and for all other later versions.

Examples

In the example below, the two plans that are for the query identifier -5471422286312252535 are captured and the statements statistics are tracked by the planid.

```
db1=# select calls, total_exec_time, planid, plan_captured_time, explain_plan
db1-# from aurora_stat_plans(true)
db1-# where queryid = '-5471422286312252535'

calls    |  total_exec_time   |   planid    |      plan_captured_time       |
                 explain_plan
---------+--------------------+-------------+-------------------------------
+------------------------------------------------------------------
 1532632 |  3209846.097107853 |  1602979607 | 2023-10-31 03:27:16.925497+00 | Update on
 pgbench_branches                                      +
         |                    |             |                               |   ->
 Bitmap Heap Scan on pgbench_branches                      +
         |                    |             |                               |
 Recheck Cond: (bid = 76)                                +
         |                    |             |                               |         -
>  Bitmap Index Scan on pgbench_branches_pkey          +
         |                    |             |                               |
     Index Cond: (bid = 76)
   61365 | 124078.18012200127 | -2054628807 | 2023-10-31 03:20:09.85429+00  | Update on
 pgbench_branches                                      +
         |                    |             |                               |   ->
 Index Scan using pgbench_branches_pkey on pgbench_branches+
         |                    |             |                               |
 Index Cond: (bid = 17)
```

aurora_stat_reset_wal_cache

Resets the counter for logical wal cache.

Syntax

To reset a specific slot

```
SELECT * FROM aurora_stat_reset_wal_cache('slot_name')
```

To reset all slots

```
SELECT * FROM aurora_stat_reset_wal_cache(NULL)
```

Arguments

`NULL` or `slot_name`

Return type

Status message, text string

- Reset the logical wal cache counter – Success message. This text is returned when the function succeeds.
- Replication slot not found. Please try again. – Failure message. This text is returned when the function doesn't succeed.

Usage notes

This function is available for the following versions.

- Aurora PostgreSQL 14.5 and higher
- Aurora PostgreSQL version 13.8 and higher
- Aurora PostgreSQL version 12.12 and higher
- Aurora PostgreSQLversion 11.17 and higher

Examples

The following example uses the `aurora_stat_reset_wal_cache` function to reset a slot named `test_results`, and then tries to reset a slot that doesn't exist.

```
=> SELECT *
     FROM aurora_stat_reset_wal_cache('test_slot');
```

```
aurora_stat_reset_wal_cache
--------------------------------------
 Reset the logical wal cache counter.
(1 row)
=> SELECT *
     FROM aurora_stat_reset_wal_cache('slot-not-exist');
aurora_stat_reset_wal_cache
-----------------------------------------------
 Replication slot not found. Please try again.
(1 row)
```

aurora_stat_statements

Displays all pg_stat_statements columns and adds more columns in the end.

Syntax

```
aurora_stat_statements(showtext boolean)
```

Arguments

showtext boolean

Return type

SETOF record with all pg_stat_statements columns and the following additional columns. For more information on pg_stat_statements columns, see pg_stat_statements.

You can reset stats for this function using pg_stat_statements_reset().

- storage_blks_read – Total number of shared blocks read from aurora storage by this statement.
- orcache_blks_hit – Total number of optimized reads cache hits by this statement.
- storage_blk_read_time – If track_io_timing is enabled, it tracks the total time the statement spent reading data file blocks from aurora storage, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.
- local_blk_read_time – If track_io_timing is enabled, it tracks the total time the statement spent reading local data file blocks, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.

- `orcache_blk_read_time` – If `track_io_timing` is enabled, it tracks the total time the statement spent reading data file blocks from optimized reads cache, in milliseconds, otherwise the value is zero. For more information, see track_io_timing.

Usage notes

To use the aurora_stat_statements() function, you must include `pg_stat_statements` extension in the `shared_preload_libraries` parameter.

This function is available in the following Aurora PostgreSQL versions:

- 15.4 and higher 15 versions
- 14.9 and higher 14 versions

Examples

The following example shows how it carries all the pg_stat_statements columns and append 5 new columns in the end:

```
=> select * from aurora_stat_statements(true) where queryid=-7342090857217643794;
-[ RECORD 1 ]---------+--------------------------------------------------
userid                | 10
dbid                  | 16419
toplevel              | t
queryid               | -7342090857217643794
query                 | CREATE TABLE quad_point_tbl AS                    +
                      |     SELECT point(unique1,unique2) AS p FROM tenk1
plans                 | 0
total_plan_time       | 0
min_plan_time         | 0
max_plan_time         | 0
mean_plan_time        | 0
stddev_plan_time      | 0
calls                 | 1
total_exec_time       | 571.844376
min_exec_time         | 571.844376
max_exec_time         | 571.844376
mean_exec_time        | 571.844376
stddev_exec_time      | 0
rows                  | 10000
shared_blks_hit       | 462
```

```
shared_blks_read      | 422
shared_blks_dirtied   | 0
shared_blks_written   | 55
local_blks_hit        | 0
local_blks_read       | 0
local_blks_dirtied    | 0
local_blks_written    | 0
temp_blks_read        | 0
temp_blks_written     | 0
blk_read_time         | 170.634621
blk_write_time        | 0
wal_records           | 0
wal_fpi               | 0
wal_bytes             | 0
storage_blks_read     | 47
orcache_blks_hit      | 375
storage_blk_read_time | 124.505772
local_blk_read_time   | 0
orcache_blk_read_time | 44.684038
```

aurora_stat_system_waits

Reports wait event information for the Aurora PostgreSQL DB instance.

Syntax

```
aurora_stat_system_waits()
```

Arguments

None

Return type

SETOF record

Usage notes

This function returns the cumulative number of waits and cumulative wait time for each wait event generated by the DB instance that you're currently connected to.

The returned recordset includes the following fields:

- type_id – The ID of the type of wait event.
- event_id – The ID of the wait event.
- waits – The number of times the wait event occurred.
- wait_time – The total amount of time in microseconds spent waiting for this event.

Statistics returned by this function are reset when a DB instance restarts.

Examples

The following example shows results from calling the aurora_stat_system_waits function.

```
=> SELECT *
    FROM aurora_stat_system_waits();
 type_id | event_id  |   waits   |  wait_time
---------+-----------+-----------+--------------
       1 |  16777219 |        11 |         12864
       1 |  16777220 |       501 |        174473
       1 |  16777270 |     53171 |      23641847
       1 |  16777271 |        23 |        319668
       1 |  16777274 |        60 |         12759
.
.
.
      10 | 167772231 |    204596 |     790945212
      10 | 167772232 |         2 |         47729
      10 | 167772234 |         1 |           888
      10 | 167772235 |         2 |            64
```

The following example shows how you can use this function together with aurora_stat_wait_event and aurora_stat_wait_type to produce more readable results.

```
=> SELECT type_name,
             event_name,
             waits,
             wait_time
        FROM aurora_stat_system_waits()
NATURAL JOIN aurora_stat_wait_event()
NATURAL JOIN aurora_stat_wait_type();

 type_name |       event_name        |  waits   |  wait_time
-----------+------------------------+----------+--------------
```

```
 LWLock     | XidGenLock             |       11 |        12864
 LWLock     | ProcArrayLock          |      501 |       174473
 LWLock     | buffer_content         |    53171 |     23641847
 LWLock     | rdsutils               |        2 |        12764
 Lock       | tuple                  |    75686 |   2033956052
 Lock       | transactionid          |  1765147 |  47267583409
 Activity   | AutoVacuumMain         |   136868 |  56305604538
 Activity   | BgWriterHibernate      |     7486 |  55266949471
 Activity   | BgWriterMain           |     7487 |   1508909964
.
.
.
 IO         | SLRURead               |        3 |        11756
 IO         | WALWrite               | 52544463 |    388850428
 IO         | XactSync               |   187073 |    597041642
 IO         | ClogRead               |        2 |        47729
 IO         | OutboundCtrlRead       |        1 |          888
 IO         | OutboundCtrlWrite      |        2 |           64
```

aurora_stat_wait_event

Lists all supported wait events for Aurora PostgreSQL. For information about Aurora PostgreSQL wait events, see Amazon Aurora PostgreSQL wait events.

Syntax

```
aurora_stat_wait_event()
```

Arguments

None

Return type

SETOF record with the following columns:

- type_id – The ID of the type of wait event.
- event_id – The ID of the wait event.
- type_name – Wait type name
- event_name – Wait event name

Usage notes

To see event names with event types instead of IDs, use this function together with other functions such as `aurora_stat_wait_type` and `aurora_stat_system_waits`. Wait event names returned by this function are the same as those returned by the `aurora_wait_report` function.

Examples

The following example shows results from calling the `aurora_stat_wait_event` function.

```
=>  SELECT *
    FROM aurora_stat_wait_event();

 type_id | event_id  |                event_name
---------+-----------+------------------------------------------
       1 |  16777216 | <unassigned:0>
       1 |  16777217 | ShmemIndexLock
       1 |  16777218 | OidGenLock
       1 |  16777219 | XidGenLock
.
.
.
       9 | 150994945 | PgSleep
       9 | 150994946 | RecoveryApplyDelay
      10 | 167772160 | BufFileRead
      10 | 167772161 | BufFileWrite
      10 | 167772162 | ControlFileRead
.
.
.
      10 | 167772226 | WALInitWrite
      10 | 167772227 | WALRead
      10 | 167772228 | WALSync
      10 | 167772229 | WALSyncMethodAssign
      10 | 167772230 | WALWrite
      10 | 167772231 | XactSync
.
.
.
      11 | 184549377 | LsnAllocate
```

The following example joins `aurora_stat_wait_type` and `aurora_stat_wait_event` to return type names and event names for improved readability.

```
=> SELECT *
    FROM aurora_stat_wait_type() t
    JOIN aurora_stat_wait_event() e
      ON t.type_id = e.type_id;

 type_id | type_name | type_id | event_id  |                event_name
---------+-----------+---------+-----------+-------------------------------------------
       1 | LWLock    |       1 |  16777216 | <unassigned:0>
       1 | LWLock    |       1 |  16777217 | ShmemIndexLock
       1 | LWLock    |       1 |  16777218 | OidGenLock
       1 | LWLock    |       1 |  16777219 | XidGenLock
       1 | LWLock    |       1 |  16777220 | ProcArrayLock
.
.
.
       3 | Lock      |       3 |  50331648 | relation
       3 | Lock      |       3 |  50331649 | extend
       3 | Lock      |       3 |  50331650 | page
       3 | Lock      |       3 |  50331651 | tuple
.
.
.
      10 | IO        |      10 | 167772214 | TimelineHistorySync
      10 | IO        |      10 | 167772215 | TimelineHistoryWrite
      10 | IO        |      10 | 167772216 | TwophaseFileRead
      10 | IO        |      10 | 167772217 | TwophaseFileSync
.
.
.
      11 | LSN       |      11 | 184549376 | LsnDurable
```

aurora_stat_wait_type

Lists all supported wait types for Aurora PostgreSQL.

Syntax

```
aurora_stat_wait_type()
```

Arguments

None

Return type

SETOF record with the following columns:

- type_id – The ID of the type of wait event.
- type_name – Wait type name.

Usage notes

To see wait event names with wait event types instead of IDs, use this function together with other functions such as `aurora_stat_wait_event` and `aurora_stat_system_waits`. Wait type names returned by this function are the same as those returned by the `aurora_wait_report` function.

Examples

The following example shows results from calling the `aurora_stat_wait_type` function.

```
=> SELECT *
     FROM aurora_stat_wait_type();
 type_id | type_name
---------+-----------
       1 | LWLock
       3 | Lock
       4 | BufferPin
       5 | Activity
       6 | Client
       7 | Extension
       8 | IPC
       9 | Timeout
      10 | IO
      11 | LSN
```

aurora_version

Returns the string value of the Amazon Aurora PostgreSQL-Compatible Edition version number.

Syntax

```
aurora_version()
```

Arguments

None

Return type

CHAR or VARCHAR string

Usage notes

This function displays the version of the Amazon Aurora PostgreSQL-Compatible Edition database engine. The version number is returned as a string formatted as *major*.*minor*.*patch*. For more information about Aurora PostgreSQL version numbers, see Aurora version number.

You can choose when to apply minor version upgrades by setting the maintenance window for your Aurora PostgreSQL DB cluster. To learn how, see Maintaining an Amazon Aurora DB cluster.

Starting with the release of Aurora PostgreSQL versions 13.3, 12.8, 11.13, 10.18, and for all other later versions, Aurora version numbers follow PostgreSQL version numbers. For more information about all Aurora PostgreSQL releases, see Amazon Aurora PostgreSQL updates in the *Release Notes for Aurora PostgreSQL*.

Examples

The following example shows the results of calling the `aurora_version` function on an Aurora PostgreSQL DB cluster running PostgreSQL 12.7, Aurora PostgreSQL release 4.2 and then running the same function on a cluster running Aurora PostgreSQL version 13.3.

```
=> SELECT * FROM aurora_version();
aurora_version
----------------
 4.2.2
SELECT * FROM aurora_version();
aurora_version
----------------
 13.3.0
```

This example shows how to use the function with various options to get more detail about the Aurora PostgreSQL version. This example has an Aurora version number that's distinct from the PostgreSQL version number.

```
=> SHOW SERVER_VERSION;
 server_version
----------------
 12.7
(1 row)

=> SELECT * FROM aurora_version();
aurora_version
----------------
 4.2.2
(1 row)

=> SELECT current_setting('server_version') AS "PostgreSQL Compatiblility";
PostgreSQL Compatiblility
---------------------------
 12.7
(1 row)

=> SELECT version() AS "PostgreSQL Compatiblility Full String";
PostgreSQL Compatiblility Full String
-----------------------------------------------------------------------------------------------
 PostgreSQL 12.7 on aarch64-unknown-linux-gnu, compiled by aarch64-unknown-linux-gnu-
gcc (GCC) 7.4.0, 64-bit
(1 row)

=> SELECT 'Aurora: '
    || aurora_version()
    || ' Compatible with PostgreSQL: '
    || current_setting('server_version') AS "Instance Version";
Instance Version
------------------------------------------------
 Aurora: 4.2.2 Compatible with PostgreSQL: 12.7
(1 row)
```

This next example uses the function with the same options in the previous example. This example doesn't have an Aurora version number that's distinct from the PostgreSQL version number.

```
=> SHOW SERVER_VERSION;
server_version
----------------
 13.3
```

```
=> SELECT * FROM aurora_version();
aurora_version
----------------
 13.3.0
=> SELECT current_setting('server_version') AS "PostgreSQL Compatiblility";
PostgreSQL Compatiblility
---------------------------
 13.3

=> SELECT version() AS "PostgreSQL Compatiblility Full String";
PostgreSQL Compatiblility Full String
-----------------------------------------------------------------------------------------------
 PostgreSQL 13.3 on x86_64-pc-linux-gnu, compiled by x86_64-pc-linux-gnu-gcc (GCC)
 7.4.0, 64-bit
=> SELECT 'Aurora: '
    || aurora_version()
    || ' Compatible with PostgreSQL: '
    || current_setting('server_version') AS "Instance Version";
Instance Version
-------------------------------------------------------
 Aurora: 13.3.0 Compatible with PostgreSQL: 13.3
```

aurora_volume_logical_start_lsn

Returns the log sequence number (LSN) used for identifying the beginning of a record in the logical write-ahead log (WAL) stream of the Aurora cluster volume.

Syntax

```
aurora_volume_logical_start_lsn()
```

Arguments

None

Return type

`pg_lsn`

Usage notes

This function identifies the beginning of the record in the logical WAL stream for a given Aurora cluster volume. You can use this function while performing major version upgrade using logical replication and Aurora fast cloning to determine the LSN at which a snapshot or database clone is taken. You can then use logical replication to continuously stream the newer data recorded after the LSN and synchronize the changes from publisher to subscriber.

For more information on using logical replication for a major version upgrade, see Using logical replication to perform a major version upgrade for Aurora PostgreSQL.

This function is available on the following versions of Aurora PostgreSQL:

- 15.2 and higher 15 versions
- 14.3 and higher 14 versions
- 13.6 and higher 13 versions
- 12.10 and higher 12 versions
- 11.15 and higher 11 versions
- 10.20 and higher 10 versions

Examples

You can obtain the log sequence number (LSN) using the following query:

```
postgres=> SELECT aurora_volume_logical_start_lsn();

aurora_volume_logical_start_lsn
---------------
0/402E2F0
(1 row)
```

aurora_wait_report

This function shows wait event activity over a period of time.

Syntax

```
aurora_wait_report([time])
```

Arguments

time (optional)

The time in seconds. Default is 10 seconds.

Return type

SETOF record with the following columns:

- type_name – Wait type name
- event_name – Wait event name
- wait – Number of waits
- wait_time – Wait time in milliseconds
- ms_per_wait – Average milliseconds by the number of an wait
- waits_per_xact – Average waits by the number of one transaction
- ms_per_xact – Average milliseconds by the number of transactions

Usage notes

This function is available as of Aurora PostgreSQL release 1.1 compatible with PostgreSQL 9.6.6 and higher versions.

To use this function, you need to first create the Aurora PostgreSQL `aurora_stat_utils` extension, as follows:

```
=> CREATE extension aurora_stat_utils;
CREATE EXTENSION
```

For more information about available Aurora PostgreSQL extension versions, see Extension versions for Amazon Aurora PostgreSQL in *Release Notes for Aurora PostgreSQL*.

This function calculates the instance-level wait events by comparing two snapshots of statistics data from aurora_stat_system_waits() function and pg_stat_database PostgreSQL Statistics Views.

For more information about `aurora_stat_system_waits()` and `pg_stat_database`, see The Statistics Collector in the *PostgreSQL documentation*.

When run, this function takes an initial snapshot, waits the number of seconds specified, and then takes a second snapshot. The function compares the two snapshots and returns the difference. This difference represents the instance's activity for that time interval.

On the writer instance, the function also displays the number of committed transactions and TPS (transactions per second). This function returns information at the instance level and includes all databases on the instance.

Examples

This example shows how to create aurora_stat_utils extension to be able to use aurora_log_report function.

```
=> CREATE extension aurora_stat_utils;
CREATE EXTENSION
```

This example shows how to check wait report for 10 seconds.

```
=> SELECT *
     FROM aurora_wait_report();
NOTICE:  committed 34 transactions in 10 seconds (tps 3)
 type_name |    event_name     | waits | wait_time | ms_per_wait | waits_per_xact |
 ms_per_xact
-----------+-------------------+-------+-----------+-------------+----------------
+-------------
 Client    | ClientRead        |    26 |  30003.00 |    1153.961 |           0.76 |
 882.441
 Activity  | WalWriterMain     |    50 |  10051.32 |     201.026 |           1.47 |
 295.627
 Timeout   | PgSleep           |     1 |  10049.52 |   10049.516 |           0.03 |
 295.574
 Activity  | BgWriterHibernate |     1 |  10048.15 |   10048.153 |           0.03 |
 295.534
 Activity  | AutoVacuumMain    |    18 |   9941.66 |     552.314 |           0.53 |
 292.402
 Activity  | BgWriterMain      |     1 |    201.09 |     201.085 |           0.03 |
   5.914
 IO        | XactSync          |    15 |     25.34 |       1.690 |           0.44 |
   0.745
 IO        | RelationMapRead   |    12 |      0.54 |       0.045 |           0.35 |
   0.016
 IO        | WALWrite          |    84 |      0.21 |       0.002 |           2.47 |
   0.006
 IO        | DataFileExtend     |      1 |      0.02 |       0.018 |            0.03 |
   0.001
```

This example shows how to check wait report for 60 seconds.

```
=> SELECT *
     FROM aurora_wait_report(60);
NOTICE:  committed 1544 transactions in 60 seconds (tps 25)
 type_name |       event_name       |  waits  | wait_time | ms_per_wait |
 waits_per_xact | ms_per_xact
-----------+------------------------+---------+-----------+-------------
+----------------+-------------
 Lock      | transactionid          |    6422 | 477000.53 |      74.276 |
 4.16 |     308.938
 Client    | ClientRead             |    8265 | 270752.99 |      32.759 |
 5.35 |     175.358
 Activity  | CheckpointerMain       |       1 |  60100.25 |   60100.246 |
 0.00 |      38.925
 Timeout   | PgSleep                |       1 |  60098.49 |   60098.493 |
 0.00 |      38.924
 Activity  | WalWriterMain          |     296 |  60010.99 |     202.740 |
 0.19 |      38.867
 Activity  | AutoVacuumMain         |     107 |  59827.84 |     559.139 |
 0.07 |      38.749
 Activity  | BgWriterMain           |     290 |  58821.83 |     202.834 |
 0.19 |      38.097
 IO        | XactSync               |    1295 |  55220.13 |      42.641 |
 0.84 |      35.764
 IO        | WALWrite               | 6602259 |  47810.94 |       0.007 |
 4276.07 |      30.966
 Lock      | tuple                  |     473 |  29880.67 |      63.173 |
 0.31 |      19.353
 LWLock    | buffer_mapping         |     142 |   3540.13 |      24.930 |
 0.09 |       2.293
 Activity  | BgWriterHibernate      |     290 |   1124.15 |       3.876 |
 0.19 |       0.728
 IO        | BufFileRead            |    7615 |    618.45 |       0.081 |
 4.93 |       0.401
 LWLock    | buffer_content         |      73 |    345.93 |       4.739 |
 0.05 |       0.224
 LWLock    | lock_manager           |      62 |    191.44 |       3.088 |
 0.04 |       0.124
```

```
IO        | RelationMapRead        |      72 |      5.16 |       0.072 |
0.05 |       0.003
LWLock    | ProcArrayLock          |       1 |      2.01 |       2.008 |
0.00 |       0.001
IO        | ControlFileWriteUpdate |       2 |      0.03 |       0.013 |
0.00 |       0.000
IO        | DataFileExtend         |       1 |      0.02 |       0.018 |
0.00 |       0.000
IO        | ControlFileSyncUpdate  |       1 |      0.00 |       0.000 |
0.00 |       0.000
```

Amazon Aurora PostgreSQL parameters

You manage your Amazon Aurora DB cluster in the same way that you manage Amazon RDS DB instances, by using parameters in a DB parameter group. However, Amazon Aurora differs from Amazon RDS in that an Aurora DB cluster has multiple DB instances. Some of the parameters that you use to manage your Amazon Aurora DB cluster apply to the entire cluster, while other parameters apply only to a given DB instance in the DB cluster, as follows:

- **DB cluster parameter group** – A DB cluster parameter group contains the set of engine configuration parameters that apply throughout the Aurora DB cluster. For example, cluster cache management is a feature of an Aurora DB cluster that's controlled by the `apg_ccm_enabled` parameter which is part of the DB cluster parameter group. The DB cluster parameter group also contains default settings for the DB parameter group for the DB instances that make up the cluster.
- **DB parameter group** – A DB parameter group is the set of engine configuration values that apply to a specific DB instance of that engine type. The DB parameter groups for the PostgreSQL DB engine are used by an RDS for PostgreSQL DB instance and Aurora PostgreSQL DB cluster. These configuration settings apply to properties that can vary among the DB instances within an Aurora cluster, such as the sizes for memory buffers.

You manage cluster-level parameters in DB cluster parameter groups. You manage instance-level parameters in DB parameter groups. You can manage parameters using the Amazon RDS console, the AWS CLI, or the Amazon RDS API. There are separate commands for managing cluster-level parameters and instance-level parameters.

- To manage cluster-level parameters in a DB cluster parameter group, use the modify-db-cluster-parameter-group AWS CLI command.

- To manage instance-level parameters in a DB parameter group for a DB instance in a DB cluster, use the modify-db-parameter-group AWS CLI command.

To learn more about the AWS CLI, see Using the AWS CLI in the *AWS Command Line Interface User Guide*.

For more information about parameter groups, see Working with parameter groups.

Viewing Aurora PostgreSQL DB cluster and DB parameters

You can view all available default parameter groups for RDS for PostgreSQL DB instances and for Aurora PostgreSQL DB clusters in the AWS Management Console. The default parameter groups for all DB engines and DB cluster types and versions are listed for each AWS Region. Any custom parameter groups are also listed.

Rather than viewing in the AWS Management Console, you can also list parameters contained in DB cluster parameter groups and DB parameter groups by using the AWS CLI or the Amazon RDS API. For example, to list parameters in a DB cluster parameter group you use the describe-db-cluster-parameters AWS CLI command as follows:

```
aws rds describe-db-cluster-parameters --db-cluster-parameter-group-name
 default.aurora-postgresql12
```

The command returns detailed JSON descriptions of each parameter. To reduce the amount of information returned, you can specify what you want by using the `--query` option. For example, you can get the parameter name, its description, and allowed values for the default Aurora PostgreSQL 12 DB cluster parameter group as follows:

For Linux, macOS, or Unix:

```
aws rds describe-db-cluster-parameters --db-cluster-parameter-group-name
 default.aurora-postgresql12 \
    --query 'Parameters[].
[{ParameterName:ParameterName,Description:Description,ApplyType:ApplyType,AllowedValues:Allowed
```

For Windows:

```
aws rds describe-db-cluster-parameters --db-cluster-parameter-group-name
 default.aurora-postgresql12 ^
```

```
    --query "Parameters[].
[{ParameterName:ParameterName,Description:Description,ApplyType:ApplyType,AllowedValues:Allowed
```

An Aurora DB cluster parameter group includes the DB instance parameter group and default values for a given Aurora DB engine. You can get the list of DB parameters from the same default Aurora PostgreSQL default parameter group by usng the describe-db-parameters AWS CLI command as shown following.

For Linux, macOS, or Unix:

```
aws rds describe-db-parameters --db-parameter-group-name default.aurora-postgresql12 \
    --query 'Parameters[].
[{ParameterName:ParameterName,Description:Description,ApplyType:ApplyType,AllowedValues:Allowed
```

For Windows:

```
aws rds describe-db-parameters --db-parameter-group-name default.aurora-postgresql12 ^
    --query "Parameters[].
[{ParameterName:ParameterName,Description:Description,ApplyType:ApplyType,AllowedValues:Allowed
```

The preceding commands return lists of parameters from the DB cluster or DB parameter group with descriptions and other details specified in the query. Following is an example response.

```
[
    [
        {
            "ParameterName": "apg_enable_batch_mode_function_execution",
            "ApplyType": "dynamic",
            "Description": "Enables batch-mode functions to process sets of rows at a
 time.",
            "AllowedValues": "0,1"
        }
    ],
    [
        {
            "ParameterName": "apg_enable_correlated_any_transform",
            "ApplyType": "dynamic",
            "Description": "Enables the planner to transform correlated ANY Sublink
 (IN/NOT IN subquery) to JOIN when possible.",
            "AllowedValues": "0,1"
        }
    ],...
```

Following are tables containing values for the default DB cluster parameter and DB parameter for Aurora PostgreSQL version 14.

Aurora PostgreSQL cluster-level parameters

You can view the cluster-level parameters available for a specific Aurora PostgreSQL version using the AWS Management console, the AWS CLI, or the Amazon RDS API. For information about viewing the parameters in an Aurora PostgreSQL DB cluster parameter groups in the RDS console, see Viewing parameter values for a DB cluster parameter group.

Some cluster-level parameters aren't available in all versions and some are being deprecated. For information about viewing the parameters of a specific Aurora PostgreSQL version, see Viewing Aurora PostgreSQL DB cluster and DB parameters.

For example, the following table lists the parameters available in the default DB cluster parameter group for Aurora PostgreSQL version 14. If you create an Aurora PostgreSQL DB cluster without specifying your own custom DB parameter group, your DB cluster is created using the default Aurora DB cluster parameter group for the version chosen, such as `default.aurora-postgresql14`, `default.aurora-postgresql13`, and so on.

For a listing of the DB instance parameters for this same default DB cluster parameter group, see Aurora PostgreSQL instance-level parameters.

Parameter name	Description	Default
ansi_constraint_trigger_ordering	Change the firing order of constraint triggers to be compatible with the ANSI SQL standard.	–
ansi_force_foreign_key_checks	Ensure referential actions such as cascaded delete or cascaded update will always occur regardless of the various trigger contexts that exist for the action.	–
ansi_qualified_update_set_target	Support table and schema qualifiers in UPDATE ... SET statements.	–
apg_ccm_enabled	Enable or disable cluster cache management for the cluster.	–
apg_enable_batch_mode_function_execution	Enables batch-mode functions to process sets of rows at a time.	–

Parameter name	Description	Default
apg_enable_correlated_any_transform	Enables the planner to transform correlated ANY Sublink (IN/NOT IN subquery) to JOIN when possible.	–
apg_enable_function_migration	Enables the planner to migrate eligible scalar functions to the FROM clause.	–
apg_enable_not_in_transform	Enables the planner to transform NOT IN subquery to ANTI JOIN when possible.	–
apg_enable_remove_redundant_inner_joins	Enables the planner to remove redundant inner joins.	–
apg_enable_semijoin_push_down	Enables the use of semijoin filters for hash joins.	–
apg_plan_mgmt.capture_plan_baselines	Capture plan baseline mode. manual - enable plan capture for any SQL statement, off - disable plan capture, automatic - enable plan capture for for statements in pg_stat_statements that satisfy the eligibility criteria.	off
apg_plan_mgmt.max_databases	Sets the maximum number of databases that that may manage queries using apg_plan_mgmt.	10
apg_plan_mgmt.max_plans	Sets the maximum number of plans that may be cached by apg_plan_mgmt.	10000
apg_plan_mgmt.plan_retention_period	Maximum number of days since a plan was last_used before a plan will be automatically deleted.	32

Parameter name	Description	Default
apg_plan_ mgmt.unapproved_plan_execution_threshold	Estimated total plan cost below which an Unapproved plan will be executed.	0
apg_plan_mgmt.use_ plan_baselines	Use only approved or fixed plans for managed statements.	false
application_name	Sets the application name to be reported in statistics and logs.	–
array_nulls	Enable input of NULL elements in arrays.	–
aurora_compute_plan_id	Monitors query execution plans to detect the execution plans contributing to current database load and to track performance statistics of execution plans over time. For more information, see Monitoring query execution plans for Aurora PostgreSQL.	on
authentication_timeout	(s) Sets the maximum allowed time to complete client authentication.	–
auto_explain.log_analyze	Use EXPLAIN ANALYZE for plan logging.	–
auto_explain.log_buffers	Log buffers usage.	–
auto_explain.log_format	EXPLAIN format to be used for plan logging.	–
auto_explain.log_min_duration	Sets the minimum execution time above which plans will be logged.	–
auto_explain.log_nested_statements	Log nested statements.	–

Parameter name	Description	Default
auto_explain.log_timing	Collect timing data, not just row counts.	–
auto_explain.log_triggers	Include trigger statistics in plans.	–
auto_explain.log_verbose	Use EXPLAIN VERBOSE for plan logging.	–
auto_explain.sample_rate	Fraction of queries to process.	–
autovacuum	Starts the autovacuum subprocess.	–
autovacuum_analyze_scale_factor	Number of tuple inserts, updates or deletes prior to analyze as a fraction of reltuples.	0.05
autovacuum_analyze_threshold	Minimum number of tuple inserts, updates or deletes prior to analyze.	–
autovacuum_freeze_max_age	Age at which to autovacuum a table to prevent transaction ID wraparound.	–
autovacuum_max_workers	Sets the maximum number of simultaneously running autovacuum worker processes.	GREATEST(DBInstanceClassMemory/64371566592,3)
autovacuum_multixact_freeze_max_age	Multixact age at which to autovacuum a table to prevent multixact wraparound.	–
autovacuum_naptime	(s) Time to sleep between autovacuum runs.	5
autovacuum_vacuum_cost_delay	(ms) Vacuum cost delay in milliseconds, for autovacuum.	5

Parameter name	Description	Default
autovacuum_vacuum_cost_limit	Vacuum cost amount available before napping, for autovacuum.	GREATEST(log(DBInstanceClassMemory/21474836480)*600,200)
autovacuum_vacuum_insert_scale_factor	Number of tuple inserts prior to vacuum as a fraction of reltuples.	–
autovacuum_vacuum_insert_threshold	Minimum number of tuple inserts prior to vacuum, or -1 to disable insert vacuums.	–
autovacuum_vacuum_scale_factor	Number of tuple updates or deletes prior to vacuum as a fraction of reltuples.	0.1
autovacuum_vacuum_threshold	Minimum number of tuple updates or deletes prior to vacuum.	–
autovacuum_work_mem	(kB) Sets the maximum memory to be used by each autovacuum worker process.	GREATEST(DBInstanceClassMemory/32768,131072)
babelfishpg_tds.default_server_name	Default Babelfish server name	Microsoft SQL Server
babelfishpg_tds.listen_addresses	Sets the host name or IP address(es) to listen TDS to.	*
babelfishpg_tds.port	Sets the TDS TCP port the server listens on.	1433
babelfishpg_tds.tds_debug_log_level	Sets logging level in TDS, 0 disables logging	1

Parameter name	Description	Default
babelfishpg_tds.tds_default_numeric_precision	Sets the default precision of numeric type to be sent in the TDS column metadata if the engine does not specify one.	38
babelfishpg_tds.tds_default_numeric_scale	Sets the default scale of numeric type to be sent in the TDS column metadata if the engine does not specify one.	8
babelfishpg_tds.tds_default_packet_size	Sets the default packet size for all the SQL Server clients being connected	4096
babelfishpg_tds.tds_default_protocol_version	Sets a default TDS protocol version for all the clients being connected	DEFAULT
babelfishpg_tds.tds_ssl_encrypt	Sets the SSL Encryption option	0
babelfishpg_tds.tds_ssl_max_protocol_version	Sets the maximum SSL/TLS protocol version to use for tds session.	TLSv1.2
babelfishpg_tds.tds_ssl_min_protocol_version	Sets the minimum SSL/TLS protocol version to use for tds session.	TLSv1.2 from Aurora PostgreSQL version 16, TLSv1 for versions older than Aurora PostgreSQL version 16
babelfishpg_tsql.default_locale	Default locale to be used for collations created by CREATE COLLATION.	en-US

Parameter name	Description	Default
babelfishpg_tsql.migration_mode	Defines if multiple user databases are supported	multi-db from Aurora PostgreSQL version 16, single-db for versions older than Aurora PostgreSQL version 16
babelfishpg_tsql.server_collation_name	Name of the default server collation	sql_latin1_general_cp1_ci_as
babelfishpg_tsql.version	Sets the output of @@VERSION variable	default
backend_flush_after	(8Kb) Number of pages after which previously performed writes are flushed to disk.	–
backslash_quote	Sets whether \\ is allowed in string literals.	–
backtrace_functions	Log backtrace for errors in these functions.	–
bytea_output	Sets the output format for bytea.	–
check_function_bodies	Check function bodies during CREATE FUNCTION.	–
client_connection_check_interval	Sets the time interval between checks for disconnection while running queries.	–
client_encoding	Sets the clients character set encoding.	UTF8
client_min_messages	Sets the message levels that are sent to the client.	–
compute_query_id	Compute query identifiers.	auto
config_file	Sets the servers main configuration file.	/rdsdbdata/config/postgresql.conf

Parameter name	Description	Default
constraint_exclusion	Enables the planner to use constraints to optimize queries.	–
cpu_index_tuple_cost	Sets the planners estimate of the cost of processing each index entry during an index scan.	–
cpu_operator_cost	Sets the planners estimate of the cost of processing each operator or function call.	–
cpu_tuple_cost	Sets the planners estimate of the cost of processing each tuple (row).	–
cron.database_name	Sets the database to store pg_cron metadata tables	postgres
cron.log_run	Log all jobs runs into the job_run_details table	on
cron.log_statement	Log all cron statements prior to execution.	off
cron.max_running_jobs	Maximum number of jobs that can run concurrently.	5
cron.use_background_workers	Enables background workers for pg_cron	on
cursor_tuple_fraction	Sets the planners estimate of the fraction of a cursors rows that will be retrieved.	–
data_directory	Sets the servers data directory.	/rdsdbdata/db
datestyle	Sets the display format for date and time values.	–
db_user_namespace	Enables per-database user names.	–
deadlock_timeout	(ms) Sets the time to wait on a lock before checking for deadlock.	–

Parameter name	Description	Default
debug_pretty_print	Indents parse and plan tree displays.	–
debug_print_parse	Logs each querys parse tree.	–
debug_print_plan	Logs each querys execution plan.	–
debug_print_rewritten	Logs each querys rewritten parse tree.	–
default_statistics_target	Sets the default statistics target.	–
default_tablespace	Sets the default tablespace to create tables and indexes in.	–
default_toast_compression	Sets the default compression method for compressible values.	–
default_transaction_deferrable	Sets the default deferrable status of new transactions.	–
default_transaction_isolation	Sets the transaction isolation level of each new transaction.	–
default_transaction_read_only	Sets the default read-only status of new transactions.	–
effective_cache_size	(8kB) Sets the planners assumption about the size of the disk cache.	SUM(DBInstanceClassMemory/12038,-50003)
effective_io_concurrency	Number of simultaneous requests that can be handled efficiently by the disk subsystem.	–
enable_async_append	Enables the planners use of async append plans.	–

Parameter name	Description	Default
enable_bitmapscan	Enables the planners use of bitmap-scan plans.	–
enable_gathermerge	Enables the planners use of gather merge plans.	–
enable_hashagg	Enables the planners use of hashed aggregati on plans.	–
enable_hashjoin	Enables the planners use of hash join plans.	–
enable_incremental _sort	Enables the planners use of incremental sort steps.	–
enable_indexonlyscan	Enables the planners use of index-only-scan plans.	–
enable_indexscan	Enables the planners use of index-scan plans.	–
enable_material	Enables the planners use of materialization.	–
enable_memoize	Enables the planners use of memoization	–
enable_mergejoin	Enables the planners use of merge join plans.	–
enable_nestloop	Enables the planners use of nested-loop join plans.	–
enable_parallel_ap pend	Enables the planners use of parallel append plans.	–
enable_parallel_hash	Enables the planners user of parallel hash plans.	–
enable_partition_p runing	Enable plan-time and run-time partition pruning.	–

Parameter name	Description	Default
enable_partitionwise_aggregate	Enables partitionwise aggregation and grouping.	–
enable_partitionwise_join	Enables partitionwise join.	–
enable_seqscan	Enables the planners use of sequential-scan plans.	–
enable_sort	Enables the planners use of explicit sort steps.	–
enable_tidscan	Enables the planners use of TID scan plans.	–
escape_string_warning	Warn about backslash escapes in ordinary string literals.	–
exit_on_error	Terminate session on any error.	–
extra_float_digits	Sets the number of digits displayed for floating-point values.	–
force_parallel_mode	Forces use of parallel query facilities.	–
from_collapse_limit	Sets the FROM-list size beyond which subqueries are not collapsed.	–
geqo	Enables genetic query optimization.	–
geqo_effort	GEQO: effort is used to set the default for other GEQO parameters.	–
geqo_generations	GEQO: number of iterations of the algorithm.	–
geqo_pool_size	GEQO: number of individuals in the population.	–
geqo_seed	GEQO: seed for random path selection.	–

Parameter name	Description	Default
geqo_selection_bias	GEQO: selective pressure within the population.	–
geqo_threshold	Sets the threshold of FROM items beyond which GEQO is used.	–
gin_fuzzy_search_limit	Sets the maximum allowed result for exact search by GIN.	–
gin_pending_list_limit	(kB) Sets the maximum size of the pending list for GIN index.	–
hash_mem_multiplier	Multiple of work_mem to use for hash tables.	–
hba_file	Sets the servers hba configuration file.	/rdsdbdata/config/pg_hba.conf
hot_standby_feedback	Allows feedback from a hot standby to the primary that will avoid query conflicts.	on
huge_pages	Reduces overhead when a DB instance is working with large contiguous chunks of memory, such as that used by shared buffers. It is turned on by default for all the DB instance classes other than t3.medium,db.t3.large,db.t4g.medium,db.t4g.large instance classes.	on
ident_file	Sets the servers ident configuration file.	/rdsdbdata/config/pg_ident.conf
idle_in_transaction_session_timeout	(ms) Sets the maximum allowed duration of any idling transaction.	86400000

Parameter name	Description	Default
idle_session_timeout	Terminate any session that has been idle (that is, waiting for a client query), but not within an open transaction, for longer than the specified amount of time	–
intervalstyle	Sets the display format for interval values.	–
join_collapse_limit	Sets the FROM-list size beyond which JOIN constructs are not flattened.	–
krb_caseins_users	Sets whether GSSAPI (Generic Security Service API) user names should be treated case-insensitively (true) or not. By default, this parameter is set to false, so Kerberos expects user names to be case sensitive. For more information, see GSSAPI Authentication in the PostgreSQL documentation.	false
lc_messages	Sets the language in which messages are displayed.	–
lc_monetary	Sets the locale for formatting monetary amounts.	–
lc_numeric	Sets the locale for formatting numbers.	–
lc_time	Sets the locale for formatting date and time values.	–
listen_addresses	Sets the host name or IP address(es) to listen to.	*
lo_compat_privileges	Enables backward compatibility mode for privilege checks on large objects.	0
log_autovacuum_min_duration	(ms) Sets the minimum execution time above which autovacuum actions will be logged.	10000

Parameter name	Description	Default
log_connections	Logs each successful connection.	–
log_destination	Sets the destination for server log output.	stderr
log_directory	Sets the destination directory for log files.	/rdsdbdata/log/error
log_disconnections	Logs end of a session, including duration.	–
log_duration	Logs the duration of each completed SQL statement.	–
log_error_verbosity	Sets the verbosity of logged messages.	–
log_executor_stats	Writes executor performance statistics to the server log.	–
log_file_mode	Sets the file permissions for log files.	0644
log_filename	Sets the file name pattern for log files.	postgresql.log.%Y-%m-%d-%H%M
logging_collector	Start a subprocess to capture stderr output and/or csvlogs into log files.	1
log_hostname	Logs the host name in the connection logs.	0
logical_decoding_work_mem	(kB) This much memory can be used by each internal reorder buffer before spilling to disk.	–
log_line_prefix	Controls information prefixed to each log line.	%t:%r:%u@%d:%p]:
log_lock_waits	Logs long lock waits.	–
log_min_duration_sample	(ms) Sets the minimum execution time above which a sample of statements will be logged. Sampling is determined by log_statement_sample_rate.	–

Parameter name	Description	Default
log_min_duration_statement	(ms) Sets the minimum execution time above which statements will be logged.	–
log_min_error_statement	Causes all statements generating error at or above this level to be logged.	–
log_min_messages	Sets the message levels that are logged.	–
log_parameter_max_length	(B) When logging statements, limit logged parameter values to first N bytes.	–
log_parameter_max_length_on_error	(B) When reporting an error, limit logged parameter values to first N bytes.	–
log_parser_stats	Writes parser performance statistics to the server log.	–
log_planner_stats	Writes planner performance statistics to the server log.	–
log_replication_commands	Logs each replication command.	–
log_rotation_age	(min) Automatic log file rotation will occur after N minutes.	60
log_rotation_size	(kB) Automatic log file rotation will occur after N kilobytes.	100000
log_statement	Sets the type of statements logged.	–
log_statement_sample_rate	Fraction of statements exceeding log_min_duration_sample to be logged.	–
log_statement_stats	Writes cumulative performance statistics to the server log.	–

Parameter name	Description	Default
log_temp_files	(kB) Log the use of temporary files larger than this number of kilobytes.	–
log_timezone	Sets the time zone to use in log messages.	UTC
log_transaction_sample_rate	Set the fraction of transactions to log for new transactions.	–
log_truncate_on_rotation	Truncate existing log files of same name during log rotation.	0
maintenance_io_concurrency	A variant of effective_io_concurrency that is used for maintenance work.	1
maintenance_work_mem	(kB) Sets the maximum memory to be used for maintenance operations.	GREATEST(DBInstanceClassMemory/63963136*1024,65536)
max_connections	Sets the maximum number of concurrent connections.	LEAST(DBInstanceClassMemory/9531392,5000)
max_files_per_process	Sets the maximum number of simultaneously open files for each server process.	–
max_locks_per_transaction	Sets the maximum number of locks per transaction.	64
max_logical_replication_workers	Maximum number of logical replication worker processes.	–
max_parallel_maintenance_workers	Sets the maximum number of parallel processes per maintenance operation.	–
max_parallel_workers	Sets the maximum number of parallel workers than can be active at one time.	GREATEST($DBInstanceVCPU/2,8)

Parameter name	Description	Default
max_parallel_workers_per_gather	Sets the maximum number of parallel processes per executor node.	–
max_pred_locks_per_page	Sets the maximum number of predicate-locked tuples per page.	–
max_pred_locks_per_relation	Sets the maximum number of predicate-locked pages and tuples per relation.	–
max_pred_locks_per_transaction	Sets the maximum number of predicate locks per transaction.	–
max_prepared_transactions	Sets the maximum number of simultaneously prepared transactions.	0
max_replication_slots	Sets the maximum number of replication slots that the server can support.	20
max_slot_wal_keep_size	(MB) Replication slots will be marked as failed, and segments released for deletion or recycling, if this much space is occupied by WAL on disk.	–
max_stack_depth	(kB) Sets the maximum stack depth, in kilobytes.	6144
max_standby_streaming_delay	(ms) Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data.	14000
max_sync_workers_per_subscription	Maximum number of synchronization workers per subscription	2
max_wal_senders	Sets the maximum number of simultaneously running WAL sender processes.	10

Parameter name	Description	Default
max_worker_processes	Sets the maximum number of concurrent worker processes.	GREATEST($DBInstanceVCPU*2,8)
min_dynamic_shared_memory	(MB) Amount of dynamic shared memory reserved at startup.	–
min_parallel_index_scan_size	(8kB) Sets the minimum amount of index data for a parallel scan.	–
min_parallel_table_scan_size	(8kB) Sets the minimum amount of table data for a parallel scan.	–
old_snapshot_threshold	(min) Time before a snapshot is too old to read pages changed after the snapshot was taken.	–
orafce.nls_date_format	Emulate oracles date output behaviour.	–
orafce.timezone	Specify timezone used for sysdate function.	–
parallel_leader_participation	Controls whether Gather and Gather Merge also run subplans.	–
parallel_setup_cost	Sets the planners estimate of the cost of starting up worker processes for parallel query.	–
parallel_tuple_cost	Sets the planners estimate of the cost of passing each tuple (row) from worker to master backend.	–
password_encryption	Encrypt passwords.	–
pgaudit.log	Specifies which classes of statements will be logged by session audit logging.	–

Parameter name	Description	Default
pgaudit.log_catalog	Specifies that session logging should be enabled in the case where all relations in a statement are in pg_catalog.	–
pgaudit.log_level	Specifies the log level that will be used for log entries.	–
pgaudit.log_parameter	Specifies that audit logging should include the parameters that were passed with the statement.	–
pgaudit.log_relation	Specifies whether session audit logging should create a separate log entry for each relation (TABLE, VIEW, etc.) referenced in a SELECT or DML statement.	–
pgaudit.log_statement_once	Specifies whether logging will include the statement text and parameters with the first log entry for a statement/substatement combination or with every entry.	–
pgaudit.role	Specifies the master role to use for object audit logging.	–
pg_bigm.enable_recheck	It specifies whether to perform Recheck which is an internal process of full text search.	on
pg_bigm.gin_key_limit	It specifies the maximum number of 2-grams of the search keyword to be used for full text search.	0
pg_bigm.last_update	It reports the last updated date of the pg_bigm module.	2013.11.22
pg_bigm.similarity_limit	It specifies the minimum threshold used by the similarity search.	0.3

Parameter name	Description	Default
pg_hint_plan.debug_print	Logs results of hint parsing.	–
pg_hint_plan.enable_hint	Force planner to use plans specified in the hint comment preceding to the query.	–
pg_hint_plan.enable_hint_table	Force planner to not get hint by using table lookups.	–
pg_hint_plan.message_level	Message level of debug messages.	–
pg_hint_plan.parse_messages	Message level of parse errors.	–
pglogical.batch_inserts	Batch inserts if possible	–
pglogical.conflict_log_level	Sets log level used for logging resolved conflicts.	–
pglogical.conflict_resolution	Sets method used for conflict resolution for resolvable conflicts.	–
pglogical.extra_connection_options	connection options to add to all peer node connections	–
pglogical.synchronous_commit	pglogical specific synchronous commit value	–
pglogical.use_spi	Use SPI instead of low-level API for applying changes	–
pgtle.clientauth_databases_to_skip	List of databases to skip for clientauth feature.	–
pgtle.clientauth_db_name	Controls which database is used for clientauth feature.	–

Parameter name	Description	Default
pgtle.clientauth_num_parallel_workers	Number of background workers used for clientauth feature.	–
pgtle.clientauth_users_to_skip	List of users to skip for clientauth feature.	–
pgtle.enable_clientauth	Enables the clientauth feature.	–
pgtle.passcheck_db_name	Sets which database is used for cluster-wide passcheck feature.	–
pg_prewarm.autoprewarm	Starts the autoprewarm worker.	–
pg_prewarm.autoprewarm_interval	Sets the interval between dumps of shared buffers	–
pg_similarity.block_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.block_threshold	Sets the threshold used by the Block similarity function.	–
pg_similarity.block_tokenizer	Sets the tokenizer for Block similarity function.	–
pg_similarity.cosine_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.cosine_threshold	Sets the threshold used by the Cosine similarity function.	–
pg_similarity.cosine_tokenizer	Sets the tokenizer for Cosine similarity function.	–
pg_similarity.dice_is_normalized	Sets if the result value is normalized or not.	–

Parameter name	Description	Default
pg_similarity.dice _threshold	Sets the threshold used by the Dice similarity measure.	–
pg_similarity.dice _tokenizer	Sets the tokenizer for Dice similarity measure.	–
pg_similarity.eucl idean_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.eucl idean_threshold	Sets the threshold used by the Euclidean similarity measure.	–
pg_similarity.eucl idean_tokenizer	Sets the tokenizer for Euclidean similarity measure.	–
pg_similarity.hamm ing_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.hamm ing_threshold	Sets the threshold used by the Block similarity metric.	–
pg_similarity.jacc ard_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jacc ard_threshold	Sets the threshold used by the Jaccard similarity measure.	–
pg_similarity.jacc ard_tokenizer	Sets the tokenizer for Jaccard similarity measure.	–
pg_similarity.jaro _is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jaro _threshold	Sets the threshold used by the Jaro similarity measure.	–

Parameter name	Description	Default
pg_similarity.jarowinkler_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jarowinkler_threshold	Sets the threshold used by the Jarowinkler similarity measure.	–
pg_similarity.levenshtein_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.levenshtein_threshold	Sets the threshold used by the Levenshtein similarity measure.	–
pg_similarity.matching_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.matching_threshold	Sets the threshold used by the Matching Coefficient similarity measure.	–
pg_similarity.matching_tokenizer	Sets the tokenizer for Matching Coefficient similarity measure.	–
pg_similarity.mongeelkan_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.mongeelkan_threshold	Sets the threshold used by the Monge-Elkan similarity measure.	–
pg_similarity.mongeelkan_tokenizer	Sets the tokenizer for Monge-Elkan similarity measure.	–
pg_similarity.nw_gap_penalty	Sets the gap penalty used by the Needleman-Wunsch similarity measure.	–
pg_similarity.nw_is_normalized	Sets if the result value is normalized or not.	–

Parameter name	Description	Default
pg_similarity.nw_threshold	Sets the threshold used by the Needleman-Wunsch similarity measure.	–
pg_similarity.overlap_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.overlap_threshold	Sets the threshold used by the Overlap Coefficient similarity measure.	–
pg_similarity.overlap_tokenizer	Sets the tokenizer for Overlap Coefficientsimilarity measure.	–
pg_similarity.qgram_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.qgram_threshold	Sets the threshold used by the Q-Gram similarity measure.	–
pg_similarity.qgram_tokenizer	Sets the tokenizer for Q-Gram measure.	–
pg_similarity.swg_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.swg_threshold	Sets the threshold used by the Smith-Waterman-Gotoh similarity measure.	–
pg_similarity.sw_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.sw_threshold	Sets the threshold used by the Smith-Waterman similarity measure.	–
pg_stat_statements.max	Sets the maximum number of statements tracked by pg_stat_statements.	–
pg_stat_statements.save	Save pg_stat_statements statistics across server shutdowns.	–

Parameter name	Description	Default
pg_stat_statements.track	Selects which statements are tracked by pg_stat_statements.	–
pg_stat_statements.track_planning	Selects whether planning duration is tracked by pg_stat_statements.	–
pg_stat_statements.track_utility	Selects whether utility commands are tracked by pg_stat_statements.	–
plan_cache_mode	Controls the planner selection of custom or generic plan.	–
port	Sets the TCP port the server listens on.	EndPointPort
postgis.gdal_enabled_drivers	Enable or disable GDAL drivers used with PostGIS in Postgres 9.3.5 and above.	ENABLE_ALL
quote_all_identifiers	When generating SQL fragments, quote all identifiers.	–
random_page_cost	Sets the planners estimate of the cost of a nonsequentially fetched disk page.	–
rdkit.dice_threshold	Lower threshold of Dice similarity. Molecules with similarity lower than threshold are not similar by # operation.	–
rdkit.do_chiral_sss	Should stereochemistry be taken into account in substructure matching. If false, no stereochemistry information is used in substructure matches.	–
rdkit.tanimoto_threshold	Lower threshold of Tanimoto similarity. Molecules with similarity lower than threshold are not similar by % operation.	–

Parameter name	Description	Default
rds.accepted_password_auth_method	Force authentication for connections with password stored locally.	md5+scram
rds.adaptive_autovacuum	RDS parameter to enable/disable adaptive autovacuum.	1
rds.babelfish_status	RDS parameter to enable/disable Babelfish for Aurora PostgreSQL.	off
rds.enable_plan_management	Enable or disable the apg_plan_mgmt extension.	0

Parameter name	Description	Default
rds.extensions	List of extensions provided by RDS	address_standardizer, address_standardizer_data_us, apg_plan_mgmt, aurora_stat_utils, amcheck, autoinc, aws_commons, aws_ml, aws_s3, aws_lambda, bool_plperl, bloom, btree_gin, btree_gist, citext, cube, dblink, dict_int, dict_xsyn, earthdistance, fuzzystrmatch, hll, hstore, hstore_plperl, insert_username, intagg, intarray, ip4r, isn, jsonb_plperl, lo, log_fdw, ltree, moddatetime, old_snapshot, oracle_fdw, orafce, pgaudit, pgcrypto, pglogical, pgrouting, pgrowlocks, pgstattuple, pgtap, pg_bigm, pg_buffercache, pg_cron, pg_freespacemap, pg_hint_plan, pg_partman, pg_prewarm, pg_proctab,

Parameter name	Description	Default
		pg_repack, pg_similarity, pg_stat_statements, pg_trgm, pg_visibility, plcoffee, plls, plperl, plpgsql, plprofiler, pltcl, plv8, postgis, postgis_tiger_geocoder, postgis_raster, postgis_topology, postgres_fdw, prefix, rdkit, rds_tools, refint, sslinfo, tablefunc, tds_fdw, test_parser, tsm_system_rows, tsm_system_time, unaccent, uuid-ossp
rds.force_admin_logging_level	See log messages for RDS admin user actions in customer databases.	–
rds.force_autovacuum_logging_level	See log messages related to autovacuum operations.	WARNING
rds.force_ssl	Force SSL connections.	0

Parameter name	Description	Default
rds.global_db_rpo	(s) Recovery point objective threshold, in seconds, that blocks user commits when it is violated. **Important** This parameter is meant for Aurora PostgreSQL-based global databases. For a nonglobal database, leave it at the default value. For more information about using this parameter, see the section called "Managing RPOs for Aurora PostgreSQL–based global databases".	–
rds.logical_replication	Enables logical decoding.	0
rds.logically_replicate_unlogged_tables	Unlogged tables are logically replicated.	1
rds.log_retention_period	Amazon RDS will delete PostgreSQL log that are older than N minutes.	4320
rds.pg_stat_ramdisk_size	Size of the stats ramdisk in MB. A nonzero value will setup the ramdisk. This parameter is only available in Aurora PostgreSQL 14 and lower versions.	0
rds.rds_superuser_reserved_connections	Sets the number of connection slots reserved for rds_superusers. This parameter is only available in versions 15 and earlier. For more information, see the PostgreSQL documentation reserved connections.	2

Parameter name	Description	Default
rds.restrict_password_commands	restricts password-related commands to members of rds_password	–
rds.superuser_variables	List of superuser-only variables for which we elevate rds_superuser modification statements.	session_replication_role
recovery_init_sync_method	Sets the method for synchronizing the data directory before crash recovery.	syncfs
remove_temp_files_after_crash	Remove temporary files after backend crash.	0
restart_after_crash	Reinitialize server after backend crash.	–
row_security	Enable row security.	–
search_path	Sets the schema search order for names that are not schema-qualified.	–
seq_page_cost	Sets the planners estimate of the cost of a sequentially fetched disk page.	–
session_replication_role	Sets the sessions behavior for triggers and rewrite rules.	–
shared_buffers	(8kB) Sets the number of shared memory buffers used by the server.	SUM(DBInstanceClassMemory/12038,-50003)
shared_preload_libraries	Lists shared libraries to preload into server.	pg_stat_statements
ssl	Enables SSL connections.	1
ssl_ca_file	Location of the SSL server authority file.	/rdsdbdata/rds-metadata/ca-cert.pem

Parameter name	Description	Default
ssl_cert_file	Location of the SSL server certificate file.	/rdsdbdata/rds-metadata/server-cert.pem
ssl_ciphers	Sets the list of allowed TLS ciphers to be used on secure connections.	–
ssl_crl_dir	Location of the SSL certificate revocation list directory.	/rdsdbdata/rds-metadata/ssl_crl_dir/
ssl_key_file	Location of the SSL server private key file	/rdsdbdata/rds-metadata/server-key.pem
ssl_max_protocol_version	Sets the maximum SSL/TLS protocol version allowed	–
ssl_min_protocol_version	Sets the minimum SSL/TLS protocol version allowed	TLSv1.2
standard_conforming_strings	Causes ... strings to treat backslashes literally.	–
statement_timeout	(ms) Sets the maximum allowed duration of any statement.	–
stats_temp_directory	Writes temporary statistics files to the specified directory.	/rdsdbdata/db/pg_stat_tmp
superuser_reserved_connections	Sets the number of connection slots reserved for superusers.	3
synchronize_seqscans	Enable synchronized sequential scans.	–
synchronous_commit	Sets the current transactions synchronization level.	on
tcp_keepalives_count	Maximum number of TCP keepalive retransmits.	–

Parameter name	Description	Default
tcp_keepalives_idle	(s) Time between issuing TCP keepalives.	–
tcp_keepalives_interval	(s) Time between TCP keepalive retransmits.	–
temp_buffers	(8kB) Sets the maximum number of temporary buffers used by each session.	–
temp_file_limit	Constrains the total amount disk space in kilobytes that a given PostgreSQL process can use for temporary files, excluding space used for explicit temporary tables	-1
temp_tablespaces	Sets the tablespace(s) to use for temporary tables and sort files.	–
timezone	Sets the time zone for displaying and interpreting time stamps.	UTC
track_activities	Collects information about executing commands.	–
track_activity_query_size	Sets the size reserved for pg_stat_activity.current_query, in bytes.	4096
track_commit_timestamp	Collects transaction commit time.	–
track_counts	Collects statistics on database activity.	–
track_functions	Collects function-level statistics on database activity.	pl
track_io_timing	Collects timing statistics on database IO activity.	1
track_wal_io_timing	Collects timing statistics for WAL I/O activity.	–

Parameter name	Description	Default
transform_null_equals	Treats expr=NULL as expr IS NULL.	–
update_process_title	Updates the process title to show the active SQL command.	–
vacuum_cost_delay	(ms) Vacuum cost delay in milliseconds.	–
vacuum_cost_limit	Vacuum cost amount available before napping.	–
vacuum_cost_page_dirty	Vacuum cost for a page dirtied by vacuum.	–
vacuum_cost_page_hit	Vacuum cost for a page found in the buffer cache.	–
vacuum_cost_page_miss	Vacuum cost for a page not found in the buffer cache.	0
vacuum_defer_cleanup_age	Number of transactions by which VACUUM and HOT cleanup should be deferred, if any.	–
vacuum_failsafe_age	Age at which VACUUM should trigger failsafe to avoid a wraparound outage.	1200000000
vacuum_freeze_min_age	Minimum age at which VACUUM should freeze a table row.	–
vacuum_freeze_table_age	Age at which VACUUM should scan whole table to freeze tuples.	–
vacuum_multixact_failsafe_age	Multixact age at which VACUUM should trigger failsafe to avoid a wraparound outage.	1200000000
vacuum_multixact_freeze_min_age	Minimum age at which VACUUM should freeze a MultiXactId in a table row.	–

Parameter name	Description	Default
vacuum_multixact_freeze_table_age	Multixact age at which VACUUM should scan whole table to freeze tuples.	–
wal_buffers	(8kB) Sets the number of disk-page buffers in shared memory for WAL.	–
wal_receiver_create_temp_slot	Sets whether a WAL receiver should create a temporary replication slot if no permanent slot is configured.	0
wal_receiver_status_interval	(s) Sets the maximum interval between WAL receiver status reports to the primary.	–
wal_receiver_timeout	(ms) Sets the maximum wait time to receive data from the primary.	30000
wal_sender_timeout	(ms) Sets the maximum time to wait for WAL replication.	–
work_mem	(kB) Sets the maximum memory to be used for query workspaces.	–
xmlbinary	Sets how binary values are to be encoded in XML.	–
xmloption	Sets whether XML data in implicit parsing and serialization operations is to be considered as documents or content fragments.	–

Aurora PostgreSQL instance-level parameters

You can view the instance-level parameters available for a specific Aurora PostgreSQL version using the AWS Management console, the AWS CLI, or the Amazon RDS API. For information about viewing the parameters in an Aurora PostgreSQL DB parameter groups in the RDS console, see Viewing parameter values for a DB parameter group.

Some instance-level parameters aren't available in all versions and some are being deprecated. For information about viewing the parameters of a specific Aurora PostgreSQL version, see Viewing Aurora PostgreSQL DB cluster and DB parameters.

For example, the following table lists the parameters that apply to a specific DB instance in an Aurora PostgreSQL DB cluster. This list was generated by running the describe-db-parameters AWS CLI command with `default.aurora-postgresql14` for the `--db-parameter-group-name` value.

For a listing of the DB cluster parameters for this same default DB parameter group, see Aurora PostgreSQL cluster-level parameters.

Parameter name	Description	Default
apg_enable_batch_mode_function_execution	Enables batch-mode functions to process sets of rows at a time.	–
apg_enable_correlated_any_transform	Enables the planner to transform correlated ANY Sublink (IN/NOT IN subquery to JOIN when possible.	–
apg_enable_function_migration	Enables the planner to migrate eligible scalar functions to the FROM clause.	–
apg_enable_not_in_transform	Enables the planner to transform NOT IN subquery to ANTI JOIN when possible.	–
apg_enable_remove_redundant_inner_joins	Enables the planner to remove redundant inner joins.	–

Parameter name	Description	Default
apg_enable_semijoin_push_down	Enables the use of semijoin filters for hash joins.	–
apg_plan_mgmt.capture_plan_baselines	Capture plan baseline mode. manual - enable plan capture for any SQL statement, off - disable plan capture, automatic - enable plan capture for for statements in pg_stat_statements that satisfy the eligibility criteria.	off
apg_plan_mgmt.max_databases	Sets the maximum number of databases that that may manage queries using apg_plan_mgmt.	10
apg_plan_mgmt.max_plans	Sets the maximum number of plans that may be cached by apg_plan_mgmt.	10000
apg_plan_mgmt.plan_retention_period	Maximum number of days since a plan was last_used before a plan will be automatically deleted.	32
apg_plan_mgmt.unapproved_plan_execution_threshold	Estimated total plan cost below which an Unapproved plan will be executed.	0
apg_plan_mgmt.use_plan_baselines	Use only approved or fixed plans for managed statements.	false
application_name	Sets the application name to be reported in statistics and logs.	–
aurora_compute_plan_id	Monitors query execution plans to detect the execution plans contributing to current database load and to track performance statistics of execution plans over time. For more information, see Monitoring query execution plans for Aurora PostgreSQL.	on

Parameter name	Description	Default
authentication_timeout	(s Sets the maximum allowed time to complete client authentication.	–
auto_explain.log_analyze	Use EXPLAIN ANALYZE for plan logging.	–
auto_explain.log_buffers	Log buffers usage.	–
auto_explain.log_format	EXPLAIN format to be used for plan logging.	–
auto_explain.log_min_duration	Sets the minimum execution time above which plans will be logged.	–
auto_explain.log_nested_statements	Log nested statements.	–
auto_explain.log_timing	Collect timing data, not just row counts.	–
auto_explain.log_triggers	Include trigger statistics in plans.	–
auto_explain.log_verbose	Use EXPLAIN VERBOSE for plan logging.	–
auto_explain.sample_rate	Fraction of queries to process.	–
babelfishpg_tds.listen_addresses	Sets the host name or IP address(es to listen TDS to.	*
babelfishpg_tds.tds_debug_log_level	Sets logging level in TDS, 0 disables logging	1
backend_flush_after	(8Kb Number of pages after which previously performed writes are flushed to disk.	–

Parameter name	Description	Default
bytea_output	Sets the output format for bytea.	–
check_function_bodies	Check function bodies during CREATE FUNCTION.	–
client_connection_check_interval	Sets the time interval between checks for disconnection while running queries.	–
client_min_messages	Sets the message levels that are sent to the client.	–
config_file	Sets the servers main configuration file.	/rdsdbdata/config/postgresql.conf
constraint_exclusion	Enables the planner to use constraints to optimize queries.	–
cpu_index_tuple_cost	Sets the planners estimate of the cost of processing each index entry during an index scan.	–
cpu_operator_cost	Sets the planners estimate of the cost of processing each operator or function call.	–
cpu_tuple_cost	Sets the planners estimate of the cost of processing each tuple (row.	–
cron.database_name	Sets the database to store pg_cron metadata tables	postgres
cron.log_run	Log all jobs runs into the job_run_details table	on
cron.log_statement	Log all cron statements prior to execution.	off
cron.max_running_jobs	Maximum number of jobs that can run concurrently.	5

Parameter name	Description	Default
cron.use_background_workers	Enables background workers for pg_cron	on
cursor_tuple_fraction	Sets the planners estimate of the fraction of a cursors rows that will be retrieved.	–
db_user_namespace	Enables per-database user names.	–
deadlock_timeout	(ms Sets the time to wait on a lock before checking for deadlock.	–
debug_pretty_print	Indents parse and plan tree displays.	–
debug_print_parse	Logs each querys parse tree.	–
debug_print_plan	Logs each querys execution plan.	–
debug_print_rewritten	Logs each querys rewritten parse tree.	–
default_statistics_target	Sets the default statistics target.	–
default_transaction_deferrable	Sets the default deferrable status of new transactions.	–
default_transaction_isolation	Sets the transaction isolation level of each new transaction.	–
default_transaction_read_only	Sets the default read-only status of new transactions.	–
effective_cache_size	(8kB Sets the planners assumption about the size of the disk cache.	SUM(DBInstanceClassMemory/12038,-50003
effective_io_concurrency	Number of simultaneous requests that can be handled efficiently by the disk subsystem.	–

Parameter name	Description	Default
enable_async_append	Enables the planners use of async append plans.	–
enable_bitmapscan	Enables the planners use of bitmap-scan plans.	–
enable_gathermerge	Enables the planners use of gather merge plans.	–
enable_hashagg	Enables the planners use of hashed aggregation plans.	–
enable_hashjoin	Enables the planners use of hash join plans.	–
enable_incremental_sort	Enables the planners use of incremental sort steps.	–
enable_indexonlyscan	Enables the planners use of index-only-scan plans.	–
enable_indexscan	Enables the planners use of index-scan plans.	–
enable_material	Enables the planners use of materialization.	–
enable_memoize	Enables the planners use of memoization	–
enable_mergejoin	Enables the planners use of merge join plans.	–
enable_nestloop	Enables the planners use of nested-loop join plans.	–
enable_parallel_append	Enables the planners use of parallel append plans.	–
enable_parallel_hash	Enables the planners user of parallel hash plans.	–

Parameter name	Description	Default
enable_partition_pruning	Enable plan-time and run-time partition pruning.	–
enable_partitionwise_aggregate	Enables partitionwise aggregation and grouping.	–
enable_partitionwise_join	Enables partitionwise join.	–
enable_seqscan	Enables the planners use of sequential-scan plans.	–
enable_sort	Enables the planners use of explicit sort steps.	–
enable_tidscan	Enables the planners use of TID scan plans.	–
escape_string_warning	Warn about backslash escapes in ordinary string literals.	–
exit_on_error	Terminate session on any error.	–
force_parallel_mode	Forces use of parallel query facilities.	–
from_collapse_limit	Sets the FROM-list size beyond which subqueries are not collapsed.	–
geqo	Enables genetic query optimization.	–
geqo_effort	GEQO: effort is used to set the default for other GEQO parameters.	–
geqo_generations	GEQO: number of iterations of the algorithm.	–
geqo_pool_size	GEQO: number of individuals in the population.	–
geqo_seed	GEQO: seed for random path selection.	–

Parameter name	Description	Default
geqo_selection_bias	GEQO: selective pressure within the populatio n.	–
geqo_threshold	Sets the threshold of FROM items beyond which GEQO is used.	–
gin_fuzzy_search_l imit	Sets the maximum allowed result for exact search by GIN.	–
gin_pending_list_l imit	(kB Sets the maximum size of the pending list for GIN index.	–
hash_mem_multiplier	Multiple of work_mem to use for hash tables.	–
hba_file	Sets the servers hba configuration file.	/rdsdbdata/config/ pg_hba.conf
hot_standby_feedba ck	Allows feedback from a hot standby to the primary that will avoid query conflicts.	on
ident_file	Sets the servers ident configuration file.	/rdsdbdata/config/ pg_ident.conf
idle_in_transactio n_session_timeout	(ms Sets the maximum allowed duration of any idling transaction.	86400000
idle_session_timeout	Terminate any session that has been idle (that is, waiting for a client query, but not within an open transaction, for longer than the specified amount of time	–
join_collapse_limit	Sets the FROM-list size beyond which JOIN constructs are not flattened.	–
lc_messages	Sets the language in which messages are displayed.	–

Parameter name	Description	Default
listen_addresses	Sets the host name or IP address(es to listen to.	*
lo_compat_privileges	Enables backward compatibility mode for privilege checks on large objects.	0
log_connections	Logs each successful connection.	–
log_destination	Sets the destination for server log output.	stderr
log_directory	Sets the destination directory for log files.	/rdsdbdata/log/error
log_disconnections	Logs end of a session, including duration.	–
log_duration	Logs the duration of each completed SQL statement.	–
log_error_verbosity	Sets the verbosity of logged messages.	–
log_executor_stats	Writes executor performance statistics to the server log.	–
log_file_mode	Sets the file permissions for log files.	0644
log_filename	Sets the file name pattern for log files.	postgresql.log.%Y-%m-%d-%H%M
logging_collector	Start a subprocess to capture stderr output and/or csvlogs into log files.	1
log_hostname	Logs the host name in the connection logs.	0
logical_decoding_work_mem	(kB This much memory can be used by each internal reorder buffer before spilling to disk.	–
log_line_prefix	Controls information prefixed to each log line.	%t:%r:%u@%d:%p]:
log_lock_waits	Logs long lock waits.	–

Parameter name	Description	Default
log_min_duration_sample	(ms Sets the minimum execution time above which a sample of statements will be logged. Sampling is determined by log_statement_sample_rate.	–
log_min_duration_statement	(ms Sets the minimum execution time above which statements will be logged.	–
log_min_error_statement	Causes all statements generating error at or above this level to be logged.	–
log_min_messages	Sets the message levels that are logged.	–
log_parameter_max_length	(B When logging statements, limit logged parameter values to first N bytes.	–
log_parameter_max_length_on_error	(B When reporting an error, limit logged parameter values to first N bytes.	–
log_parser_stats	Writes parser performance statistics to the server log.	–
log_planner_stats	Writes planner performance statistics to the server log.	–
log_replication_commands	Logs each replication command.	–
log_rotation_age	(min Automatic log file rotation will occur after N minutes.	60
log_rotation_size	(kB Automatic log file rotation will occur after N kilobytes.	100000
log_statement	Sets the type of statements logged.	–
log_statement_sample_rate	Fraction of statements exceeding log_min_duration_sample to be logged.	–

Parameter name	Description	Default
log_statement_stats	Writes cumulative performance statistics to the server log.	–
log_temp_files	(kB Log the use of temporary files larger than this number of kilobytes.	–
log_timezone	Sets the time zone to use in log messages.	UTC
log_truncate_on_rotation	Truncate existing log files of same name during log rotation.	0
maintenance_io_concurrency	A variant of effective_io_concurrency that is used for maintenance work.	1
maintenance_work_mem	(kB Sets the maximum memory to be used for maintenance operations.	GREATEST(DBInstanceClassMemory/63963136*1024,65536
max_connections	Sets the maximum number of concurrent connections.	LEAST(DBInstanceClassMemory/9531392,5000
max_files_per_process	Sets the maximum number of simultaneously open files for each server process.	–
max_locks_per_transaction	Sets the maximum number of locks per transaction.	64
max_parallel_maintenance_workers	Sets the maximum number of parallel processes per maintenance operation.	–
max_parallel_workers	Sets the maximum number of parallel workers than can be active at one time.	GREATEST($DBInstanceVCPU/2,8
max_parallel_workers_per_gather	Sets the maximum number of parallel processes per executor node.	–

Parameter name	Description	Default
max_pred_locks_per_page	Sets the maximum number of predicate-locked tuples per page.	–
max_pred_locks_per_relation	Sets the maximum number of predicate-locked pages and tuples per relation.	–
max_pred_locks_per_transaction	Sets the maximum number of predicate locks per transaction.	–
max_slot_wal_keep_size	(MB Replication slots will be marked as failed, and segments released for deletion or recycling, if this much space is occupied by WAL on disk.	–
max_stack_depth	(kB Sets the maximum stack depth, in kilobytes.	6144
max_standby_streaming_delay	(ms Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data.	14000
max_worker_processes	Sets the maximum number of concurrent worker processes.	GREATEST($DBInstanceVCPU*2,8
min_dynamic_shared_memory	(MB Amount of dynamic shared memory reserved at startup.	–
min_parallel_index_scan_size	(8kB Sets the minimum amount of index data for a parallel scan.	–
min_parallel_table_scan_size	(8kB Sets the minimum amount of table data for a parallel scan.	–
old_snapshot_threshold	(min Time before a snapshot is too old to read pages changed after the snapshot was taken.	–

Parameter name	Description	Default
parallel_leader_participation	Controls whether Gather and Gather Merge also run subplans.	–
parallel_setup_cost	Sets the planners estimate of the cost of starting up worker processes for parallel query.	–
parallel_tuple_cost	Sets the planners estimate of the cost of passing each tuple (row from worker to master backend.	–
pgaudit.log	Specifies which classes of statements will be logged by session audit logging.	–
pgaudit.log_catalog	Specifies that session logging should be enabled in the case where all relations in a statement are in pg_catalog.	–
pgaudit.log_level	Specifies the log level that will be used for log entries.	–
pgaudit.log_parameter	Specifies that audit logging should include the parameters that were passed with the statement.	–
pgaudit.log_relation	Specifies whether session audit logging should create a separate log entry for each relation (TABLE, VIEW, etc. referenced in a SELECT or DML statement.	–
pgaudit.log_statement_once	Specifies whether logging will include the statement text and parameters with the first log entry for a statement/substatement combination or with every entry.	–
pgaudit.role	Specifies the master role to use for object audit logging.	–

Parameter name	Description	Default
pg_bigm.enable_recheck	It specifies whether to perform Recheck which is an internal process of full text search.	on
pg_bigm.gin_key_limit	It specifies the maximum number of 2-grams of the search keyword to be used for full text search.	0
pg_bigm.last_update	It reports the last updated date of the pg_bigm module.	2013.11.22
pg_bigm.similarity_limit	It specifies the minimum threshold used by the similarity search.	0.3
pg_hint_plan.debug_print	Logs results of hint parsing.	–
pg_hint_plan.enable_hint	Force planner to use plans specified in the hint comment preceding to the query.	–
pg_hint_plan.enable_hint_table	Force planner to not get hint by using table lookups.	–
pg_hint_plan.message_level	Message level of debug messages.	–
pg_hint_plan.parse_messages	Message level of parse errors.	–
pglogical.batch_inserts	Batch inserts if possible	–
pglogical.conflict_log_level	Sets log level used for logging resolved conflicts.	–
pglogical.conflict_resolution	Sets method used for conflict resolution for resolvable conflicts.	–

Parameter name	Description	Default
pglogical.extra_connection_options	connection options to add to all peer node connections	–
pglogical.synchronous_commit	pglogical specific synchronous commit value	–
pglogical.use_spi	Use SPI instead of low-level API for applying changes	–
pg_similarity.block_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.block_threshold	Sets the threshold used by the Block similarity function.	–
pg_similarity.block_tokenizer	Sets the tokenizer for Block similarity function.	–
pg_similarity.cosine_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.cosine_threshold	Sets the threshold used by the Cosine similarity function.	–
pg_similarity.cosine_tokenizer	Sets the tokenizer for Cosine similarity function.	–
pg_similarity.dice_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.dice_threshold	Sets the threshold used by the Dice similarity measure.	–
pg_similarity.dice_tokenizer	Sets the tokenizer for Dice similarity measure.	–
pg_similarity.euclidean_is_normalized	Sets if the result value is normalized or not.	–

Parameter name	Description	Default
pg_similarity.euclidean_threshold	Sets the threshold used by the Euclidean similarity measure.	–
pg_similarity.euclidean_tokenizer	Sets the tokenizer for Euclidean similarity measure.	–
pg_similarity.hamming_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.hamming_threshold	Sets the threshold used by the Block similarity metric.	–
pg_similarity.jaccard_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jaccard_threshold	Sets the threshold used by the Jaccard similarity measure.	–
pg_similarity.jaccard_tokenizer	Sets the tokenizer for Jaccard similarity measure.	–
pg_similarity.jaro_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jaro_threshold	Sets the threshold used by the Jaro similarity measure.	–
pg_similarity.jarowinkler_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.jarowinkler_threshold	Sets the threshold used by the Jarowinkler similarity measure.	–
pg_similarity.levenshtein_is_normalized	Sets if the result value is normalized or not.	–

Parameter name	Description	Default
pg_similarity.levenshtein_threshold	Sets the threshold used by the Levenshtein similarity measure.	–
pg_similarity.matching_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.matching_threshold	Sets the threshold used by the Matching Coefficient similarity measure.	–
pg_similarity.matching_tokenizer	Sets the tokenizer for Matching Coefficient similarity measure.	–
pg_similarity.mongeelkan_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.mongeelkan_threshold	Sets the threshold used by the Monge-Elkan similarity measure.	–
pg_similarity.mongeelkan_tokenizer	Sets the tokenizer for Monge-Elkan similarity measure.	–
pg_similarity.nw_gap_penalty	Sets the gap penalty used by the Needleman-Wunsch similarity measure.	–
pg_similarity.nw_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.nw_threshold	Sets the threshold used by the Needleman-Wunsch similarity measure.	–
pg_similarity.overlap_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.overlap_threshold	Sets the threshold used by the Overlap Coefficient similarity measure.	–
pg_similarity.overlap_tokenizer	Sets the tokenizer for Overlap Coefficientsimilarity measure.	–

Parameter name	Description	Default
pg_similarity.qgram_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.qgram_threshold	Sets the threshold used by the Q-Gram similarity measure.	–
pg_similarity.qgram_tokenizer	Sets the tokenizer for Q-Gram measure.	–
pg_similarity.swg_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.swg_threshold	Sets the threshold used by the Smith-Waterman-Gotoh similarity measure.	–
pg_similarity.sw_is_normalized	Sets if the result value is normalized or not.	–
pg_similarity.sw_threshold	Sets the threshold used by the Smith-Waterman similarity measure.	–
pg_stat_statements.max	Sets the maximum number of statements tracked by pg_stat_statements.	–
pg_stat_statements.save	Save pg_stat_statements statistics across server shutdowns.	–
pg_stat_statements.track	Selects which statements are tracked by pg_stat_statements.	–
pg_stat_statements.track_planning	Selects whether planning duration is tracked by pg_stat_statements.	–
pg_stat_statements.track_utility	Selects whether utility commands are tracked by pg_stat_statements.	–
postgis.gdal_enabled_drivers	Enable or disable GDAL drivers used with PostGIS in Postgres 9.3.5 and above.	ENABLE_ALL

Parameter name	Description	Default
quote_all_identifiers	When generating SQL fragments, quote all identifiers.	–
random_page_cost	Sets the planners estimate of the cost of a nonsequentially fetched disk page.	–
rds.enable_memory_management	Improves memory management capabilities in Aurora PostgreSQL 12.17, 13.13, 14.10, 15.5, and higher versions that prevents stability issues and database restarts caused by insufficient free memory. For more information, see Improved memory management in Aurora PostgreSQL.	True
rds.force_admin_logging_level	See log messages for RDS admin user actions in customer databases.	–
rds.log_retention_period	Amazon RDS will delete PostgreSQL log that are older than N minutes.	4320
rds.memory_allocation_guard	Improves memory management capabilities in Aurora PostgreSQL 11.21, 12.16, 13.12, 14.9, 15.4, and older versions that prevents stability issues and database restarts caused by insufficient free memory. For more information, see Improved memory management in Aurora PostgreSQL.	False
rds.pg_stat_ramdisk_size	Size of the stats ramdisk in MB. A nonzero value will setup the ramdisk.	0
rds.rds_superuser_reserved_connections	Sets the number of connection slots reserved for rds_superusers. This parameter is only available in versions 15 and earlier. For more information, see the PostgreSQL documentation reserved connections.	2

Parameter name	Description	Default
rds.superuser_variables	List of superuser-only variables for which we elevate rds_superuser modification statements.	session_replication_role
remove_temp_files_after_crash	Remove temporary files after backend crash.	0
restart_after_crash	Reinitialize server after backend crash.	–
row_security	Enable row security.	–
search_path	Sets the schema search order for names that are not schema-qualified.	–
seq_page_cost	Sets the planners estimate of the cost of a sequentially fetched disk page.	–
session_replication_role	Sets the sessions behavior for triggers and rewrite rules.	–
shared_buffers	(8kB Sets the number of shared memory buffers used by the server.	SUM(DBInstanceClassMemory/12038,-50003
shared_preload_libraries	Lists shared libraries to preload into server.	pg_stat_statements
ssl_ca_file	Location of the SSL server authority file.	/rdsdbdata/rds-metadata/ca-cert.pem
ssl_cert_file	Location of the SSL server certificate file.	/rdsdbdata/rds-metadata/server-cert.pem
ssl_crl_dir	Location of the SSL certificate revocation list directory.	/rdsdbdata/rds-metadata/ssl_crl_dir/

Parameter name	Description	Default
ssl_key_file	Location of the SSL server private key file	/rdsdbdata/rds-metadata/server-key.pem
standard_conforming_strings	Causes ... strings to treat backslashes literally.	–
statement_timeout	(ms Sets the maximum allowed duration of any statement.	–
stats_temp_directory	Writes temporary statistics files to the specified directory.	/rdsdbdata/db/pg_stat_tmp
superuser_reserved_connections	Sets the number of connection slots reserved for superusers.	3
synchronize_seqscans	Enable synchronized sequential scans.	–
tcp_keepalives_count	Maximum number of TCP keepalive retransmits.	–
tcp_keepalives_idle	(s Time between issuing TCP keepalives.	–
tcp_keepalives_interval	(s Time between TCP keepalive retransmits.	–
temp_buffers	(8kB Sets the maximum number of temporary buffers used by each session.	–
temp_file_limit	Constrains the total amount disk space in kilobytes that a given PostgreSQL process can use for temporary files, excluding space used for explicit temporary tables	-1
temp_tablespaces	Sets the tablespace(s to use for temporary tables and sort files.	–
track_activities	Collects information about executing commands.	–

Parameter name	Description	Default
track_activity_query_size	Sets the size reserved for pg_stat_activity.current_query, in bytes.	4096
track_counts	Collects statistics on database activity.	–
track_functions	Collects function-level statistics on database activity.	pl
track_io_timing	Collects timing statistics on database IO activity.	1
transform_–_equals	Treats expr=– as expr IS –.	–
update_process_title	Updates the process title to show the active SQL command.	–
wal_receiver_status_interval	(s Sets the maximum interval between WAL receiver status reports to the primary.	–
work_mem	(kB Sets the maximum memory to be used for query workspaces.	–
xmlbinary	Sets how binary values are to be encoded in XML.	–
xmloption	Sets whether XML data in implicit parsing and serialization operations is to be considered as documents or content fragments.	–

Amazon Aurora PostgreSQL wait events

The following are common wait events for Aurora PostgreSQL. To learn more about wait events and tuning your Aurora PostgreSQL DB cluster, see Tuning with wait events for Aurora PostgreSQL.

Activity:ArchiverMain

The archiver process is waiting for activity.

Activity:AutoVacuumMain

The autovacuum launcher process is waiting for activity.

Activity:BgWriterHibernate

The background writer process is hibernating while waiting for activity.

Activity:BgWriterMain

The background writer process is waiting for activity.

Activity:CheckpointerMain

The checkpointer process is waiting for activity.

Activity:LogicalApplyMain

The logical replication apply process is waiting for activity.

Activity:LogicalLauncherMain

The logical replication launcher process is waiting for activity.

Activity:PgStatMain

The statistics collector process is waiting for activity.

Activity:RecoveryWalAll

A process is waiting for the write-ahead log (WAL) from a stream at recovery.

Activity:RecoveryWalStream

The startup process is waiting for the write-ahead log (WAL) to arrive during streaming recovery.

Activity:SysLoggerMain

The syslogger process is waiting for activity.

Activity:WalReceiverMain

The write-ahead log (WAL) receiver process is waiting for activity.

Activity:WalSenderMain

The write-ahead log (WAL) sender process is waiting for activity.

Activity:WalWriterMain

The write-ahead log (WAL) writer process is waiting for activity.

BufferPin:BufferPin

A process is waiting to acquire an exclusive pin on a buffer.

Client:GSSOpenServer

A process is waiting to read data from the client while establishing a Generic Security Service Application Program Interface (GSSAPI) session.

Client:ClientRead

A backend process is waiting to receive data from a PostgreSQL client. For more information, see Client:ClientRead.

Client:ClientWrite

A backend process is waiting to send more data to a PostgreSQL client. For more information, see Client:ClientWrite.

Client:LibPQWalReceiverConnect

A process is waiting in the write-ahead log (WAL) receiver to establish connection to remote server.

Client:LibPQWalReceiverReceive

A process is waiting in the write-ahead log (WAL) receiver to receive data from remote server.

Client:SSLOpenServer

A process is waiting for Secure Sockets Layer (SSL) while attempting connection.

Client:WalReceiverWaitStart

A process is waiting for startup process to send initial data for streaming replication.

Client:WalSenderWaitForWAL

A process is waiting for the write-ahead log (WAL) to be flushed in the WAL sender process.

Client:WalSenderWriteData

A process is waiting for any activity when processing replies from the write-ahead log (WAL) receiver in the WAL sender process.

CPU

A backend process is active in or is waiting for CPU. For more information, see CPU.

Extension:extension

A backend process is waiting for a condition defined by an extension or module.

IO:AuroraOptimizedReadsCacheRead

A process is waiting for a read from Optimized Reads tiered cache because the page isn't available in shared memory.

IO:AuroraOptimizedReadsCacheSegmentTruncate

A process is waiting for an Optimized Reads tiered cache segment file to be truncated.

IO:AuroraOptimizedReadsCacheWrite

The background writer process is waiting to write in Optimized Reads tiered cache.

IO:AuroraStorageLogAllocate

A session is allocating metadata and preparing for a transaction log write.

IO:BufFileRead

When operations require more memory than the amount defined by working memory parameters, the engine creates temporary files on disk. This wait event occurs when operations read from the temporary files. For more information, see IO:BufFileRead and IO:BufFileWrite.

IO:BufFileWrite

When operations require more memory than the amount defined by working memory parameters, the engine creates temporary files on disk. This wait event occurs when operations write to the temporary files. For more information, see IO:BufFileRead and IO:BufFileWrite.

IO:ControlFileRead

A process is waiting for a read from the `pg_control` file.

IO:ControlFileSync

A process is waiting for the `pg_control` file to reach durable storage.

IO:ControlFileSyncUpdate

A process is waiting for an update to the `pg_control` file to reach durable storage.

IO:ControlFileWrite

A process is waiting for a write to the `pg_control` file.

IO:ControlFileWriteUpdate

A process is waiting for a write to update the `pg_control` file.

IO:CopyFileRead

A process is waiting for a read during a file copy operation.

IO:CopyFileWrite

A process is waiting for a write during a file copy operation.

IO:DataFileExtend

A process is waiting for a relation data file to be extended.

IO:DataFileFlush

A process is waiting for a relation data file to reach durable storage.

IO:DataFileImmediateSync

A process is waiting for an immediate synchronization of a relation data file to durable storage.

IO:DataFilePrefetch

A process is waiting for an asynchronous prefetch from a relation data file.

IO:DataFileSync

A process is waiting for changes to a relation data file to reach durable storage.

IO:DataFileRead

A backend process tried to find a page in the shared buffers, didn't find it, and so read it from storage. For more information, see IO:DataFileRead.

IO:DataFileTruncate

A process is waiting for a relation data file to be truncated.

IO:DataFileWrite

A process is waiting for a write to a relation data file.

IO:DSMFillZeroWrite

A process is waiting to write zero bytes to a dynamic shared memory backing file.

IO:LockFileAddToDataDirRead

A process is waiting for a read while adding a line to the data directory lock file.

IO:LockFileAddToDataDirSync

A process is waiting for data to reach durable storage while adding a line to the data directory lock file.

IO:LockFileAddToDataDirWrite

A process is waiting for a write while adding a line to the data directory lock file.

IO:LockFileCreateRead

A process is waiting to read while creating the data directory lock file.

IO:LockFileCreateSync

A process is waiting for data to reach durable storage while creating the data directory lock file.

IO:LockFileCreateWrite

A process is waiting for a write while creating the data directory lock file.

IO:LockFileReCheckDataDirRead

A process is waiting for a read during recheck of the data directory lock file.

IO:LogicalRewriteCheckpointSync

A process is waiting for logical rewrite mappings to reach durable storage during a checkpoint.

IO:LogicalRewriteMappingSync

A process is waiting for mapping data to reach durable storage during a logical rewrite.

IO:LogicalRewriteMappingWrite

A process is waiting for a write of mapping data during a logical rewrite.

IO:LogicalRewriteSync

A process is waiting for logical rewrite mappings to reach durable storage.

IO:LogicalRewriteTruncate

A process is waiting for the truncation of mapping data during a logical rewrite.

IO:LogicalRewriteWrite

A process is waiting for a write of logical rewrite mappings.

IO:RelationMapRead

A process is waiting for a read of the relation map file.

IO:RelationMapSync

A process is waiting for the relation map file to reach durable storage.

IO:RelationMapWrite

A process is waiting for a write to the relation map file.

IO:ReorderBufferRead

A process is waiting for a read during reorder buffer management.

IO:ReorderBufferWrite

A process is waiting for a write during reorder buffer management.

IO:ReorderLogicalMappingRead

A process is waiting for a read of a logical mapping during reorder buffer management.

IO:ReplicationSlotRead

A process is waiting for a read from a replication slot control file.

IO:ReplicationSlotRestoreSync

A process is waiting for a replication slot control file to reach durable storage while restoring it to memory.

IO:ReplicationSlotSync

A process is waiting for a replication slot control file to reach durable storage.

IO:ReplicationSlotWrite

A process is waiting for a write to a replication slot control file.

IO:SLRUFlushSync

A process is waiting for simple least-recently used (SLRU) data to reach durable storage during a checkpoint or database shutdown.

IO:SLRURead

A process is waiting for a read of a simple least-recently used (SLRU) page.

IO:SLRUSync

A process is waiting for simple least-recently used (SLRU) data to reach durable storage following a page write.

IO:SLRUWrite

A process is waiting for a write of a simple least-recently used (SLRU) page.

IO:SnapbuildRead

A process is waiting for a read of a serialized historical catalog snapshot.

IO:SnapbuildSync

A process is waiting for a serialized historical catalog snapshot to reach durable storage.

IO:SnapbuildWrite

A process is waiting for a write of a serialized historical catalog snapshot.

IO:TimelineHistoryFileSync

A process is waiting for a timeline history file received through streaming replication to reach durable storage.

IO:TimelineHistoryFileWrite

A process is waiting for a write of a timeline history file received through streaming replication.

IO:TimelineHistoryRead

A process is waiting for a read of a timeline history file.

IO:TimelineHistorySync

A process is waiting for a newly created timeline history file to reach durable storage.

IO:TimelineHistoryWrite

A process is waiting for a write of a newly created timeline history file.

IO:TwophaseFileRead

A process is waiting for a read of a two phase state file.

IO:TwophaseFileSync

A process is waiting for a two phase state file to reach durable storage.

IO:TwophaseFileWrite

A process is waiting for a write of a two phase state file.

IO:WALBootstrapSync

A process is waiting for the write-ahead log (WAL) to reach durable storage during bootstrapping.

IO:WALBootstrapWrite

A process is waiting for a write of a write-ahead log (WAL) page during bootstrapping.

IO:WALCopyRead

A process is waiting for a read when creating a new write-ahead log (WAL) segment by copying an existing one.

IO:WALCopySync

A process is waiting for a new write-ahead log (WAL) segment created by copying an existing one to reach durable storage.

IO:WALCopyWrite

A process is waiting for a write when creating a new write-ahead log (WAL) segment by copying an existing one.

IO:WALInitSync

A process is waiting for a newly initialized write-ahead log (WAL) file to reach durable storage.

IO:WALInitWrite

A process is waiting for a write while initializing a new write-ahead log (WAL) file.

IO:WALRead

A process is waiting for a read from a write-ahead log (WAL) file.

IO:WALSenderTimelineHistoryRead

A process is waiting for a read from a timeline history file during a WAL sender timeline command.

IO:WALSync

A process is waiting for a write-ahead log (WAL) file to reach durable storage.

IO:WALSyncMethodAssign

A process is waiting for data to reach durable storage while assigning a new write-ahead log (WAL) sync method.

IO:WALWrite

A process is waiting for a write to a write-ahead log (WAL) file.

IO:XactSync

A backend process is waiting for the Aurora storage subsystem to acknowledge the commit of a regular transaction, or the commit or rollback of a prepared transaction. For more information, see IO:XactSync.

IPC:BackupWaitWalArchive

A process is waiting for write-ahead log (WAL) files required for a backup to be successfully archived.

IPC:AuroraOptimizedReadsCacheWriteStop

A process is waiting for the background writer to stop writing into Optimized Reads tiered cache.

IPC:BgWorkerShutdown

A process is waiting for a background worker to shut down.

IPC:BgWorkerStartup

A process is waiting for a background worker to start.

IPC:BtreePage

A process is waiting for the page number needed to continue a parallel B-tree scan to become available.

IPC:CheckpointDone

A process is waiting for a checkpoint to complete.

IPC:CheckpointStart

A process is waiting for a checkpoint to start.

IPC:ClogGroupUpdate

A process is waiting for the group leader to update the transaction status at a transaction's end.

IPC:DamRecordTxAck

A backend process has generated a database activity streams event and is waiting for the event to become durable. For more information, see IPC:DamRecordTxAck.

IPC:ExecuteGather

A process is waiting for activity from a child process while executing a Gather plan node.

IPC:Hash/Batch/Allocating

A process is waiting for an elected parallel hash participant to allocate a hash table.

IPC:Hash/Batch/Electing

A process is electing a parallel hash participant to allocate a hash table.

IPC:Hash/Batch/Loading

A process is waiting for other parallel hash participants to finish loading a hash table.

IPC:Hash/Build/Allocating

A process is waiting for an elected parallel hash participant to allocate the initial hash table.

IPC:Hash/Build/Electing

A process is electing a parallel hash participant to allocate the initial hash table.

IPC:Hash/Build/HashingInner

A process is waiting for other parallel hash participants to finish hashing the inner relation.

IPC:Hash/Build/HashingOuter

A process is waiting for other parallel hash participants to finish partitioning the outer relation.

IPC:Hash/GrowBatches/Allocating

A process is waiting for an elected parallel hash participant to allocate more batches.

IPC:Hash/GrowBatches/Deciding

A process is electing a parallel hash participant to decide on future batch growth.

IPC:Hash/GrowBatches/Electing

A process is electing a parallel hash participant to allocate more batches.

IPC:Hash/GrowBatches/Finishing

A process is waiting for an elected parallel hash participant to decide on future batch growth.

IPC:Hash/GrowBatches/Repartitioning

A process is waiting for other parallel hash participants to finishing repartitioning.

IPC:Hash/GrowBuckets/Allocating

A process is waiting for an elected parallel hash participant to finish allocating more buckets.

IPC:Hash/GrowBuckets/Electing

A process is electing a parallel hash participant to allocate more buckets.

IPC:Hash/GrowBuckets/Reinserting

A process is waiting for other parallel hash participants to finish inserting tuples into new buckets.

IPC:HashBatchAllocate

A process is waiting for an elected parallel hash participant to allocate a hash table.

IPC:HashBatchElect

A process is waiting to elect a parallel hash participant to allocate a hash table.

IPC:HashBatchLoad

A process is waiting for other parallel hash participants to finish loading a hash table.

IPC:HashBuildAllocate

A process is waiting for an elected parallel hash participant to allocate the initial hash table.

IPC:HashBuildElect

A process is waiting to elect a parallel hash participant to allocate the initial hash table.

IPC:HashBuildHashInner

A process is waiting for other parallel hash participants to finish hashing the inner relation.

IPC:'HashBuildHashOuter

A process is waiting for other parallel hash participants to finish partitioning the outer relation.

IPC:HashGrowBatchesAllocate

A process is waiting for an elected parallel hash participant to allocate more batches.

IPC:'HashGrowBatchesDecide

A process is waiting to elect a parallel hash participant to decide on future batch growth.

IPC:HashGrowBatchesElect

A process is waiting to elect a parallel hash participant to allocate more batches.

IPC:HashGrowBatchesFinish

A process is waiting for an elected parallel hash participant to decide on future batch growth.

IPC:HashGrowBatchesRepartition

A process is waiting for other parallel hash participants to finish repartitioning.

IPC:HashGrowBucketsAllocate

A process is waiting for an elected parallel hash participant to finish allocating more buckets.

IPC:HashGrowBucketsElect

A process is waiting to elect a parallel hash participant to allocate more buckets.

IPC:HashGrowBucketsReinsert

A process is waiting for other parallel hash participants to finish inserting tuples into new buckets.

IPC:LogicalSyncData

A process is waiting for a logical replication remote server to send data for initial table synchronization.

IPC:LogicalSyncStateChange

A process is waiting for a logical replication remote server to change state.

IPC:MessageQueueInternal

A process is waiting for another process to be attached to a shared message queue.

IPC:MessageQueuePutMessage

A process is waiting to write a protocol message to a shared message queue.

IPC:MessageQueueReceive

A process is waiting to receive bytes from a shared message queue.

IPC:MessageQueueSend

A process is waiting to send bytes to a shared message queue.

IPC:ParallelBitmapScan

A process is waiting for a parallel bitmap scan to become initialized.

IPC:ParallelCreateIndexScan

A process is waiting for parallel CREATE INDEX workers to finish a heap scan.

IPC:ParallelFinish

A process is waiting for parallel workers to finish computing.

IPC:ProcArrayGroupUpdate

A process is waiting for the group leader to clear the transaction ID at the end of a parallel operation.

IPC:ProcSignalBarrier

A process is waiting for a barrier event to be processed by all backends.

IPC:Promote

A process is waiting for standby promotion.

IPC:RecoveryConflictSnapshot

A process is waiting for recovery conflict resolution for a vacuum cleanup.

IPC:RecoveryConflictTablespace

A process is waiting for recovery conflict resolution for dropping a tablespace.

IPC:RecoveryPause

A process is waiting for recovery to be resumed.

IPC:ReplicationOriginDrop

A process is waiting for a replication origin to become inactive so it can be dropped.

IPC:ReplicationSlotDrop

A process is waiting for a replication slot to become inactive so it can be dropped.

IPC:SafeSnapshot

A process is waiting to obtain a valid snapshot for a READ ONLY DEFERRABLE transaction.

IPC:SyncRep

A process is waiting for confirmation from a remote server during synchronous replication.

IPC:XactGroupUpdate

A process is waiting for the group leader to update the transaction status at the end of a parallel operation.

Lock:advisory

A backend process requested an advisory lock and is waiting for it. For more information, see Lock:advisory.

Lock:extend

A backend process is waiting for a lock to be released so that it can extend a relation. This lock is needed because only one backend process can extend a relation at a time. For more information, see Lock:extend.

Lock:frozenid

A process is waiting to update `pg_database.datfrozenxid` and `pg_database.datminmxid`.

Lock:object

A process is waiting to get a lock on a nonrelation database object.

Lock:page

A process is waiting to get a lock on a page of a relation.

Lock:Relation

A backend process is waiting to acquire a lock on a relation that is locked by another transaction. For more information, see Lock:Relation.

Lock:spectoken

A process is waiting to get a speculative insertion lock.

Lock:speculative token

A process is waiting to acquire a speculative insertion lock.

Lock:transactionid

A transaction is waiting for a row-level lock. For more information, see Lock:transactionid.

Lock:tuple

A backend process is waiting to acquire a lock on a tuple while another backend process holds a conflicting lock on the same tuple. For more information, see Lock:tuple.

Lock:userlock

A process is waiting to get a user lock.

Lock:virtualxid

A process is waiting to get a virtual transaction ID lock.

LWLock:AddinShmemInit

A process is waiting to manage an extension's space allocation in shared memory.

LWLock:AddinShmemInitLock

A process is waiting to manage space allocation in shared memory.

LWLock:async

A process is waiting for I/O on an async (notify) buffer.

LWLock:AsyncCtlLock

A process is waiting to read or update a shared notification state.

LWLock:AsyncQueueLock

A process is waiting to read or update notification messages.

LWLock:AuroraOptimizedReadsCacheMapping

A process is waiting to associate a data block with a page in the Optimized Reads tiered cache.

LWLock:AutoFile

A process is waiting to update the `postgresql.auto.conf` file.

LWLock:AutoFileLock

A process is waiting to update the `postgresql.auto.conf` file.

LWLock:Autovacuum

A process is waiting to read or update the current state of autovacuum workers.

LWLock:AutovacuumLock

An autovacuum worker or launcher is waiting to update or read the current state of autovacuum workers.

LWLock:AutovacuumSchedule

A process is waiting to ensure that a table selected for autovacuum still needs vacuuming.

LWLock:AutovacuumScheduleLock

A process is waiting to ensure that the table it has selected for a vacuum still needs vacuuming.

LWLock:BackendRandomLock

A process is waiting to generate a random number.

LWLock:BackgroundWorker

A process is waiting to read or update background worker state.

LWLock:BackgroundWorkerLock

A process is waiting to read or update the background worker state.

LWLock:BtreeVacuum

A process is waiting to read or update vacuum-related information for a B-tree index.

LWLock:BtreeVacuumLock

A process is waiting to read or update vacuum-related information for a B-tree index.

LWLock:buffer_content

A backend process is waiting to acquire a lightweight lock on the contents of a shared memory buffer. For more information, see LWLock:buffer_content (BufferContent).

LWLock:buffer_mapping

A backend process is waiting to associate a data block with a buffer in the shared buffer pool. For more information, see LWLock:buffer_mapping.

LWLock:BufferIO

A backend process wants to read a page into shared memory. The process is waiting for other processes to finish their I/O for the page. For more information, see LWLock:BufferIO (IPC:BufferIO).

LWLock:Checkpoint

A process is waiting to begin a checkpoint.

LWLock:CheckpointLock

A process is waiting to perform checkpoint.

LWLock:CheckpointerComm

A process is waiting to manage `fsync` requests.

LWLock:CheckpointerCommLock

A process is waiting to manage `fsync` requests.

LWLock:clog

A process is waiting for I/O on a clog (transaction status) buffer.

LWLock:CLogControlLock

A process is waiting to read or update transaction status.

LWLock:CLogTruncationLock

A process is waiting to run `txid_status` or update the oldest transaction ID available to it.

LWLock:commit_timestamp

A process is waiting for I/O on a commit timestamp buffer.

LWLock:CommitTs

A process is waiting to read or update the last value set for a transaction commit timestamp.

LWLock:CommitTsBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a commit timestamp.

LWLock:CommitTsControlLock

A process is waiting to read or update transaction commit timestamps.

LWLock:CommitTsLock

A process is waiting to read or update the last value set for the transaction timestamp.

LWLock:CommitTsSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a commit timestamp.

LWLock:ControlFile

A process is waiting to read or update the `pg_control` file or create a new write-ahead log (WAL) file.

LWLock:ControlFileLock

A process is waiting to read or update the control file or creation of a new write-ahead log (WAL) file.

LWLock:DynamicSharedMemoryControl

A process is waiting to read or update dynamic shared memory allocation information.

LWLock:DynamicSharedMemoryControlLock

A process is waiting to read or update the dynamic shared memory state.

LWLock:lock_manager

A backend process is waiting to add or examine locks for backend processes. Or it's waiting to join or exit a locking group that is used by parallel query. For more information, see LWLock:lock_manager.

LWLock:LockFastPath

A process is waiting to read or update a process's fast-path lock information.

LWLock:LogicalRepWorker

A process is waiting to read or update the state of logical replication workers.

LWLock:LogicalRepWorkerLock

A process is waiting for an action on a logical replication worker to finish.

LWLock:multixact_member

A process is waiting for I/O on a multixact_member buffer.

LWLock:multixact_offset

A process is waiting for I/O on a multixact offset buffer.

LWLock:MultiXactGen

A process is waiting to read or update shared multixact state.

LWLock:MultiXactGenLock

A process is waiting to read or update a shared multixact state.

LWLock:MultiXactMemberBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a multixact member. For more information, see LWLock:MultiXact.

LWLock:MultiXactMemberControlLock

A process is waiting to read or update multixact member mappings.

LWLock:MultiXactMemberSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a multixact member. For more information, see LWLock:MultiXact.

LWLock:MultiXactOffsetBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a multixact offset. For more information, see LWLock:MultiXact.

LWLock:MultiXactOffsetControlLock

A process is waiting to read or update multixact offset mappings.

LWLock:MultiXactOffsetSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a multixact offset. For more information, see LWLock:MultiXact.

LWLock:MultiXactTruncation

A process is waiting to read or truncate multixact information.

LWLock:MultiXactTruncationLock

A process is waiting to read or truncate multixact information.

LWLock:NotifyBuffer

A process is waiting for I/O on the simple least-recently used (SLRU) buffer for a NOTIFY message.

LWLock:NotifyQueue

A process is waiting to read or update NOTIFY messages.

LWLock:NotifyQueueTail

A process is waiting to update a limit on NOTIFY message storage.

LWLock:NotifyQueueTailLock

A process is waiting to update limit on notification message storage.

LWLock:NotifySLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a NOTIFY message.

LWLock:OidGen

A process is waiting to allocate a new object ID (OID).

LWLock:OidGenLock

A process is waiting to allocate or assign an object ID (OID).

LWLock:oldserxid

A process is waiting for I/O on an oldserxid buffer.

LWLock:OldSerXidLock

A process is waiting to read or record conflicting serializable transactions.

LWLock:OldSnapshotTimeMap

A process is waiting to read or update old snapshot control information.

LWLock:OldSnapshotTimeMapLock

A process is waiting to read or update old snapshot control information.

LWLock:parallel_append

A process is waiting to choose the next subplan during parallel append plan execution.

LWLock:parallel_hash_join

A process is waiting to allocate or exchange a chunk of memory or update counters during a parallel hash plan execution.

LWLock:parallel_query_dsa

A process is waiting for a lock on dynamic shared memory allocation for a parallel query.

LWLock:ParallelAppend

A process is waiting to choose the next subplan during parallel append plan execution.

LWLock:ParallelHashJoin

A process is waiting to synchronize workers during plan execution for a parallel hash join.

Lwlock:ParallelQueryDSA

A process is waiting for dynamic shared memory allocation for a parallel query.

Lwlock:PerSessionDSA

A process is waiting for dynamic shared memory allocation for a parallel query.

Lwlock:PerSessionRecordType

A process is waiting to access a parallel query's information about composite types.

Lwlock:PerSessionRecordTypmod

A process is waiting to access a parallel query's information about type modifiers that identify anonymous record types.

Lwlock:PerXactPredicateList

A process is waiting to access the list of predicate locks held by the current serializable transaction during a parallel query.

Lwlock:predicate_lock_manager

A process is waiting to add or examine predicate lock information.

Lwlock:PredicateLockManager

A process is waiting to access predicate lock information used by serializable transactions.

Lwlock:proc

A process is waiting to read or update the fast-path lock information.

LWLock:ProcArray

A process is waiting to access the shared per-process data structures (typically, to get a snapshot or report a session's transaction ID).

LWLock:ProcArrayLock

A process is waiting to get a snapshot or clearing a transaction Id at a transaction's end.

LWLock:RelationMapping

A process is waiting to read or update a `pg_filenode.map` file (used to track the file-node assignments of certain system catalogs).

LWLock:RelationMappingLock

A process is waiting to update the relation map file used to store catalog-to-file-node mapping.

LWLock:RelCacheInit

A process is waiting to read or update a `pg_internal.init` file (a relation cache initialization file).

LWLock:RelCacheInitLock

A process is waiting to read or write a relation cache initialization file.

LWLock:replication_origin

A process is waiting to read or update the replication progress.

LWLock:replication_slot_io

A process is waiting for I/O on a replication slot.

LWLock:ReplicationOrigin

A process is waiting to create, drop, or use a replication origin.

LWLock:ReplicationOriginLock

A process is waiting to set up, drop, or use a replication origin.

LWLock:ReplicationOriginState

A process is waiting to read or update the progress of one replication origin.

LWLock:ReplicationSlotAllocation

A process is waiting to allocate or free a replication slot.

LWLock:ReplicationSlotAllocationLock

A process is waiting to allocate or free a replication slot.

LWLock:ReplicationSlotControl

A process is waiting to read or update a replication slot state.

LWLock:ReplicationSlotControlLock

A process is waiting to read or update the replication slot state.

LWLock:ReplicationSlotIO

A process is waiting for I/O on a replication slot.

LWLock:SerialBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a serializable transaction conflict.

LWLock:SerializableFinishedList

A process is waiting to access the list of finished serializable transactions.

LWLock:SerializableFinishedListLock

A process is waiting to access the list of finished serializable transactions.

LWLock:SerializablePredicateList

A process is waiting to access the list of predicate locks held by serializable transactions.

LWLock:SerializablePredicateLockListLock

A process is waiting to perform an operation on a list of locks held by serializable transactions.

LWLock:SerializableXactHash

A process is waiting to read or update information about serializable transactions.

LWLock:SerializableXactHashLock

A process is waiting to retrieve or store information about serializable transactions.

LWLock:SerialSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a serializable transaction conflict.

LWLock:SharedTidBitmap

A process is waiting to access a shared tuple identifier (TID) bitmap during a parallel bitmap index scan.

LWLock:SharedTupleStore

A process is waiting to access a shared tuple store during a parallel query.

LWLock:ShmemIndex

A process is waiting to find or allocate space in shared memory.

LWLock:ShmemIndexLock

A process is waiting to find or allocate space in shared memory.

LWLock:SInvalRead

A process is waiting to retrieve messages from the shared catalog invalidation queue.

LWLock:SInvalReadLock

A process is waiting to retrieve or remove messages from a shared invalidation queue.

LWLock:SInvalWrite

A process is waiting to add a message to the shared catalog invalidation queue.

LWLock:SInvalWriteLock

A process is waiting to add a message in a shared invalidation queue.

LWLock:subtrans

A process is waiting for I/O on a subtransaction buffer.

LWLock:SubtransBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a subtransaction.

LWLock:SubtransControlLock

A process is waiting to read or update subtransaction information.

LWLock:SubtransSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a subtransaction.

LWLock:SyncRep

A process is waiting to read or update information about the state of synchronous replication.

LWLock:SyncRepLock

A process is waiting to read or update information about synchronous replicas.

LWLock:SyncScan

A process is waiting to select the starting location of a synchronized table scan.

LWLock:SyncScanLock

A process is waiting to get the start location of a scan on a table for synchronized scans.

LWLock:TablespaceCreate

A process is waiting to create or drop a tablespace.

LWLock:TablespaceCreateLock

A process is waiting to create or drop the tablespace.

LWLock:tbm

A process is waiting for a shared iterator lock on a tree bitmap (TBM).

LWLock:TwoPhaseState

A process is waiting to read or update the state of prepared transactions.

LWLock:TwoPhaseStateLock

A process is waiting to read or update the state of prepared transactions.

LWLock:wal_insert

A process is waiting to insert the write-ahead log (WAL) into a memory buffer.

LWLock:WALBufMapping

A process is waiting to replace a page in write-ahead log (WAL) buffers.

LWLock:WALBufMappingLock

A process is waiting to replace a page in write-ahead log (WAL) buffers.

LWLock:WALInsert

A process is waiting to insert write-ahead log (WAL) data into a memory buffer.

LWLock:WALWrite

A process is waiting for write-ahead log (WAL) buffers to be written to disk.

LWLock:WALWriteLock

A process is waiting for write-ahead log (WAL) buffers to be written to disk.

LWLock:WrapLimitsVacuum

A process is waiting to update limits on transaction ID and multixact consumption.

LWLock:WrapLimitsVacuumLock

A process is waiting to update limits on transaction ID and multixact consumption.

LWLock:XactBuffer

A process is waiting for I/O on a simple least-recently used (SLRU) buffer for a transaction status.

LWLock:XactSLRU

A process is waiting to access the simple least-recently used (SLRU) cache for a transaction status.

LWLock:XactTruncation

A process is waiting to run pg_xact_status or update the oldest transaction ID available to it.

LWLock:XidGen

A process is waiting to allocate a new transaction ID.

LWLock:XidGenLock

A process is waiting to allocate or assign a transaction ID.

Timeout:BaseBackupThrottle

A process is waiting during base backup when throttling activity.

Timeout:PgSleep

A backend process has called the pg_sleep function and is waiting for the sleep timeout to expire. For more information, see Timeout:PgSleep.

Timeout:RecoveryApplyDelay

A process is waiting to apply write-ahead log (WAL) during recovery because of a delay setting.

Timeout:RecoveryRetrieveRetryInterval

A process is waiting during recovery when write-ahead log (WAL) data is not available from any source (pg_wal, archive, or stream).

Timeout:VacuumDelay

A process is waiting in a cost-based vacuum delay point.

For a complete list of PostgreSQL wait events, see The Statistics Collector > Wait Event tables in the PostgreSQL documentation.

Amazon Aurora PostgreSQL updates

Following, you can find information about Amazon Aurora PostgreSQL engine version releases and updates. You can also find information about how to upgrade your Aurora PostgreSQL engine. For more information about Aurora releases in general, see Amazon Aurora versions.

> **Tip**
>
> You can minimize the downtime required for a DB cluster upgrade by using a blue/green deployment. For more information, see *Using Blue/Green Deployments for database updates*.

Topics

- Identifying versions of Amazon Aurora PostgreSQL
- Amazon Aurora PostgreSQL releases and engine versions
- Extension versions for Amazon Aurora PostgreSQL
- Upgrading Amazon Aurora PostgreSQL DB clusters
- Aurora PostgreSQL long-term support (LTS) releases

Identifying versions of Amazon Aurora PostgreSQL

Amazon Aurora includes certain features that are general to Aurora and available to all Aurora DB clusters. Aurora includes other features that are specific to a particular database engine that Aurora supports. These features are available only to those Aurora DB clusters that use that database engine, such as Aurora PostgreSQL.

An Aurora database release typically has two version numbers, the database engine version number and the Aurora version number. If an Aurora PostgreSQL release has an Aurora version number, it's included after the engine version number in the Amazon Aurora PostgreSQL releases and engine versions listing.

Aurora version number

Aurora version numbers use the `major.minor.patch` naming scheme. An Aurora patch version includes important bug fixes added to a minor version after its release. To learn more about

Amazon Aurora major, minor, and patch releases, see Amazon Aurora major versions, Amazon Aurora minor versions, and Amazon Aurora patch versions.

You can find out the Aurora version number of your Aurora PostgreSQL DB instance with the following SQL query:

```
postgres=> SELECT aurora_version();
```

Starting with the release of PostgreSQL versions 13.3, 12.8, 11.13, 10.18, and for all other later versions, Aurora version numbers align more closely to the PostgreSQL engine version. For example, querying an Aurora PostgreSQL 13.3 DB cluster returns the following:

```
aurora_version
----------------
 13.3.1
(1 row)
```

Prior releases, such as Aurora PostgreSQL 10.14 DB cluster, return version numbers similar to the following:

```
aurora_version
----------------
 2.7.3
(1 row)
```

PostgreSQL engine version numbers

Starting with PostgreSQL 10, PostgreSQL database engine versions use a *major*.*minor* numbering scheme for all releases. Some examples include PostgreSQL 10.18, PostgreSQL 12.7, and PostgreSQL 13.3.

Releases prior to PostgreSQL 10 used a *major*.*major*.*minor* numbering scheme in which the first two digits make up the major version number and a third digit denotes a minor version. For example, PostgreSQL 9.6 was a major version, with minor versions 9.6.21 or 9.6.22 indicated by the third digit.

Note

The PostgreSQL engine version 9.6 is no longer supported. To upgrade, see Upgrading Amazon Aurora PostgreSQL DB clusters. For version policies and release timelines, see How long Amazon Aurora major versions remain available.

You can find out the PostgreSQL database engine version number with the following SQL query:

```
postgres=> SELECT version();
```

For an Aurora PostgreSQL 13.3 DB cluster, the results are as follows:

```
version
----------------------------------------------------------------------------------------------
 PostgreSQL 13.3 on x86_64-pc-linux-gnu, compiled by x86_64-pc-linux-gnu-gcc (GCC)
 7.4.0, 64-bit
(1 row)
```

Amazon Aurora PostgreSQL releases and engine versions

Amazon Aurora PostgreSQL-Compatible Edition releases are updated regularly. Updates are applied to Aurora PostgreSQL DB clusters during system maintenance windows. When updates are applied depends on the type of update, the AWS Region, and maintenance window setting for the DB cluster. Many of the listed releases include both a PostgreSQL version number and an Amazon Aurora version number. However, starting with the release of PostgreSQL versions 13.3, 12.8, 11.13, 10.18, and for all other later versions, Aurora version numbers aren't used. To identify the version numbers of your Aurora PostgreSQL database, see Identifying versions of Amazon Aurora PostgreSQL.

For information about extensions and modules, see Extension versions for Amazon Aurora PostgreSQL.

Note

For information about Amazon Aurora version policies and release timelines, see How long Amazon Aurora major versions remain available.

For information about support for Amazon Aurora see Amazon RDS FAQs.

To determine which Aurora PostgreSQL DB engine versions are available in an AWS Region, use the describe-db-engine-versions AWS CLI command. For example:

```
aws rds describe-db-engine-versions --engine aurora-postgresql --query '*[].
[EngineVersion]' --output text --region aws-region
```

For a list of AWS Regions, see Aurora PostgreSQL Region availability.

For details about the PostgreSQL versions that are available on Aurora PostgreSQL, see the *Release Notes for Aurora PostgreSQL*.

Extension versions for Amazon Aurora PostgreSQL

You can install and configure various PostgreSQL extensions for use with Aurora PostgreSQL DB clusters. For example, you can use the PostgreSQL `pg_partman` extension to automate the creation and maintenance of table partitions. To learn more about this and other extensions available for Aurora PostgreSQL, see Working with extensions and foreign data wrappers.

For details about the PostgreSQL extensions that are supported on Aurora PostgreSQL, see Extension versions for Amazon Aurora PostgreSQL in *Release Notes for Aurora PostgreSQL*.

Upgrading Amazon Aurora PostgreSQL DB clusters

Amazon Aurora makes new versions of the PostgreSQL database engine available in AWS Regions only after extensive testing. You can upgrade your Aurora PostgreSQL DB clusters to the new version when it's available in your Region.

Depending on the version of Aurora PostgreSQL that your DB cluster is currently running, an upgrade to the new release is either a minor upgrade or a major upgrade. For example, upgrading an Aurora PostgreSQL 11.15 DB cluster to Aurora PostgreSQL 13.6 is a *major version upgrade*. Upgrading an Aurora PostgreSQL 13.3 DB cluster to Aurora PostgreSQL 13.7 is a *minor version upgrade*. In the following topics, you can find information about how to perform both types of upgrades.

Contents

Overview of the Aurora PostgreSQL upgrade processes

The differences between major and minor version upgrades are as follows:

Minor version upgrades and patches

Minor version upgrades and patches include only those changes that are backward-compatible with existing applications. Minor version upgrades and patches become available to you only after Aurora PostgreSQL tests and approves them.

Minor version upgrades can be applied for you automatically by Aurora. When you create a new Aurora PostgreSQL DB cluster, the **Enable minor version upgrade** option is preselected. Unless you turn off this option, minor version upgrades are applied automatically during your scheduled maintenance window. For more information about the automatic minor version upgrade (AmVU) option and how to modify your Aurora DB cluster to use it, see Automatic minor version upgrades for Aurora DB clusters.

If the automatic minor version upgrade option isn't set for your Aurora PostgreSQL DB cluster, your Aurora PostgreSQL isn't automatically upgraded to the new minor version. Instead, when a new minor version is released in your AWS Region and your Aurora PostgreSQL DB cluster is running an older minor version, Aurora prompts you to upgrade. It does so by adding a recommendation to the maintenance tasks for your cluster.

Patches aren't considered an upgrade, and they aren't applied automatically. Aurora PostgreSQL prompts you to apply any patches by adding a recommendation to maintenance tasks for your Aurora PostgreSQL DB cluster. For more information, see How to perform minor version upgrades and apply patches.

Note

Patches that resolve security or other critical issues are also added as maintenance tasks. However, these patches are required. Make sure to apply security patches to your Aurora PostgreSQL DB cluster when they become available in your pending maintenance tasks.

The upgrade process involves the possibility of brief outages as each instance in the cluster is upgraded to the new version. However, after Aurora PostgreSQL versions 14.3.3, 13.7.3, 12.11.3, 11.16.3, 10.21.3 and other higher releases of these minor versions and newer major versions, the upgrade process uses the zero-downtime patching (ZDP) feature. This feature minimizes outages, and in most cases completely eliminates them. For more information, see Minor release upgrades and zero-downtime patching.

Note

ZDP isn't supported in the following cases:

- When Aurora PostgreSQL DB clusters are configured as Aurora Serverless v1.
- When Aurora PostgreSQL DB clusters are configured as Aurora global database in the secondary AWS Regions.
- During the upgrade of reader instances in Aurora global database.
- During OS patches and OS upgrades.

ZDP is supported for Aurora PostgreSQL DB clusters that are configured as Aurora Serverless v2.

Major version upgrades

Unlike for minor version upgrades and patches, Aurora PostgreSQL doesn't have an automatic major version upgrade option. New major PostgreSQL versions might contain database changes that aren't backward-compatible with existing applications. The new functionality can cause your existing applications to stop working correctly.

To prevent any issues, we strongly recommend that you follow the process outlined in Testing an upgrade of your production DB cluster to a new major version before upgrading the DB instances in your Aurora PostgreSQL DB clusters. First ensure that your applications can run on the new version by following that procedure. Then you can manually upgrade your Aurora PostgreSQL DB cluster to the new version.

The upgrade process involves the possibility of brief outage when all the instances in the cluster are upgraded to the new version. The preliminary planning process also takes time. We recommend that you always perform upgrade tasks during your cluster's maintenance window or when operations are minimal. For more information, see How to perform a major version upgrade.

> **Note**
>
> Both minor version upgrades and major version upgrades might involve brief outages. For that reason, we recommend strongly that you perform or schedule upgrades during your maintenance window or during other periods of low utilization.

Aurora PostgreSQL DB clusters occasionally require operating system updates. These updates might include a newer version of glibc library. During such updates, we recommend you to follow the guidelines as described in Collations supported in Aurora PostgreSQL.

Getting a list of available versions in your AWS Region

You can get a list of all engine versions available as upgrade targets for your Aurora PostgreSQL DB cluster by querying your AWS Region using the describe-db-engine-versions AWS CLI command, as follows.

For Linux, macOS, or Unix:

```
aws rds describe-db-engine-versions \
  --engine aurora-postgresql \
  --engine-version version-number \
  --query 'DBEngineVersions[*].ValidUpgradeTarget[*].{EngineVersion:EngineVersion}' \
  --output text
```

For Windows:

```
aws rds describe-db-engine-versions ^
  --engine aurora-postgresql ^
  --engine-version version-number ^
  --query "DBEngineVersions[*].ValidUpgradeTarget[*].{EngineVersion:EngineVersion}" ^
  --output text
```

For example, to identify the valid upgrade targets for an Aurora PostgreSQL version 12.10 DB cluster, run the following AWS CLI command:

For Linux, macOS, or Unix:

```
aws rds describe-db-engine-versions \
  --engine aurora-postgresql \
  --engine-version 12.10 \
  --query 'DBEngineVersions[*].ValidUpgradeTarget[*].{EngineVersion:EngineVersion}' \
  --output text
```

For Windows:

```
aws rds describe-db-engine-versions ^
  --engine aurora-postgresql ^
  --engine-version 12.10 ^
  --query "DBEngineVersions[*].ValidUpgradeTarget[*].{EngineVersion:EngineVersion}" ^
  --output text
```

In this table, you can find both major and minor version upgrade targets that are available for various Aurora PostgreSQL DB versions.

Curre sourc versi	**Upgrade targets**			
16.1	16			
15.6	16			
15.5	16	16	15	
15.4	16	16	15	15

Current source version	Upgrade targets																						
15.3	16	16	15	15	15																		
15.2	16	16	15	15	15	15																	
14.11	16	15																					
14.10	16	16	15	15																			
14.9	16	16	15	15	15	14	14																
14.8	16	16	15	15	15	15	15	14	14	14													
14.7	16	16	15	15	15	15	15	14	14	14	14												
14.6	16	16	15	15	15	15	15	14	14	14	14	14											
14.5	16	16	15	15	15	15	15	14	14	14	14	14	14										
14.4	16	16	15	15	15	15	15	14	14	14	14	14	14	14									
14.3	16	16	15	15	15	15	15	14	14	14	14	14	14	14	14								
13.14	16	15	14																				
13.13	16	16	15	15	14	14																	
13.12	16	16	15	15	15	14	14	14															
13.11	16	16	15	15	15	15	14	14	14	14													
13.10	16	16	15	15	15	15	15	14	14	14	14	14	13	13	13	13							
13.9	16	16	15	15	15	15	15	14	14	14	14	14	14	13	13	13							
13.8	16	16	15	15	15	15	15	14	14	14	14	14	14	14	13	13	13	13	13	13			
13.7	16	16	15	15	15	15	15	14	14	14	14	14	14	14	14	14	13	13	13	13	13	13	13

Curre sourc versi	Upgrade targets																										
12.18	16	15	14	13																							
12.17	16	16	15	15	14	14	13																				
12.16	16	16	15	15	15	14	14	14	13	13	13																
12.15	16	16	15	15	15	15	14	14	14	14	13	13	13	13													
12.14	16	16	15	15	15	15	15	15	14	14	14	14	14	13	13	13	13	13	12								
12.13	16	16	15	15	15	15	15	14	14	14	14	14	14	13	13	13	13	13	13	12	12	12	12				
12.12	16	16	15	15	15	15	15	14	14	14	14	14	14	14	13	13	13	13	13	13	12	12	12	13	12	12	12	[illegible]
12.11	16	16	15	15	15	15	15	14	14	14	14	14	14	14	14	13	13	13	13	13	13	13	13	12	12	12	12	[illegible]
12.9	16	16	15	15	15	15	15	14	14	14	14	14	13	13	13	13	13	13	13	13	12	12	12	12	12	12	12	[illegible]
11.21	16	16	15	15	15	14	14	14	13	13	13	12	12															
11.9	16	16	15	15	15	15	15	14	14	14	14	14	14	13	13	13	13	13	13	12	12	12	12	12	12	12	12	11.21

For any version that you're considering, always check the availability of your cluster's DB instance class. For example, `db.r4` isn't supported for Aurora PostgreSQL 13. If your Aurora PostgreSQL DB cluster currently uses a db.r4 instance class, you need to move to db.r5 before trying to upgrade. For more information about DB instance classes, including which ones are Graviton2-based and which ones are Intel-based, see Aurora DB instance classes.

How to perform a major version upgrade

Major version upgrades might contain database changes that are not backward-compatible with previous versions of the database. New functionality in a new version can cause your existing applications to stop working correctly. To avoid issues, Amazon Aurora doesn't apply major version upgrades automatically. Rather, we recommend that you carefully plan for a major version upgrade by following these steps:

1. Choose the major version that you want from the list of available targets from those listed for your version in the table. You can get a precise list of versions available in your AWS Region for your current version by using the AWS CLI. For details, see Getting a list of available versions in your AWS Region.
2. Verify that your applications work as expected on a trial deployment of the new version. For information about the complete process, see Testing an upgrade of your production DB cluster to a new major version.
3. After verifying that your applications work as expected on the trial deployment, you can upgrade your cluster. For details, see Upgrading the Aurora PostgreSQL engine to a new major version.

> **Note**
>
> You can perform a major version upgrade from Babelfish for Aurora PostgreSQL 13-based versions starting from 13.6 to Aurora PostgreSQL 14-based versions starting from 14.6. Babelfish for Aurora PostgreSQL 13.4 and 13.5 don't support major version upgrade.

You can get a list of engine versions available as major version upgrade targets for your Aurora PostgreSQL DB cluster by querying your AWS Region using the describe-db-engine-versions AWS CLI command, as follows.

For Linux, macOS, or Unix:

```
aws rds describe-db-engine-versions \
  --engine aurora-postgresql \
  --engine-version version-number \
  --query 'DBEngineVersions[].ValidUpgradeTarget[?IsMajorVersionUpgrade == `true`].
{EngineVersion:EngineVersion}' \
  --output text
```

For Windows:

```
aws rds describe-db-engine-versions ^
  --engine aurora-postgresql ^
  --engine-version version-number ^
  --query "DBEngineVersions[].ValidUpgradeTarget[?IsMajorVersionUpgrade == `true`].
{EngineVersion:EngineVersion}" ^
```

```
  --output text
```

In some cases, the version that you want to upgrade to isn't a target for your current version. In such cases, use the information in the versions table to perform minor version upgrades until your cluster is at a version that has your chosen target in its row of targets.

Testing an upgrade of your production DB cluster to a new major version

Each new major version includes enhancements to the query optimizer that are designed to improve performance. However, your workload might include queries that result in a worse performing plan in the new version. That's why we recommend that you test and review performance before upgrading in production. You can manage query plan stability across versions by using the Query Plan Management (QPM) extension, as detailed in Ensuring plan stability after a major version upgrade.

Before upgrading your production Aurora PostgreSQL DB clusters to a new major version, we strongly recommend that you test the upgrade to verify that all your applications work correctly:

1. Have a version-compatible parameter group ready.

   If you are using a custom DB instance or DB cluster parameter group, you can choose from two options:

   a. Specify the default DB instance, DB cluster parameter group, or both for the new DB engine version.

   b. Create your own custom parameter group for the new DB engine version.

   If you associate a new DB instance or DB cluster parameter group as a part of the upgrade request, make sure to reboot the database after the upgrade completes to apply the parameters. If a DB instance needs to be rebooted to apply the parameter group changes, the instance's parameter group status shows `pending-reboot`. You can view an instance's parameter group status in the console or by using a CLI command such as describe-db-instances or describe-db-clusters.

2. Check for unsupported usage:

   - Commit or roll back all open prepared transactions before attempting an upgrade. You can use the following query to verify that there are no open prepared transactions on your instance.

     ```
     SELECT count(*) FROM pg_catalog.pg_prepared_xacts;
     ```

- Remove all uses of the *reg** data types before attempting an upgrade. Except for `regtype` and `regclass`, you can't upgrade the *reg** data types. The pg_upgrade utility (used by Amazon Aurora to do the upgrade) can't persist this data type. To learn more about this utility, see [pg_upgrade]() in the PostgreSQL documentation.

  To verify that there are no uses of unsupported *reg** data types, use the following query for each database.

  ```
  SELECT count(*) FROM pg_catalog.pg_class c, pg_catalog.pg_namespace n,
   pg_catalog.pg_attribute a
    WHERE c.oid = a.attrelid
        AND NOT a.attisdropped
        AND a.atttypid IN ('pg_catalog.regproc'::pg_catalog.regtype,
                           'pg_catalog.regprocedure'::pg_catalog.regtype,
                           'pg_catalog.regoper'::pg_catalog.regtype,
                           'pg_catalog.regoperator'::pg_catalog.regtype,
                           'pg_catalog.regconfig'::pg_catalog.regtype,
                           'pg_catalog.regdictionary'::pg_catalog.regtype)
        AND c.relnamespace = n.oid
        AND n.nspname NOT IN ('pg_catalog', 'information_schema');
  ```

- If you are upgrading an Aurora PostgreSQL version 10.18 or higher DB cluster that has the `pgRouting` extension installed, drop the extension before upgrading to version 12.4 or higher.

  If you are upgrading an Aurora PostgreSQL 10.x version that has the extension `pg_repack` version 1.4.3 installed, drop the extension before upgrading to any higher version.

3. Check for template1 and template0 databases.

   For a successful upgrade, template 1 and template 0 databases must exist and should be listed as a template. To check on this, use the following command:

   ```
   SELECT datname, datistemplate FROM pg_database;

   datname    | datistemplate
   -----------+---------------
   template0  | t
   rdsadmin   | f
   template1  | t
   postgres   | f
   ```

In the command output, the `datistemplate` value for template1 and template0 databases should be `t`.

4. Drop logical replication slots.

   The upgrade process can't proceed if the Aurora PostgreSQL DB cluster is using any logical replication slots. Logical replication slots are typically used for short-term data migration tasks, such as migrating data using AWS DMS or for replicating tables from the database to data lakes, BI tools, or other targets. Before upgrading, make sure that you know the purpose of any logical replication slots that exist, and confirm that it's okay to delete them. You can check for logical replication slots using the following query:

   ```
   SELECT * FROM pg_replication_slots;
   ```

   If logical replication slots are still being used, you shouldn't delete them, and you can't proceed with the upgrade. However, if the logical replication slots aren't needed, you can delete them using the following SQL:

   ```
   SELECT pg_drop_replication_slot(slot_name);
   ```

   Logical replication scenarios that use the `pglogical` extension also need to have slots dropped from the publisher node for a successful major version upgrade on that node. However, you can restart the replication process from the subscriber node after the upgrade. For more information, see Reestablishing logical replication after a major upgrade.

5. Perform a backup.

   The upgrade process creates a DB cluster snapshot of your DB cluster during upgrading. If you also want to do a manual backup before the upgrade process, see Creating a DB cluster snapshot for more information.

6. Upgrade certain extensions to the latest available version before performing the major version upgrade. The extensions to update include the following:

   - `pgRouting`
   - `postgis_raster`
   - `postgis_tiger_geocoder`
   - `postgis_topology`
   - `address_standardizer`

- address_standardizer_data_us

Run the following command for each extension that's currently installed.

```
ALTER EXTENSION PostgreSQL-extension UPDATE TO 'new-version';
```

For more information, see Upgrading PostgreSQL extensions. To learn more about upgrading PostGIS, see Step 6: Upgrade the PostGIS extension.

7. If you're upgrading to version 11.x, drop the extensions that it doesn't support before performing the major version upgrade. The extensions to drop include:

   - chkpass
   - tsearch2

8. Drop unknown data types, depending on your target version.

   PostgreSQL version 10 doesn't support the unknown data type. If a version 9.6 database uses the unknown data type, an upgrade to version 10 shows an error message such as the following.

   ```
   Database instance is in a state that cannot be upgraded: PreUpgrade checks failed: 
   The instance could not be upgraded because the 'unknown' data type is used in user
    tables. 
   Please remove all usages of the 'unknown' data type and try again."
   ```

   To find the unknown data type in your database so that you can remove such columns or change them to supported data types, use the following SQL code for each database.

   ```
   SELECT n.nspname, c.relname, a.attname
       FROM pg_catalog.pg_class c,
       pg_catalog.pg_namespace n,
       pg_catalog.pg_attribute a
       WHERE c.oid = a.attrelid AND NOT a.attisdropped AND
       a.atttypid = 'pg_catalog.unknown'::pg_catalog.regtype AND
       c.relkind IN ('r','m','c') AND
       c.relnamespace = n.oid AND
       n.nspname !~ '^pg_temp_' AND
       n.nspname !~ '^pg_toast_temp_' AND n.nspname NOT IN ('pg_catalog',
    'information_schema');
   ```

9. Perform a dry-run upgrade.

We highly recommend testing a major version upgrade on a duplicate of your production database before trying the upgrade on your production database. You can monitor the execution plans on the duplicate test instance for any possible execution plan regressions and to evaluate its performance. To create a duplicate test instance, you can either restore your database from a recent snapshot or clone your database. For more information, see Restoring from a snapshot or Cloning a volume for an Amazon Aurora DB cluster.

For more information, see Upgrading the Aurora PostgreSQL engine to a new major version.

10. Upgrade your production instance.

    When your dry-run major version upgrade is successful, you should be able to upgrade your production database with confidence. For more information, see Upgrading the Aurora PostgreSQL engine to a new major version.

    > **Note**
    >
    > During the upgrade process, Aurora PostgreSQL takes a DB cluster snapshot if the cluster's backup retention period is greater than 0. You can't do a point-in-time restore of your cluster during this process. Later, you can perform a point-in-time restore to times before the upgrade began and after the automatic snapshot of your instance has completed. However, you can't perform a point-in-time restore to a previous minor version.

    For information about an upgrade in progress, you can use Amazon RDS to view two logs that the pg_upgrade utility produces. These are `pg_upgrade_internal.log` and `pg_upgrade_server.log`. Amazon Aurora appends a timestamp to the file name for these logs. You can view these logs as you can any other log. For more information, see Monitoring Amazon Aurora log files.

11. Upgrade PostgreSQL extensions. The PostgreSQL upgrade process doesn't upgrade any PostgreSQL extensions. For more information, see Upgrading PostgreSQL extensions.

Post-upgrade recommendations

After you complete a major version upgrade, we recommend the following:

- Run the ANALYZE operation to refresh the pg_statistic table. You should do this for every database on all your PostgreSQL DB instances. Optimizer statistics aren't transferred during a major version upgrade, so you need to regenerate all statistics to avoid performance issues. Run the command without any parameters to generate statistics for all regular tables in the current database, as follows:

  ```
  ANALYZE VERBOSE;
  ```

  The VERBOSE flag is optional, but using it shows you the progress. For more information, see ANALYZE in the PostgreSQL documentation.

  > **Note**
  >
  > Run ANALYZE on your system after the upgrade to avoid performance issues.

- If you upgraded to PostgreSQL version 10, run REINDEX on any hash indexes you have. Hash indexes were changed in version 10 and must be rebuilt. To locate invalid hash indexes, run the following SQL for each database that contains hash indexes.

  ```
  SELECT idx.indrelid::regclass AS table_name,
     idx.indexrelid::regclass AS index_name
  FROM pg_catalog.pg_index idx
     JOIN pg_catalog.pg_class cls ON cls.oid = idx.indexrelid
     JOIN pg_catalog.pg_am am ON am.oid = cls.relam
  WHERE am.amname = 'hash'
  AND NOT idx.indisvalid;
  ```

- We recommend that you test your application on the upgraded database with a similar workload to verify that everything works as expected. After the upgrade is verified, you can delete this test instance.

Upgrading the Aurora PostgreSQL engine to a new major version

When you initiate the upgrade process to a new major version, Aurora PostgreSQL takes a snapshot of the Aurora DB cluster before it makes any changes to your cluster. This snapshot is created for major version upgrades only, not minor version upgrades. When the upgrade process completes, you can find this snapshot among the manual snapshots listed under **Snapshots** in the RDS console. The snapshot name includes preupgrade as its prefix, the name of your Aurora

PostgreSQL DB cluster, the source version, the target version, and the date and timestamp, as shown in the following example.

```
preupgrade-docs-lab-apg-global-db-12-8-to-13-6-2022-05-19-00-19
```

After the upgrade completes, you can use the snapshot that Aurora created and stored in your manual snapshot list to restore the DB cluster to its previous version, if necessary.

Tip

In general, snapshots provide many ways to restore your Aurora DB cluster to various points in time. To learn more, see Restoring from a DB cluster snapshot and Restoring a DB cluster to a specified time. However, Aurora PostgreSQL doesn't support using a snapshot to restore to a previous minor version.

During the major version upgrade process, Aurora allocates a volume and clones the source Aurora PostgreSQL DB cluster. If the upgrade fails for any reason, Aurora PostgreSQL uses the clone to roll back the upgrade. After more than 15 clones of a source volume are allocated, subsequent clones become full copies and take longer. This can cause the upgrade process also to take longer. If Aurora PostgreSQL rolls back the upgrade, be aware of the following:

- You might see billing entries and metrics for both the original volume and the cloned volume allocated during the upgrade. Aurora PostgreSQL cleans up the extra volume after the cluster backup retention window is beyond the time of the upgrade.
- The next cross-Region snapshot copy from this cluster will be a full copy instead of an incremental copy.

To safely upgrade the DB instances that make up your cluster, Aurora PostgreSQL uses the pg_upgrade utility. After the writer upgrade completes, each reader instance experiences a brief outage while it's upgraded to the new major version. To learn more about this PostgreSQL utility, see pg_upgrade in the PostgreSQL documentation.

You can upgrade your Aurora PostgreSQL DB cluster to a new version by using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To upgrade the engine version of a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to upgrade.
3. Choose **Modify**. The **Modify DB cluster** page appears.
4. For **Engine version**, choose the new version.
5. Choose **Continue** and check the summary of modifications.
6. To apply the changes immediately, choose **Apply immediately**. Choosing this option can cause an outage in some cases. For more information, see Modifying an Amazon Aurora DB cluster.
7. On the confirmation page, review your changes. If they are correct, choose **Modify Cluster** to save your changes.

   Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

AWS CLI

To upgrade the engine version of a DB cluster, use the modify-db-cluster AWS CLI command. Specify the following parameters:

- `--db-cluster-identifier` – The name of the DB cluster.
- `--engine-version` – The version number of the database engine to upgrade to. For information about valid engine versions, use the AWS CLI describe-db-engine-versions command.
- `--allow-major-version-upgrade` – A required flag when the `--engine-version` parameter is a different major version than the DB cluster's current major version.
- `--no-apply-immediately` – Apply changes during the next maintenance window. To apply changes immediately, use `--apply-immediately`.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --engine-version new_version \
    --allow-major-version-upgrade \
    --no-apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --engine-version new_version ^
    --allow-major-version-upgrade ^
    --no-apply-immediately
```

RDS API

To upgrade the engine version of a DB cluster, use the ModifyDBCluster operation. Specify the following parameters:

- `DBClusterIdentifier` – The name of the DB cluster, for example *mydbcluster*.
- `EngineVersion` – The version number of the database engine to upgrade to. For information about valid engine versions, use the DescribeDBEngineVersions operation.
- `AllowMajorVersionUpgrade` – A required flag when the `EngineVersion` parameter is a different major version than the DB cluster's current major version.
- `ApplyImmediately` – Whether to apply changes immediately or during the next maintenance window. To apply changes immediately, set the value to `true`. To apply changes during the next maintenance window, set the value to `false`.

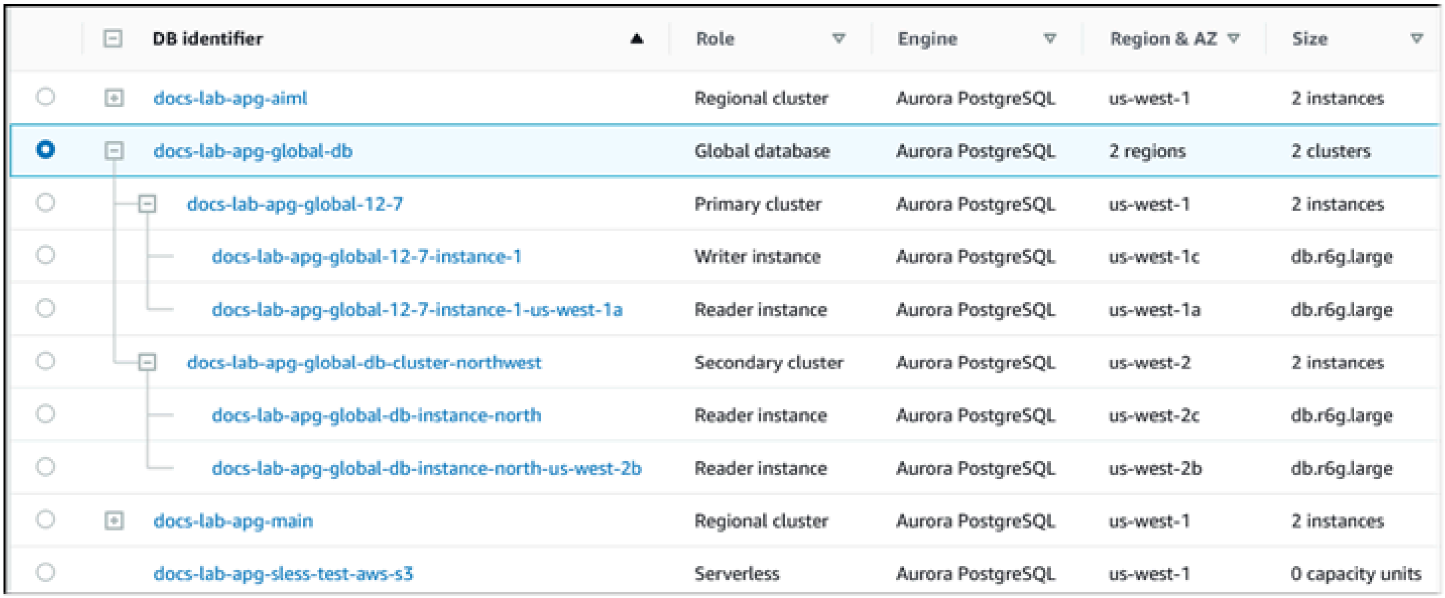
Major upgrades for global databases

For an Aurora global database cluster, the upgrade process upgrades all DB clusters that make up your Aurora global database at the same time. It does so to ensure that each runs the same Aurora PostgreSQL version. It also ensures that any changes to system tables, data file formats, and so on, are automatically replicated to all secondary clusters.

To upgrade a global database cluster to a new major version of Aurora PostgreSQL, we recommend that you test your applications on the upgraded version, as detailed in Testing an upgrade of your production DB cluster to a new major version. Be sure to prepare your DB cluster parameter group and DB parameter group settings for each AWS Region in your Aurora global database before the upgrade as detailed in step 1. of Testing an upgrade of your production DB cluster to a new major version.

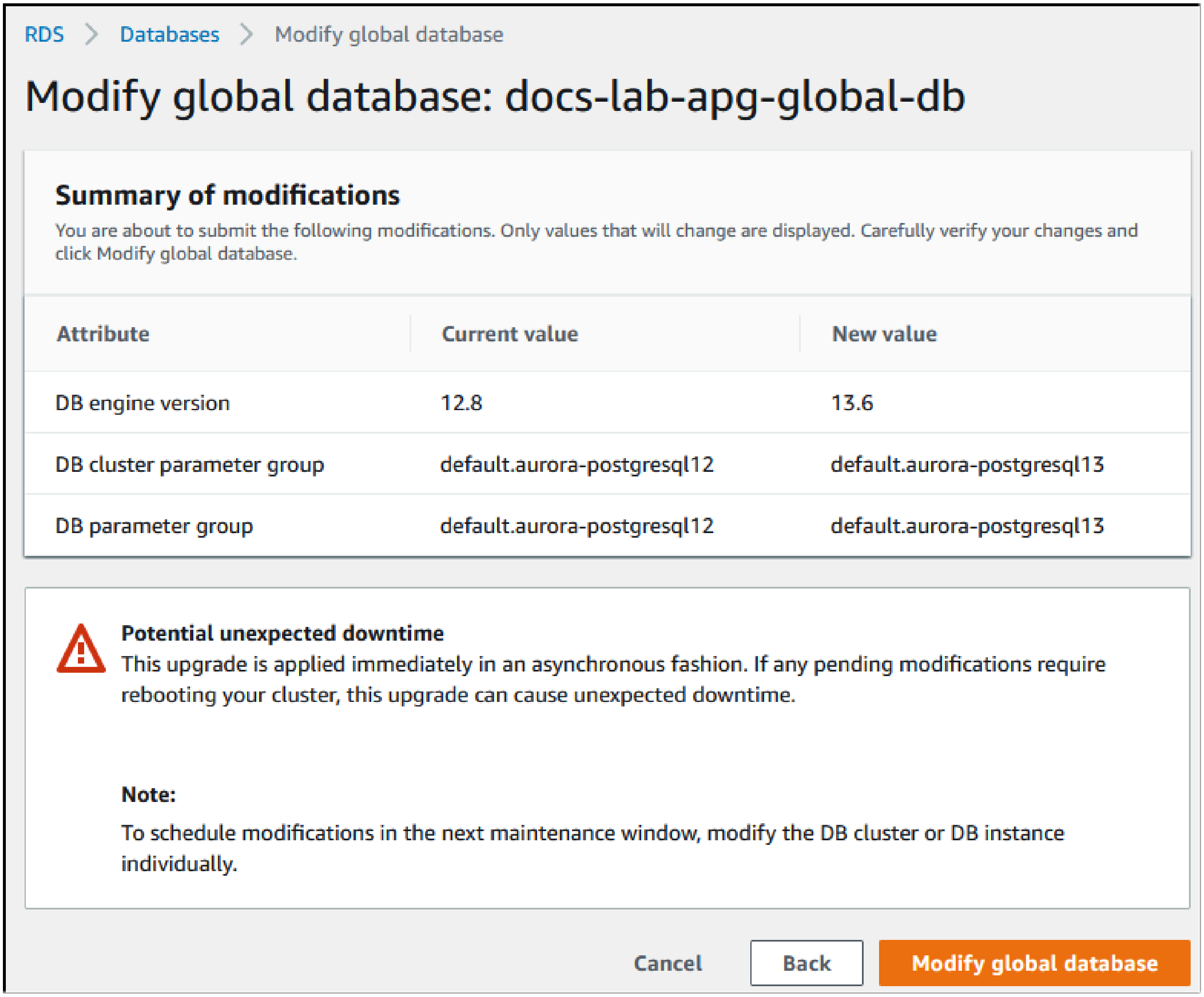
If your Aurora PostgreSQL global database cluster has a recovery point objective (RPO) set for its `rds.global_db_rpo` parameter, make sure to reset the parameter before upgrading. The major version upgrade process doesn't work if the RPO is turned on. By default, this parameter is turned off. For more information about Aurora PostgreSQL global databases and RPO, see Managing RPOs for Aurora PostgreSQL–based global databases.

If you verify that your applications can run as expected on the trial deployment of the new version, you can start the upgrade process. To do so, see Upgrading the Aurora PostgreSQL engine to a new major version. Be sure to choose the top-level item from the **Databases** list in the RDS console, **Global database**, as shown in the following image.



	DB identifier	Role	Engine	Region & AZ	Size
○	⊞ docs-lab-apg-aiml	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances
●	⊟ docs-lab-apg-global-db	Global database	Aurora PostgreSQL	2 regions	2 clusters
○	⊟ docs-lab-apg-global-12-7	Primary cluster	Aurora PostgreSQL	us-west-1	2 instances
○	docs-lab-apg-global-12-7-instance-1	Writer instance	Aurora PostgreSQL	us-west-1c	db.r6g.large
○	docs-lab-apg-global-12-7-instance-1-us-west-1a	Reader instance	Aurora PostgreSQL	us-west-1a	db.r6g.large
○	⊟ docs-lab-apg-global-db-cluster-northwest	Secondary cluster	Aurora PostgreSQL	us-west-2	2 instances
○	docs-lab-apg-global-db-instance-north	Reader instance	Aurora PostgreSQL	us-west-2c	db.r6g.large
○	docs-lab-apg-global-db-instance-north-us-west-2b	Reader instance	Aurora PostgreSQL	us-west-2b	db.r6g.large
○	⊞ docs-lab-apg-main	Regional cluster	Aurora PostgreSQL	us-west-1	2 instances
○	docs-lab-apg-sless-test-aws-s3	Serverless	Aurora PostgreSQL	us-west-1	0 capacity units

As with any modification, you can confirm that you want the process to proceed when prompted.



Modify global database: docs-lab-apg-global-db

Summary of modifications

You are about to submit the following modifications. Only values that will change are displayed. Carefully verify your changes and click Modify global database.

Attribute	Current value	New value
DB engine version	12.8	13.6
DB cluster parameter group	default.aurora-postgresql12	default.aurora-postgresql13
DB parameter group	default.aurora-postgresql12	default.aurora-postgresql13

Potential unexpected downtime
This upgrade is applied immediately in an asynchronous fashion. If any pending modifications require rebooting your cluster, this upgrade can cause unexpected downtime.

Note:

To schedule modifications in the next maintenance window, modify the DB cluster or DB instance individually.

Rather than using the console, you can start the upgrade process by using the AWS CLI or the RDS API. As with the console, you operate on the Aurora global database cluster rather than any of its constituents, as follows:

- Use the modify-global-cluster AWS CLI command to start the upgrade for your Aurora global database by using the AWS CLI.
- Use the ModifyGlobalCluster API to start the upgrade.

Before performing a minor version upgrade

We recommend that you perform the following actions to reduce the downtime during a minor version upgrade:

- The Aurora DB cluster maintenance should be performed during a period of low traffic. Use Performance Insights to identify these time periods in order to configure the maintenance windows correctly. For more information on Performance Insights, see Monitoring DB load with Performance Insights on Amazon RDS. For more information on DB cluster maintenance window, Adjusting the preferred DB cluster maintenance window.
- Use AWS SDKs that support exponential backoff and jitter as a best practice. For more information, see Exponential Backoff And Jitter.

How to perform minor version upgrades and apply patches

Minor version upgrades and patches become available in AWS Regions only after rigorous testing. Before releasing upgrades and patches, Aurora PostgreSQL tests to ensure that known security issues, bugs, and other issues that emerge after the release of the minor community version don't disrupt overall Aurora PostgreSQL fleet stability.

As Aurora PostgreSQL makes new minor versions available, the instances that make up your Aurora PostgreSQL DB cluster can be automatically upgraded during your specified maintenance window. For this to happen, your Aurora PostgreSQL DB cluster must have the **Enable auto minor version upgrade** option turned on. All DB instances that make up your Aurora PostgreSQL DB cluster must have the automatic minor version upgrade (AmVU) option turned on so that the minor upgrade to be applied throughout the cluster.

> **Tip**
>
> Make sure that the **Enable auto minor version upgrade** option is turned on for all PostgreSQL DB instances that make up your Aurora PostgreSQL DB cluster. This option must be turned on for every instance in the DB cluster to work. For information on how to set **Auto minor version upgrade**, and how the setting works when applied at the cluster and instance levels, see Automatic minor version upgrades for Aurora DB clusters.

You can check the value of the **Enable auto minor version upgrade** option for all your Aurora PostgreSQL DB clusters by using the describe-db-instances AWS CLI command with the following query.

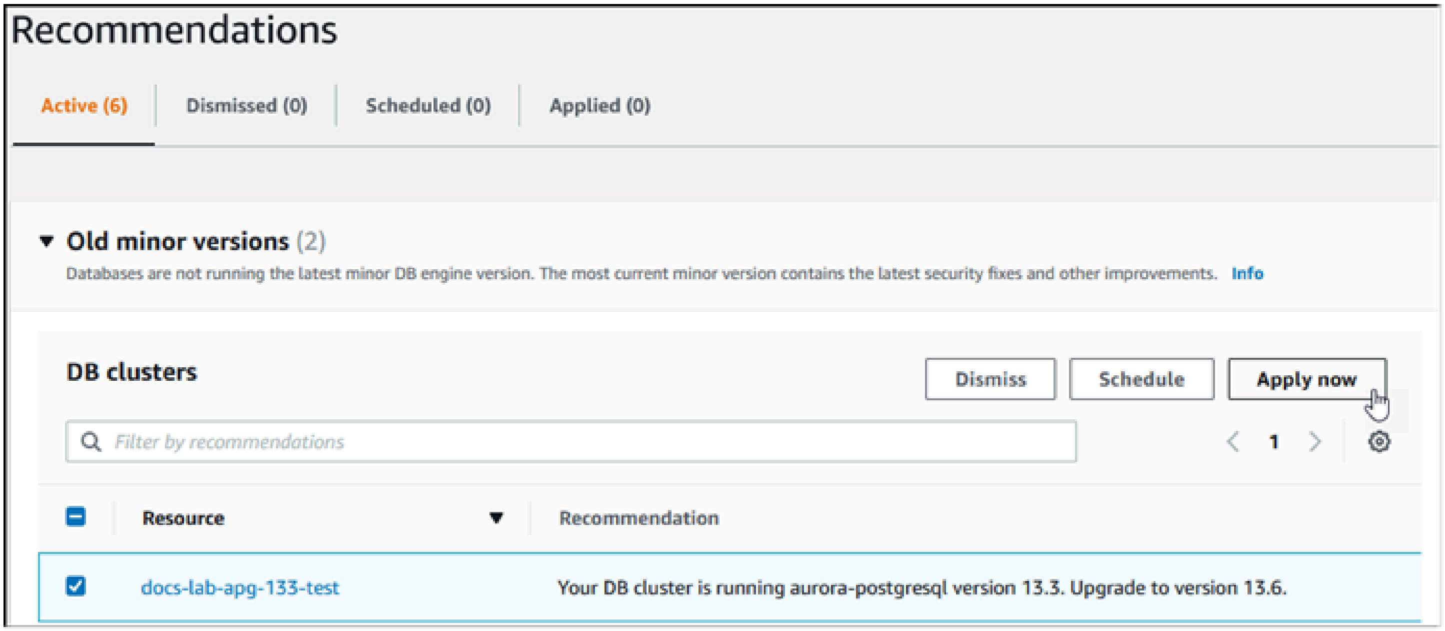
```
aws rds describe-db-instances \
  --query '*[].
{DBClusterIdentifier:DBClusterIdentifier,DBInstanceIdentifier:DBInstanceIdentifier,AutoMinorVer
```

This query returns a list of all Aurora DB clusters and their instances with a `true` or `false` value for the status of the `AutoMinorVersionUpgrade` setting. The command as shown assumes that you have your AWS CLI configured to target your default AWS Region.

For more information about the AmVU option and how to modify your Aurora DB cluster to use it, see Automatic minor version upgrades for Aurora DB clusters.

You can upgrade your Aurora PostgreSQL DB clusters to new minor versions either by responding to maintenance tasks, or by modifying the cluster to use the new version.

You can identify any available upgrades or patches for your Aurora PostgreSQL DB clusters by using the RDS console and opening the **Recommendations** menu. There, you can find a list of various maintenance issues such as **Old minor versions**. Depending on your production environment, you can choose to **Schedule** the upgrade or take immediate action, by choosing **Apply now**, as shown following.



Databases are not running the latest minor DB engine version. The most current minor version contains the latest security fixes and other improvements.

	Resource	Recommendation
☑	docs-lab-apg-133-test	Your DB cluster is running aurora-postgresql version 13.3. Upgrade to version 13.6.

To learn more about how to maintain an Aurora DB cluster, including how to manually apply patches and minor version upgrades, see Maintaining an Amazon Aurora DB cluster.

Minor release upgrades and zero-downtime patching

Upgrading an Aurora PostgreSQL DB cluster involves the possibility of an outage. During the upgrade process, the database is shut down as it's being upgraded. If you start the upgrade while the database is busy, you lose all connections and transactions that the DB cluster is processing. If you wait until the database is idle to perform the upgrade, you might have to wait a long time.

The zero-downtime patching (ZDP) feature improves the upgrading process. With ZDP, both minor version upgrades and patches can be applied with minimal impact to your Aurora PostgreSQL DB cluster. ZDP is used when applying patches or newer minor version upgrades to Aurora PostgreSQL versions and other higher releases of these minor versions and newer major versions. That is, upgrading to new minor versions from any of these releases onward uses ZDP.

The following table shows the Aurora PostgreSQL versions and DB instance classes where ZDP is available:

Version	db.r* instance classes	db.t* instance classes	db.x* instance classes	db.serverless instance class
10.21.0 and higher 10.21 versions	Yes	Yes	Yes	N/A
11.16.0 and higher 11.16 versions	Yes	Yes	Yes	N/A
11.17 and higher versions	Yes	Yes	Yes	N/A
12.11.0 and higher 12.11 versions	Yes	Yes	Yes	N/A
12.12 and higher versions	Yes	Yes	Yes	N/A

Version	db.r* instance classes	db.t* instance classes	db.x* instance classes	db.serverless instance class
13.7.0 and higher 13.7 versions	Yes	Yes	Yes	N/A
13.8 and higher versions	Yes	Yes	Yes	Yes
14.3.1 and higher 14.3 versions	Yes	Yes	Yes	N/A
14.4.0 and higher 14.4 versions	Yes	Yes	Yes	N/A
14.5 and higher versions	Yes	Yes	Yes	Yes
15.3 and higher versions	Yes	Yes	Yes	Yes

ZDP works by preserving current client connections to your Aurora PostgreSQL DB cluster throughout the Aurora PostgreSQL upgrade process. However, in the following cases, connections will be dropped for ZDP to complete:

- Long running query or transactions are in progress.
- Data definition language (DDL) statements are running.
- Temporary tables or table locks are in use.
- All sessions are listening on notification channels.
- A cursor in ‘WITH HOLD’ status is in use.
- TLSv1.3 or TLSv1.1 connections are in use.

During the upgrade process using ZDP, the database engine looks for a quiet point to pause all new transactions. This action safeguards the database during patches and upgrades. To make sure that your applications run smoothly with paused transactions, we recommend integrating retry logic into your code. This approach ensures that the system can manage any brief downtime without failing and can retry the new transactions after the upgrade.

When ZDP completes successfully, application sessions are maintained except for those with dropped connections, and the database engine restarts while the upgrade is still in progress. Although the database engine restart can cause a temporary drop in throughput, this typically lasts only for a few seconds or at most, approximately one minute.

In some cases, zero-downtime patching (ZDP) might not succeed. For example, parameter changes that are in a `pending` state on your Aurora PostgreSQL DB cluster or its instances interfere with ZDP.

You can find metrics and events for ZDP operations in **Events** page in the console. The events include the start of the ZDP upgrade and completion of the upgrade. In this event you can find how long the process took, and the numbers of preserved and dropped connections that occurred during the restart. You can find details in your database error log.

Upgrading the Aurora PostgreSQL engine to a new minor version

You can upgrade your Aurora PostgreSQL DB cluster to a new minor version by using the console, the AWS CLI, or the RDS API. Before performing the upgrade, we recommend that you follow the same best practice that we recommend for major version upgrades. As with new major versions, new minor versions can also have optimizer improvements, such as fixes, that can cause query plan regressions. To ensure plan stability, we recommend that you use the Query Plan Management (QPM) extension as detailed in Ensuring plan stability after a major version upgrade.

Console

To upgrade the engine version of your Aurora PostgreSQL DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB cluster that you want to upgrade.
3. Choose **Modify**. The **Modify DB cluster** page appears.
4. For **Engine version**, choose the new version.

5. Choose **Continue** and check the summary of modifications.
6. To apply the changes immediately, choose **Apply immediately**. Choosing this option can cause an outage in some cases. For more information, see Modifying an Amazon Aurora DB cluster.
7. On the confirmation page, review your changes. If they are correct, choose **Modify Cluster** to save your changes.

   Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

AWS CLI

To upgrade the engine version of a DB cluster, use the modify-db-cluster AWS CLI command with the following parameters:

- `--db-cluster-identifier` – The name of your Aurora PostgreSQL DB cluster.
- `--engine-version` – The version number of the database engine to upgrade to. For information about valid engine versions, use the AWS CLI describe-db-engine-versions command.
- `--no-apply-immediately` – Apply changes during the next maintenance window. To apply changes immediately, use `--apply-immediately` instead.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --engine-version new_version \
    --no-apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --engine-version new_version ^
    --no-apply-immediately
```

RDS API

To upgrade the engine version of a DB cluster, use the ModifyDBCluster operation. Specify the following parameters:

- `DBClusterIdentifier` – The name of the DB cluster, for example *`mydbcluster`*.
- `EngineVersion` – The version number of the database engine to upgrade to. For information about valid engine versions, use the DescribeDBEngineVersions operation.
- `ApplyImmediately` – Whether to apply changes immediately or during the next maintenance window. To apply changes immediately, set the value to `true`. To apply changes during the next maintenance window, set the value to `false`.

Upgrading PostgreSQL extensions

Upgrading your Aurora PostgreSQL DB cluster to a new major or minor version doesn't upgrade the PostgreSQL extensions at the same time. For most extensions, you upgrade the extension after the major or minor version upgrade completes. However, in some cases, you upgrade the extension before you upgrade the Aurora PostgreSQL DB engine. For more information, see the list of extensions to update in Testing an upgrade of your production DB cluster to a new major version.

Installing PostgreSQL extensions requires `rds_superuser` privileges. Typically, an `rds_superuser` delegates permissions over specific extensions to relevant users (roles), to facilitate the management of a given extension. That means that the task of upgrading all the extensions in your Aurora PostgreSQL DB cluster might involve many different users (roles). Keep this in mind especially if you want to automate the upgrade process by using scripts. For more information about PostgreSQL privileges and roles, see Security with Amazon Aurora PostgreSQL.

> **Note**
>
> For information about updating the PostGIS extension, see Managing spatial data with the PostGIS extension (Step 6: Upgrade the PostGIS extension).
> To update the `pg_repack` extension, drop the extension and then create the new version in the upgraded DB instance. For more information, see pg_repack installation in the `pg_repack` documentation.

To update an extension after an engine upgrade, use the `ALTER EXTENSION UPDATE` command.

```
ALTER EXTENSION extension_name UPDATE TO 'new_version';
```

To list your currently installed extensions, use the PostgreSQL pg_extension catalog in the following command.

```
SELECT * FROM pg_extension;
```

To view a list of the specific extension versions that are available for your installation, use the PostgreSQL pg_available_extension_versions view in the following command.

```
SELECT * FROM pg_available_extension_versions;
```

Alternative blue/green upgrade technique

In some situations, your top priority is to perform an immediate switchover from the old cluster to an upgraded one. In such situations, you can use a multistep process that runs the old and new clusters side-by-side. Here, you replicate data from the old cluster to the new one until you are ready for the new cluster to take over. For details, see *Using Blue/Green Deployments for database updates*.

Aurora PostgreSQL long-term support (LTS) releases

Each new Aurora PostgreSQL version remains available for a certain amount of time for you to use when you create or upgrade a DB cluster. After this period, you must upgrade any clusters that use that version. You can manually upgrade your cluster before the support period ends, or Aurora can automatically upgrade it for you when its Aurora PostgreSQL version is no longer supported.

Aurora designates certain Aurora PostgreSQL versions as long-term support (LTS) releases. Database clusters that use LTS releases can stay on the same version longer and undergo fewer upgrade cycles than clusters that use non-LTS releases. LTS minor versions include only bug fixes (through patch versions); an LTS version doesn't include new features released after its introduction.

Once a year, DB clusters running on an LTS minor version are patched to the latest patch version of the LTS release. We do this patching to help ensure that you benefit from cumulative security and stability fixes. We might patch an LTS minor version more frequently if there are critical fixes, such as for security, that need to be applied.

Note

To remain on an LTS minor version for the duration of its lifecycle, make sure to turn off **Auto minor version upgrade** for your DB instances. To avoid automatically upgrading your DB cluster from the LTS minor version, set **Auto minor version upgrade** to **No** on all DB instances in your Aurora cluster.

We recommend that you upgrade to the latest release, instead of using the LTS release, for most of your Aurora PostgreSQL clusters. Doing so takes advantage of Aurora as a managed service and gives you access to the latest features and bug fixes. LTS releases are intended for clusters with the following characteristics:

- You can't afford downtime on your Aurora PostgreSQL application for upgrades outside of rare occurrences for critical patches.
- The testing cycle for the cluster and associated applications takes a long time for each update to the Aurora PostgreSQL database engine.
- The database version for your Aurora PostgreSQL cluster has all the DB engine features and bug fixes that your application needs.

The current LTS releases for Aurora PostgreSQL are as follows:

- PostgreSQL 14.6. It was released on January 20, 2023. For more information, see PostgreSQL 14.6 in the *Release Notes for Aurora PostgreSQL*.
- PostgreSQL 13.9. It was released on January 20, 2023. For more information, see PostgreSQL 13.9 in the *Release Notes for Aurora PostgreSQL*.
- PostgreSQL 12.9. It was released on February 25, 2022. For more information, see PostgreSQL 12.9 in the *Release Notes for Aurora PostgreSQL*.
- PostgreSQL 11.9 (Aurora PostgreSQL release 3.4. It was released on December 11, 2020. For more information about this version, see PostgreSQL 11.9, Aurora PostgreSQL release 3.4 in the *Release Notes for Aurora PostgreSQL*.

For information about how to identify Aurora and database engine versions, see Identifying versions of Amazon Aurora PostgreSQL.

Using Amazon Aurora global databases

Amazon Aurora global databases span multiple AWS Regions, enabling low latency global reads and providing fast recovery from the rare outage that might affect an entire AWS Region. An Aurora global database has a primary DB cluster in one Region, and up to five secondary DB clusters in different Regions.

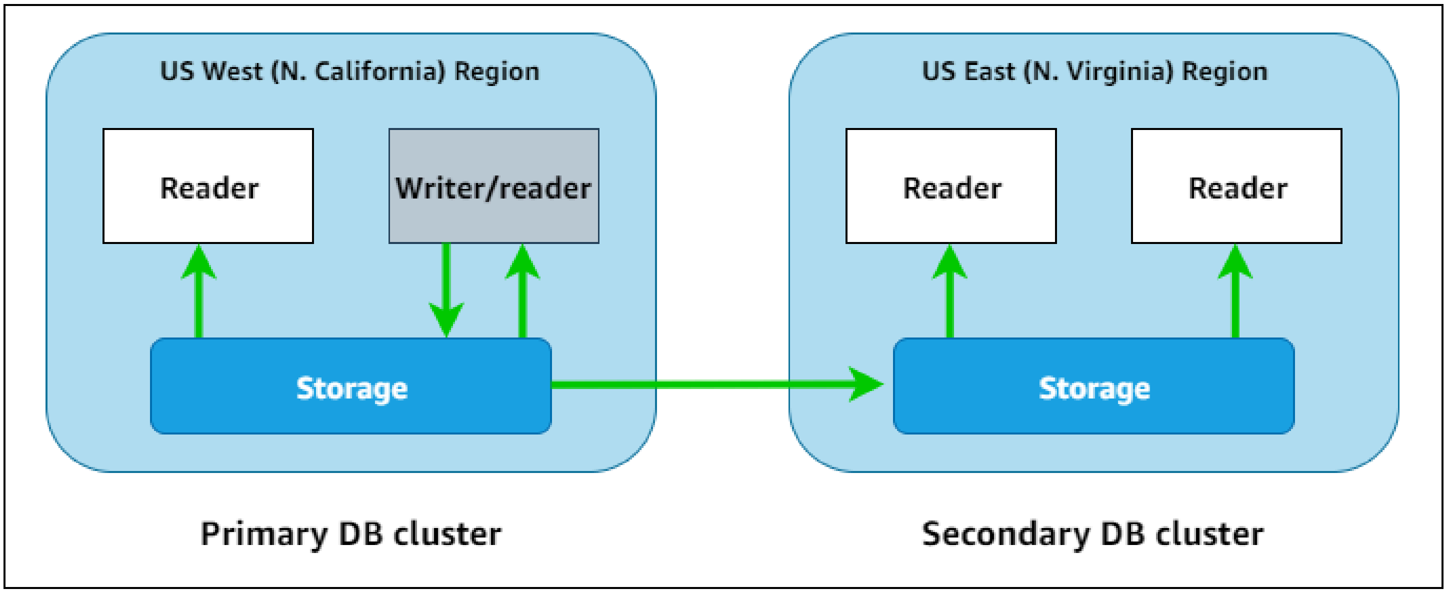
Topics

Overview of Amazon Aurora global databases

By using an Amazon Aurora global database, you can run your globally distributed applications using a single Aurora database that spans multiple AWS Regions.

An Aurora global database consists of one *primary* AWS Region where your data is written, and up to five read-only *secondary* AWS Regions. You issue write operations directly to the primary DB cluster in the primary AWS Region. Aurora replicates data to the secondary AWS Regions using dedicated infrastructure, with latency typically under a second.

In the following diagram, you can find an example Aurora global database that spans two AWS Regions.



You can scale up each secondary cluster independently, by adding one or more Aurora Replicas (read-only Aurora DB instances) to serve read-only workloads.

Only the primary cluster performs write operations. Clients that perform write operations connect to the DB cluster endpoint of the primary DB cluster. As shown in the diagram, Aurora global database uses the cluster storage volume and not the database engine for replication. To learn more, see Overview of Amazon Aurora storage.

Aurora global databases are designed for applications with a worldwide footprint. The read-only secondary DB clusters (AWS Regions) allow you to support read operations closer to application users. By using the write forwarding feature, you can also configure an Aurora global database so that secondary clusters send data to the primary. For more information, see Using write forwarding in an Amazon Aurora global database.

An Aurora global database supports two different operations for changing the Region of your primary DB cluster, depending on the scenario: *global database switchover* and *global database failover*.

- For planned operational procedures such as Regional rotation, use *global database switchover* (previously called "managed planned failover"). With this feature, you can relocate the primary cluster of a healthy Aurora global database to one of its secondary Regions with no data loss. To learn more, see Performing switchovers for Amazon Aurora global databases.
- To recover your Aurora global database after an outage in the primary Region, use *global database failover*. With this feature, you fail over your primary DB cluster to another Region

(cross-Region failover). To learn more, see Performing managed failovers for Aurora global databases.

Advantages of Amazon Aurora global databases

By using Aurora global databases, you can get the following advantages:

- **Global reads with local latency –** If you have offices around the world, you can use an Aurora global database to keep your main sources of information updated in the primary AWS Region. Offices in your other Regions can access the information in their own Region, with local latency.
- **Scalable secondary Aurora DB clusters –** You can scale your secondary clusters by adding more read-only instances (Aurora Replicas) to a secondary AWS Region. The secondary cluster is read-only, so it can support up to 16 read-only Aurora Replica instances rather than the usual limit of 15 for a single Aurora cluster.
- **Fast replication from primary to secondary Aurora DB clusters –** The replication performed by an Aurora global database has little performance impact on the primary DB cluster. The resources of the DB instances are fully devoted to serve application read and write workloads.
- **Recovery from Region-wide outages –** The secondary clusters allow you to make an Aurora global database available in a new primary AWS Region more quickly (lower RTO) and with less data loss (lower RPO) than traditional replication solutions.

Region and version availability

Feature availability and support vary across specific versions of each Aurora database engine, and across AWS Regions. For more information on version and Region availability with Aurora and global databases, see Supported Regions and DB engines for Aurora global databases.

Limitations of Amazon Aurora global databases

The following limitations currently apply to Aurora global databases:

- Aurora global databases are available in certain AWS Regions and for specific Aurora MySQL and Aurora PostgreSQL versions only. For more information, see Supported Regions and DB engines for Aurora global databases.

- Aurora global databases have specific configuration requirements for supported Aurora DB instance classes, maximum number of AWS Regions, and so on. For more information, see Configuration requirements of an Amazon Aurora global database.
- For Aurora MySQL with MySQL 5.7 compatibility, Aurora global database switchovers require version 2.09.1 or a higher minor version.
- You can perform managed cross-Region switchovers or failovers on an Aurora global database only if the primary and secondary DB clusters have the same major, minor, and patch level engine versions. However, the patch levels can be different if the minor engine versions are one of the following.

Database engine	Minor engine versions
Aurora PostgreSQL	• Version 14.5 or higher minor version • Version 13.8 or higher minor version • Version 12.12 or higher minor version • Version 11.17 or higher minor version

  For more information, see Patch level compatibility for managed cross-Region switchovers and failovers.
- Aurora global databases currently don't support the following Aurora features:
  - Aurora Serverless v1
  - Backtracking in Aurora
- For limitations with using the RDS Proxy feature with global databases, see Limitations for RDS Proxy with global databases.
- Automatic minor version upgrade doesn't apply to Aurora MySQL and Aurora PostgreSQL clusters that are part of an Aurora global database. Note that you can specify this setting for a DB instance that is part of a global database cluster, but the setting has no effect.
- Aurora global databases currently don't support Aurora Auto Scaling for secondary DB clusters.
- To use database activity streams on Aurora global databases running Aurora MySQL 5.7, the engine version must be version 2.08 or higher. For information about database activity streams, see Monitoring Amazon Aurora with Database Activity Streams.
- The following limitations currently apply to upgrading Aurora global databases:

  - You can't apply a custom parameter group to the global database cluster while you're performing a major version upgrade of that Aurora global database. You create your custom parameter groups in each Region of the global cluster and you apply them manually to the Regional clusters after the upgrade.
  - With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on. For more information on the methods that you can use, see Major version upgrades.
  - With an Aurora global database based on Aurora PostgreSQL, you can't perform a major version upgrade of the Aurora DB engine if the recovery point objective (RPO) feature is turned on. For information about the RPO feature, see Managing RPOs for Aurora PostgreSQL–based global databases.
  - With an Aurora global database based on Aurora MySQL, you can't perform a minor version upgrade from version 3.01 or 3.02 to 3.03 or higher by using the standard process. For details about the process to use, see Upgrading Aurora MySQL by modifying the engine version.

  For information about upgrading an Aurora global database, see Upgrading an Amazon Aurora global database.

- You can't stop or start the Aurora DB clusters in your Aurora global database individually. To learn more, see Stopping and starting an Amazon Aurora DB cluster.
- Aurora Replicas attached to the secondary Aurora DB cluster can restart under certain circumstances. If the primary AWS Region's writer DB instance restarts or fails over, Aurora Replicas in secondary Regions also restart. The secondary cluster is then unavailable until all replicas are back in sync with the primary DB cluster's writer instance. The behavior of the primary cluster when rebooting or failing over is the same as for a singular, nonglobal DB cluster. For more information, see Replication with Amazon Aurora.

  Be sure that you understand the impacts to your Aurora global database before making changes to your primary DB cluster. To learn more, see Recovering an Amazon Aurora global database from an unplanned outage.
- Aurora global databases currently don't support the `inaccessible-encryption-credentials-recoverable` status when Amazon Aurora loses access to the AWS KMS key for the DB cluster. In these cases, the encrypted DB cluster goes directly into the terminal `inaccessible-encryption-credentials` state. For more information about these states, see Viewing DB cluster status.

- Aurora PostgreSQL–based DB clusters running in an Aurora global database have the following limitations:
  - Cluster cache management isn't supported for Aurora PostgreSQL DB clusters that are part of Aurora global databases.
  - If the primary DB cluster of your Aurora global database is based on a replica of an Amazon RDS PostgreSQL instance, you can't create a secondary cluster. Don't attempt to create a secondary from that cluster using the AWS Management Console, the AWS CLI, or the `CreateDBCluster` API operation. Attempts to do so time out, and the secondary cluster isn't created.

We recommend that you create secondary DB clusters for your Aurora global databases by using the same version of the Aurora DB engine as the primary. For more information, see Creating an Amazon Aurora global database.

Getting started with Amazon Aurora global databases

To get started with Aurora global databases, first decide which Aurora DB engine you want to use and in which AWS Regions. Only specific versions of the Aurora MySQL and Aurora PostgreSQL database engines in certain AWS Regions support Aurora global databases. For the complete list, see Supported Regions and DB engines for Aurora global databases.

You can create an Aurora global database in one of the following ways:

- **Create a new Aurora global database with new Aurora DB clusters and Aurora DB instances** – You can do this by following the steps in Creating an Amazon Aurora global database. After you create the primary Aurora DB cluster, you then add the secondary AWS Region by following the steps in Adding an AWS Region to an Amazon Aurora global database.
- **Use an existing Aurora DB cluster that supports the Aurora global database feature and add an AWS Region to it** – You can do this only if your existing Aurora DB cluster uses a DB engine version that supports the Aurora global mode or is global-compatible. For some DB engine versions, this mode is explicit, but for others, it's not.

  Check whether you can choose **Add region** for **Action** on the AWS Management Console when your Aurora DB cluster is selected. If you can, you can use that Aurora DB cluster for your Aurora global cluster. For more information, see Adding an AWS Region to an Amazon Aurora global database.

Before creating an Aurora global database, we recommend that you understand all configuration requirements.

Topics

Configuration requirements of an Amazon Aurora global database

An Aurora global database spans at least two AWS Regions. The primary AWS Region supports an Aurora DB cluster that has one writer Aurora DB instance. A secondary AWS Region runs a read-only Aurora DB cluster made up entirely of Aurora Replicas. At least one secondary AWS Region is required, but an Aurora global database can have up to five secondary AWS Regions. The table lists the maximum Aurora DB clusters, Aurora DB instances, and Aurora Replicas allowed in an Aurora global database.

Description	Primary AWS Region	Secondary AWS Regions
Aurora DB clusters	1	5 (maximum)
Writer instances	1	0
Read-only instances (Aurora replicas), per Aurora DB cluster	15 (max)	16 (total)
Read-only instances (max allowed, given actual number of secondary Regions)	15 - *s*	*s* = total number of secondary AWS Regions

The Aurora DB clusters that make up an Aurora global database have the following specific requirements:

- **DB instance class requirements –** An Aurora global database requires DB instance classes that are optimized for memory-intensive applications. For information about the memory optimized DB instance classes, see DB instance classes. We recommend that you use a db.r5 or higher instance class.
- **AWS Region requirements –** An Aurora global database needs a primary Aurora DB cluster in one AWS Region, and at least one secondary Aurora DB cluster in a different Region. You can create up to five secondary (read-only) Aurora DB clusters, and each must be in a different Region. In other words, no two Aurora DB clusters in an Aurora global database can be in the same AWS Region.
- **Naming requirements –** The names you choose for each of your Aurora DB clusters must be unique, across all AWS Regions. You can't use the same name for different Aurora DB clusters even though they're in different Regions.
- **Capacity requirements for Aurora Serverless v2 –** For a global database with Aurora Serverless v2, the minimum capacity required for the DB cluster in the primary AWS Region is 8 ACUs.

Before you can follow the procedures in this section, you need an AWS account. Complete the setup tasks for working with Amazon Aurora. For more information, see Setting up your environment for Amazon Aurora. You also need to complete other preliminary steps for creating any Aurora DB cluster. To learn more, see Creating an Amazon Aurora DB cluster.

Creating an Amazon Aurora global database

In some cases, you might have an existing Aurora provisioned DB cluster running an Aurora database engine that's global-compatible. If so, you can add another AWS Region to it to create your Aurora global database. To do so, see Adding an AWS Region to an Amazon Aurora global database.

To create an Aurora global database by using the AWS Management Console, the AWS CLI, or the RDS API, use the following steps.

Console

The steps for creating an Aurora global database begin by signing in to an AWS Region that supports the Aurora global database feature. For a complete list, see Supported Regions and DB engines for Aurora global databases.

One of the following steps is choosing a virtual private cloud (VPC) based on Amazon VPC for your Aurora DB cluster. To use your own VPC, we recommend that you create it in advance so it's available for you to choose. At the same time, create any related subnets, and as needed a subnet group and security group. To learn how, see Tutorial: Create an Amazon VPC for use with a DB instance.

For general information about creating an Aurora DB cluster, see Creating an Amazon Aurora DB cluster.

To create an Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Create database**. On the **Create database** page, do the following:

   - For the database creation method, choose **Standard create**. (Don't choose Easy create.)
   - For `Engine type` in the **Engine options** section, choose the applicable engine type, **Aurora (MySQL Compatible)** or **Aurora (PostgreSQL Compatible)**.
3. Continue creating your Aurora global database by using the steps from the following procedures.

Creating a global database using Aurora MySQL

The following steps apply to all versions of Aurora MySQL.

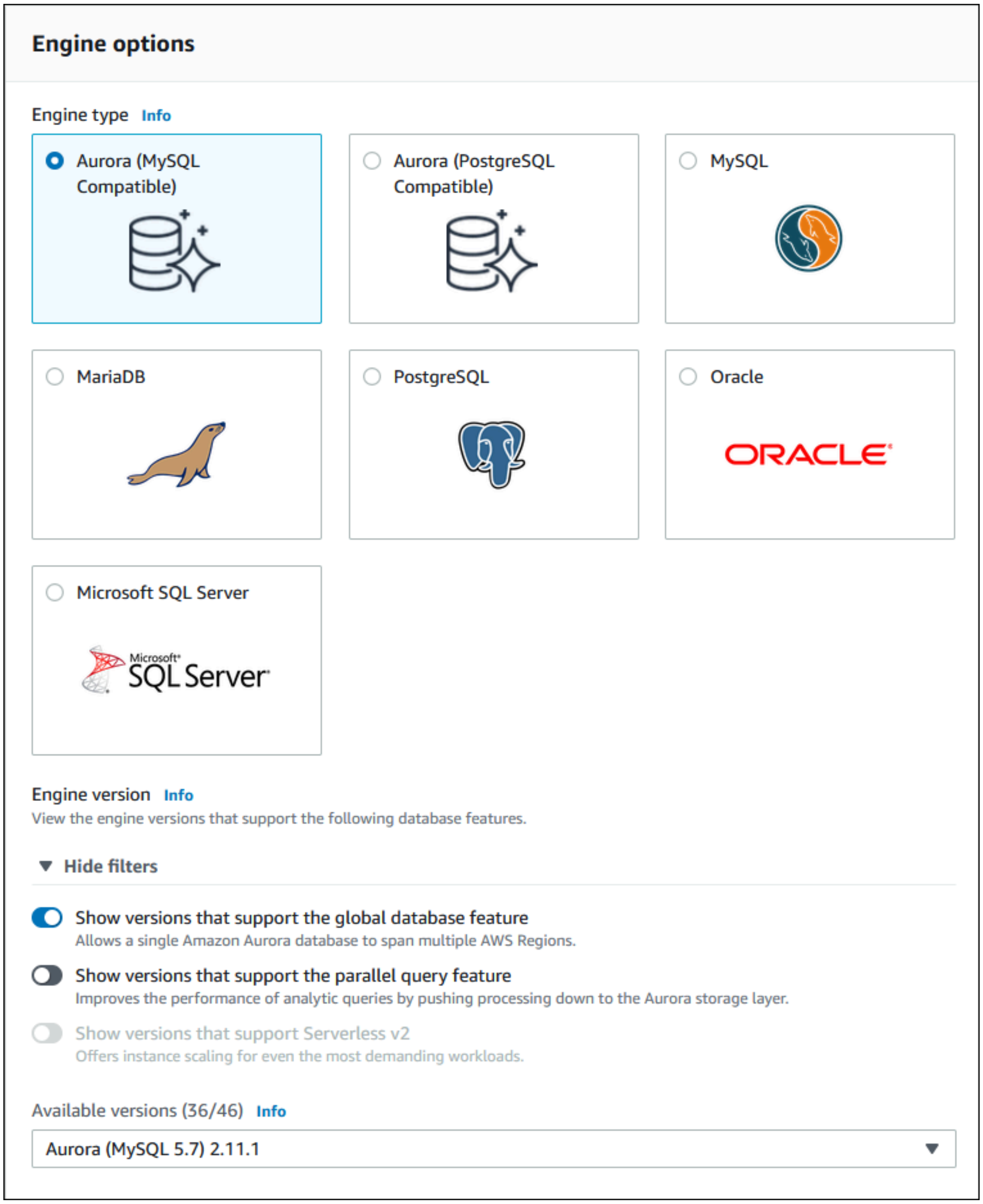
To create an Aurora global database using Aurora MySQL

Complete the **Create database** page.

1. For **Engine options**, choose the following:

   a. Expand **Show filters**, and then turn on **Show versions that support the global database feature**.

   b. For **Engine version**, choose the version of Aurora MySQL that you want to use for your Aurora global database.



Engine options

Engine type Info

● Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

Engine version Info

View the engine versions that support the following database features.

▼ Hide filters

Show versions that support the global database feature: on

Allows a single Amazon Aurora database to span multiple AWS Regions.

Show versions that support the parallel query feature: off

Improves the performance of analytic queries by pushing processing down to the Aurora storage layer.

Show versions that support Serverless v2: off (unavailable)

Offers instance scaling for even the most demanding workloads.

Available versions (36/46) Info

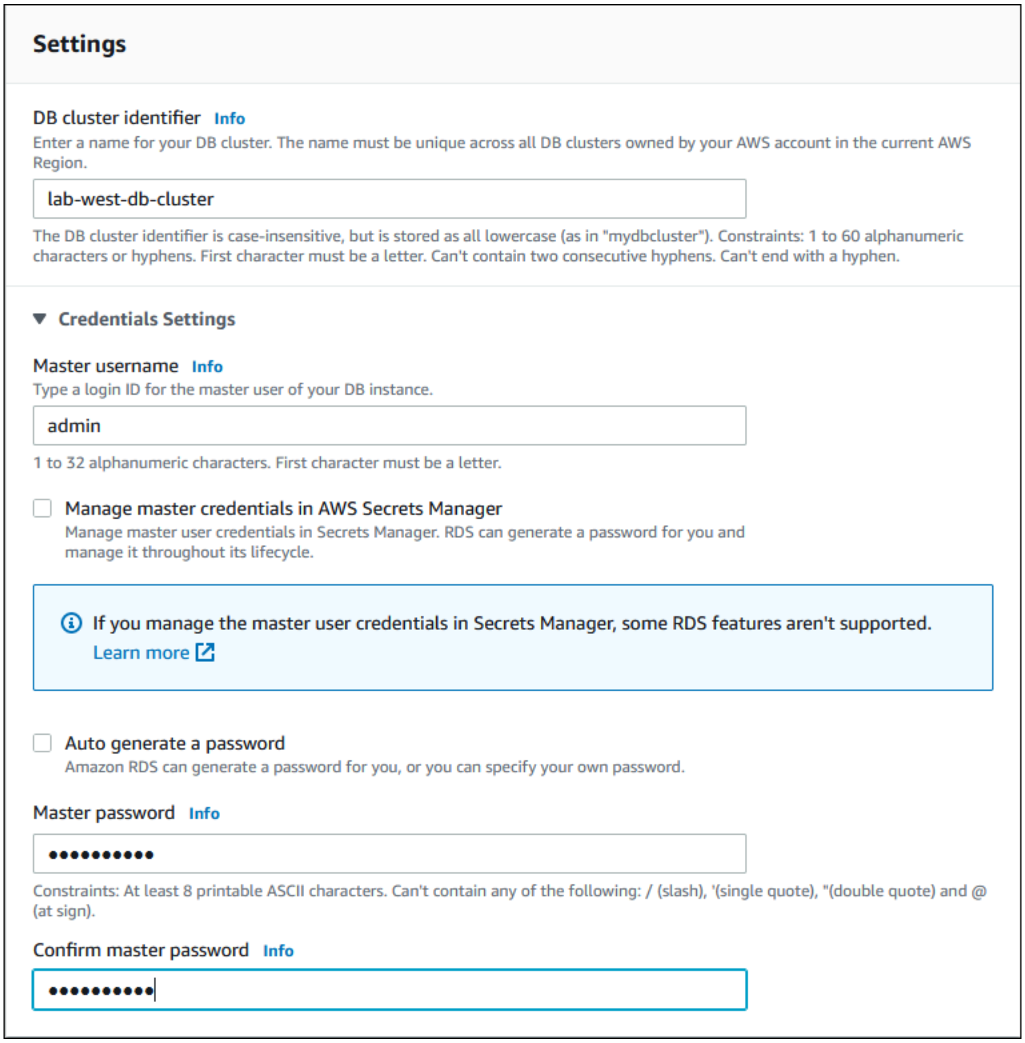
Aurora (MySQL 5.7) 2.11.1

2. For **Templates**, choose **Production**. Or you can choose Dev/Test if appropriate for your use case. Don't use Dev/Test in production environments.

3. For **Settings**, do the following:

   a. Enter a meaningful name for the DB cluster identifier. When you finish creating the Aurora global database, this name identifies the primary DB cluster.

   b. Enter your own password for the `admin` user account for the DB instance, or have Aurora generate one for you. If you choose to autogenerate a password, you get an option to copy the password.



Settings

DB cluster identifier Info

Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

lab-west-db-cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

admin

1 to 32 alphanumeric characters. First character must be a letter.

□ Manage master credentials in AWS Secrets Manager
Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

ⓘ If you manage the master user credentials in Secrets Manager, some RDS features aren't supported.
Learn more

□ Auto generate a password
Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info

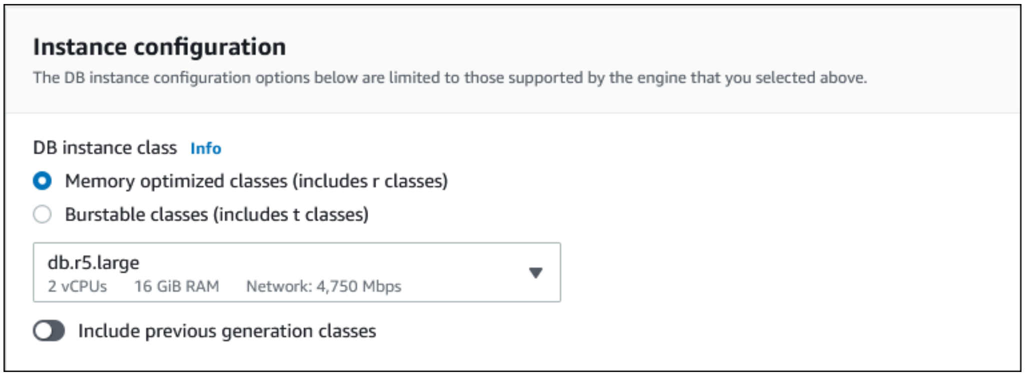
••••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password Info

••••••••••

4. For **DB instance class**, choose `db.r5.large` or another memory optimized DB instance class. We recommend that you use a db.r5 or higher instance class.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

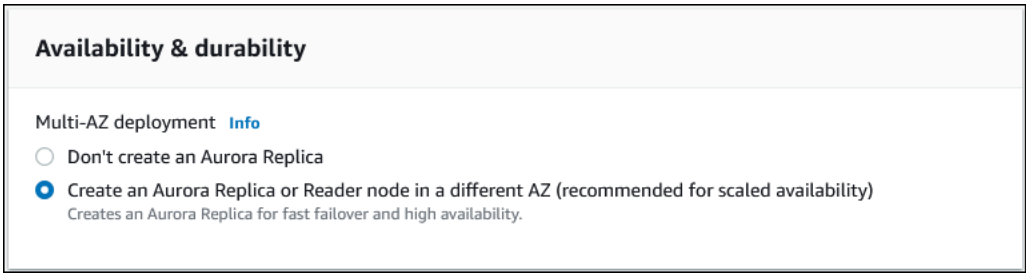
● Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

db.r5.large
2 vCPUs 16 GiB RAM Network: 4,750 Mbps

Include previous generation classes

5. For **Availability & durability**, we recommend that you choose to have Aurora create an Aurora Replica in a different Availability Zone (AZ) for you. If you don't create an Aurora Replica now, you need to do it later.



Availability & durability

Multi-AZ deployment Info

○ Don't create an Aurora Replica

● Create an Aurora Replica or Reader node in a different AZ (recommended for scaled availability)
Creates an Aurora Replica for fast failover and high availability.

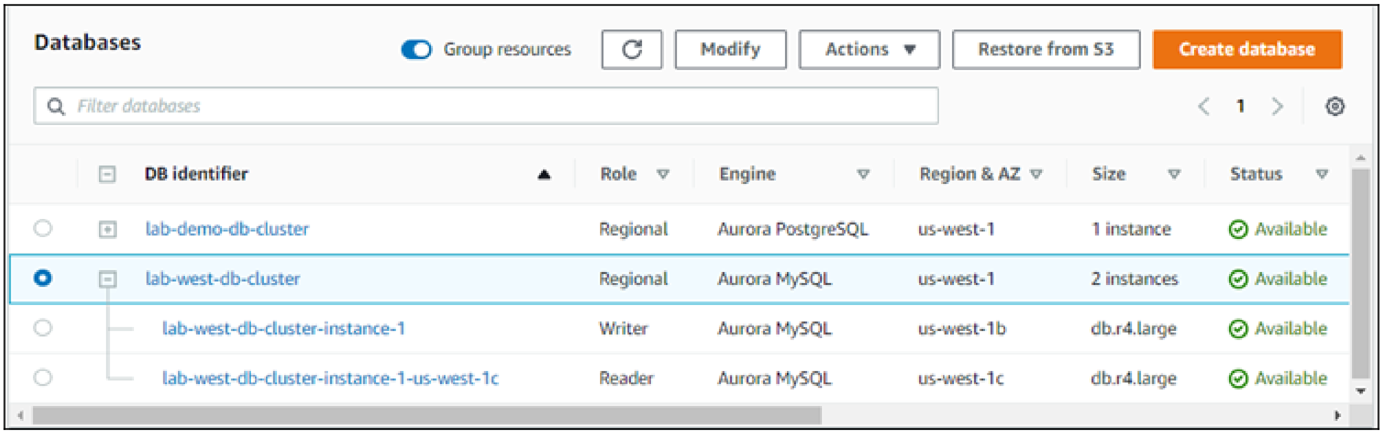
6. For **Connectivity**, choose the virtual private cloud (VPC) based on Amazon VPC that defines the virtual networking environment for this DB instance. You can choose the defaults to simplify this task.
7. Complete the **Database authentication** settings. To simplify the process, you can choose **Password authentication** now and set up AWS Identity and Access Management (IAM) later.
8. For **Additional configuration**, do the following:

   a. Enter a name for **Initial database name** to create the primary Aurora DB instance for this cluster. This is the writer node for the Aurora primary DB cluster.

      Leave the defaults selected for the DB cluster parameter group and DB parameter group, unless you have your own custom parameter groups that you want to use.

   b. Clear the **Enable backtrack** check box if it's selected. Aurora global databases don't support backtracking. Otherwise, accept the other default settings for **Additional configuration**.
9. Choose **Create database**.

   It can take several minutes for Aurora to complete the process of creating the Aurora DB instance, its Aurora Replica, and the Aurora DB cluster. You can tell when the Aurora DB cluster is ready to use as the primary DB cluster in an Aurora global database by its status. When that's so, its status and that of the writer and replica node is **Available**, as shown following.



	DB identifier	Role	Engine	Region & AZ	Size	Status
○	lab-demo-db-cluster	Regional	Aurora PostgreSQL	us-west-1	1 instance	✓ Available
●	lab-west-db-cluster	Regional	Aurora MySQL	us-west-1	2 instances	✓ Available
○	lab-west-db-cluster-instance-1	Writer	Aurora MySQL	us-west-1b	db.r4.large	✓ Available
○	lab-west-db-cluster-instance-1-us-west-1c	Reader	Aurora MySQL	us-west-1c	db.r4.large	✓ Available

When your primary DB cluster is available, create the Aurora global database by adding a secondary cluster to it. To do this, follow the steps in Adding an AWS Region to an Amazon Aurora global database.

Creating a global database using Aurora PostgreSQL

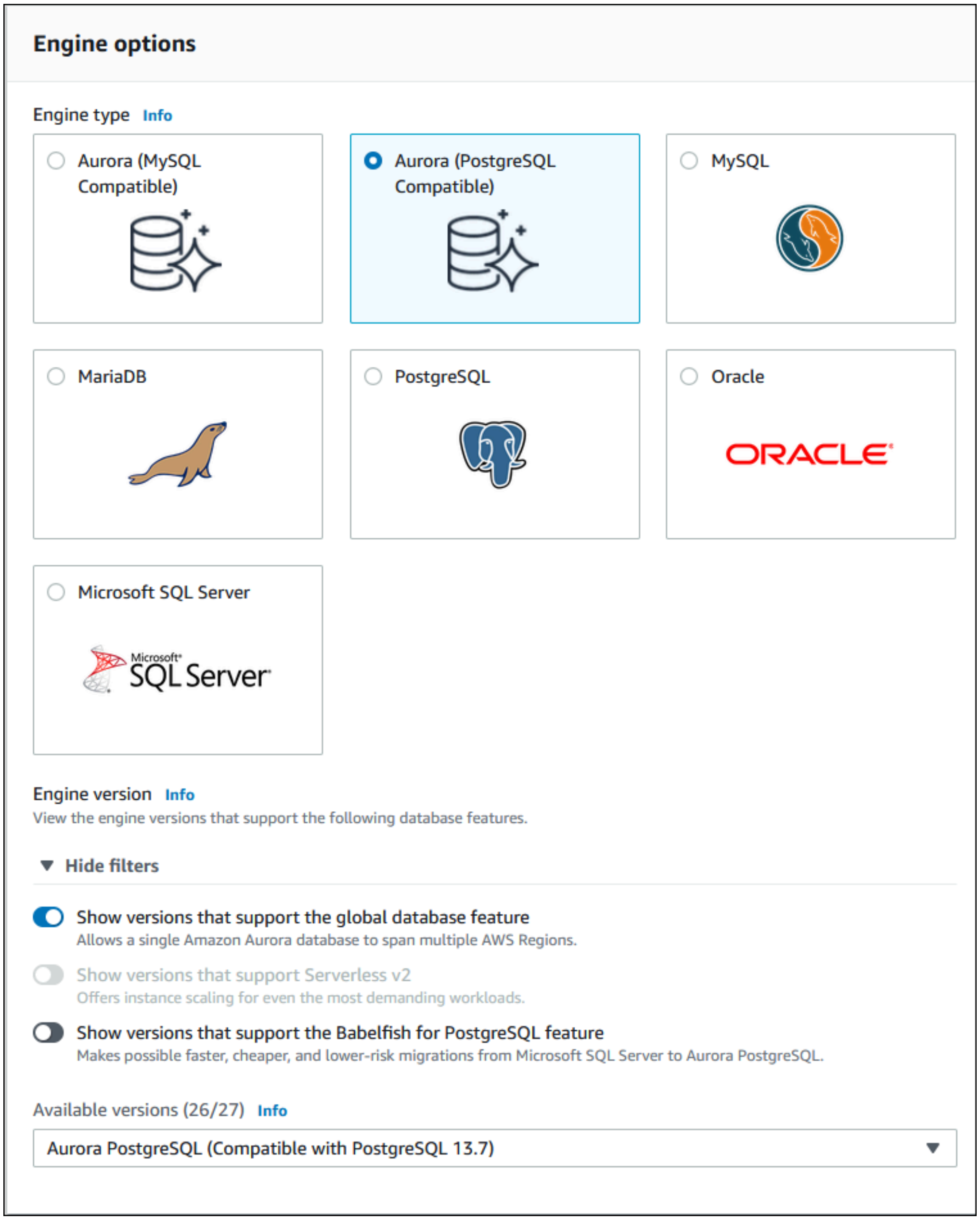
To create an Aurora global database using Aurora PostgreSQL

Complete the **Create database** page.

1. For **Engine options**, choose the following:

   a. Expand **Show filters**, and then turn on **Show versions that support the global database feature**.

   b. For **Engine version**, choose the version of Aurora PostgreSQL that you want to use for your Aurora global database.



Engine options

Engine type Info

○ Aurora (MySQL Compatible)

● Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

Engine version Info

View the engine versions that support the following database features.

▼ Hide filters

Show versions that support the global database feature (on)
Allows a single Amazon Aurora database to span multiple AWS Regions.

Show versions that support Serverless v2 (off, disabled)
Offers instance scaling for even the most demanding workloads.

Show versions that support the Babelfish for PostgreSQL feature (off)
Makes possible faster, cheaper, and lower-risk migrations from Microsoft SQL Server to Aurora PostgreSQL.

Available versions (26/27) Info

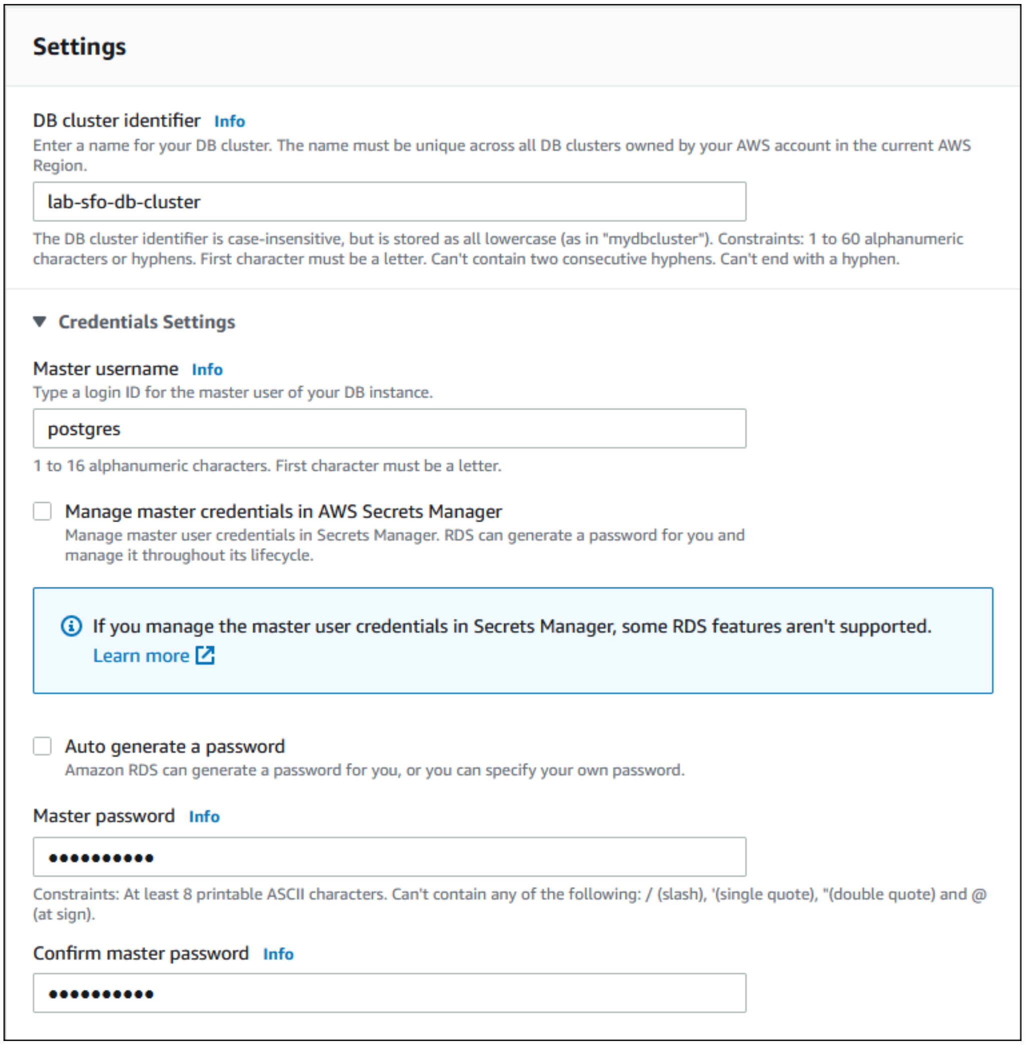
Aurora PostgreSQL (Compatible with PostgreSQL 13.7) ▼

2. For **Templates**, choose **Production**. Or you can choose Dev/Test if appropriate. Don't use Dev/Test in production environments.

3. For **Settings**, do the following:

   a. Enter a meaningful name for the DB cluster identifier. When you finish creating the Aurora global database, this name identifies the primary DB cluster.

   b. Enter your own password for the default admin account for the DB cluster, or have Aurora generate one for you. If you choose Auto generate a password, you get an option to copy the password.



Settings

DB cluster identifier Info

Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

lab-sfo-db-cluster

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB instance.

postgres

1 to 16 alphanumeric characters. First character must be a letter.

□ Manage master credentials in AWS Secrets Manager

Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

ⓘ If you manage the master user credentials in Secrets Manager, some RDS features aren't supported.
Learn more

□ Auto generate a password

Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info

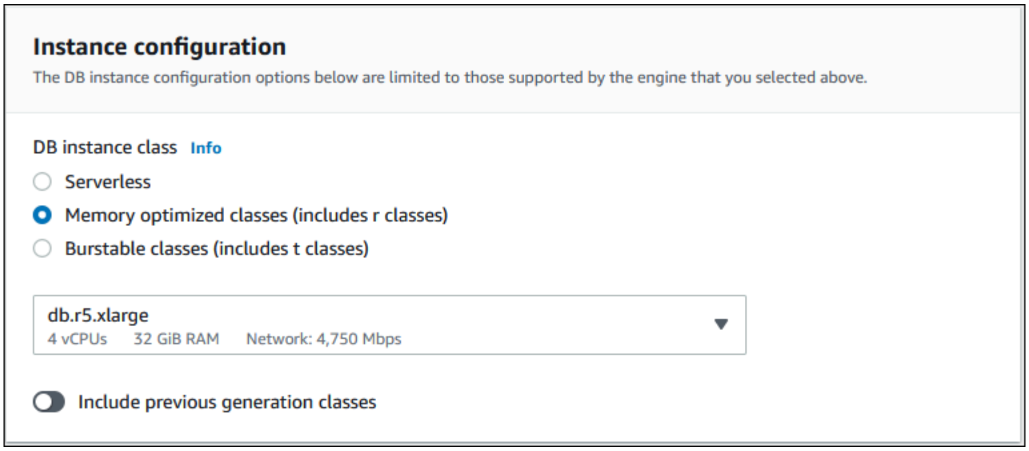
••••••••••

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password Info

••••••••••

4. For **DB instance class**, choose `db.r5.large` or another memory optimized DB instance class. We recommend that you use a db.r5 or higher instance class.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

○ Serverless

● Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

db.r5.xlarge
4 vCPUs 32 GiB RAM Network: 4,750 Mbps

Include previous generation classes

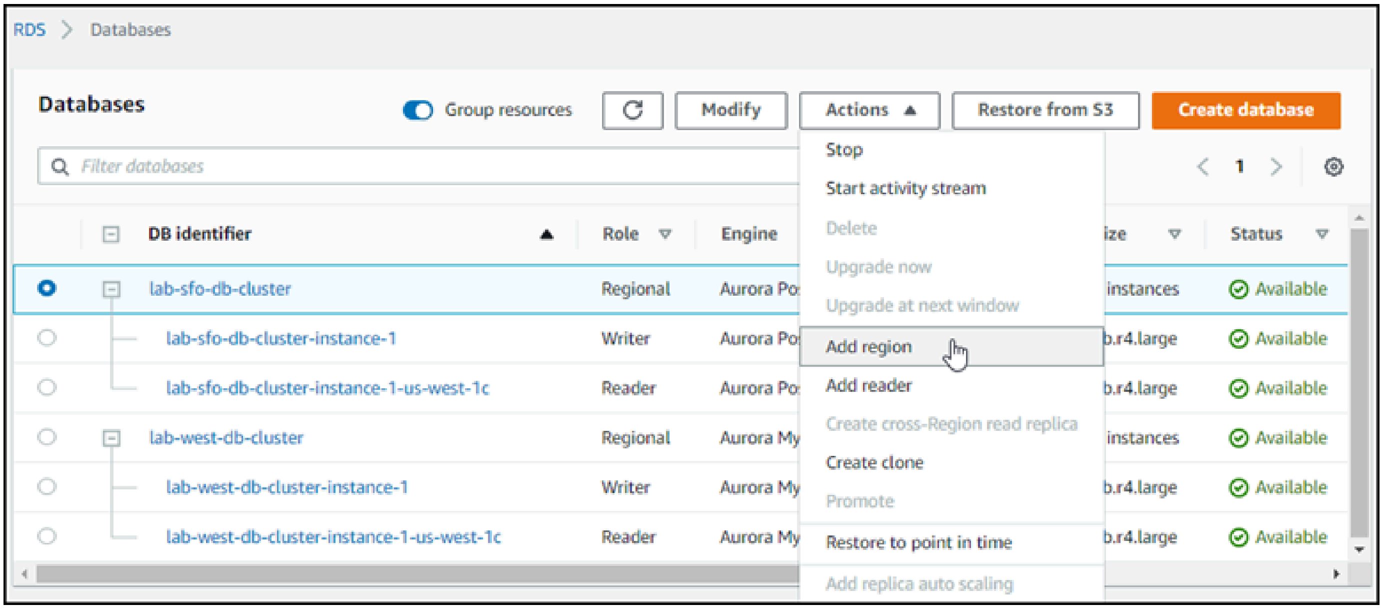
5. For **Availability & durability**, we recommend that you choose to have Aurora create an Aurora Replica in a different AZ for you. If you don't create an Aurora Replica now, you need to do it later.
6. For **Connectivity**, choose the virtual private cloud (VPC) based on Amazon VPC that defines the virtual networking environment for this DB instance. You can choose the defaults to simplify this task.
7. (Optional) Complete the **Database authentication** settings. Password authentication is always enabled. To simplify the process, you can skip this section and set up IAM or password and Kerberos authentication later.
8. For **Additional configuration**, do the following:

   a. Enter a name for **Initial database name** to create the primary Aurora DB instance for this cluster. This is the writer node for the Aurora primary DB cluster.

      Leave the defaults selected for the DB cluster parameter group and DB parameter group, unless you have your own custom parameter groups that you want to use.

   b. Accept all other default settings for **Additional configuration**, such as Encryption, Log exports, and so on.
9. Choose **Create database**.

   It can take several minutes for Aurora to complete the process of creating the Aurora DB instance, its Aurora Replica, and the Aurora DB cluster. When the cluster is ready to use, the Aurora DB cluster and its writer and replica nodes display **Available** status. This becomes the primary DB cluster of your Aurora global database, after you add a secondary.



	DB identifier ▲	Role	Engine	ize	Status
●	lab-sfo-db-cluster	Regional	Aurora Po	instances	Available
○	lab-sfo-db-cluster-instance-1	Writer	Aurora Po	b.r4.large	Available
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora Po	b.r4.large	Available
○	lab-west-db-cluster	Regional	Aurora My	instances	Available
○	lab-west-db-cluster-instance-1	Writer	Aurora My	b.r4.large	Available
○	lab-west-db-cluster-instance-1-us-west-1c	Reader	Aurora My	b.r4.large	Available

When your primary DB cluster is available, create one or more secondary clusters by following the steps in Adding an AWS Region to an Amazon Aurora global database.

AWS CLI

The AWS CLI commands in the procedures following accomplish the following tasks:

1. Create an Aurora global database, giving it a name and specifying the Aurora database engine type that you plan to use.
2. Create an Aurora DB cluster for the Aurora global database.
3. Create the Aurora DB instance for the cluster. This is the primary Aurora DB cluster for the global database.
4. Create a second DB instance for Aurora DB cluster. This is a reader to complete the Aurora DB cluster.
5. Create a second Aurora DB cluster in another Region and then add it to your Aurora global database, by following the steps in Adding an AWS Region to an Amazon Aurora global database.

Follow the procedure for your Aurora database engine.

Creating a global database using Aurora MySQL

To create an Aurora global database using Aurora MySQL

1. Use the create-global-cluster CLI command, passing the name of the AWS Region, Aurora database engine, and version.

   For Linux, macOS, or Unix:

   ```
   aws rds create-global-cluster --region primary_region  \
       --global-cluster-identifier global_database_id \
       --engine aurora-mysql \
       --engine-version version # optional
   ```

   For Windows:

   ```
   aws rds create-global-cluster ^
       --global-cluster-identifier global_database_id ^
       --engine aurora-mysql ^
       --engine-version version # optional
   ```

   This creates an "empty" Aurora global database, with just a name (identifier) and Aurora database engine. It can take a few minutes for the Aurora global database to be available. Before going to the next step, use the describe-global-clusters CLI command to see if it's available.

   ```
   aws rds describe-global-clusters --region primary_region --global-cluster-
   identifier global_database_id
   ```

   When the Aurora global database is available, you can create its primary Aurora DB cluster.

2. To create a primary Aurora DB cluster, use the create-db-cluster CLI command. Include the name of your Aurora global database by using the `--global-cluster-identifier` parameter.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster \
     --region primary_region \
     --db-cluster-identifier primary_db_cluster_id \
   ```

```
  --master-username userid \
  --master-user-password password \
  --engine aurora-mysql \
  --engine-version version \
  --global-cluster-identifier global_database_id
```

For Windows:

```
aws rds create-db-cluster ^
  --region primary_region ^
  --db-cluster-identifier primary_db_cluster_id ^
  --master-username userid ^
  --master-user-password password ^
  --engine aurora-mysql ^
  --engine-version version ^
  --global-cluster-identifier global_database_id
```

Use the `describe-db-clusters` AWS CLI command to confirm that the Aurora DB cluster is ready. To single out a specific Aurora DB cluster, use `--db-cluster-identifier` parameter. Or you can leave out the Aurora DB cluster name in the command to get details about all your Aurora DB clusters in the given Region.

```
aws rds describe-db-clusters --region primary_region --db-cluster-
identifier primary_db_cluster_id
```

When the response shows `"Status": "available"` for the cluster, it's ready to use.

3. Create the DB instance for your primary Aurora DB cluster. To do so, use the `create-db-instance` CLI command. Give the command your Aurora DB cluster's name, and specify the configuration details for the instance. You don't need to pass the `--master-username` and `--master-user-password` parameters in the command, because it gets those from the Aurora DB cluster.

   For the `--db-instance-class`, you can use only those from the memory optimized classes, such as `db.r5.large`. We recommend that you use a db.r5 or higher instance class. For information about these classes, see DB instance classes.

   For Linux, macOS, or Unix:

```
aws rds create-db-instance \
  --db-cluster-identifier primary_db_cluster_id \
  --db-instance-class instance_class \
  --db-instance-identifier db_instance_id \
  --engine aurora-mysql \
  --engine-version version \
  --region primary_region
```

For Windows:

```
aws rds create-db-instance ^
  --db-cluster-identifier primary_db_cluster_id ^
  --db-instance-class instance_class ^
  --db-instance-identifier db_instance_id ^
  --engine aurora-mysql ^
  --engine-version version ^
  --region primary_region
```

The `create-db-instance` operation might take some time to complete. Check the status to see if the Aurora DB instance is available before continuing.

```
aws rds describe-db-clusters --db-cluster-identifier primary_db_cluster_id
```

When the command returns a status of "available," you can create another Aurora DB instance for your primary DB cluster. This is the reader instance (the Aurora Replica) for the Aurora DB cluster.

4. To create another Aurora DB instance for the cluster, use the [`create-db-instance`] CLI command.

   For Linux, macOS, or Unix:

```
aws rds create-db-instance \
  --db-cluster-identifier primary_db_cluster_id \
  --db-instance-class instance_class \
  --db-instance-identifier replica_db_instance_id \
  --engine aurora-mysql
```

   For Windows:

```
aws rds create-db-instance ^
  --db-cluster-identifier primary_db_cluster_id ^
```

```
    --db-instance-class instance_class ^
    --db-instance-identifier replica_db_instance_id ^
    --engine aurora-mysql
```

When the DB instance is available, replication begins from the writer node to the replica. Before continuing, check that the DB instance is available with the `describe-db-instances` CLI command.

At this point, you have an Aurora global database with its primary Aurora DB cluster containing a writer DB instance and an Aurora Replica. You can now add a read-only Aurora DB cluster in a different Region to complete your Aurora global database. To do so, follow the steps in Adding an AWS Region to an Amazon Aurora global database.

Creating a global database using Aurora PostgreSQL

When you create Aurora objects for an Aurora global database by using the following commands, it can take a few minutes for each to become available. We recommend that after completing any given command, you check the specific Aurora object's status to make sure that the status is available.

To do so, use the `describe-global-clusters` CLI command.

```
aws rds describe-global-clusters --region primary_region
    --global-cluster-identifier global_database_id
```

To create an Aurora global database using Aurora PostgreSQL

1. Use the `create-global-cluster` CLI command.

   For Linux, macOS, or Unix:

   ```
   aws rds create-global-cluster --region primary_region  \
       --global-cluster-identifier global_database_id \
       --engine aurora-postgresql \
       --engine-version version # optional
   ```

   For Windows:

   ```
   aws rds create-global-cluster ^
       --global-cluster-identifier global_database_id ^
   ```

```
    --engine aurora-postgresql ^
    --engine-version version # optional
```

When the Aurora global database is available, you can create its primary Aurora DB cluster.

2. To create a primary Aurora DB cluster, use the `create-db-cluster` CLI command. Include the name of your Aurora global database by using the `--global-cluster-identifier` parameter.

   For Linux, macOS, or Unix:

   ```
   aws rds create-db-cluster \
     --region primary_region \
     --db-cluster-identifier primary_db_cluster_id \
     --master-username userid \
     --master-user-password password \
     --engine aurora-postgresql \
     --engine-version version \
     --global-cluster-identifier global_database_id
   ```

   For Windows:

   ```
   aws rds create-db-cluster ^
     --region primary_region ^
     --db-cluster-identifier primary_db_cluster_id ^
     --master-username userid ^
     --master-user-password password ^
     --engine aurora-postgresql ^
     --engine-version version ^
     --global-cluster-identifier global_database_id
   ```

   Check that the Aurora DB cluster is ready. When the response from the following command shows `"Status": "available"` for the Aurora DB cluster, you can continue.

   ```
   aws rds describe-db-clusters --region primary_region --db-cluster-
   identifier primary_db_cluster_id
   ```

3. Create the DB instance for your primary Aurora DB cluster. To do so, use the `create-db-instance` CLI command.

   Pass the name of your Aurora DB cluster with the `--db-cluster-identifier` parameter.

You don't need to pass the `--master-username` and `--master-user-password` parameters in the command, because it gets those from the Aurora DB cluster.

For the `--db-instance-class`, you can use only those from the memory optimized classes, such as `db.r5.large`. We recommend that you use a db.r5 or higher instance class. For information about these classes, see DB instance classes.

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
  --db-cluster-identifier primary_db_cluster_id \
  --db-instance-class instance_class \
  --db-instance-identifier db_instance_id \
  --engine aurora-postgresql \
  --engine-version version \
  --region primary_region
```

For Windows:

```
aws rds create-db-instance ^
  --db-cluster-identifier primary_db_cluster_id ^
  --db-instance-class instance_class ^
  --db-instance-identifier db_instance_id ^
  --engine aurora-postgresql ^
  --engine-version version ^
  --region primary_region
```

4. Check the status of the Aurora DB instance before continuing.

```
aws rds describe-db-clusters --db-cluster-identifier primary_db_cluster_id
```

If the response shows that Aurora DB instance status is "available," you can create another Aurora DB instance for your primary DB cluster.

5. To create an Aurora Replica for Aurora DB cluster, use the `create-db-instance` CLI command.

For Linux, macOS, or Unix:

```
aws rds create-db-instance \
  --db-cluster-identifier primary_db_cluster_id \
  --db-instance-class instance_class \
  --db-instance-identifier replica_db_instance_id \
  --engine aurora-postgresql
```

For Windows:

```
aws rds create-db-instance ^
  --db-cluster-identifier primary_db_cluster_id ^
  --db-instance-class instance_class ^
  --db-instance-identifier replica_db_instance_id ^
  --engine aurora-postgresql
```

When the DB instance is available, replication begins from the writer node to the replica. Before continuing, check that the DB instance is available with the `describe-db-instances` CLI command.

Your Aurora global database exists, but it has only its primary Region with an Aurora DB cluster made up of a writer DB instance and an Aurora Replica. You can now add a read-only Aurora DB cluster in a different Region to complete your Aurora global database. To do so, follow the steps in Adding an AWS Region to an Amazon Aurora global database.

RDS API

To create an Aurora global database with the RDS API, run the CreateGlobalCluster operation.

Adding an AWS Region to an Amazon Aurora global database

An Aurora global database needs at least one secondary Aurora DB cluster in a different AWS Region than the primary Aurora DB cluster. You can attach up to five secondary DB clusters to your Aurora global database. For each secondary DB cluster that you add to your Aurora global database, reduce the number of Aurora Replicas allowed to the primary DB cluster by one.

For example, if your Aurora global database has 5 secondary Regions, your primary DB cluster can have only 10 (rather than 15) Aurora Replicas. For more information, see Configuration requirements of an Amazon Aurora global database.

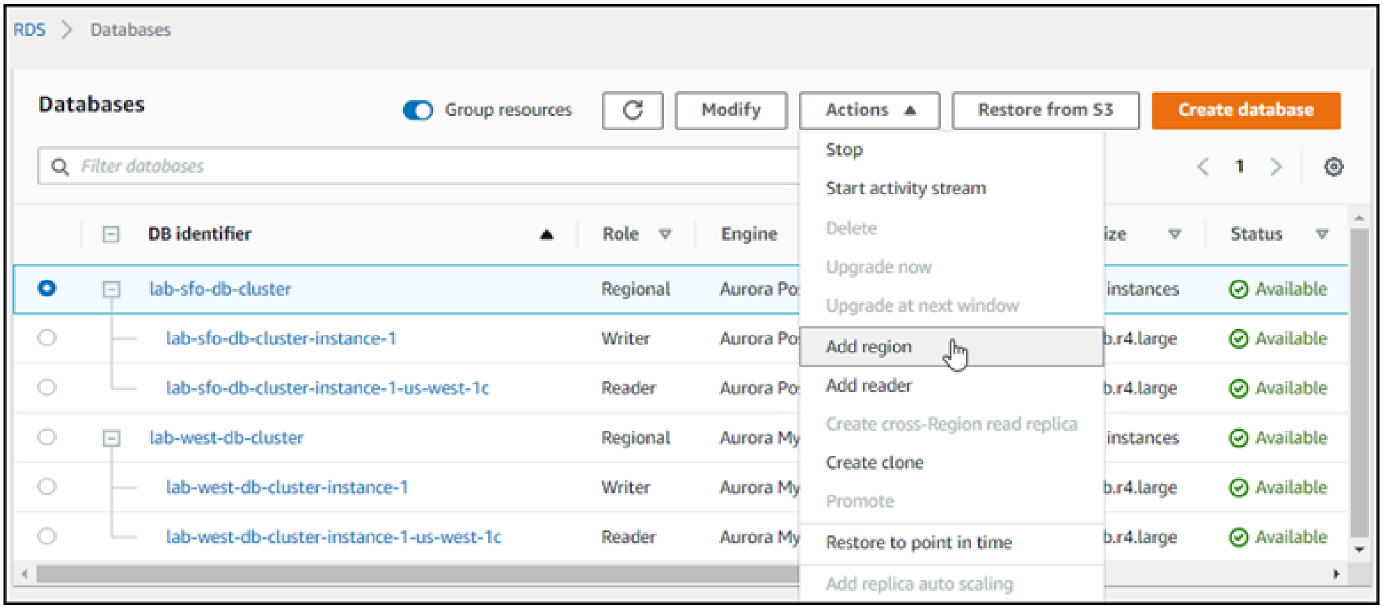
The number of Aurora Replicas (reader instances) in the primary DB cluster determines the number of secondary DB clusters you can add. The total number of reader instances in the primary DB cluster plus the number of secondary clusters can't exceed 15. For example, if you have 14 reader instances in the primary DB cluster and 1 secondary cluster, you can't add another secondary cluster to the global database.

> **Note**
>
> For Aurora MySQL version 3, when you create a secondary cluster, make sure that the value of `lower_case_table_names` matches the value in the primary cluster. This setting is a database parameter that affects how the server handles identifier case sensitivity. For more information about database parameters, see Working with parameter groups.
>
> We recommend that when you create a secondary cluster, you use the same DB engine version for the primary and secondary. If necessary, upgrade the primary to be the same version as the secondary. For more information, see Patch level compatibility for managed cross-Region switchovers and failovers.

Console

To add an AWS Region to an Aurora global database

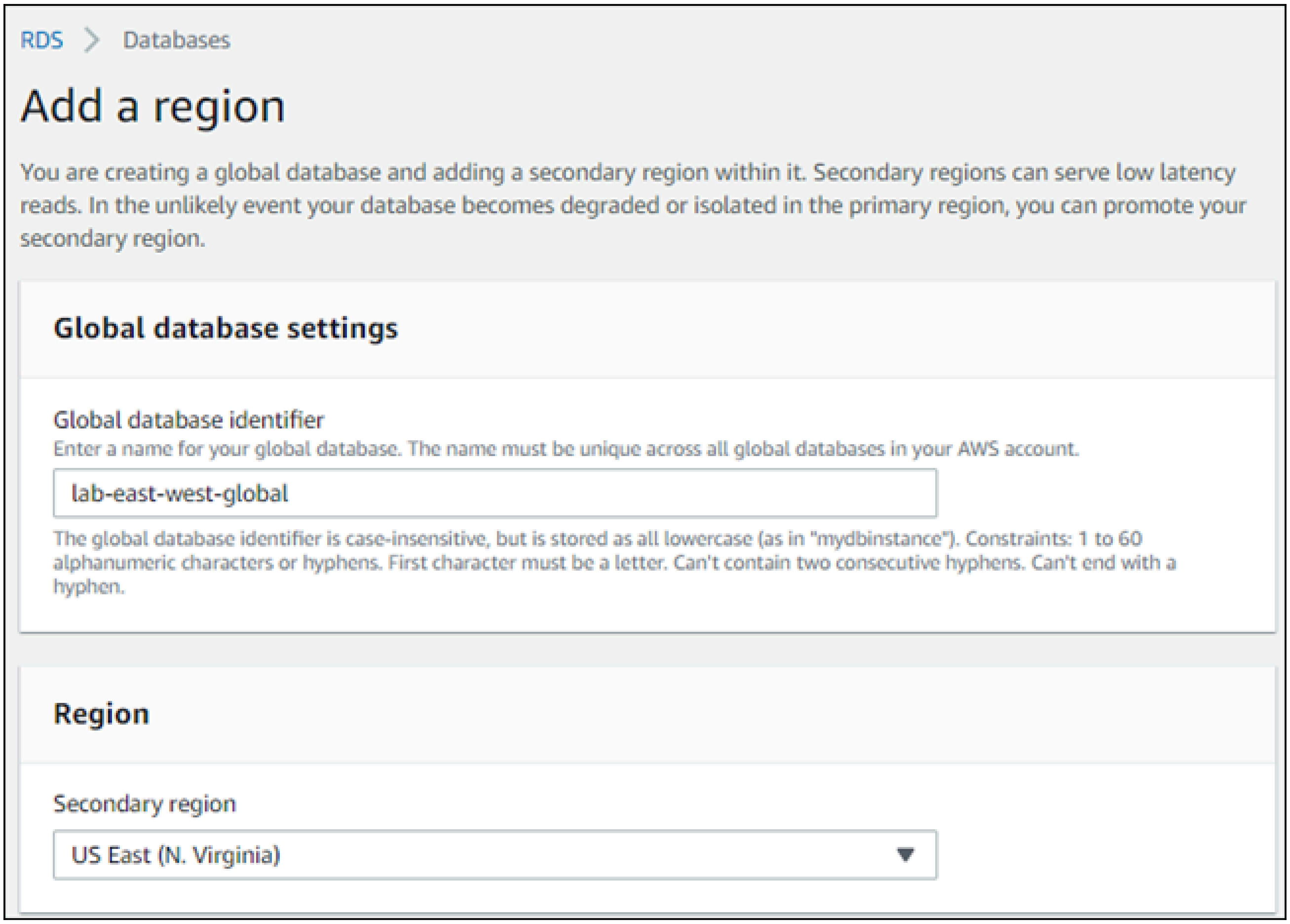
1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane of the AWS Management Console, choose **Databases**.
3. Choose the Aurora global database that needs a secondary Aurora DB cluster. Ensure that the primary Aurora DB cluster is `Available`.
4. For **Actions**, choose **Add region**.



	DB identifier ▲	Role	Engine	ize	Status
●	lab-sfo-db-cluster	Regional	Aurora Po	instances	Available
○	lab-sfo-db-cluster-instance-1	Writer	Aurora Po	b.r4.large	Available
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora Po	b.r4.large	Available
○	lab-west-db-cluster	Regional	Aurora My	instances	Available
○	lab-west-db-cluster-instance-1	Writer	Aurora My	b.r4.large	Available
○	lab-west-db-cluster-instance-1-us-west-1c	Reader	Aurora My	b.r4.large	Available

5. On the **Add a region** page, choose the secondary AWS Region.

   You can't choose an AWS Region that already has a secondary Aurora DB cluster for the same Aurora global database. Also, it can't be the same Region as the primary Aurora DB cluster.



Add a region

You are creating a global database and adding a secondary region within it. Secondary regions can serve low latency reads. In the unlikely event your database becomes degraded or isolated in the primary region, you can promote your secondary region.

Global database settings

Global database identifier
Enter a name for your global database. The name must be unique across all global databases in your AWS account.

lab-east-west-global

The global database identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

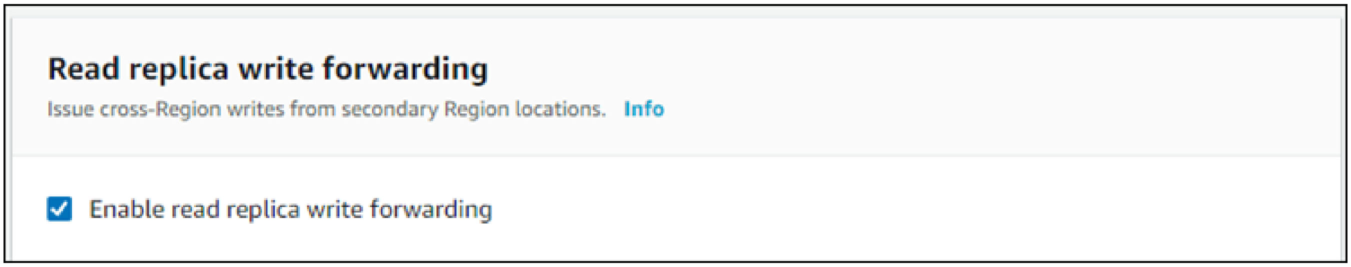
Region

Secondary region

US East (N. Virginia) ▼

6. Complete the remaining fields for the secondary Aurora cluster in the new AWS Region. These are the same configuration options as for any Aurora DB cluster instance, except for the following option for Aurora MySQL–based Aurora global databases only:

   - Enable read replica write forwarding – This optional setting let's your Aurora global database's secondary DB clusters forward write operations to the primary cluster. For more information, see [Using write forwarding in an Amazon Aurora global database](#).



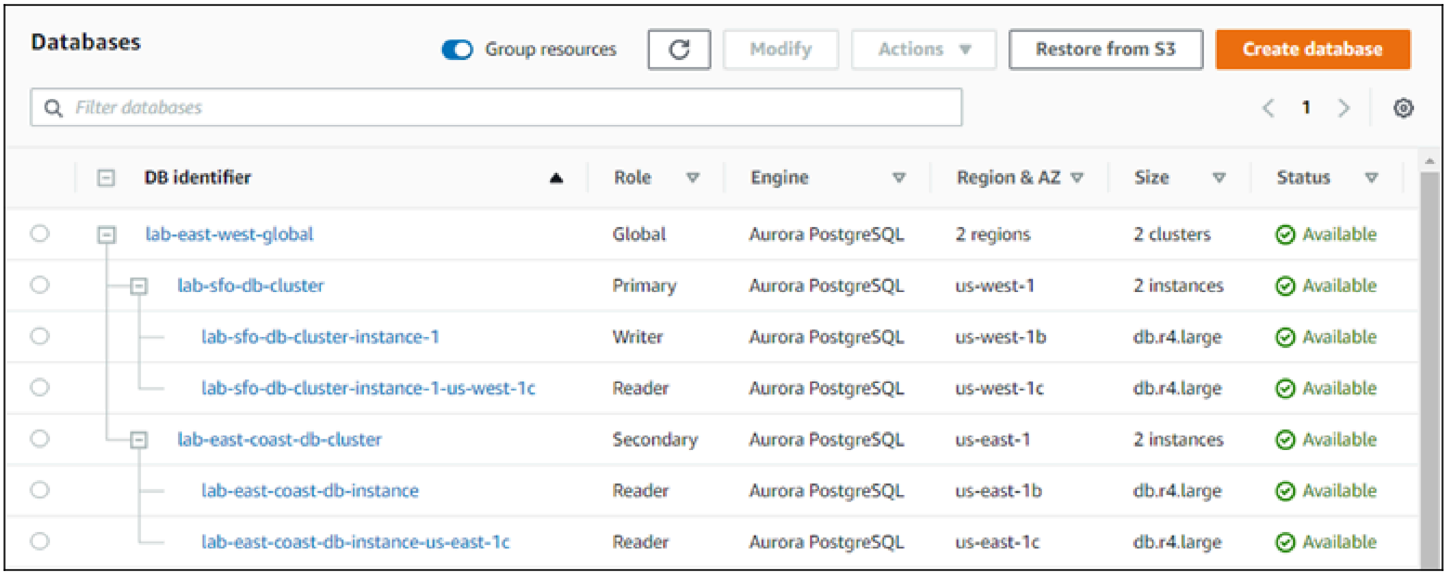
Read replica write forwarding

Issue cross-Region writes from secondary Region locations. Info

☑ Enable read replica write forwarding

7. Choose **Add region**.

After you finish adding the Region to your Aurora global database, you can see it in the list of **Databases** in the AWS Management Console as shown in the screenshot.



DB identifier	Role	Engine	Region & AZ	Size	Status
lab-east-west-global	Global	Aurora PostgreSQL	2 regions	2 clusters	Available
lab-sfo-db-cluster	Primary	Aurora PostgreSQL	us-west-1	2 instances	Available
lab-sfo-db-cluster-instance-1	Writer	Aurora PostgreSQL	us-west-1b	db.r4.large	Available
lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora PostgreSQL	us-west-1c	db.r4.large	Available
lab-east-coast-db-cluster	Secondary	Aurora PostgreSQL	us-east-1	2 instances	Available
lab-east-coast-db-instance	Reader	Aurora PostgreSQL	us-east-1b	db.r4.large	Available
lab-east-coast-db-instance-us-east-1c	Reader	Aurora PostgreSQL	us-east-1c	db.r4.large	Available

AWS CLI

To add a secondary AWS Region to an Aurora global database

1. Use the [`create-db-cluster`](#) CLI command with the name (`--global-cluster-identifier`) of your Aurora global database. For other parameters, do the following:

2. For `--region`, choose a different AWS Region than that of your Aurora primary Region.

3. Choose specific values for the `--engine` and `--engine-version` parameters. These values are the same as those for the primary Aurora DB cluster in your Aurora global database.
4. For an encrypted cluster, specify your primary AWS Region as the `--source-region` for encryption.

The following example creates a new Aurora DB cluster and attaches it to an Aurora global database as a read-only secondary Aurora DB cluster. In the last step, an Aurora DB instance is added to the new Aurora DB cluster.

For Linux, macOS, or Unix:

```
aws rds --region secondary_region \
  create-db-cluster \
    --db-cluster-identifier secondary_cluster_id \
    --global-cluster-identifier global_database_id \
    --engine aurora-mysql|aurora-postgresql
    --engine-version version

aws rds --region secondary_region \
  create-db-instance \
    --db-instance-class instance_class \
    --db-cluster-identifier secondary_cluster_id \
    --db-instance-identifier db_instance_id \
    --engine aurora-mysql|aurora-postgresql
```

For Windows:

```
aws rds --region secondary_region ^
  create-db-cluster ^
    --db-cluster-identifier secondary_cluster_id ^
    --global-cluster-identifier global_database_id_id ^
    --engine aurora-mysql|aurora-postgresql ^
    --engine-version version

aws rds --region secondary_region ^
  create-db-instance ^
    --db-instance-class instance_class ^
    --db-cluster-identifier secondary_cluster_id ^
    --db-instance-identifier db_instance_id ^
    --engine aurora-mysql|aurora-postgresql
```

RDS API

To add a new AWS Region to an Aurora global database with the RDS API, run the CreateDBCluster operation. Specify the identifier of the existing global database using the `GlobalClusterIdentifier` parameter.

Creating a headless Aurora DB cluster in a secondary Region

Although an Aurora global database requires at least one secondary Aurora DB cluster in a different AWS Region than the primary, you can use a *headless* configuration for the secondary cluster. A headless secondary Aurora DB cluster is one without a DB instance. This type of configuration can lower expenses for an Aurora global database. In an Aurora DB cluster, compute and storage are decoupled. Without the DB instance, you're not charged for compute, only for storage. If it's set up correctly, a headless secondary's storage volume is kept in-sync with the primary Aurora DB cluster.

You add the secondary cluster as you normally do when creating an Aurora global database. However, after the primary Aurora DB cluster begins replication to the secondary, you delete the Aurora read-only DB instance from the secondary Aurora DB cluster. This secondary cluster is now considered "headless" because it no longer has a DB instance. Yet, the storage volume is kept in sync with the primary Aurora DB cluster.

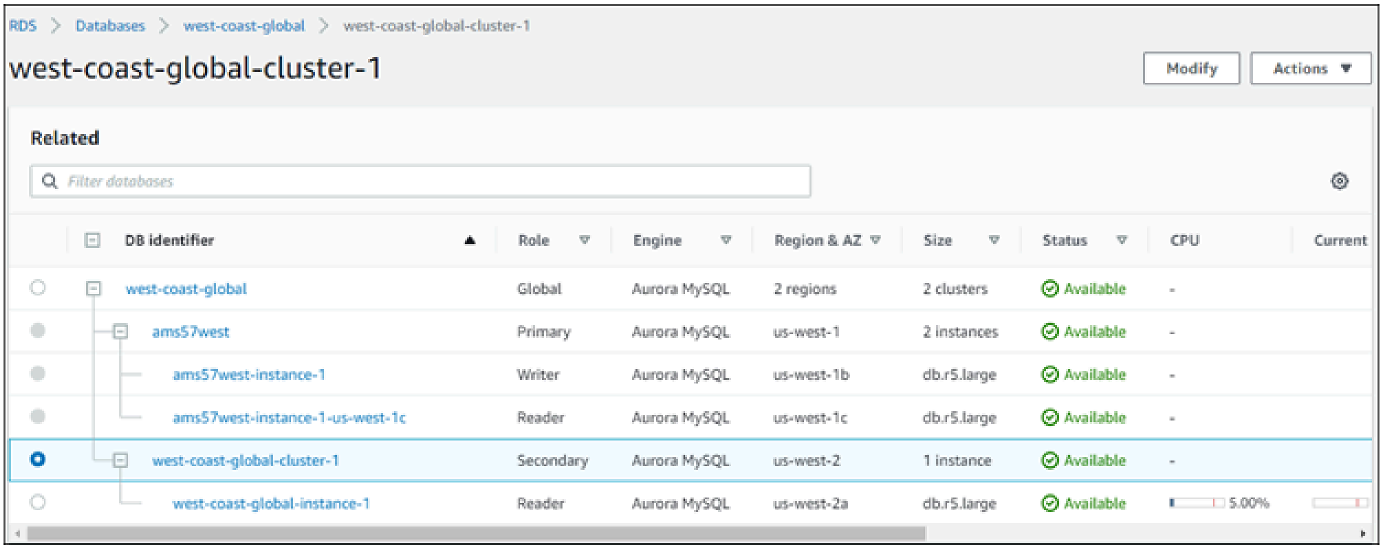
> ⚠ **Warning**
>
> With Aurora PostgreSQL, to create a headless cluster in a secondary AWS Region, use the AWS CLI or RDS API to add the secondary AWS Region. Skip the step to create the reader DB instance for the secondary cluster. Currently, creating a headless cluster isn't supported in the RDS Console. For the CLI and API procedures to use, see Adding an AWS Region to an Amazon Aurora global database.
> If your global database is using an engine version lower than 13.4, 12.8, or 11.13, creating a reader DB instance in a secondary Region and subsequently deleting it could lead to an Aurora PostgreSQL vacuum issue on the primary Region's writer DB instance. If you encounter this issue, restart the primary Region's writer DB instance after you delete the secondary Region's reader DB instance.

To add a headless secondary Aurora DB cluster to your Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane of the AWS Management Console, choose **Databases**.
3. Choose the Aurora global database that needs a secondary Aurora DB cluster. Ensure that the primary Aurora DB cluster is `Available`.
4. For **Actions**, choose **Add region**.
5. On the **Add a region** page, choose the secondary AWS Region.

   You can't choose an AWS Region that already has a secondary Aurora DB cluster for the same Aurora global database. Also, it can't be the same Region as the primary Aurora DB cluster.
6. Complete the remaining fields for the secondary Aurora cluster in the new AWS Region. These are the same configuration options as for any Aurora DB cluster instance.

   For an Aurora MySQL–based Aurora global database, disregard the **Enable read replica write forwarding** option. This option has no function after you delete the reader instance.
7. Choose **Add region**. After you finish adding the Region to your Aurora global database, you can see it in the list of Databases in the AWS Management Console as shown in the screenshot.



RDS > Databases > west-coast-global > west-coast-global-cluster-1

west-coast-global-cluster-1

Related

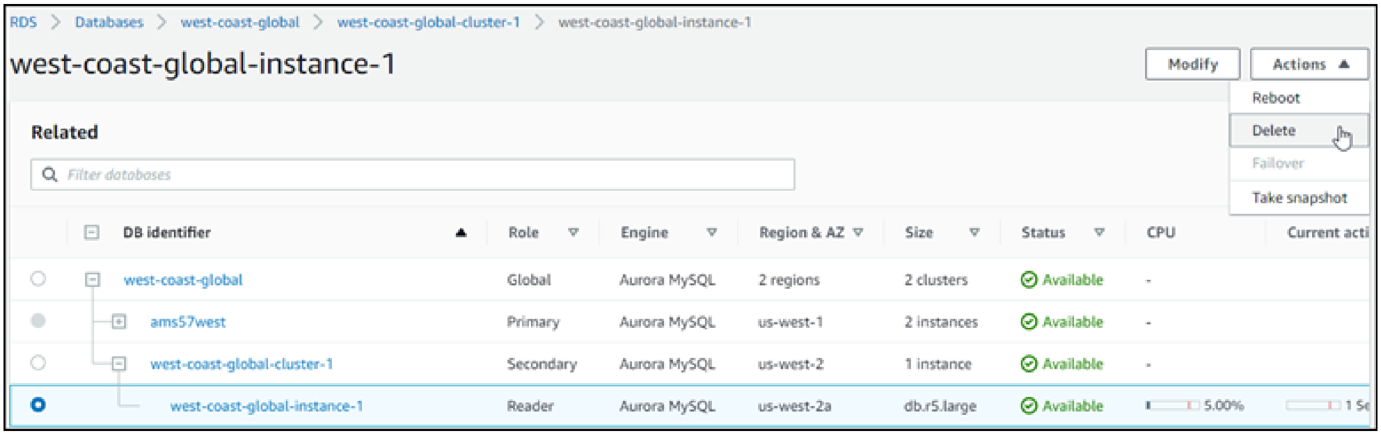
	DB identifier	Role	Engine	Region & AZ	Size	Status	CPU	Current
○	west-coast-global	Global	Aurora MySQL	2 regions	2 clusters	✓ Available	-	
●	ams57west	Primary	Aurora MySQL	us-west-1	2 instances	✓ Available	-	
●	ams57west-instance-1	Writer	Aurora MySQL	us-west-1b	db.r5.large	✓ Available	-	
●	ams57west-instance-1-us-west-1c	Reader	Aurora MySQL	us-west-1c	db.r5.large	✓ Available	-	
●	west-coast-global-cluster-1	Secondary	Aurora MySQL	us-west-2	1 instance	✓ Available	-	
○	west-coast-global-instance-1	Reader	Aurora MySQL	us-west-2a	db.r5.large	✓ Available	5.00%	

8. Check the status of the secondary Aurora DB cluster and its reader instance before continuing, by using the AWS Management Console or the AWS CLI. For example:

```
$ aws rds describe-db-clusters --db-cluster-identifier secondary-cluster-id --query
 '*[].[Status]' --output text
```

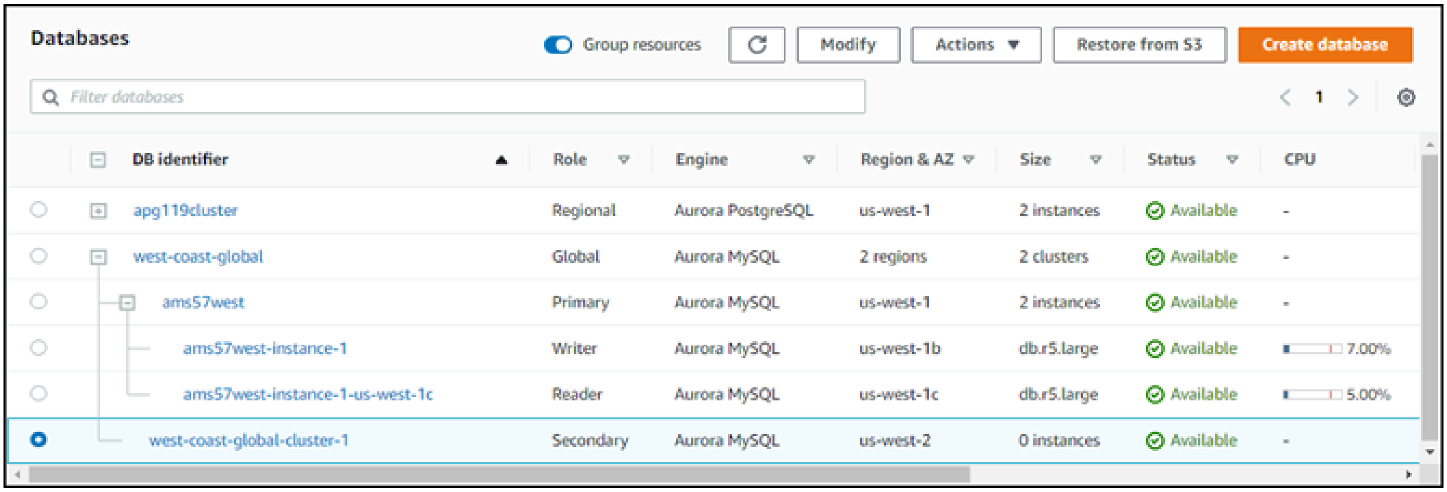
It can take several minutes for the status of a newly added secondary Aurora DB cluster to change from `creating` to `available`. When the Aurora DB cluster is available, you can delete the reader instance.

9. Select the reader instance in the secondary Aurora DB cluster, and then choose **Delete**.


RDS > Databases > west-coast-global > west-coast-global-cluster-1 > west-coast-global-instance-1
west-coast-global-instance-1
Modify
Actions ▲
Reboot
Delete
Failover
Take snapshot
Related
Filter databases

	DB identifier ▲	Role	Engine	Region & AZ	Size	Status	CPU	Current acti
○	west-coast-global	Global	Aurora MySQL	2 regions	2 clusters	Available	-	
●	ams57west	Primary	Aurora MySQL	us-west-1	2 instances	Available	-	
○	west-coast-global-cluster-1	Secondary	Aurora MySQL	us-west-2	1 instance	Available	-	
●	west-coast-global-instance-1	Reader	Aurora MySQL	us-west-2a	db.r5.large	Available	5.00%	1 Se

After deleting the reader instance, the secondary cluster remains part of the Aurora global database. It has no instance associated with it, as shown following.



	DB identifier ▲	Role	Engine	Region & AZ	Size	Status	CPU
○	apg119cluster	Regional	Aurora PostgreSQL	us-west-1	2 instances	Available	-
○	west-coast-global	Global	Aurora MySQL	2 regions	2 clusters	Available	-
○	ams57west	Primary	Aurora MySQL	us-west-1	2 instances	Available	-
○	ams57west-instance-1	Writer	Aurora MySQL	us-west-1b	db.r5.large	Available	7.00%
○	ams57west-instance-1-us-west-1c	Reader	Aurora MySQL	us-west-1c	db.r5.large	Available	5.00%
●	west-coast-global-cluster-1	Secondary	Aurora MySQL	us-west-2	0 instances	Available	-

You can use this headless secondary Aurora DB cluster to manually recover your Amazon Aurora global database from an unplanned outage in the primary AWS Region if such an outage occurs.

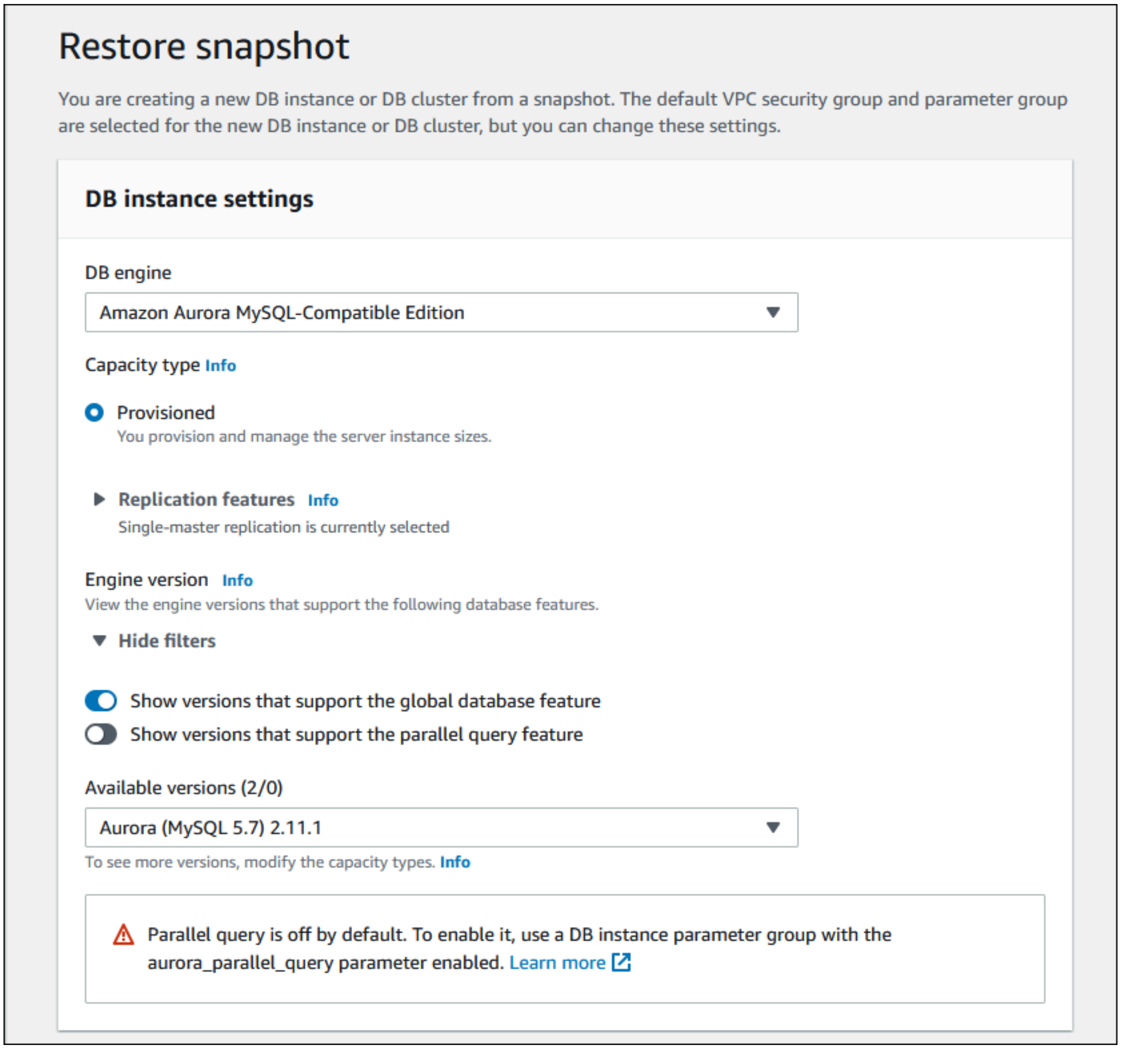
Using a snapshot for your Amazon Aurora global database

You can restore a snapshot of an Aurora DB cluster or from an Amazon RDS DB instance to use as the starting point for your Aurora global database. You restore the snapshot and create a new Aurora provisioned DB cluster at the same time. You then add another AWS Region to the restored DB cluster, thus turning it into an Aurora global database. Any Aurora DB cluster that you create using a snapshot in this way becomes the primary cluster of your Aurora global database.

The snapshot that you use can be from a `provisioned` or from a `serverless` Aurora DB cluster.

During the restore process, choose the same DB engine type as the snapshot. For example, suppose that you want to restore a snapshot that was made from an Aurora Serverless DB cluster running Aurora PostgreSQL. In this case, you create an Aurora PostgreSQL DB cluster using that same Aurora DB engine and version.

The restored DB cluster assumes the role of primary cluster for the Aurora global database when you add an AWS Region to it. All data contained in this primary cluster is replicated to any secondary clusters that you add to your Aurora global database.



Restore snapshot

You are creating a new DB instance or DB cluster from a snapshot. The default VPC security group and parameter group are selected for the new DB instance or DB cluster, but you can change these settings.

DB instance settings

DB engine

Amazon Aurora MySQL-Compatible Edition

Capacity type Info

● Provisioned
You provision and manage the server instance sizes.

▶ Replication features Info
Single-master replication is currently selected

Engine version Info
View the engine versions that support the following database features.

▼ Hide filters

Show versions that support the global database feature
Show versions that support the parallel query feature

Available versions (2/0)

Aurora (MySQL 5.7) 2.11.1

To see more versions, modify the capacity types. Info

⚠ Parallel query is off by default. To enable it, use a DB instance parameter group with the aurora_parallel_query parameter enabled. Learn more

Managing an Amazon Aurora global database

You perform most management operations on the individual clusters that make up an Aurora global database. When you choose **Group related resources** on the **Databases** page in the console, you see the primary cluster and secondary clusters grouped under the associated global database. To find the AWS Regions where a global database's DB clusters are running, its Aurora DB engine and version, and its identifier, use its **Configuration** tab.

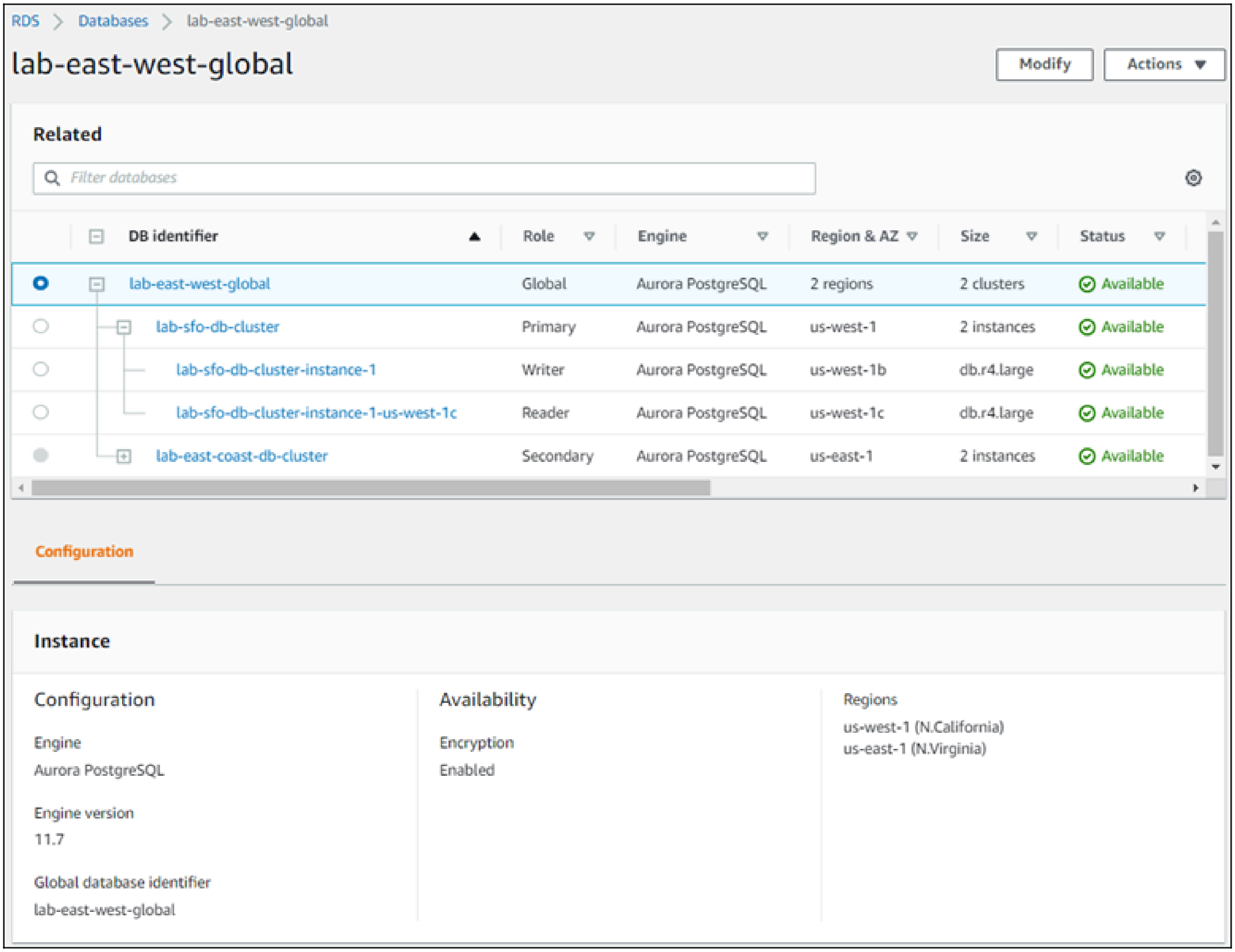
The cross-Region database failover processes are available to Aurora global databases only, not for a single Aurora DB cluster. To learn more, see Using switchover or failover in an Amazon Aurora global database.

To recover an Aurora global database from an unplanned outage in its primary Region, see Recovering an Amazon Aurora global database from an unplanned outage.

Topics

Modifying an Amazon Aurora global database

The **Databases** page in the AWS Management Console lists all your Aurora global databases, showing the primary cluster and secondary clusters for each one. The Aurora global database has its own configuration settings. Specifically, it has AWS Regions associated with its primary and secondary clusters, as shown in the screenshot following.



lab-east-west-global

Related

	DB identifier	Role	Engine	Region & AZ	Size	Status
●	lab-east-west-global	Global	Aurora PostgreSQL	2 regions	2 clusters	Available
○	lab-sfo-db-cluster	Primary	Aurora PostgreSQL	us-west-1	2 instances	Available
○	lab-sfo-db-cluster-instance-1	Writer	Aurora PostgreSQL	us-west-1b	db.r4.large	Available
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora PostgreSQL	us-west-1c	db.r4.large	Available
○	lab-east-coast-db-cluster	Secondary	Aurora PostgreSQL	us-east-1	2 instances	Available

Configuration

Instance

Configuration

Engine
Aurora PostgreSQL

Engine version
11.7

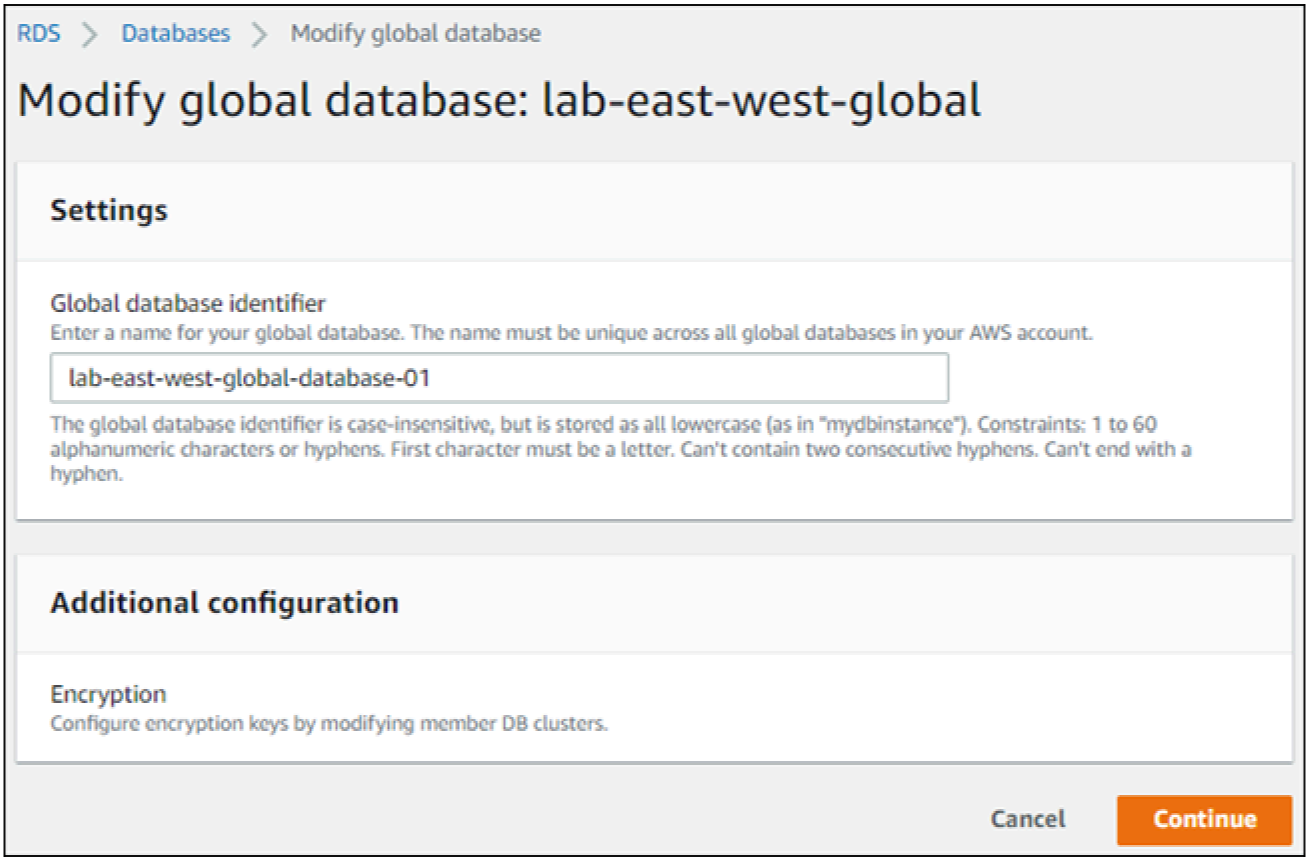
Global database identifier
lab-east-west-global

Availability

Encryption
Enabled

Regions
us-west-1 (N.California)
us-east-1 (N.Virginia)

When you make changes to the Aurora global database, you have a chance to cancel changes, as shown in the following screenshot.



Modify global database: lab-east-west-global

Settings

Global database identifier

Enter a name for your global database. The name must be unique across all global databases in your AWS account.

lab-east-west-global-database-01

The global database identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

Additional configuration

Encryption

Configure encryption keys by modifying member DB clusters.

When you choose **Continue**, you confirm the changes.

Modifying parameters for an Aurora global database

You can configure the Aurora DB cluster parameter groups independently for each Aurora cluster within the Aurora global database. Most parameters work the same as for other kinds of Aurora clusters. We recommend that you keep settings consistent among all the clusters in a global database. Doing this helps to avoid unexpected behavior changes if you promote a secondary cluster to be the primary.

For example, use the same settings for time zones and character sets to avoid inconsistent behavior if a different cluster takes over as the primary cluster.

The `aurora_enable_repl_bin_log_filtering` and `aurora_enable_replica_log_compression` configuration settings have no effect.

Removing a cluster from an Amazon Aurora global database

You can remove Aurora DB clusters from your Aurora global database for several different reasons. For example, you might want to remove an Aurora DB cluster from an Aurora global database if the primary cluster becomes degraded or isolated. It then becomes a standalone provisioned Aurora DB cluster that could be used to create a new Aurora global database. To learn more, see Recovering an Amazon Aurora global database from an unplanned outage.

You also might want to remove Aurora DB clusters because you want to delete an Aurora global database that you no longer need. You can't delete the Aurora global database until after you remove (detach) all associated Aurora DB clusters, leaving the primary for last. For more information, see Deleting an Amazon Aurora global database.

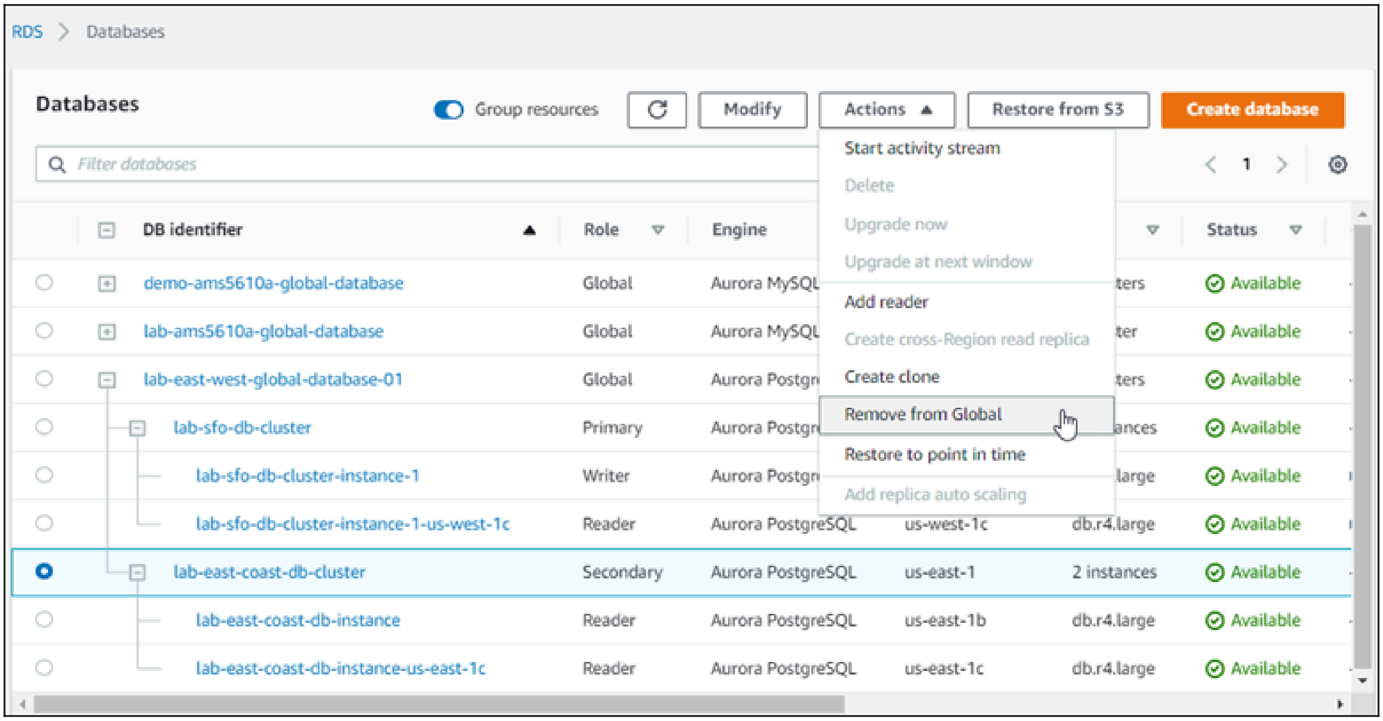
When an Aurora DB cluster is detached from the Aurora global database, it's no longer synchronized with the primary. It becomes a standalone provisioned Aurora DB cluster with full read/write capabilities.

You can remove Aurora DB clusters from your Aurora global database using the AWS Management Console, the AWS CLI, or the RDS API.

Console

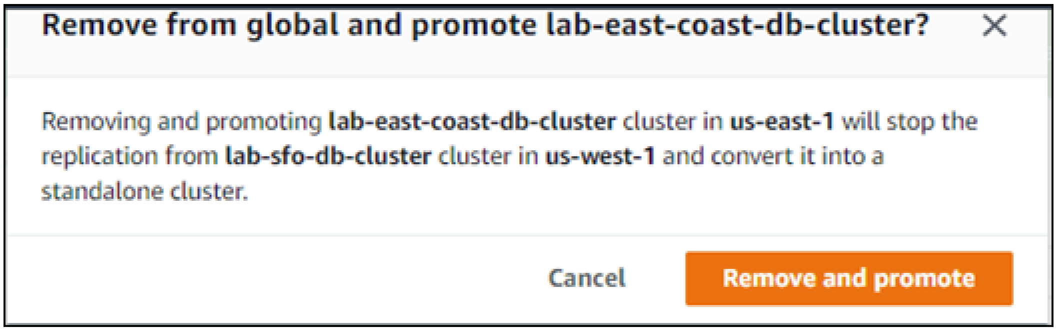
To remove an Aurora cluster from an Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose the cluster on the **Databases** page.
3. For **Actions**, choose **Remove from Global**.



	DB identifier ▲	Role	Engine			Status
○	demo-ams5610a-global-database	Global	Aurora MySQL		ters	✓ Available
○	lab-ams5610a-global-database	Global	Aurora MySQL		ter	✓ Available
○	lab-east-west-global-database-01	Global	Aurora Postgr		ters	✓ Available
○	lab-sfo-db-cluster	Primary	Aurora Postgr		ances	✓ Available
○	lab-sfo-db-cluster-instance-1	Writer	Aurora Postgr		large	✓ Available
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora PostgreSQL	us-west-1c	db.r4.large	✓ Available
●	lab-east-coast-db-cluster	Secondary	Aurora PostgreSQL	us-east-1	2 instances	✓ Available
○	lab-east-coast-db-instance	Reader	Aurora PostgreSQL	us-east-1b	db.r4.large	✓ Available
○	lab-east-coast-db-instance-us-east-1c	Reader	Aurora PostgreSQL	us-east-1c	db.r4.large	✓ Available

You see a prompt to confirm that you want to detach the secondary from the Aurora global database.

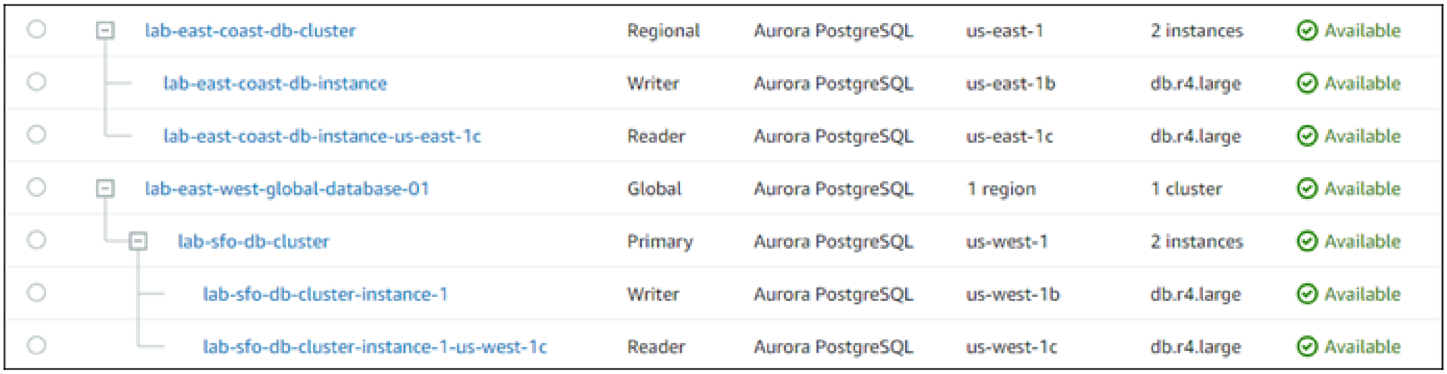


Remove from global and promote lab-east-coast-db-cluster?

Removing and promoting **lab-east-coast-db-cluster** cluster in **us-east-1** will stop the replication from **lab-sfo-db-cluster** cluster in **us-west-1** and convert it into a standalone cluster.

4. Choose **Remove and promote** to remove the cluster from the global database.

The Aurora DB cluster is no longer serving as a secondary in the Aurora global database, and is no longer synchronized with the primary DB cluster. It is a standalone Aurora DB cluster with full read/write capability.



○	⊟ lab-east-coast-db-cluster	Regional	Aurora PostgreSQL	us-east-1	2 instances	✓ Available
○	lab-east-coast-db-instance	Writer	Aurora PostgreSQL	us-east-1b	db.r4.large	✓ Available
○	lab-east-coast-db-instance-us-east-1c	Reader	Aurora PostgreSQL	us-east-1c	db.r4.large	✓ Available
○	⊟ lab-east-west-global-database-01	Global	Aurora PostgreSQL	1 region	1 cluster	✓ Available
○	⊟ lab-sfo-db-cluster	Primary	Aurora PostgreSQL	us-west-1	2 instances	✓ Available
○	lab-sfo-db-cluster-instance-1	Writer	Aurora PostgreSQL	us-west-1b	db.r4.large	✓ Available
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora PostgreSQL	us-west-1c	db.r4.large	✓ Available

After you remove or delete all secondary clusters, then you can remove the primary cluster the same way. You can't detach (remove) the primary Aurora DB cluster from an Aurora global database until after you remove all secondary clusters.

The Aurora global database might remain in the **Databases** list, with zero Regions and AZs. You can delete if you no longer want to use this Aurora global database. For more information, see Deleting an Amazon Aurora global database.

AWS CLI

To remove an Aurora cluster from an Aurora global database, run the remove-from-global-cluster CLI command with the following parameters:

- `--global-cluster-identifier` – The name (identifier) of your Aurora global database.
- `--db-cluster-identifier` – The name of each Aurora DB cluster to remove from the Aurora global database. Remove all secondary Aurora DB clusters before removing the primary.

The following examples first remove a secondary cluster and then the primary cluster from an Aurora global database.

For Linux, macOS, or Unix:

```
aws rds --region secondary_region \
  remove-from-global-cluster \
    --db-cluster-identifier secondary_cluster_ARN \
    --global-cluster-identifier global_database_id

aws rds --region primary_region \
  remove-from-global-cluster \
    --db-cluster-identifier primary_cluster_ARN \
    --global-cluster-identifier global_database_id
```

Repeat the `remove-from-global-cluster --db-cluster-identifier` *`secondary_cluster_ARN`* command for each secondary AWS Region in your Aurora global database.

For Windows:

```
aws rds --region secondary_region ^
  remove-from-global-cluster ^
    --db-cluster-identifier secondary_cluster_ARN ^
    --global-cluster-identifier global_database_id

aws rds --region primary_region ^
  remove-from-global-cluster ^
    --db-cluster-identifier primary_cluster_ARN ^
    --global-cluster-identifier global_database_id
```

Repeat the `remove-from-global-cluster --db-cluster-identifier` *`secondary_cluster_ARN`* command for each secondary AWS Region in your Aurora global database.

RDS API

To remove an Aurora cluster from an Aurora global database with the RDS API, run the RemoveFromGlobalCluster action.

Deleting an Amazon Aurora global database

Because an Aurora global database typically holds business-critical data, you can't delete the global database and its associated clusters in a single step. To delete an Aurora global database, do the following:

- Remove all secondary DB clusters from the Aurora global database. Each cluster becomes a standalone Aurora DB cluster. To learn how, see Removing a cluster from an Amazon Aurora global database.
- From each standalone Aurora DB cluster, delete all Aurora Replicas.
- Remove the primary DB cluster from the Aurora global database. This becomes a standalone Aurora DB cluster.
- From the Aurora primary DB cluster, first delete all Aurora Replicas, then delete the writer DB instance.

Deleting the writer instance from the newly standalone Aurora DB cluster also typically removes the Aurora DB cluster and the Aurora global database.

For more general information, see Deleting a DB instance from an Aurora DB cluster.

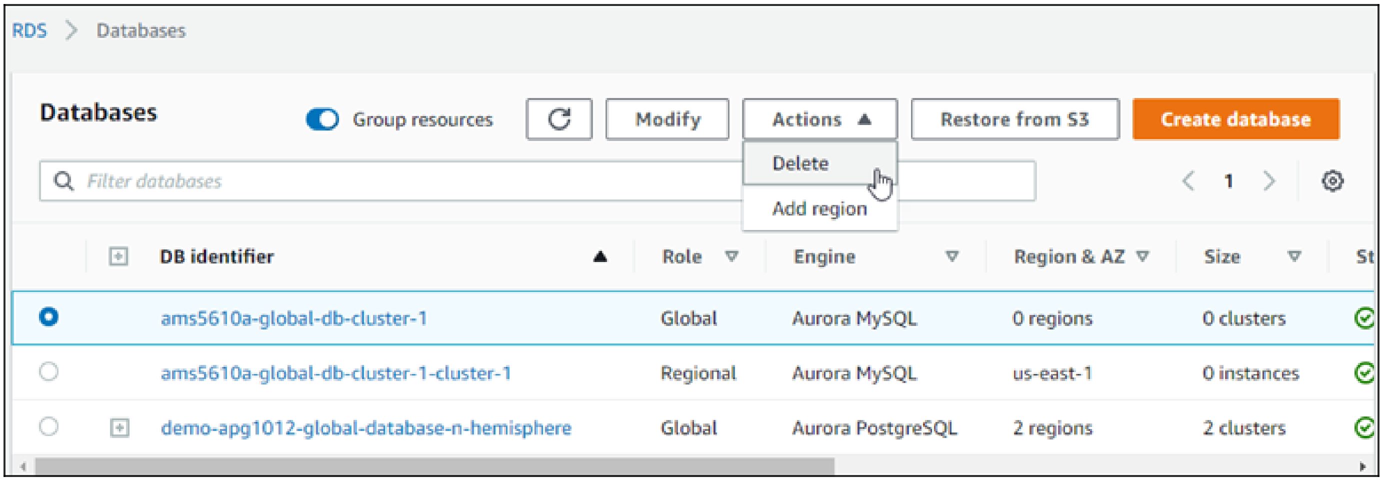
To delete an Aurora global database, you can use the AWS Management Console, the AWS CLI, or the RDS API.

Console

To delete an Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** and find the Aurora global database you want to delete in the listing.
3. Confirm that all clusters are removed from the Aurora global database. The Aurora global database should show 0 Regions and AZs and a size of 0 clusters.

   If the Aurora global database contains any Aurora DB clusters, you can't delete it. If necessary, detach the primary and secondary Aurora DB clusters from the Aurora global database. For more information, see Removing a cluster from an Amazon Aurora global database.
4. Choose your Aurora global database in the list, and then choose **Delete** from the **Actions** menu.



	DB identifier	Role	Engine	Region & AZ	Size	St
●	ams5610a-global-db-cluster-1	Global	Aurora MySQL	0 regions	0 clusters	✓
○	ams5610a-global-db-cluster-1-cluster-1	Regional	Aurora MySQL	us-east-1	0 instances	✓
○	demo-apg1012-global-database-n-hemisphere	Global	Aurora PostgreSQL	2 regions	2 clusters	✓

AWS CLI

To delete an Aurora global database, run the delete-global-cluster CLI command with the name of the AWS Region and the Aurora global database identifier, as shown in the following example.

For Linux, macOS, or Unix:

```
aws rds --region primary_region delete-global-cluster \
   --global-cluster-identifier global_database_id
```

For Windows:

```
aws rds --region primary_region delete-global-cluster ^
   --global-cluster-identifier global_database_id
```

RDS API

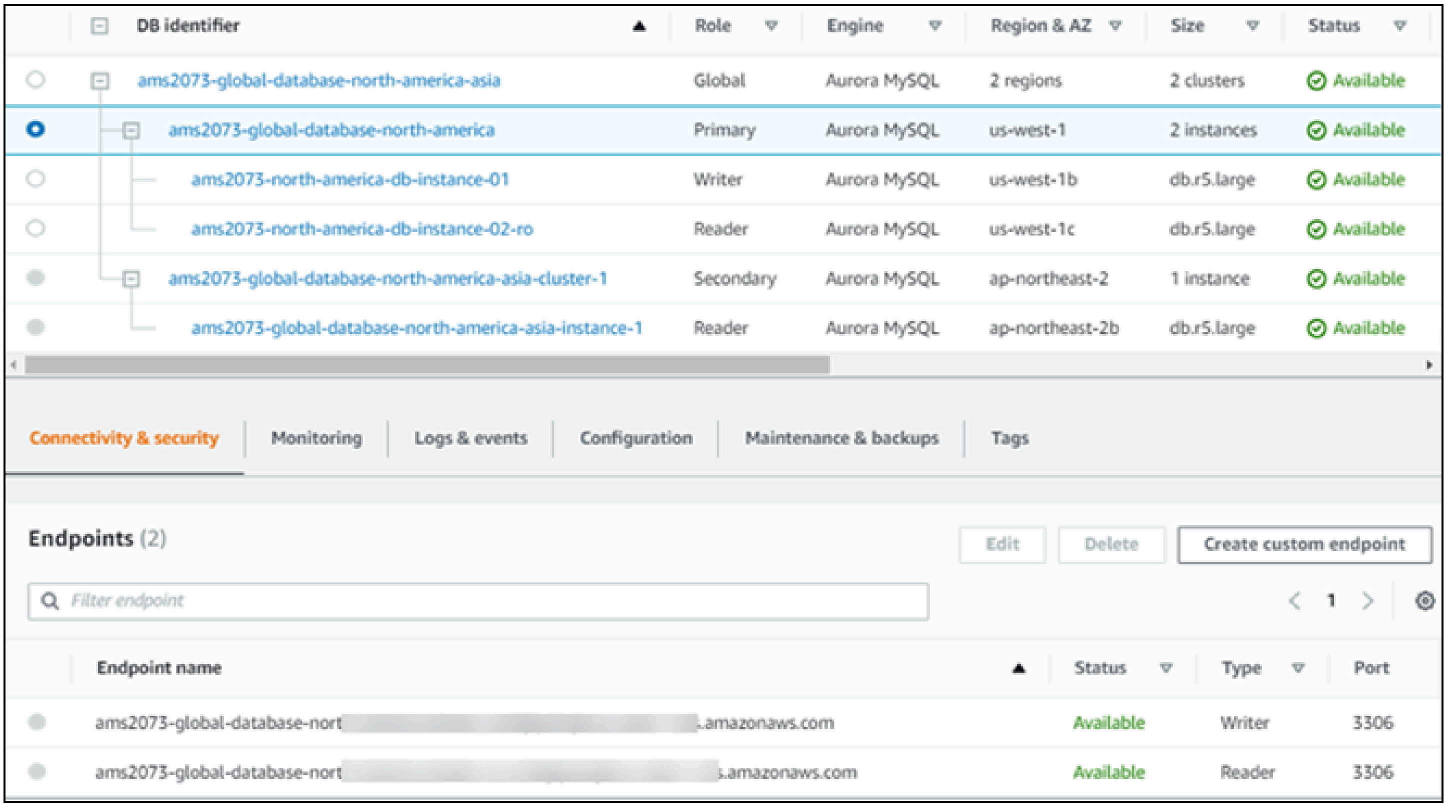
To delete a cluster that is part of an Aurora global database, run the DeleteGlobalCluster API operation.

Connecting to an Amazon Aurora global database

How you connect to an Aurora global database depends on whether you need to write to the database or read from the database:

- For read-only requests or queries, you connect to the reader endpoint for the Aurora cluster in your AWS Region.
- To run data manipulation language (DML) or data definition language (DDL) statements, you connect to the cluster endpoint for the primary cluster. This endpoint might be in a different AWS Region than your application.

When you view an Aurora global database in the console, you can see all the general-purpose endpoints associated with all of its clusters. The following screenshot shows an example. There is a single cluster endpoint associated with the primary cluster that you use for write operations. The primary cluster and each secondary cluster has a reader endpoint that you use for read-only queries. To minimize latency, choose whichever reader endpoint is in your AWS Region or the AWS Region closest to you. The following shows an Aurora MySQL example.



DB identifier	Role	Engine	Region & AZ	Size	Status
ams2073-global-database-north-america-asia	Global	Aurora MySQL	2 regions	2 clusters	Available
ams2073-global-database-north-america	Primary	Aurora MySQL	us-west-1	2 instances	Available
ams2073-north-america-db-instance-01	Writer	Aurora MySQL	us-west-1b	db.r5.large	Available
ams2073-north-america-db-instance-02-ro	Reader	Aurora MySQL	us-west-1c	db.r5.large	Available
ams2073-global-database-north-america-asia-cluster-1	Secondary	Aurora MySQL	ap-northeast-2	1 instance	Available
ams2073-global-database-north-america-asia-instance-1	Reader	Aurora MySQL	ap-northeast-2b	db.r5.large	Available

Endpoint name	Status	Type	Port
ams2073-global-database-nort[redacted].amazonaws.com	Available	Writer	3306
ams2073-global-database-nort[redacted].amazonaws.com	Available	Reader	3306

Using write forwarding in an Amazon Aurora global database

You can reduce the number of endpoints that you need to manage for applications running on your Aurora global database, by using *write forwarding*. With write forwarding enabled, secondary clusters in an Aurora global database forward SQL statements that perform write operations to the primary cluster. The primary cluster updates the source and then propagates resulting changes back to all secondary AWS Regions.

The write forwarding configuration saves you from implementing your own mechanism to send write operations from a secondary AWS Region to the primary Region. Aurora handles the cross-Region networking setup. Aurora also transmits all necessary session and transactional context for each statement. The data is always changed first on the primary cluster and then replicated to the secondary clusters in the Aurora global database. This way, the primary cluster is the source of truth and always has an up-to-date copy of all your data.

Topics

- Using write forwarding in an Aurora MySQL global database
- Using write forwarding in an Aurora PostgreSQL global database

Using write forwarding in an Aurora MySQL global database

Topics

Region and version availability of write forwarding in Aurora MySQL

Write forwarding is supported with Aurora MySQL 2.08.1 and higher versions, in every Region where Aurora MySQL-based global databases are available.

For information on version and Region availability of Aurora MySQL global databases, see Aurora global databases with Aurora MySQL.

Enabling write forwarding in Aurora MySQL

By default, write forwarding isn't enabled when you add a secondary cluster to an Aurora global database.

To enable write forwarding using the AWS Management Console, select the **Turn on global write forwarding** check box under **Read replica write forwarding** when you add a Region for a global database. For an existing secondary cluster, modify the cluster to **Turn on global write forwarding**. To turn off write forwarding, clear the **Turn on global write forwarding** check box when adding the Region or modifying the secondary cluster.

To enable write forwarding using the AWS CLI, use the `--enable-global-write-forwarding` option. This option works when you create a new secondary cluster using the `create-db-cluster` command. It also works when you modify an existing secondary cluster using the modify-db-cluster command. It requires that the global database uses an Aurora version that supports write forwarding. You can turn write forwarding off by using the --no-enable-global-write-forwarding option with these same CLI commands.

To enable write forwarding using the Amazon RDS API, set the EnableGlobalWriteForwarding parameter to true. This parameter works when you create a new secondary cluster using the CreateDBCluster operation. It also works when you modify an existing secondary cluster using the ModifyDBCluster operation. It requires that the global database uses an Aurora version that supports write forwarding. You can turn write forwarding off by setting the EnableGlobalWriteForwarding parameter to false.

> **Note**
>
> For a database session to use write forwarding, specify a setting for the aurora_replica_read_consistency configuration parameter. Do this in every session that uses the write forwarding feature. For information about this parameter, see Isolation and consistency for write forwarding in Aurora MySQL.
> The RDS Proxy feature doesn't support the SESSION value for the aurora_replica_read_consistency variable. Setting this value can cause unexpected behavior.

The following CLI examples show how you can set up an Aurora global database with write forwarding enabled or disabled. The highlighted items represent the commands and options that are important to specify and keep consistent when setting up the infrastructure for an Aurora global database.

The following example creates an Aurora global database, a primary cluster, and a secondary cluster with write forwarding enabled. Substitute your own choices for the user name, password, and primary and secondary AWS Regions.

```
# Create overall global database.
aws rds create-global-cluster --global-cluster-identifier write-forwarding-test \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-1

# Create primary cluster, in the same AWS Region as the global database.
aws rds create-db-cluster --global-cluster-identifier write-forwarding-test \
  --db-cluster-identifier write-forwarding-test-cluster-1 \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --master-username user_name --master-user-password password \
  --region us-east-1

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-1 \
  --db-instance-identifier write-forwarding-test-cluster-1-instance-1 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-1

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-1 \
  --db-instance-identifier write-forwarding-test-cluster-1-instance-2 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-1

# Create secondary cluster, in a different AWS Region than the global database,
# with write forwarding enabled.
aws rds create-db-cluster --global-cluster-identifier write-forwarding-test \
  --db-cluster-identifier write-forwarding-test-cluster-2 \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-2 \
  --enable-global-write-forwarding

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-2 \
  --db-instance-identifier write-forwarding-test-cluster-2-instance-1 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-2

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-2 \
  --db-instance-identifier write-forwarding-test-cluster-2-instance-2 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-east-2
```

The following example continues from the previous one. It creates a secondary cluster without write forwarding enabled, then enables write forwarding. After this example finishes, all secondary clusters in the global database have write forwarding enabled.

```
# Create secondary cluster, in a different AWS Region than the global database,
# without write forwarding enabled.
aws rds create-db-cluster --global-cluster-identifier write-forwarding-test \
  --db-cluster-identifier write-forwarding-test-cluster-2 \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-west-1

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-2 \
  --db-instance-identifier write-forwarding-test-cluster-2-instance-1 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-west-1

aws rds create-db-instance --db-cluster-identifier write-forwarding-test-cluster-2 \
  --db-instance-identifier write-forwarding-test-cluster-2-instance-2 \
  --db-instance-class db.r5.large \
  --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.1 \
  --region us-west-1

aws rds modify-db-cluster --db-cluster-identifier write-forwarding-test-cluster-2 \
  --region us-east-2 \
  --enable-global-write-forwarding
```

Checking if a secondary cluster has write forwarding enabled in Aurora MySQL

To determine whether you can use write forwarding from a secondary cluster, you can check whether the cluster has the attribute `"GlobalWriteForwardingStatus": "enabled"`.

In the AWS Management Console, on the **Configuration** tab of the details page for the cluster, you see the status **Enabled** for **Global read replica write forwarding**.

To see the status of the global write forwarding setting for all of your clusters, run the following AWS CLI command.

A secondary cluster shows the value `"enabled"` or `"disabled"` to indicate if write forwarding is turned on or off. A value of `null` indicates that write forwarding isn't available for that cluster. Either the cluster isn't part of a global database, or is the primary cluster instead of a secondary cluster. The value can also be `"enabling"` or `"disabling"` if write forwarding is in the process of being turned on or off.

Example

```
aws rds describe-db-clusters \
--query '*[].
{DBClusterIdentifier:DBClusterIdentifier,GlobalWriteForwardingStatus:GlobalWriteForwardingStatu
```

```
[
    {
        "GlobalWriteForwardingStatus": "enabled",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-1"
    },
    {
        "GlobalWriteForwardingStatus": "disabled",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-2"
    },
    {
        "GlobalWriteForwardingStatus": null,
        "DBClusterIdentifier": "non-global-cluster"
    }
]
```

To find all secondary clusters that have global write forwarding enabled, run the following command. This command also returns the cluster's reader endpoint. You use the secondary cluster's reader endpoint when you use write forwarding from the secondary to the primary in your Aurora global database.

Example

```
aws rds describe-db-clusters --query 'DBClusters[].
{DBClusterIdentifier:DBClusterIdentifier,GlobalWriteForwardingStatus:GlobalWriteForwardingStatu
 | [?GlobalWriteForwardingStatus == `enabled`]'
[
    {
        "GlobalWriteForwardingStatus": "enabled",
        "ReaderEndpoint": "aurora-write-forwarding-test-replica-1.cluster-ro-
cnpexample.us-west-2.rds.amazonaws.com",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-1"
    }
]
```

Application and SQL compatibility with write forwarding in Aurora MySQL

You can use the following kinds of SQL statements with write forwarding:

- Data manipulation language (DML) statements, such as `INSERT`, `DELETE`, and `UPDATE`. There are some restrictions on the properties of these statements that you can use with write forwarding, as described following.

- `SELECT ... LOCK IN SHARE MODE` and `SELECT FOR UPDATE` statements.
- PREPARE and EXECUTE statements.

Certain statements aren't allowed or can produce stale results when you use them in a global database with write forwarding. Thus, the `EnableGlobalWriteForwarding` setting is turned off by default for secondary clusters. Before turning it on, check to make sure that your application code isn't affected by any of these restrictions.

The following restrictions apply to the SQL statements you use with write forwarding. In some cases, you can use the statements on secondary clusters with write forwarding enabled at the cluster level. This approach works if write forwarding isn't turned on within the session by the `aurora_replica_read_consistency` configuration parameter. Trying to use a statement when it's not allowed because of write forwarding causes an error message with the following format.

```
ERROR 1235 (42000): This version of MySQL doesn't yet support 'operation with write
 forwarding'.
```

Data definition language (DDL)

Connect to the primary cluster to run DDL statements. You can't run them from reader DB instances.

Updating a permanent table using data from a temporary table

You can use temporary tables on secondary clusters with write forwarding enabled. However, you can't use a DML statement to modify a permanent table if the statement refers to a temporary table. For example, you can't use an `INSERT ... SELECT` statement that takes the data from a temporary table. The temporary table exists on the secondary cluster and isn't available when the statement runs on the primary cluster.

XA transactions

You can't use the following statements on a secondary cluster when write forwarding is turned on within the session. You can use these statements on secondary clusters that don't have write forwarding enabled, or within sessions where the `aurora_replica_read_consistency` setting is empty. Before turning on write forwarding within a session, check if your code uses these statements.

```
XA {START|BEGIN} xid [JOIN|RESUME]
XA END xid [SUSPEND [FOR MIGRATE]]
XA PREPARE xid
XA COMMIT xid [ONE PHASE]
XA ROLLBACK xid
XA RECOVER [CONVERT XID]
```

LOAD statements for permanent tables

You can't use the following statements on a secondary cluster with write forwarding enabled.

```
LOAD DATA INFILE 'data.txt' INTO TABLE t1;
        LOAD XML LOCAL INFILE 'test.xml' INTO TABLE t1;
```

You can load data into a temporary table on a secondary cluster. However, make sure that you run any LOAD statements that refer to permanent tables only on the primary cluster.

Plugin statements

You can't use the following statements on a secondary cluster with write forwarding enabled.

```
INSTALL PLUGIN example SONAME 'ha_example.so';
UNINSTALL PLUGIN example;
```

SAVEPOINT statements

You can't use the following statements on a secondary cluster when write forwarding is turned on within the session. You can use these statements on secondary clusters that don't have write forwarding enabled, or within sessions where the aurora_replica_read_consistency setting is blank. Check if your code uses these statements before turning on write forwarding within a session.

```
SAVEPOINT t1_save;
ROLLBACK TO SAVEPOINT t1_save;
RELEASE SAVEPOINT t1_save;
```

Isolation and consistency for write forwarding in Aurora MySQL

In sessions that use write forwarding, you can only use the REPEATABLE READ isolation level. Although you can also use the READ COMMITTED isolation level with read-only clusters in secondary AWS Regions, that isolation level doesn't work with write forwarding. For information about the `REPEATABLE READ` and `READ COMMITTED` isolation levels, see [Aurora MySQL isolation levels](#).

You can control the degree of read consistency on a secondary cluster. The read consistency level determines how much waiting the secondary cluster does before each read operation to ensure that some or all changes are replicated from the primary cluster. You can adjust the read consistency level to ensure that all forwarded write operations from your session are visible in the secondary cluster before any subsequent queries. You can also use this setting to ensure that queries on the secondary cluster always see the most current updates from the primary cluster. This is so even for those submitted by other sessions or other clusters. To specify this type of behavior for your application, you choose a value for the session-level parameter `aurora_replica_read_consistency`.

> ⚠ **Important**
>
> Always set the `aurora_replica_read_consistency` parameter for any session for which you want to forward writes. If you don't, Aurora doesn't enable write forwarding for that session. This parameter has an empty value by default, so choose a specific value when you use this parameter. The `aurora_replica_read_consistency` parameter has an effect only on secondary clusters that have write forwarding enabled.
> For Aurora MySQL version 2 and version 3 lower than 3.04, use `aurora_replica_read_consistency` as a session variable. For Aurora MySQL version 3.04 and higher, you can use `aurora_replica_read_consistency` as either a session variable or as a DB cluster parameter.

For the `aurora_replica_read_consistency` parameter, you can specify the values `EVENTUAL`, `SESSION`, and `GLOBAL`.

As you increase the consistency level, your application spends more time waiting for changes to be propagated between AWS Regions. You can choose the balance between fast response time and ensuring that changes made in other locations are fully available before your queries run.

With the read consistency set to `EVENTUAL`, queries in a secondary AWS Region that uses write forwarding might see data that is slightly stale due to replication lag. Results of write operations in the same session aren't visible until the write operation is performed on the primary Region and replicated to the current Region. The query doesn't wait for the updated results to be available.

Thus, it might retrieve the older data or the updated data, depending on the timing of the statements and the amount of replication lag.

With the read consistency set to SESSION, all queries in a secondary AWS Region that uses write forwarding see the results of all changes made in that session. The changes are visible regardless of whether the transaction is committed. If necessary, the query waits for the results of forwarded write operations to be replicated to the current Region. It doesn't wait for updated results from write operations performed in other Regions or in other sessions within the current Region.

With the read consistency set to GLOBAL, a session in a secondary AWS Region sees changes made by that session. It also sees all committed changes from both the primary AWS Region and other secondary AWS Regions. Each query might wait for a period that varies depending on the amount of session lag. The query proceeds when the secondary cluster is up-to-date with all committed data from the primary cluster, as of the time that the query began.

For more information about all the parameters involved with write forwarding, see Configuration parameters for write forwarding in Aurora MySQL.

Examples of using write forwarding

These examples use aurora_replica_read_consistency as a session variable. For Aurora MySQL version 3.04 and higher, you can use aurora_replica_read_consistency as either a session variable or as a DB cluster parameter.

In the following example, the primary cluster is in the US East (N. Virginia) Region. The secondary cluster is in the US East (Ohio) Region. The example shows the effects of running INSERT statements followed by SELECT statements. Depending on the value of the aurora_replica_read_consistency setting, the results might differ depending on the timing of the statements. To achieve higher consistency, you might wait briefly before issuing the SELECT statement. Or Aurora can automatically wait until the results finish replicating before proceeding with SELECT.

In this example, there is a read consistency setting of eventual. Running an INSERT statement immediately followed by a SELECT statement still returns the value of COUNT(*). This value reflects the number of rows before the new row is inserted. Running the SELECT again a short time later returns the updated row count. The SELECT statements don't wait.

```
mysql> set aurora_replica_read_consistency = 'eventual';
mysql> select count(*) from t1;
```

```
+----------+
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)
mysql> insert into t1 values (6); select count(*) from t1;
+----------+
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.00 sec)
```

With a read consistency setting of `session`, a `SELECT` statement immediately after an `INSERT` waits until the changes from the `INSERT` statement are visible. Subsequent `SELECT` statements don't wait.

```
mysql> set aurora_replica_read_consistency = 'session';
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.01 sec)
mysql> insert into t1 values (6); select count(*) from t1; select count(*) from t1;
Query OK, 1 row affected (0.08 sec)
+----------+
| count(*) |
+----------+
|        7 |
+----------+
1 row in set (0.37 sec)
+----------+
| count(*) |
+----------+
|        7 |
+----------+
1 row in set (0.00 sec)
```

With the read consistency setting still set to `session`, introducing a brief wait after performing an `INSERT` statement makes the updated row count available by the time the next `SELECT` statement runs.

```
mysql> insert into t1 values (6); select sleep(2); select count(*) from t1;
Query OK, 1 row affected (0.07 sec)
+----------+
| sleep(2) |
+----------+
|        0 |
+----------+
1 row in set (2.01 sec)
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.00 sec)
```

With a read consistency setting of `global`, each `SELECT` statement waits to ensure that all data changes as of the start time of the statement are visible before performing the query. The amount of waiting for each `SELECT` statement varies, depending on the amount of replication lag between the primary and secondary clusters.

```
mysql> set aurora_replica_read_consistency = 'global';
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.75 sec)
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.37 sec)
mysql> select count(*) from t1;
+----------+
| count(*) |
+----------+
|        8 |
+----------+
1 row in set (0.66 sec)
```

Running multipart statements with write forwarding in Aurora MySQL

A DML statement might consist of multiple parts, such as a `INSERT ... SELECT` statement or a `DELETE ... WHERE` statement. In this case, the entire statement is forwarded to the primary cluster and run there.

Transactions with write forwarding in Aurora MySQL

Whether the transaction is forwarded to the primary cluster depends on the access mode of the transaction. You can specify the access mode for the transaction by using the `SET TRANSACTION` statement or the `START TRANSACTION` statement. You can also specify the transaction access mode by changing the value of the Aurora MySQL session variable `tx_read_only`. You can only change this session value while you're connected to a secondary cluster that has write forwarding enabled.

If a long-running transaction doesn't issue any statement for a substantial period of time, it might exceed the idle timeout period. This period has a default of one minute. You can increase it up to one day. A transaction that exceeds the idle timeout is canceled by the primary cluster. The next subsequent statement you submit receives a timeout error. Then Aurora rolls back the transaction.

This type of error can occur in other cases when write forwarding becomes unavailable. For example, Aurora cancels any transactions that use write forwarding if you restart the primary cluster or if you turn off the write forwarding configuration setting.

Configuration parameters for write forwarding in Aurora MySQL

The Aurora cluster parameter groups include settings for the write forwarding feature. Because these are cluster parameters, all DB instances in each cluster have the same values for these variables. Details about these parameters are summarized in the following table, with usage notes after the table.

Name	Scope	Type	Default value	Valid values
`aurora_fwd_master_idle_timeout` (Aurora MySQL version 2)	Global	unsigned integer	60	1–86,400
`aurora_fwd_master_max_connections_pct` (Aurora MySQL version 2)	Global	unsigned long integer	10	0–90
`aurora_fwd_writer_idle_timeout` (Aurora MySQL version 3)	Global	unsigned integer	60	1–86,400
`aurora_fwd_writer_max_connections_pct` (Aurora MySQL version 3)	Global	unsigned long integer	10	0–90
`aurora_replica_read_consistency`	Session	Enum	'' (null)	`EVENTUAL`, `SESSION`, `GLOBAL`

To control incoming write requests from secondary clusters, use these settings on the primary cluster:

- `aurora_fwd_master_idle_timeout`, `aurora_fwd_writer_idle_timeout`: The number of seconds the primary cluster waits for activity on a connection that's forwarded from a secondary cluster before closing it. If the session remains idle beyond this period, Aurora cancels the session.
- `aurora_fwd_master_max_connections_pct`, `aurora_fwd_writer_max_connections_pct`: The upper limit on database connections that can be used on a writer DB instance to handle queries forwarded from readers. It's expressed as a percentage of the `max_connections` setting for the writer DB instance in the primary cluster. For example, if `max_connections` is 800 and `aurora_fwd_master_max_connections_pct` or `aurora_fwd_writer_max_connections_pct` is 10, then the writer allows a maximum of 80 simultaneous forwarded sessions. These connections come from the same connection pool managed by the `max_connections` setting.

This setting applies only on the primary cluster, when one or more secondary clusters have write forwarding enabled. If you decrease the value, existing connections aren't affected. Aurora takes the new value of the setting into account when attempting to create a new connection from a secondary cluster. The default value is 10, representing 10% of the `max_connections` value. If you enable query forwarding on any of the secondary clusters, this setting must have a nonzero value for write operations from secondary clusters to succeed. If the value is zero, the write operations receive the error code `ER_CON_COUNT_ERROR` with the message `Not enough connections on writer to handle your request`.

The `aurora_replica_read_consistency` parameter is a session-level parameter that enables write forwarding. You use it in each session. You can specify `EVENTUAL`, `SESSION`, or `GLOBAL` for read consistency level. To learn more about consistency levels, see [Isolation and consistency for write forwarding in Aurora MySQL](). The following rules apply to this parameter:

- This is a session-level parameter. The default value is '' (empty).
- Write forwarding is available in a session only if `aurora_replica_read_consistency` is set to `EVENTUAL` or `SESSION` or `GLOBAL`. This parameter is relevant only in reader instances of secondary clusters that have write forwarding enabled and that are in an Aurora global database.
- You can't set this variable (when empty) or unset (when already set) inside a multistatement transaction. However, you can change it from one valid value (`EVENTUAL`, `SESSION`, or `GLOBAL`) to another valid value (`EVENTUAL`, `SESSION`, or `GLOBAL`) during such a transaction.
- The variable can't be `SET` when write forwarding isn't enabled on the secondary cluster.
- Setting the session variable on a primary cluster doesn't have any effect. If you try to modify this variable on a primary cluster, you receive an error.

Amazon CloudWatch metrics for write forwarding in Aurora MySQL

The following Amazon CloudWatch metrics and Aurora MySQL status variables apply to the primary cluster when you use write forwarding on one or more secondary clusters. These metrics are all measured on the writer DB instance in the primary cluster.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
`ForwardingMasterDMLLatency`	–	Milliseconds	Average time to process each forwarded DML statement on the writer DB instance. It doesn't include the time for the secondary cluster to forward the write request, or the time to replicate changes back to the secondary cluster. For Aurora MySQL version 2.
`ForwardingMasterDMLThroughput`	–	Count per second	Number of forwarded DML statements processed each second by this writer DB instance. For Aurora MySQL version 2.
`ForwardingMasterOpenSessions`	`Aurora_fwd_master_open_sessions`	Count	Number of forwarded sessions on the writer DB instance. For Aurora MySQL version 2.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
–	Aurora_fwd_master_dml_stmt_count	Count	Total number of DML statements forwarded to this writer DB instance. For Aurora MySQL version 2.
–	Aurora_fwd_master_dml_stmt_duration	Microseconds	Total duration of DML statements forwarded to this writer DB instance. For Aurora MySQL version 2.
–	Aurora_fwd_master_select_stmt_count	Count	Total number of SELECT statements forwarded to this writer DB instance. For Aurora MySQL version 2.
–	Aurora_fwd_master_select_stmt_duration	Microseconds	Total duration of SELECT statements forwarded to this writer DB instance. For Aurora MySQL version 2.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
`ForwardingWriterDMLLatency`	–	Milliseconds	Average time to process each forwarded DML statement on the writer DB instance. It doesn't include the time for the secondary cluster to forward the write request, or the time to replicate changes back to the secondary cluster. For Aurora MySQL version 3.
`ForwardingWriterDMLThroughput`	–	Count per second	Number of forwarded DML statements processed each second by this writer DB instance. For Aurora MySQL version 3.
`ForwardingWriterOpenSessions`	`Aurora_fwd_writer_open_sessions`	Count	Number of forwarded sessions on the writer DB instance. For Aurora MySQL version 3.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
–	`Aurora_fwd_writer_dml_stmt_count`	Count	Total number of DML statements forwarded to this writer DB instance. For Aurora MySQL version 3.
–	`Aurora_fwd_writer_dml_stmt_duration`	Microseconds	Total duration of DML statements forwarded to this writer DB instance.
–	`Aurora_fwd_writer_select_stmt_count`	Count	Total number of SELECT statements forwarded to this writer DB instance. For Aurora MySQL version 3.
–	`Aurora_fwd_writer_select_stmt_duration`	Microseconds	Total duration of SELECT statements forwarded to this writer DB instance. For Aurora MySQL version 3.

The following CloudWatch metrics and Aurora MySQL status variables apply to each secondary cluster. These metrics are measured on each reader DB instance in a secondary cluster with write forwarding enabled.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
`ForwardingReplicaDMLLatency`	–	Milliseconds	Average response time of forwarded DMLs on the replica.
`ForwardingReplicaDMLThroughput`	–	Count per second	Number of forwarded DML statements processed each second.
`ForwardingReplicaOpenSessions`	`Aurora_fwd_replica_open_sessions`	Count	Number of sessions that are using write forwarding on a reader DB instance.
`ForwardingReplicaReadWaitLatency`	–	Milliseconds	Average wait time that a `SELECT` statement on a reader DB instance waits to catch up to the primary cluster. The degree to which the reader DB instance waits before processing a query depends on the `aurora_replica_read_consistency` setting.
`ForwardingReplicaReadWaitThroughput`	–	Count per second	Total number of `SELECT` statements processed each second in all sessions

CloudWatch metric	Aurora MySQL status variable	Unit	Description
			that are forwarding writes.
`ForwardingReplicaSelectLatency`	(–)	Milliseconds	Forwarded `SELECT` latency, average over all forwarded `SELECT` statements within the monitoring period.
`ForwardingReplicaSelectThroughput`	–	Count per second	Forwarded `SELECT` throughput per second average within the monitoring period.
–	`Aurora_fwd_replica_dml_stmt_count`	Count	Total number of DML statements forwarded from this reader DB instance.
–	`Aurora_fwd_replica_dml_stmt_duration`	Microseconds	Total duration of all DML statements forwarded from this reader DB instance.

CloudWatch metric	Aurora MySQL status variable	Unit	Description
–	`Aurora_fwd_replica_errors_session_limit`	Count	Number of sessions rejected by the primary cluster due to one of the following error conditions: • writer full • Too many forwarded statements in progress.
–	`Aurora_fwd_replica_read_wait_count`	Count	Total number of read-after-write waits on this reader DB instance.
–	`Aurora_fwd_replica_read_wait_duration`	Microseconds	Total duration of waits due to the read consistency setting on this reader DB instance.
–	`Aurora_fwd_replica_select_stmt_count`	Count	Total number of SELECT statements forwarded from this reader DB instance.
–	`Aurora_fwd_replica_select_stmt_duration`	Microseconds	Total duration of SELECT statements forwarded from this reader DB instance.

Using write forwarding in an Aurora PostgreSQL global database

Topics

Region and version availability of write forwarding in Aurora PostgreSQL

Write forwarding is supported with Aurora PostgreSQL version 15.4 and higher minor versions, and version 14.9 and higher minor versions. Write forwarding is available in every Region where Aurora PostgreSQL-based global databases are available.

For more information on version and Region availability of Aurora PostgreSQL global databases, see Aurora global databases with Aurora PostgreSQL.

Enabling write forwarding in Aurora PostgreSQL

By default, write forwarding isn't enabled when you add a secondary cluster to an Aurora global database. You can enable write forwarding for your secondary DB cluster while you're creating it or anytime after you create it. If needed, you can disable it later. Enabling or disabling write forwarding doesn't cause downtime or a reboot.

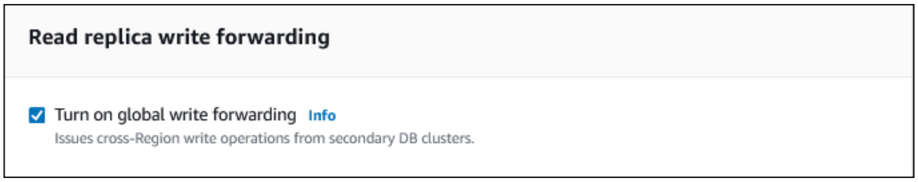
Console

In the console, you can enable or disable write forwarding when you create or modify a secondary DB cluster.

Enabling or disabling write forwarding when creating a secondary DB cluster

When you create a new secondary DB cluster, you enable write forwarding by selecting the **Turn on global write forwarding** check box under **Read replica write forwarding**. Or clear the check box to disable it. To create a secondary DB cluster, follow the instructions for your DB engine in Creating an Amazon Aurora DB cluster.

The following screenshot shows the **Read replica write forwarding** section with the **Turn on global write forwarding** check box selected.



Read replica write forwarding

☑ Turn on global write forwarding Info
Issues cross-Region write operations from secondary DB clusters.

Enabling or disabling write forwarding when modifying a secondary DB cluster

In the console, you can modify a secondary DB cluster to enable or disable write forwarding.

To enable or disable write forwarding for a secondary DB cluster by using the console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases**.
3. Choose the secondary DB cluster, and choose **Modify**.
4. In the **Read replica write forwarding** section, check or clear the **Turn on global write forwarding** check box.
5. Choose **Continue**.
6. For **Schedule modifications**, choose **Apply immediately**. If you choose **Apply during the next scheduled maintenance window**, Aurora ignores this setting and turns on write forwarding immediately.
7. Choose **Modify cluster**.

AWS CLI

To enable write forwarding by using the AWS CLI, use the `--enable-global-write-forwarding` option. This option works when you create a new secondary cluster using the create-db-cluster command. It also works when you modify an existing secondary cluster using the modify-db-cluster command. It requires that the global database uses an Aurora version that supports write forwarding. You can disable write forwarding by using the `--no-enable-global-write-forwarding` option with these same CLI commands.

The following procedures describe how to enable or disable write forwarding for a secondary DB cluster in your global cluster by using the AWS CLI.

To enable or disable write forwarding for an existing secondary DB cluster

- Call the [modify-db-cluster](https://docs.aws.amazon.com/cli/latest/reference/rds/modify-db-cluster.html) AWS CLI command and supply the following values:

  - `--db-cluster-identifier` – The name of the DB cluster.
  - `--enable-global-write-forwarding` to turn on or `--no-enable-global-write-forwarding` to turn off.

  The following example enables write forwarding for DB cluster `sample-secondary-db-cluster`.

  For Linux, macOS, or Unix:

  ```
  aws rds modify-db-cluster \
      --db-cluster-identifier sample-secondary-db-cluster \
      --enable-global-write-forwarding
  ```

  For Windows:

  ```
  aws rds modify-db-cluster ^
      --db-cluster-identifier sample-secondary-db-cluster ^
      --enable-global-write-forwarding
  ```

RDS API

To enable write forwarding using the Amazon RDS API, set the `EnableGlobalWriteForwarding` parameter to `true`. This parameter works when you create a new secondary cluster using the CreateDBCluster operation. It also works when you modify an existing secondary cluster using the ModifyDBCluster operation. It requires that the global database uses an Aurora version that supports write forwarding. You can disable write forwarding by setting the `EnableGlobalWriteForwarding` parameter to `false`.

Checking if a secondary cluster has write forwarding enabled in Aurora PostgreSQL

To determine whether you can use write forwarding from a secondary cluster, you can check whether the cluster has the attribute `"GlobalWriteForwardingStatus": "enabled"`.

In the AWS Management Console, you see `Read replica write forwarding` on the **Configuration** tab of the details page for the cluster. To see the status of the global write forwarding setting for all of your clusters, run the following AWS CLI command.

A secondary cluster shows the value `"enabled"` or `"disabled"` to indicate if write forwarding is turned on or off. A value of `null` indicates that write forwarding isn't available for that cluster. Either the cluster isn't part of a global database, or is the primary cluster instead of a secondary cluster. The value can also be `"enabling"` or `"disabling"` if write forwarding is in the process of being turned on or off.

Example

```
aws rds describe-db-clusters --query '*[].
{DBClusterIdentifier:DBClusterIdentifier,GlobalWriteForwardingStatus:GlobalWriteForwardingStatu
[
    {
        "GlobalWriteForwardingStatus": "enabled",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-1"
    },
    {
        "GlobalWriteForwardingStatus": "disabled",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-2"
    },
    {
        "GlobalWriteForwardingStatus": null,
        "DBClusterIdentifier": "non-global-cluster"
    }
]
```

To find all secondary clusters that have global write forwarding enabled, run the following command. This command also returns the cluster's reader endpoint. You use the secondary cluster's reader endpoint when you use write forwarding from the secondary to the primary in your Aurora global database.

Example

```
aws rds describe-db-clusters --query 'DBClusters[].
{DBClusterIdentifier:DBClusterIdentifier,GlobalWriteForwardingStatus:GlobalWriteForwardingStatu
 | [?GlobalWriteForwardingStatus == `enabled`]'
[
    {
        "GlobalWriteForwardingStatus": "enabled",
        "ReaderEndpoint": "aurora-write-forwarding-test-replica-1.cluster-ro-
cnpexample.us-west-2.rds.amazonaws.com",
        "DBClusterIdentifier": "aurora-write-forwarding-test-replica-1"
    }
]
```

Application and SQL compatibility with write forwarding in Aurora PostgreSQL

Certain statements aren't allowed or can produce stale results when you use them in a global database with write forwarding. In addition, user defined functions and user defined procedures aren't supported. Thus, the `EnableGlobalWriteForwarding` setting is turned off by default for secondary clusters. Before turning it on, check to make sure that your application code isn't affected by any of these restrictions.

You can use the following kinds of SQL statements with write forwarding:

- Data manipulation language (DML) statements, such as `INSERT`, `DELETE`, and `UPDATE`
- `SELECT FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }` statements
- `PREPARE` and `EXECUTE` statements
- `EXPLAIN` statements with the statements in this list

The following kinds of SQL statements aren't supported with write forwarding:

- Data definition language (DDL) statements
- `ANALYZE`
- `CLUSTER`
- `COPY`
- Cursors – Cursors aren't supported, so make sure to close them before using write forwarding.
- `GRANT|REVOKE|REASSIGN OWNED|SECURITY LABEL`
- `LOCK`

- SAVEPOINT statements
- SELECT INTO
- SET CONSTRAINTS
- TRUNCATE
- VACUUM

Isolation and consistency for write forwarding in Aurora PostgreSQL

In sessions that use write forwarding, you can use the REPEATABLE READ and READ COMMITTED isolation levels. However, the SERIALIZABLE isolation level isn't supported.

You can control the degree of read consistency on a secondary cluster. The read consistency level determines how much waiting the secondary cluster does before each read operation to ensure that some or all changes are replicated from the primary cluster. You can adjust the read consistency level to ensure that all forwarded write operations from your session are visible in the secondary cluster before any subsequent queries. You can also use this setting to ensure that queries on the secondary cluster always see the most current updates from the primary cluster. This is so even for those submitted by other sessions or other clusters. To specify this type of behavior for your application, you choose the appropriate value for the session-level parameter apg_write_forward.consistency_mode. The apg_write_forward.consistency_mode parameter has an effect only on secondary clusters that have write forwarding enabled.

Note

For the apg_write_forward.consistency_mode parameter, you can specify the value SESSION, EVENTUAL, GLOBAL, or OFF. By default, the value is set to SESSION. Setting the value to OFF disables write forwarding in the session.

As you increase the consistency level, your application spends more time waiting for changes to be propagated between AWS Regions. You can choose the balance between fast response time and ensuring that changes made in other locations are fully available before your queries run.

With each available consistency mode setting, the effect is as follows:

- SESSION – All queries in a secondary AWS Region that uses write forwarding see the results of all changes made in that session. The changes are visible regardless of whether the transaction is committed. If necessary, the query waits for the results of forwarded write operations to be replicated to the current Region. It doesn't wait for updated results from write operations performed in other Regions or in other sessions within the current Region.

- `EVENTUAL` – Queries in a secondary AWS Region that uses write forwarding might see data that is slightly stale due to replication lag. Results of write operations in the same session aren't visible until the write operation is performed on the primary Region and replicated to the current Region. The query doesn't wait for the updated results to be available. Thus, it might retrieve the older data or the updated data, depending on the timing of the statements and the amount of replication lag.
- `GLOBAL` – A session in a secondary AWS Region sees changes made by that session. It also sees all committed changes from both the primary AWS Region and other secondary AWS Regions. Each query might wait for a period that varies depending on the amount of session lag. The query proceeds when the secondary cluster is up-to-date with all committed data from the primary cluster, as of the time that the query began.
- `OFF` – Write forwarding is disabled in the session.

For more information about all the parameters involved with write forwarding, see Configuration parameters for write forwarding in Aurora PostgreSQL.

Running multipart statements with write forwarding in Aurora PostgreSQL

A DML statement might consist of multiple parts, such as a `INSERT ... SELECT` statement or a `DELETE ... WHERE` statement. In this case, the entire statement is forwarded to the primary cluster and run there.

Configuration parameters for write forwarding in Aurora PostgreSQL

The Aurora cluster parameter groups include settings for the write forwarding feature. Because these are cluster parameters, all DB instances in each cluster have the same values for these variables. Details about these parameters are summarized in the following table, with usage notes after the table.

Name	Scope	Type	Default value	Valid values
`apg_write_forward.connect_timeout`	Session	seconds	30	0–2147483647
`apg_write_forward.consistency_mode`	Session	enum	Session	`SESSION`, `EVENTUAL`, `GLOBAL`, `OFF`
`apg_write_forward.idle_in_transaction_session_timeout`	Session	milliseconds	86400000	0–2147483647
`apg_write_forward.idle_session_timeout`	Session	milliseconds	300000	0–2147483647
`apg_write_forward.max_forwarding_connections_percent`	Global	int	25	1–100

The `apg_write_forward.max_forwarding_connections_percent` parameter is the upper limit on database connection slots that can be used to handle queries forwarded from readers. It is expressed as a percentage of the `max_connections` setting for the writer DB instance in the primary cluster. For example, if `max_connections` is `800` and `apg_write_forward.max_forwarding_connections_percent` is `10`, then the writer allows a maximum of 80 simultaneous forwarded sessions. These connections come from the same connection pool managed by the `max_connections` setting. This setting applies only on the primary cluster when at least one secondary clusters has write forwarding enabled.

Use the following settings on the secondary cluster:

- `apg_write_forward.consistency_mode` – A session-level parameter that controls the degree of read consistency on the secondary cluster. Valid values are `SESSION`, `EVENTUAL`, `GLOBAL`, or `OFF`. By default, the value is set to `SESSION`. Setting the value to `OFF` disables write forwarding in the session. To learn more about consistency levels, see Isolation and consistency for write forwarding in Aurora PostgreSQL. This parameter is relevant only in reader instances of secondary clusters that have write forwarding enabled and that are in an Aurora global database.

- `apg_write_forward.connect_timeout` – The maximum number of seconds the secondary cluster waits when establishing a connection to the primary cluster before giving up. A value of `0` means to wait indefinitely.
- `apg_write_forward.idle_in_transaction_session_timeout` – The number of milliseconds the primary cluster waits for activity on a connection that's forwarded from a secondary cluster that has an open transaction before closing it. If the session remains idle in transaction beyond this period, Aurora terminates the session. A value of `0` disables the timeout.
- `apg_write_forward.idle_session_timeout` – The number of milliseconds the primary cluster waits for activity on a connection that's forwarded from a secondary cluster before closing it. If the session remains idle beyond this period, Aurora terminates the session. A value of `0` disables the timeout.

Amazon CloudWatch metrics for write forwarding in Aurora PostgreSQL

The following Amazon CloudWatch metrics apply to the primary cluster when you use write forwarding on one or more secondary clusters. These metrics are all measured on the writer DB instance in the primary cluster.

CloudWatch Metric	Units and description
`AuroraForwardingWriterDMLThroughput`	Count (per second). Number of forwarded DML statements processed each second by this writer DB instance.
`AuroraForwardingWriterOpenSessions`	Count. Number of open sessions on this writer DB instance processing forwarded queries.
`AuroraForwardingWriterTotalSessions`	Count. Total number of forwarded sessions on this writer DB instance.

The following CloudWatch metrics apply to each secondary cluster. These metrics are measured on each reader DB instance in a secondary cluster with write forwarding enabled.

CloudWatch Metric	Unit and description
`AuroraForwardingReplicaCommitThroughput`	Count (per second). Number of commits in sessions forwarded by this replica each second.
`AuroraForwardingReplicaDMLLatency`	Milliseconds. Average response time in milliseconds of forwarded DMLs on replica.
`AuroraForwardingReplicaDMLThroughput`	Count (per second). Number of forwarded DML statements processed on this replica each second.
`AuroraForwardingReplicaErrorSessionsLimit`	Count. Number of sessions rejected by the primary cluster because the limit for max connections or max write forward connections was reached.
`AuroraForwardingReplicaOpenSessions`	Count. The number of sessions that are using write forwarding on a replica instance.
`AuroraForwardingReplicaReadWaitLatency`	Milliseconds. Average wait time in milliseconds that the replica waits to be consistent with the LSN of the primary cluster. The degree to which the reader DB instance waits depends on the `apg_write_forward.consistency_mode` setting. For information about this setting, see the section called "Configuration parameters for write forwarding in Aurora PostgreSQL".

Wait events for write forwarding in Aurora PostgreSQL

Amazon Aurora generates the following wait events when you use write forwarding with Aurora PostgreSQL.

Topics

- IPC:AuroraWriteForwardConnect
- IPC:AuroraWriteForwardConsistencyPoint
- IPC:AuroraWriteForwardExecute
- IPC:AuroraWriteForwardGetGlobalConsistencyPoint
- IPC:AuroraWriteForwardXactAbort
- IPC:AuroraWriteForwardXactCommit
- IPC:AuroraWriteForwardXactStart

IPC:AuroraWriteForwardConnect

The `IPC:AuroraWriteForwardConnect` event occurs when a backend process on the secondary DB cluster is waiting for a connection to the writer node of the primary DB cluster to be opened.

Likely causes of increased waits

This event increases as the number of connection attempts from a secondary Region's reader node to the writer node of the primary DB cluster increases.

Actions

Reduce the number of simultaneous connections from a secondary node to the primary Region's writer node.

IPC:AuroraWriteForwardConsistencyPoint

The `IPC:AuroraWriteForwardConsistencyPoint` event describes how long a query from a node on the secondary DB cluster will wait for the results of forwarded write operations to be replicated to the current Region. This event is only generated if the session-level parameter `apg_write_forward.consistency_mode` is set to one of the following:

- `SESSION` – queries on a secondary node wait for the results of all changes made in that session.
- `GLOBAL` – queries on a secondary node wait for the results of changes made by that session, plus all committed changes from both the primary Region and other secondary Regions in the global cluster.

For more information about the `apg_write_forward.consistency_mode` parameter settings, see the section called “Configuration parameters for write forwarding in Aurora PostgreSQL”.

Likely causes of increased waits

Common causes for longer wait times include the following:

- Increased replica lag, as measured by the Amazon CloudWatch `ReplicaLag` metric. For more information about this metric, see Monitoring Aurora PostgreSQL replication.
- Increased load on the primary Region's writer node or on the secondary node.

Actions

Change your consistency mode, depending on your application's requirements.

IPC:AuroraWriteForwardExecute

The `IPC:AuroraWriteForwardExecute` event occurs when a backend process on the secondary DB cluster is waiting for a forwarded query to complete and obtain results from the writer node of the primary DB cluster.

Likely causes of increased waits

Common causes for increased waits include the following:

- Fetching a large number of rows from the primary Region's writer node.
- Increased network latency between the secondary node and primary Region's writer node increases the time it takes the secondary node to receive data from the writer node.
- Increased load on the secondary node can delay transmission of the query request from the secondary node to the primary Region's writer node.
- Increased load on the primary Region's writer node can delay transmission of data from the writer node to the secondary node.

Actions

We recommend different actions depending on the causes of your wait event.

- Optimize queries to retrieve only necessary data.
- Optimize data manipulation language (DML) operations to only modify necessary data.
- If the secondary node or primary Region's writer node is constrained by CPU or network bandwidth, consider changing it to an instance type with more CPU capacity or more network bandwidth.

IPC:AuroraWriteForwardGetGlobalConsistencyPoint

The `IPC:AuroraWriteForwardGetGlobalConsistencyPoint` event occurs when a backend process on the secondary DB cluster that's using the GLOBAL consistency mode is waiting to obtain the global consistency point from the writer node before executing a query.

Likely causes of increased waits

Common causes for increased waits include the following:

- Increased network latency between the secondary node and primary Region's writer node increases the time it takes the secondary node to receive data from the writer node.
- Increased load on the secondary node can delay transmission of the query request from the secondary node to the primary Region's writer node.
- Increased load on the primary Region's writer node can delay transmission of data from the writer node to the secondary node.

Actions

We recommend different actions depending on the causes of your wait event.

- Change your consistency mode, depending on your application's requirements.
- If the secondary node or primary Region's writer node is constrained by CPU or network bandwidth, consider changing it to an instance type with more CPU capacity or more network bandwidth.

IPC:AuroraWriteForwardXactAbort

The `IPC:AuroraWriteForwardXactAbort` event occurs when a backend process on the secondary DB cluster is waiting for the result of a remote cleanup query. Cleanup queries are issued to return the process to the appropriate state after a write-forwarded transaction is aborted. Amazon Aurora performs them either because an error was found or because an user issued an explicit `ABORT` command or cancelled a running query.

Likely causes of increased waits

Common causes for increased waits include the following:

- Increased network latency between the secondary node and primary Region's writer node increases the time it takes the secondary node to receive data from the writer node.
- Increased load on the secondary node can delay transmission of the cleanup query request from the secondary node to the primary Region's writer node.
- Increased load on the primary Region's writer node can delay transmission of data from the writer node to the secondary node.

Actions

We recommend different actions depending on the causes of your wait event.

- Investigate the cause of the aborted transaction.
- If the secondary node or primary Region's writer node is constrained by CPU or network bandwidth, consider changing it to an instance type with more CPU capacity or more network bandwidth.

IPC:AuroraWriteForwardXactCommit

The `IPC:AuroraWriteForwardXactCommit` event occurs when a backend process on the secondary DB cluster is waiting for the result of a forwarded commit transaction command.

Likely causes of increased waits

Common causes for increased waits include the following:

- Increased network latency between the secondary node and primary Region's writer node increases the time it takes the secondary node to receive data from the writer node.
- Increased load on the secondary node can delay transmission of the query request from the secondary node to the primary Region's writer node.
- Increased load on the primary Region's writer node can delay transmission of data from the writer node to the secondary node.

Actions

If the secondary node or primary Region's writer node is constrained by CPU or network bandwidth, consider changing it to an instance type with more CPU capacity or more network bandwidth.

IPC:AuroraWriteForwardXactStart

The `IPC:AuroraWriteForwardXactStart` event occurs when a backend process on the secondary DB cluster is waiting for the result of a forwarded start transaction command.

Likely causes of increased waits

Common causes for increased waits include the following:

- Increased network latency between the secondary node and primary Region's writer node increases the time it takes the secondary node to receive data from the writer node.
- Increased load on the secondary node can delay transmission of the query request from the secondary node to the primary Region's writer node.
- Increased load on the primary Region's writer node can delay transmission of data from the writer node to the secondary node.

Actions

If the secondary node or primary Region's writer node is constrained by CPU or network bandwidth, consider changing it to an instance type with more CPU capacity or more network bandwidth.

Using switchover or failover in an Amazon Aurora global database

An Aurora global database provides more business continuity and disaster recovery (BCDR) protection than the standard high availability provided by an Aurora DB cluster in a single AWS Region. By using an Aurora global database, you can plan for and recover from true Regional disasters or complete service-level outages quickly. Recovery from disaster is typically driven by the following two business objectives:

- **Recovery time objective (RTO)** – The time it takes a system to return to a working state after a disaster or service outage. In other words, RTO measures downtime. For an Aurora global database, RTO can be in the order of minutes.
- **Recovery point objective (RPO)** – The amount of data that can be lost (measured in time) after a disaster or service outage. This data loss is usually due to asynchronous replication lag. For an Aurora global database, RPO is typically measured in seconds. With an Aurora PostgreSQL–based global database, you can use the `rds.global_db_rpo` parameter to set and track the upper

bound on RPO, but doing so might affect transaction processing on the primary cluster's writer node. For more information, see Managing RPOs for Aurora PostgreSQL–based global databases.

Switching over or failing over an Aurora global database involves promoting a DB cluster in one of your global database's secondary Regions to be the primary DB cluster. The term "regional outage" is often used to describe a variety of failure scenarios. A worst case scenario could be a wide-spread outage from a catastrophic event that affects hundreds of square miles. However, most outages are much more localized, affecting only a small subset of cloud services or customer systems. Consider the full scope of the outage to make sure cross-Region failover is the proper solution and to choose the appropriate failover method for the situation. Whether you should use the failover or switchover approach depends on the specific outage scenario:

- **Failover –** Use this approach to recover from an unplanned outage. With this approach, you perform a cross-Region failover to one of the secondary DB clusters in your Aurora global database. The RPO for this approach is typically a non-zero value measured in seconds. The amount of data loss depends on the Aurora global database replication lag across the AWS Regions at the time of the failure. To learn more, see Recovering an Amazon Aurora global database from an unplanned outage.
- **Switchover –** This operation was previously called "managed planned failover." Use this approach for controlled scenarios, such as operational maintenance and other planned operational procedures. Because this feature synchronizes secondary DB clusters with the primary before making any other changes, RPO is 0 (no data loss). To learn more, see Performing switchovers for Amazon Aurora global databases.

> **Note**
>
> If you want to switch over or fail over to a headless secondary Aurora DB cluster, you need to first add a DB instance to it. For more information about headless DB clusters, see Creating a headless Aurora DB cluster in a secondary Region.

Topics

Recovering an Amazon Aurora global database from an unplanned outage

On very rare occasions, your Aurora global database might experience an unexpected outage in its primary AWS Region. If this happens, your primary Aurora DB cluster and its writer node aren't available, and the replication between the primary and secondary DB clusters stops. To minimize both downtime (RTO) and data loss (RPO), you can work quickly to perform a cross-Region failover.

There are two methods for failing over in a disaster recovery situation:

- Managed failover – This method is recommended for disaster recovery. When you use this method, Aurora automatically adds back the old primary Region to the global database as a secondary Region when it becomes available again. Thus, the original topology of your global cluster is maintained. To learn how to use this method, see Performing managed failovers for Aurora global databases.
- Manual failover – This alternative method can be used when managed failover isn't an option, for example, when your primary and secondary Regions are running incompatible engine versions. To learn how to use this method, see Performing manual failovers for Aurora global databases.

Important

Both failover methods can result in a loss of write transaction data that wasn't replicated to the chosen secondary before the failover event occurred. However, the recovery process that promotes a DB instance on the chosen secondary DB cluster to be the primary writer DB instance guarantees that the data is in a transactionally consistent state.

Performing managed failovers for Aurora global databases

This approach is intended for business continuity in the event of a true Regional disaster or complete service-level outage.

During a managed failover, your primary cluster is failed over to your choice of secondary Region while your Aurora global database's existing replication topology is maintained. The chosen secondary cluster promotes one of its read-only nodes to full writer status. This step allows the cluster to assume the role of primary cluster. Your database is unavailable for a short time while this cluster is assuming its new role. Data that wasn't replicated from the old primary to the chosen secondary cluster is missing when this secondary becomes the new primary.

> **Note**
>
> You can only perform a managed cross-Region database failover on an Aurora global database if the primary and secondary DB clusters have the same major, minor, and patch level engine versions. However, the patch levels can be different, depending on the minor engine version. For more information, see Patch level compatibility for managed cross-Region switchovers and failovers. If your engine versions are incompatible, you can perform the failover manually by following the steps in Performing manual failovers for Aurora global databases.

To minimize data loss, we recommend that you do the following before using this feature:

- Take applications offline to prevent writes from being sent to the primary cluster of Aurora global database.
- Check lag times for all secondary Aurora DB clusters in the Aurora global database. Choosing the secondary Region with the least replication lag can minimize data loss with the current failed primary Region. For all Aurora PostgreSQL-based global databases and for Aurora MySQL-based global databases starting with engine versions 3.04.0 and higher, or 2.12.0 and higher, use Amazon CloudWatch to view the `AuroraGlobalDBRPOLag` metric for all secondary DB clusters. For lower minor versions of Aurora MySQL-based global databases, view the `AuroraGlobalDBReplicationLag` metric instead. These metrics show you how far behind (in milliseconds) replication to a secondary cluster is to the primary DB cluster.

  For more information about CloudWatch metrics for Aurora, see Cluster-level metrics for Amazon Aurora.

During a managed failover, the chosen secondary DB cluster is promoted to its new role as primary. However, it doesn't inherit the various configuration options of the primary DB cluster. A mismatch in configuration can lead to performance issues, workload incompatibilities, and other anomalous behavior. To avoid such issues, we recommend that you resolve differences between your Aurora global database clusters for the following:

- **Configure Aurora DB cluster parameter group for the new primary, if necessary** – You can configure your Aurora DB cluster parameter groups independently for each Aurora cluster in your Aurora global database. Therefore, when you promote a secondary DB cluster to take over the primary role, the parameter group from the secondary might be configured differently than for the primary. If so, modify the promoted secondary DB cluster's parameter group to conform to your primary cluster's settings. To learn how, see Modifying parameters for an Aurora global database.

- **Configure monitoring tools and options, such as Amazon CloudWatch Events and alarms –** Configure the promoted DB cluster with the same logging ability, alarms, and so on as needed for the global database. As with parameter groups, configuration for these features isn't inherited from the primary during the failover process. Some CloudWatch metrics, such as replication lag, are only available for secondary Regions. Thus, a failover changes how to view those metrics and set alarms on them, and could require changes to any predefined dashboards. For more information about Aurora DB clusters and monitoring, see Overview of monitoring Amazon Aurora.
- **Configure integrations with other AWS services –** If your Aurora global database integrates with AWS services, such as AWS Secrets Manager, AWS Identity and Access Management, Amazon S3, and AWS Lambda, you need to make sure these are configured as needed. For more information about integrating Aurora global databases with IAM, Amazon S3 and Lambda, see Using Amazon Aurora global databases with other AWS services. To learn more about Secrets Manager, see How to automate replication of secrets in AWS Secrets Manager across AWS Regions.

Typically, the chosen secondary cluster assumes the primary role within a few minutes. As soon as the new primary Region's writer node is available, you can connect your applications to it and resume your workloads. After Aurora promotes the new primary cluster, it automatically rebuilds all additional secondary Region clusters.

Because Aurora global databases use asynchronous replication, the replication lag in each secondary Region can vary. Aurora rebuilds these secondary Regions to have the exact same point-in-time data as the new primary Region cluster. The duration of the complete rebuilding task can take a few minutes to several hours, depending on the size of the storage volume and the distance between the Regions. When the secondary Region clusters finish rebuilding from the new primary Region, they become available for read access.

As soon as the new primary writer is promoted and available, the new primary Region's cluster can handle read and write operations for the Aurora global database. Make sure to change the endpoint for your application to use the new endpoint. If you accepted the provided names when you created the Aurora global database, you can change the endpoint by removing the `-ro` from the promoted cluster's endpoint string in your application.

For example, the secondary cluster's endpoint `my-global.cluster-`**`ro`**`-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` becomes `my-global.cluster-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` when that cluster is promoted to primary.

If you are using RDS Proxy, make sure to redirect your application's write operations to the appropriate read/write endpoint of the proxy that's associated with the new primary cluster. This proxy endpoint might be the default endpoint or a custom read/write endpoint. For more information see How RDS Proxy endpoints work with global databases.

To restore the global database cluster's original topology, Aurora monitors the availability of the old primary Region. As soon as that Region is healthy and available again, Aurora automatically adds it back to the global cluster as a secondary Region. Before creating the new storage volume in the old primary Region, Aurora tries to take a snapshot of the old storage volume at the point of failure. It does this so that you can use it to recover any of the missing data. If this operation is successful, Aurora places this snapshot named "rds:unplanned-global-failover-*`name-of-old-primary-DB-cluster`*-*`timestamp`*" in the snapshot section of the AWS Management Console. You can also see this snapshot listed in the information returned by the DescribeDBClusterSnapshots API operation.

Note

The snapshot of the old storage volume is a system snapshot that's subject to the backup retention period configured on the old primary cluster. To preserve this snapshot outside of the retention period, you can copy it to save it as a manual snapshot. To learn more about copying snapshots, including pricing, see Copying a DB cluster snapshot.

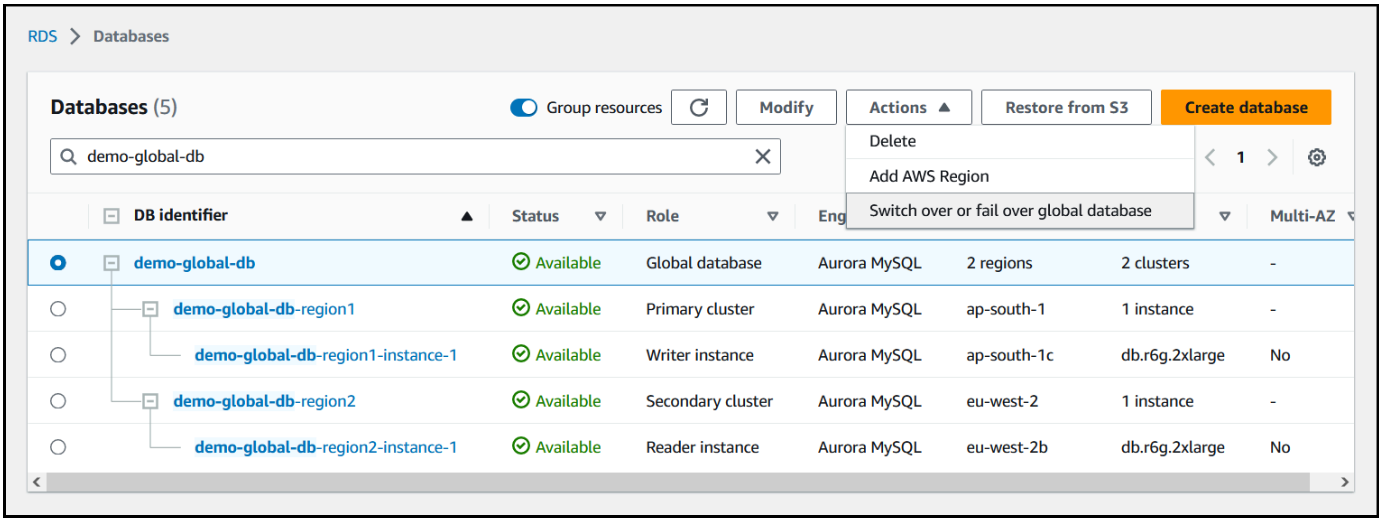
After the original topology is restored, you can fail back your global database to the original primary Region by performing a switchover operation when it makes the most sense for your business and workload. To do so, follow the steps in Performing switchovers for Amazon Aurora global databases.

You can fail over your Aurora global database using the AWS Management Console, the AWS CLI, or the RDS API.

Console

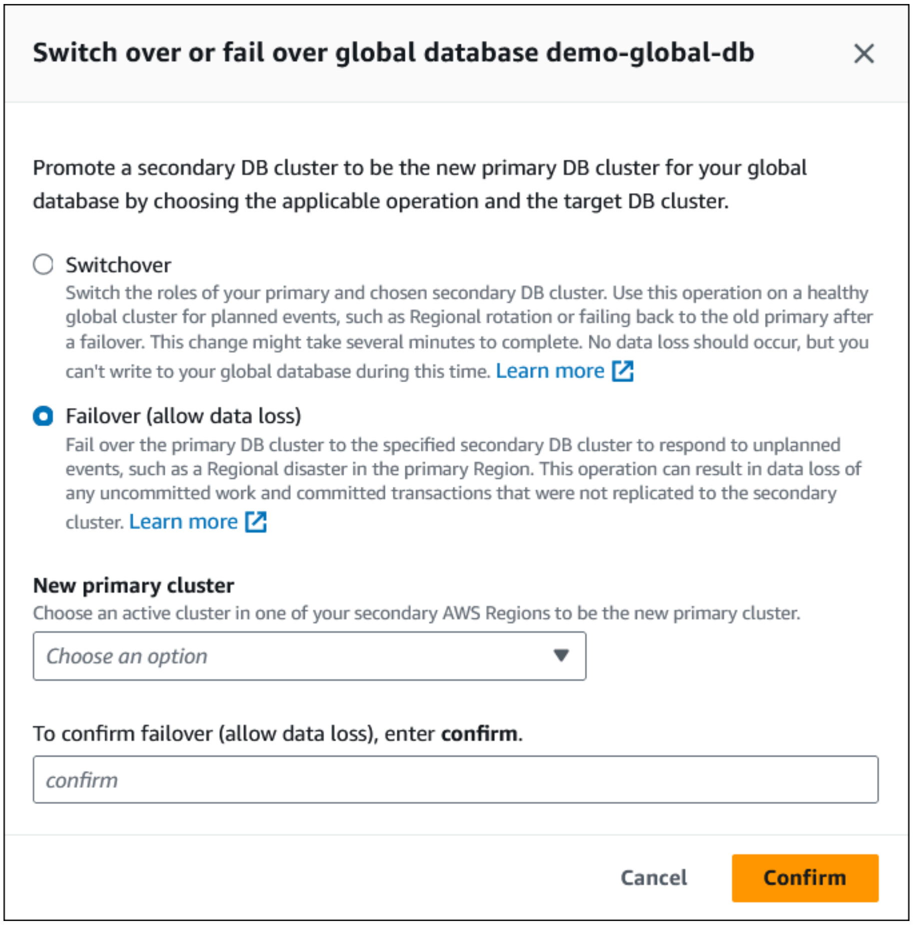
To perform the managed failover on your Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** and find the Aurora global database you want to fail over.
3. Choose **Switch over or fail over global database** from the **Actions** menu.



DB identifier	Status	Role	Eng			Multi-AZ
● demo-global-db	Available	Global database	Aurora MySQL	2 regions	2 clusters	-
○ demo-global-db-region1	Available	Primary cluster	Aurora MySQL	ap-south-1	1 instance	-
○ demo-global-db-region1-instance-1	Available	Writer instance	Aurora MySQL	ap-south-1c	db.r6g.2xlarge	No
○ demo-global-db-region2	Available	Secondary cluster	Aurora MySQL	eu-west-2	1 instance	-
○ demo-global-db-region2-instance-1	Available	Reader instance	Aurora MySQL	eu-west-2b	db.r6g.2xlarge	No

4. Choose **Failover (allow data loss)**.



Switch over or fail over global database demo-global-db

Promote a secondary DB cluster to be the new primary DB cluster for your global database by choosing the applicable operation and the target DB cluster.

○ Switchover
Switch the roles of your primary and chosen secondary DB cluster. Use this operation on a healthy global cluster for planned events, such as Regional rotation or failing back to the old primary after a failover. This change might take several minutes to complete. No data loss should occur, but you can't write to your global database during this time. Learn more

● Failover (allow data loss)
Fail over the primary DB cluster to the specified secondary DB cluster to respond to unplanned events, such as a Regional disaster in the primary Region. This operation can result in data loss of any uncommitted work and committed transactions that were not replicated to the secondary cluster. Learn more

New primary cluster
Choose an active cluster in one of your secondary AWS Regions to be the new primary cluster.

Choose an option

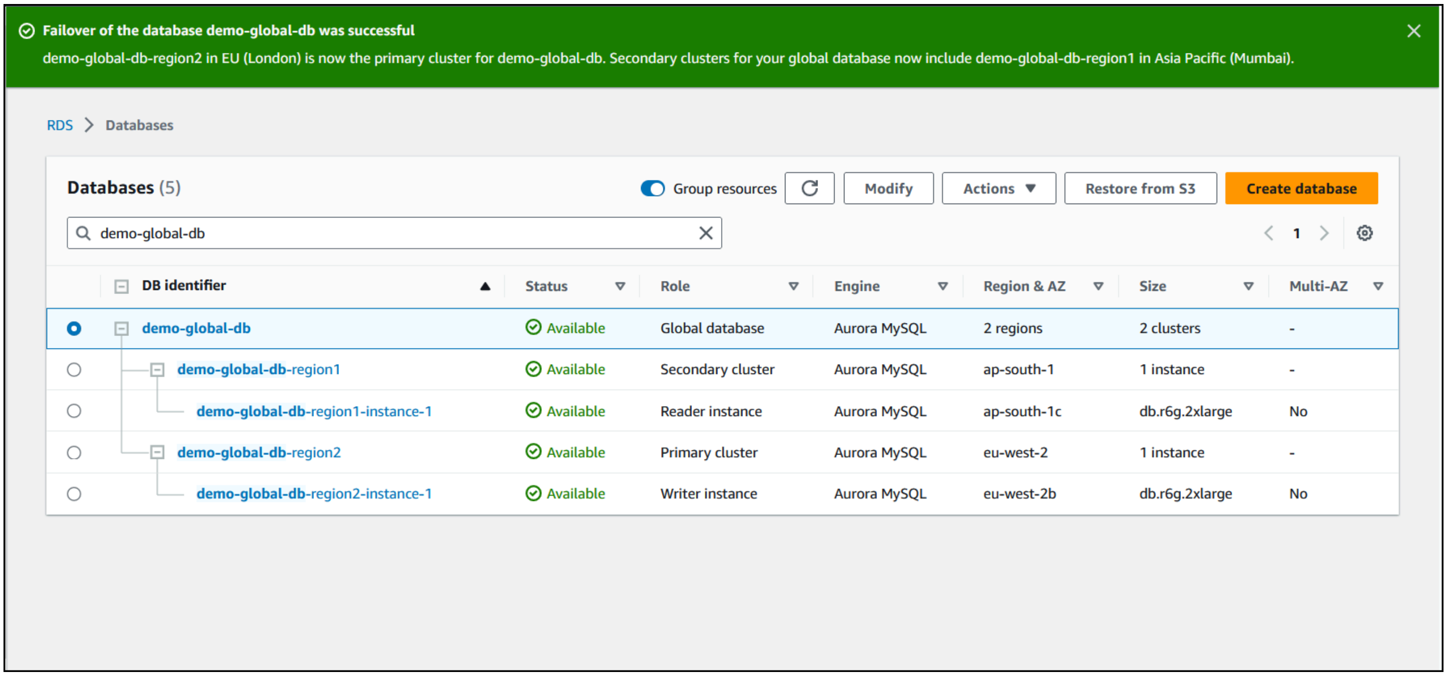
To confirm failover (allow data loss), enter **confirm**.

confirm

5. For **New primary cluster**, choose an active cluster in one of your secondary AWS Regions to be the new primary cluster.

6. Enter **`confirm`**, and then choose **Confirm**.

When the failover completes, you can see the Aurora DB clusters and their current state in the **Databases** list, as shown in the following image.



Failover of the database demo-global-db was successful

demo-global-db-region2 in EU (London) is now the primary cluster for demo-global-db. Secondary clusters for your global database now include demo-global-db-region1 in Asia Pacific (Mumbai).

Databases (5)

	DB identifier	Status	Role	Engine	Region & AZ	Size	Multi-AZ
●	demo-global-db	✓ Available	Global database	Aurora MySQL	2 regions	2 clusters	-
○	demo-global-db-region1	✓ Available	Secondary cluster	Aurora MySQL	ap-south-1	1 instance	-
○	demo-global-db-region1-instance-1	✓ Available	Reader instance	Aurora MySQL	ap-south-1c	db.r6g.2xlarge	No
○	demo-global-db-region2	✓ Available	Primary cluster	Aurora MySQL	eu-west-2	1 instance	-
○	demo-global-db-region2-instance-1	✓ Available	Writer instance	Aurora MySQL	eu-west-2b	db.r6g.2xlarge	No

AWS CLI

To perform the managed failover on an Aurora global database

Use the `failover-global-cluster` CLI command to fail over your Aurora global database. With the command, pass values for the following parameters.

- `--region` – Specify the AWS Region where the secondary DB cluster that you want to be the new primary for the Aurora global database is running.
- `--global-cluster-identifier` – Specify the name of your Aurora global database.
- `--target-db-cluster-identifier` – Specify the Amazon Resource Name (ARN) of the Aurora DB cluster that you want to promote to be the new primary for the Aurora global database.
- `--allow-data-loss` – Explicitly make this a failover operation instead of a switchover operation. A failover operation can result in some data loss if the asynchronous replication components haven't completed sending all replicated data to the secondary Region.

For Linux, macOS, or Unix:

```
aws rds --region region_of_selected_secondary \
   failover-global-cluster --global-cluster-identifier global_database_id \
   --target-db-cluster-identifier arn_of_secondary_to_promote \
    --allow-data-loss
```

For Windows:

```
aws rds --region region_of_selected_secondary ^
    failover-global-cluster --global-cluster-identifier global_database_id ^
    --target-db-cluster-identifier arn_of_secondary_to_promote ^
    --allow-data-loss
```

RDS API

To fail over an Aurora global database, run the FailoverGlobalCluster API operation.

Performing manual failovers for Aurora global databases

In some scenarios, you might not be able to use the managed failover process. One example is if your primary and secondary DB clusters aren't running compatible engine versions. In this case, you can follow this manual process to fail over your global database to your target secondary Region.

> **Tip**
>
> We recommend that you understand this process before using it. Have a plan ready to quickly proceed at the first sign of a Region-wide issue. You can be ready to identify the secondary Region with the least replication lag by using Amazon CloudWatch regularly to track lag times for the secondary clusters. Make sure to test your plan to check that your procedures are complete and accurate, and that staff are trained to perform a disaster recovery failover before it really happens.

To manually fail over to a secondary cluster after an unplanned outage in the primary Region

1. Stop issuing DML statements and other write operations to the primary Aurora DB cluster in the AWS Region with the outage.
2. Identify an Aurora DB cluster from a secondary AWS Region to use as a new primary DB cluster. If you have two or more secondary AWS Regions in your Aurora global database, choose the secondary cluster that has the least replication lag.
3. Detach your chosen secondary DB cluster from the Aurora global database.

   Removing a secondary DB cluster from an Aurora global database immediately stops the replication from the primary to this secondary and promotes it to a standalone provisioned

Aurora DB cluster with full read/write capabilities. Any other secondary Aurora DB clusters associated with the primary cluster in the Region with the outage are still available and can accept calls from your application. They also consume resources. Because you're recreating the Aurora global database, remove the other secondary DB clusters before creating the new Aurora global database in the following steps. Doing this avoids data inconsistencies among the DB clusters in the Aurora global database (*split-brain* issues).

For detailed steps for detaching, see Removing a cluster from an Amazon Aurora global database.

4. Reconfigure your application to send all write operations to this now standalone Aurora DB cluster using its new endpoint. If you accepted the provided names when you created the Aurora global database, you can change the endpoint by removing the `-ro` from the cluster's endpoint string in your application.

   For example, the secondary cluster's endpoint `my-global.cluster-`**`ro`**`-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` becomes `my-global.cluster-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` when that cluster is detached from the Aurora global database.

   This Aurora DB cluster becomes the primary cluster of a new Aurora global database when you start adding Regions to it in the next step.

   If you are using RDS Proxy, make sure to redirect your application's write operations to the appropriate read/write endpoint of the proxy that's associated with the new primary cluster. This proxy endpoint might be the default endpoint or a custom read/write endpoint. For more information see How RDS Proxy endpoints work with global databases.

5. Add an AWS Region to the DB cluster. When you do this, the replication process from primary to secondary begins. For detailed steps to add a Region, see Adding an AWS Region to an Amazon Aurora global database.

6. Add more AWS Regions as needed to recreate the topology needed to support your application.

Make sure that application writes are sent to the correct Aurora DB cluster before, during, and after making these changes. Doing this avoids data inconsistencies among the DB clusters in the Aurora global database (*split-brain* issues).

If you reconfigured in response to an outage in an AWS Region, you can make that AWS Region the primary again after the outage is resolved. To do so, you add the old AWS Region to your new global database, and then use the switchover process to switch its role. Your Aurora global database must use a version of Aurora PostgreSQL or Aurora MySQL that supports switchovers. For more information, see Performing switchovers for Amazon Aurora global databases.

Performing switchovers for Amazon Aurora global databases

> **Note**
>
> Switchovers were previously called "managed planned failovers."

By using switchovers, you can change the Region of your primary cluster on a routine basis. This approach is intended for controlled scenarios, such as operational maintenance and other planned operational procedures.

There are three common use cases for using switchovers.

- For "regional rotation" requirements imposed on specific industries. For example, financial service regulations might want tier-0 systems to switch to a different Region for several months to ensure that disaster recovery procedures are regularly exercised.
- For multi-Region "follow-the-sun" applications. For example, a business might want to provide lower latency writes in different Regions based on business hours across different time zones.
- As a zero-data-loss method to fail back to the original primary Region after a failover.

> **Note**
>
> Switchovers are designed to be used on a healthy Aurora global database. To recover from an unplanned outage, follow the appropriate procedure in Recovering an Amazon Aurora global database from an unplanned outage.
> To perform a switchover, your target secondary DB cluster must be running the exact same engine version as the primary, including the patch level, depending on the engine version. For more information, see Patch level compatibility for managed cross-Region switchovers and failovers. Before you begin the switchover, check the engine versions in your global cluster to make sure that they support managed cross-Region switchover, and upgrade them if needed.

During a switchover, Aurora switches over your primary cluster to your chosen secondary Region while it maintains your global database's existing replication topology. Before it starts the switchover process, Aurora waits for all secondary Region clusters to be fully synchronized with the primary Region cluster. Then, the DB cluster in the primary Region becomes read-only and the chosen secondary cluster promotes one of its read-only nodes to full writer status. Promoting this node to a writer allows that secondary cluster to assume the role of primary cluster. Because all secondary clusters were synchronized with the primary at the beginning of the process, the new primary continues operations for the Aurora global database without losing any data. Your database is unavailable for a short time while the primary and selected secondary clusters are assuming their new roles.

To optimize application availability, we recommend that you do the following before using this feature:

- Perform this operation during nonpeak hours or at another time when writes to the primary DB cluster are minimal.
- Take applications offline to prevent writes from being sent to the primary cluster of Aurora global database.
- Check lag times for all secondary Aurora DB clusters in the Aurora global database. For all Aurora PostgreSQL-based global databases and for Aurora MySQL-based global databases starting with engine versions 3.04.0 and higher or 2.12.0 and higher, use Amazon CloudWatch to view the `AuroraGlobalDBRPOLag` metric for all secondary DB clusters. For lower minor versions of Aurora MySQL-based global databases, view the `AuroraGlobalDBReplicationLag` metric instead. These metrics show you how far behind (in milliseconds) replication to a secondary cluster is to the primary DB cluster. This value is directly proportional to the time it takes for Aurora to complete the switchover. Therefore, the larger the lag value, the longer the switchover will take.

  For more information about CloudWatch metrics for Aurora, see [Cluster-level metrics for Amazon Aurora](#).

During a switchover, the chosen secondary DB cluster is promoted to its new role as primary. However, it doesn't inherit the various configuration options of the primary DB cluster. A mismatch in configuration can lead to performance issues, workload incompatibilities, and other anomalous behavior. To avoid such issues, we recommend that you resolve differences between your Aurora global database clusters for the following:

- **Configure Aurora DB cluster parameter group for the new primary, if necessary** – You can configure your Aurora DB cluster parameter groups independently for each Aurora cluster in your Aurora global database. That means that when you promote a secondary DB cluster to take over the primary role, the parameter group from the secondary might be configured differently than for the primary. If so, modify the promoted secondary DB cluster's parameter group to conform to your primary cluster's settings. To learn how, see Modifying parameters for an Aurora global database.
- **Configure monitoring tools and options, such as Amazon CloudWatch Events and alarms** – Configure the promoted DB cluster with the same logging ability, alarms, and so on as needed for the global database. As with parameter groups, configuration for these features isn't inherited from the primary during the switchover process. Some CloudWatch metrics, such as replication lag, are only available for secondary Regions. Thus, a switchover changes how to view those metrics and set alarms on them, and could require changes to any predefined dashboards. For more information about Aurora DB clusters and monitoring, see Overview of monitoring Amazon Aurora.
- **Configure integrations with other AWS services** – If your Aurora global database integrates with AWS services, such as AWS Secrets Manager, AWS Identity and Access Management, Amazon S3, and AWS Lambda, make sure to configure your integrations with these services as needed. For more information about integrating Aurora global databases with IAM, Amazon S3 and Lambda, see Using Amazon Aurora global databases with other AWS services. To learn more about Secrets Manager, see How to automate replication of secrets in AWS Secrets Manager across AWS Regions.

Note

Typically, the role switchover can take up to several minutes. However, building additional secondary clusters can take a few minutes to several hours, depending on the size of your database and the physical distance between the Regions.

When the switchover process completes, the promoted Aurora DB cluster can handle write operations for the Aurora global database. Make sure to change the endpoint for your application to use the new endpoint. If you accepted the provided names when you created the Aurora global database, you can change the endpoint by removing the `-ro` from the promoted cluster's endpoint string in your application.

For example, the secondary cluster's endpoint `my-global.cluster-`**`ro`**`-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` becomes `my-global.cluster-aaaaaabbbbbb.us-west-1.rds.amazonaws.com` when that cluster is promoted to primary.

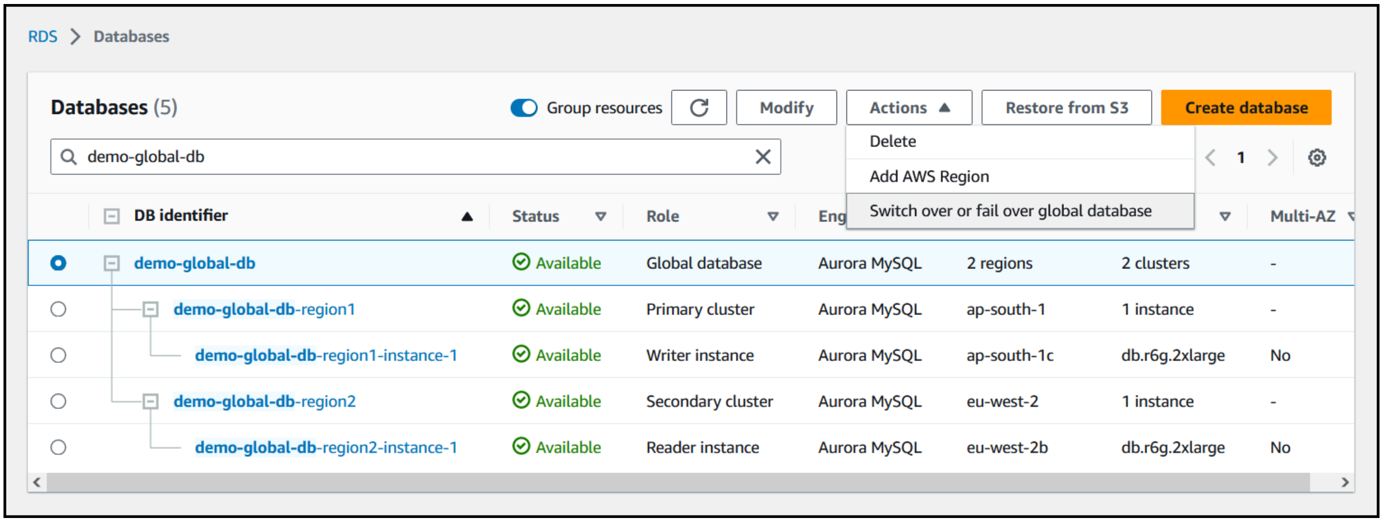
If you are using RDS Proxy, make sure to redirect your application's write operations to the appropriate read/write endpoint of the proxy that's associated with the new primary cluster. This proxy endpoint might be the default endpoint or a custom read/write endpoint. For more information see How RDS Proxy endpoints work with global databases.

You can switch over your Aurora global database using the AWS Management Console, the AWS CLI, or the RDS API.

Console

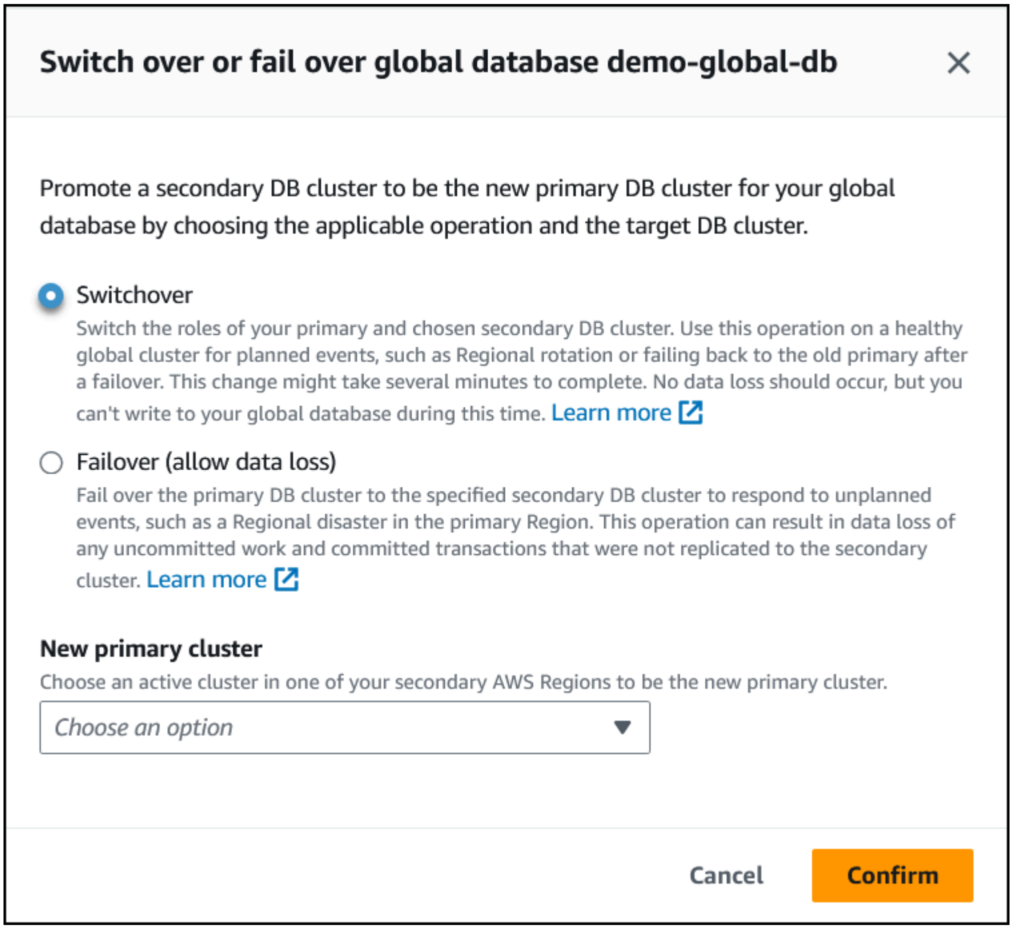
To perform the switchover on your Aurora global database

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** and find the Aurora global database you want to switch over.
3. Choose **Switch over or fail over global database** from the **Actions** menu.



	DB identifier	Status	Role	Eng			Multi-AZ
●	demo-global-db	Available	Global database	Aurora MySQL	2 regions	2 clusters	-
○	demo-global-db-region1	Available	Primary cluster	Aurora MySQL	ap-south-1	1 instance	-
○	demo-global-db-region1-instance-1	Available	Writer instance	Aurora MySQL	ap-south-1c	db.r6g.2xlarge	No
○	demo-global-db-region2	Available	Secondary cluster	Aurora MySQL	eu-west-2	1 instance	-
○	demo-global-db-region2-instance-1	Available	Reader instance	Aurora MySQL	eu-west-2b	db.r6g.2xlarge	No

4. Choose **Switchover**.



Switch over or fail over global database demo-global-db

Promote a secondary DB cluster to be the new primary DB cluster for your global database by choosing the applicable operation and the target DB cluster.

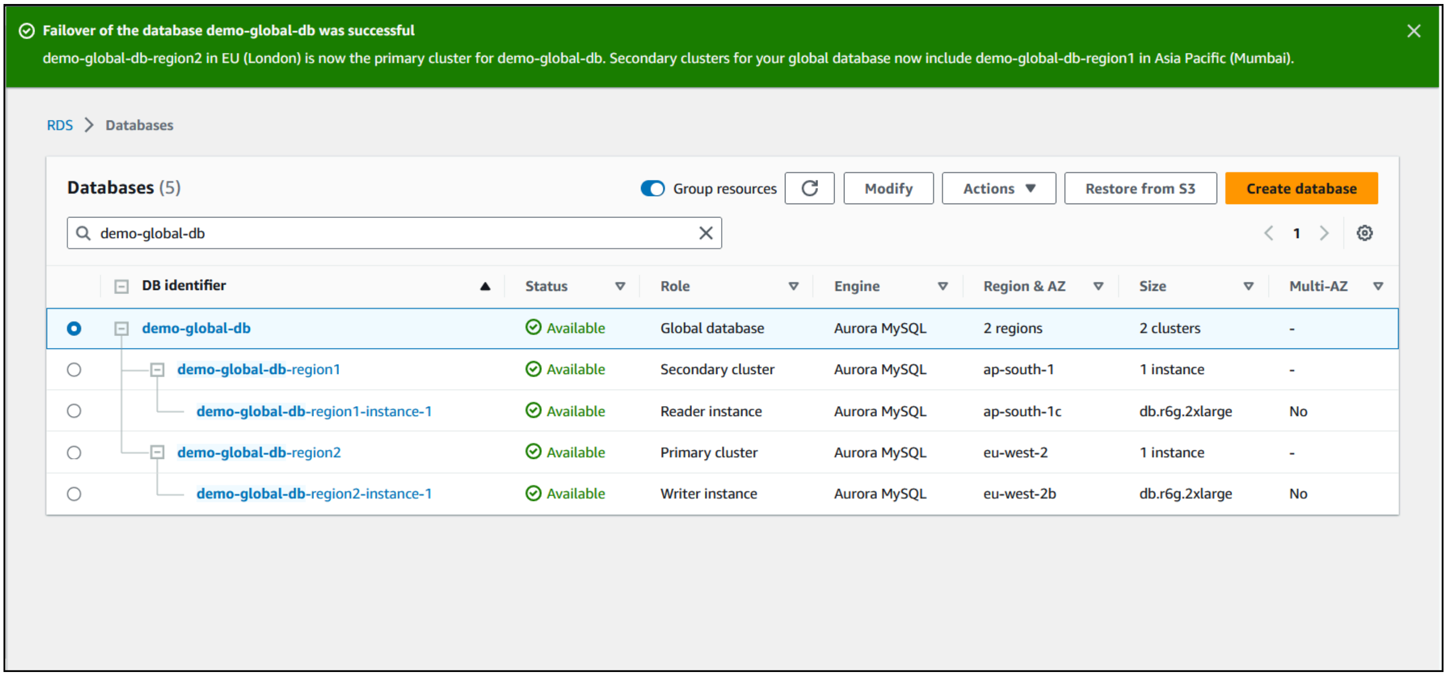
● Switchover
Switch the roles of your primary and chosen secondary DB cluster. Use this operation on a healthy global cluster for planned events, such as Regional rotation or failing back to the old primary after a failover. This change might take several minutes to complete. No data loss should occur, but you can't write to your global database during this time. Learn more

○ Failover (allow data loss)
Fail over the primary DB cluster to the specified secondary DB cluster to respond to unplanned events, such as a Regional disaster in the primary Region. This operation can result in data loss of any uncommitted work and committed transactions that were not replicated to the secondary cluster. Learn more

New primary cluster
Choose an active cluster in one of your secondary AWS Regions to be the new primary cluster.
Choose an option

5. For **New primary cluster**, choose an active cluster in one of your secondary AWS Regions to be the new primary cluster.
6. Choose **Confirm**.

When the switchover completes, you can see the Aurora DB clusters and their current roles in the **Databases** list, as shown in the following image.



Failover of the database demo-global-db was successful

demo-global-db-region2 in EU (London) is now the primary cluster for demo-global-db. Secondary clusters for your global database now include demo-global-db-region1 in Asia Pacific (Mumbai).

	DB identifier	Status	Role	Engine	Region & AZ	Size	Multi-AZ
●	demo-global-db	Available	Global database	Aurora MySQL	2 regions	2 clusters	-
○	demo-global-db-region1	Available	Secondary cluster	Aurora MySQL	ap-south-1	1 instance	-
○	demo-global-db-region1-instance-1	Available	Reader instance	Aurora MySQL	ap-south-1c	db.r6g.2xlarge	No
○	demo-global-db-region2	Available	Primary cluster	Aurora MySQL	eu-west-2	1 instance	-
○	demo-global-db-region2-instance-1	Available	Writer instance	Aurora MySQL	eu-west-2b	db.r6g.2xlarge	No

AWS CLI

To perform the switchover on an Aurora global database

Use the `switchover-global-cluster` CLI command to switch over your Aurora global database. With the command, pass values for the following parameters.

- `--region` – Specify the AWS Region where the primary DB cluster of the Aurora global database is running.
- `--global-cluster-identifier` – Specify the name of your Aurora global database.
- `--target-db-cluster-identifier` – Specify the Amazon Resource Name (ARN) of the Aurora DB cluster that you want to promote to be the primary for the Aurora global database.

For Linux, macOS, or Unix:

```
aws rds --region region_of_primary \
   switchover-global-cluster --global-cluster-identifier global_database_id \
  --target-db-cluster-identifier arn_of_secondary_to_promote
```

For Windows:

```
aws rds --region region_of_primary ^
   switchover-global-cluster --global-cluster-identifier global_database_id ^
```

```
    --target-db-cluster-identifier arn_of_secondary_to_promote
```

RDS API

To switch over an Aurora global database, run the SwitchoverGlobalCluster API operation.

Managing RPOs for Aurora PostgreSQL–based global databases

With an Aurora PostgreSQL–based global database, you can manage the recovery point objective (RPO) for your Aurora global database by using the `rds.global_db_rpo` parameter. RPO represents the maximum amount of data that can be lost in the event of an outage.

When you set an RPO for your Aurora PostgreSQL–based global database, Aurora monitors the *RPO lag time* of all secondary clusters to make sure that at least one secondary cluster stays within the target RPO window. RPO lag time is another time-based metric.

The RPO is used when your database resumes operations in a new AWS Region after a failover. Aurora evaluates RPO and RPO lag times to commit (or block) transactions on the primary as follows:

- Commits the transaction if at least one secondary DB cluster has an RPO lag time less than the RPO.
- Blocks the transaction if all secondary DB clusters have RPO lag times that are larger than the RPO. It also logs the event to the PostgreSQL log file and emits "wait" events that show the blocked sessions.

In other words, if all secondary clusters are behind the target RPO, Aurora pauses transactions on the primary cluster until at least one of the secondary clusters catches up. Paused transactions are resumed and committed as soon as the lag time of at least one secondary DB cluster becomes less than the RPO. The result is that no transactions can commit until the RPO is met.

The `rds.global_db_rpo` parameter is dynamic. If you decide that you don't want all write transactions to stall until the lag decreases sufficiently, you can reset it quickly. In this case, Aurora recognizes and implements the change after a short delay.

> **Important**
>
> In a global database with only two Regions, we recommend keeping the `rds.global_db_rpo` parameter's default value in the secondary Region's parameter

group. Otherwise, failing over to this Region due to a loss of the primary Region could cause Aurora to pause transactions. Instead, wait until Aurora completes rebuilding the cluster in the old failed Region before changing this parameter to enforce a maximum RPO.

If you set this parameter as outlined in the following, you can then also monitor the metrics that it generates. You can do so by using `psql` or another tool to query the Aurora global database's primary DB cluster and obtain detailed information about your Aurora PostgreSQL–based global database's operations. To learn how, see Monitoring Aurora PostgreSQL-based global databases.

Topics

- Setting the recovery point objective
- Viewing the recovery point objective
- Disabling the recovery point objective

Setting the recovery point objective

The `rds.global_db_rpo` parameter controls the RPO setting for a PostgreSQL database. This parameter is supported by Aurora PostgreSQL. Valid values for `rds.global_db_rpo` range from 20 seconds to 2,147,483,647 seconds (68 years). Choose a realistic value to meet your business need and use case. For example, you might want to allow up to 10 minutes for your RPO, in which case you set the value to 600.

You can set this value for your Aurora PostgreSQL–based global database by using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To set the RPO

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose the primary cluster of your Aurora global database and open the **Configuration** tab to find its DB cluster parameter group. For example, the default parameter group for a primary DB cluster running Aurora PostgreSQL 11.7 is `default.aurora-postgresql11`.

   Parameter groups can't be edited directly. Instead, you do the following:

- Create a custom DB cluster parameter group using the appropriate default parameter group as the starting point. For example, create a custom DB cluster parameter group based on the `default.aurora-postgresql11`.
- On your custom DB parameter group, set the value of the **rds.global_db_rpo** parameter to meet your use case. Valid values range from 20 seconds up to the maximum integer value of 2,147,483,647 (68 years).
- Apply the modified DB cluster parameter group to your Aurora DB cluster.

For more information, see Modifying parameters in a DB cluster parameter group.

AWS CLI

To set the `rds.global_db_rpo` parameter, use the modify-db-cluster-parameter-group CLI command. In the command, specify the name of your primary cluster's parameter group and values for RPO parameter.

The following example sets the RPO to 600 seconds (10 minutes) for the primary DB cluster's parameter group named `my_custom_global_parameter_group`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name my_custom_global_parameter_group \
    --parameters
 "ParameterName=rds.global_db_rpo,ParameterValue=600,ApplyMethod=immediate"
```

For Windows:

```
aws rds modify-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name my_custom_global_parameter_group ^
    --parameters
 "ParameterName=rds.global_db_rpo,ParameterValue=600,ApplyMethod=immediate"
```

RDS API

To modify the `rds.global_db_rpo` parameter, use the Amazon RDS ModifyDBClusterParameterGroup API operation.

Viewing the recovery point objective

The recovery point objective (RPO) of a global database is stored in the `rds.global_db_rpo` parameter for each DB cluster. You can connect to the endpoint for the secondary cluster you want to view and use `psql` to query the instance for this value.

```
db-name=>show rds.global_db_rpo;
```

If this parameter isn't set, the query returns the following:

```
rds.global_db_rpo
-------------------
 -1
(1 row)
```

This next response is from a secondary DB cluster that has 1 minute RPO setting.

```
rds.global_db_rpo
-------------------
 60
(1 row)
```

You can also use the CLI to get values for find out if `rds.global_db_rpo` is active on any of the Aurora DB clusters by using the CLI to get values of all `user` parameters for the cluster.

For Linux, macOS, or Unix:

```
aws rds describe-db-cluster-parameters \
 --db-cluster-parameter-group-name lab-test-apg-global \
 --source user
```

For Windows:

```
aws rds describe-db-cluster-parameters ^
 --db-cluster-parameter-group-name lab-test-apg-global *
 --source user
```

The command returns output similar to the following for all `user` parameters. that aren't `default-engine` or `system` DB cluster parameters.

```
{
    "Parameters": [
        {
            "ParameterName": "rds.global_db_rpo",
            "ParameterValue": "60",
            "Description": "(s) Recovery point objective threshold, in seconds, that
 blocks user commits when it is violated.",
            "Source": "user",
            "ApplyType": "dynamic",
            "DataType": "integer",
            "AllowedValues": "20-2147483647",
            "IsModifiable": true,
            "ApplyMethod": "immediate",
            "SupportedEngineModes": [
                "provisioned"
            ]
        }
    ]
}
```

To learn more about viewing parameters of the cluster parameter group, see Viewing parameter values for a DB cluster parameter group.

Disabling the recovery point objective

To disable the RPO, reset the `rds.global_db_rpo` parameter. You can reset parameters using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To disable the RPO

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose your primary DB cluster parameter group.
4. Choose **Edit parameters**.
5. Choose the box next to the **rds.global_db_rpo** parameter.
6. Choose **Reset**.
7. When the screen shows **Reset parameters in DB parameter group**, choose **Reset parameters**.

For more information on how to reset a parameter with the console, see Modifying parameters in a DB cluster parameter group.

AWS CLI

To reset the `rds.global_db_rpo` parameter, use the reset-db-cluster-parameter-group command.

For Linux, macOS, or Unix:

```
aws rds reset-db-cluster-parameter-group \
    --db-cluster-parameter-group-name global_db_cluster_parameter_group \
    --parameters "ParameterName=rds.global_db_rpo,ApplyMethod=immediate"
```

For Windows:

```
aws rds reset-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name global_db_cluster_parameter_group ^
    --parameters "ParameterName=rds.global_db_rpo,ApplyMethod=immediate"
```

RDS API

To reset the `rds.global_db_rpo` parameter, use the Amazon RDS API ResetDBClusterParameterGroup operation.

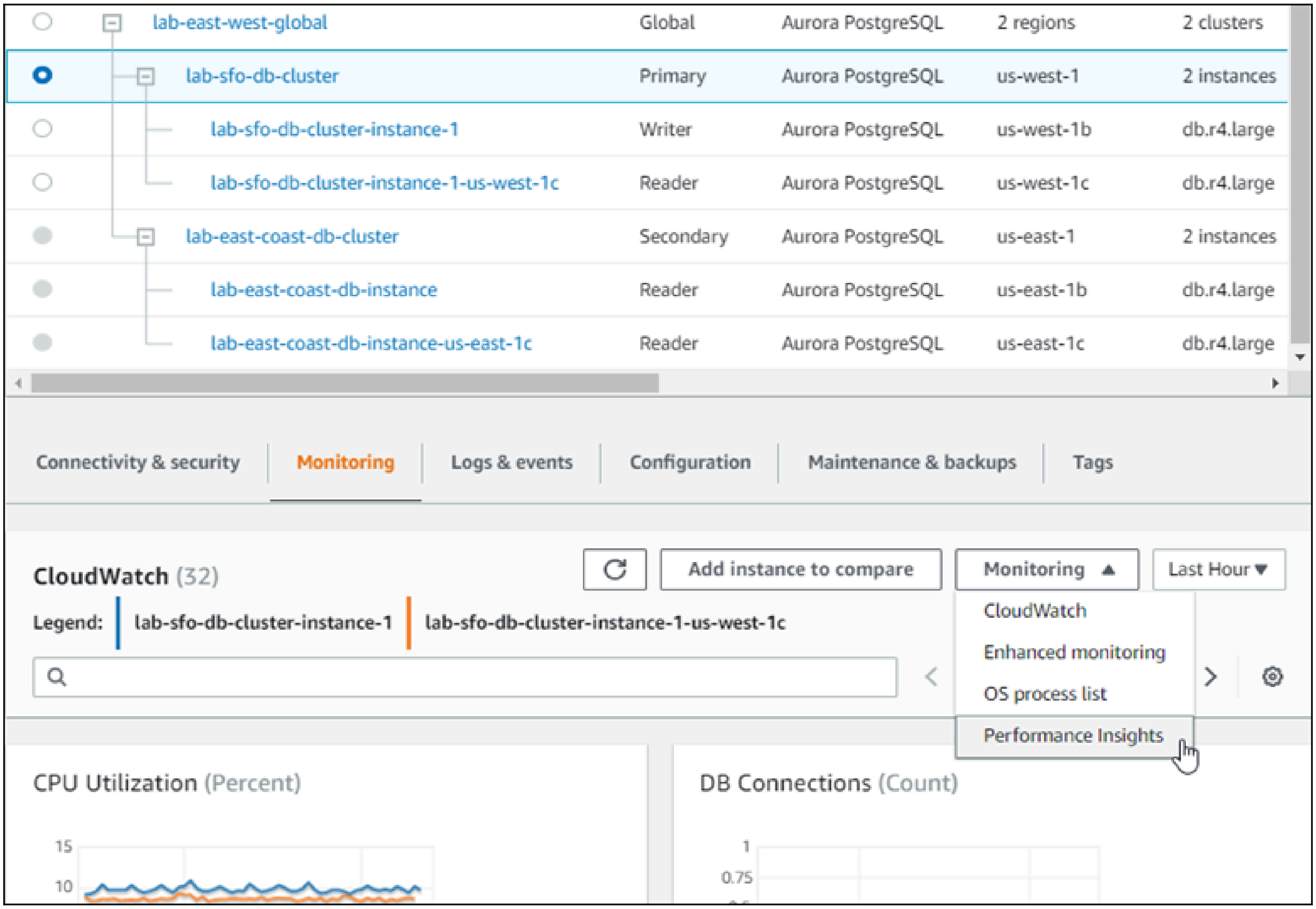
Monitoring an Amazon Aurora global database

When you create the Aurora DB clusters that make up your Aurora global database, you can choose many options that let you monitor your DB cluster's performance. These options include the following:

- Amazon RDS Performance Insights – Enables performance schema in the underlying Aurora database engine. To learn more about Performance Insights and Aurora global databases, see Monitoring an Amazon Aurora global database with Amazon RDS Performance Insights.
- Enhanced monitoring – Generates metrics for process or thread utilization on the CPU. To learn about enhanced monitoring, see Monitoring OS metrics with Enhanced Monitoring.
- Amazon CloudWatch Logs – Publishes specified log types to CloudWatch Logs. Error logs are published by default, but you can choose other logs specific to your Aurora database engine.

  - For Aurora MySQL–based Aurora DB clusters, you can export the audit log, general log, and slow query log.
  - For Aurora PostgreSQL–based Aurora DB clusters, you can export the PostgreSQL log.
- For Aurora MySQL–based global databases, you can query specific `information_schema` tables to check the status of your Aurora global database and its instances. To learn how, see Monitoring Aurora MySQL-based global databases.
- For Aurora PostgreSQL–based global databases, you can use specific functions to check the status of your Aurora global database and its instances. To learn how, see Monitoring Aurora PostgreSQL-based global databases.

The following screenshot shows some of the options available on the Monitoring tab of a primary Aurora DB cluster in an Aurora global database.



○	lab-east-west-global	Global	Aurora PostgreSQL	2 regions	2 clusters
●	lab-sfo-db-cluster	Primary	Aurora PostgreSQL	us-west-1	2 instances
○	lab-sfo-db-cluster-instance-1	Writer	Aurora PostgreSQL	us-west-1b	db.r4.large
○	lab-sfo-db-cluster-instance-1-us-west-1c	Reader	Aurora PostgreSQL	us-west-1c	db.r4.large
○	lab-east-coast-db-cluster	Secondary	Aurora PostgreSQL	us-east-1	2 instances
○	lab-east-coast-db-instance	Reader	Aurora PostgreSQL	us-east-1b	db.r4.large
○	lab-east-coast-db-instance-us-east-1c	Reader	Aurora PostgreSQL	us-east-1c	db.r4.large

For more information, see Monitoring metrics in an Amazon Aurora cluster.

Monitoring an Amazon Aurora global database with Amazon RDS Performance Insights

You can use Amazon RDS Performance Insights for your Aurora global databases. You enable this feature individually, for each Aurora DB cluster in your Aurora global database. To do so, you choose **Enable Performance Insights** in the **Additional configuration** section of the Create database page. Or you can modify your Aurora DB clusters to use this feature after they are up and running. You can enable or turn off Performance Insights for each cluster that's part of your Aurora global database.

The reports created by Performance Insights apply to each cluster in the global database. When you add a new secondary AWS Region to an Aurora global database that's already using Performance Insights, be sure that you enable Performance Insights in the newly added cluster. It doesn't inherit the Performance Insights setting from the existing global database.

You can switch AWS Regions while viewing the Performance Insights page for a DB instance that's attached to a global database. However, you might not see performance information immediately after switching AWS Regions. Although the DB instances might have identical names in each AWS Region, the associated Performance Insights URL is different for each DB instance. After switching AWS Regions, choose the name of the DB instance again in the Performance Insights navigation pane.

For DB instances associated with a global database, the factors affecting performance might be different in each AWS Region. For example, the DB instances in each AWS Region might have different capacity.

To learn more about using Performance Insights, see Monitoring DB load with Performance Insights on Amazon Aurora.

Monitoring Aurora global databases with Database Activity Streams

By using the Database Activity Streams feature, you can monitor and set alarms for auditing activity in the DB clusters in your global database. You start a database activity stream on each DB cluster separately. Each cluster delivers audit data to its own Kinesis stream within its own AWS Region. For more information, see Monitoring Amazon Aurora with Database Activity Streams.

Monitoring Aurora MySQL-based global databases

To view the status of an Aurora MySQL-based global database, query the information_schema.aurora_global_db_status and information_schema.aurora_global_db_instance_status tables.

> **Note**
>
> The `information_schema.aurora_global_db_status` and `information_schema.aurora_global_db_instance_status` tables are only available with Aurora MySQL version 3.04.0 and higher global databases.

To monitor an Aurora MySQL-based global database

1. Connect to the global database primary cluster endpoint using a MySQL client. For more information about how to connect, see Connecting to an Amazon Aurora global database.
2. Query the `information_schema.aurora_global_db_status` table in a mysql command to list the primary and secondary volumes. This query returns the lag times of the global database secondary DB clusters, as in the following example.

   ```
   mysql> select * from information_schema.aurora_global_db_status;
   ```

   ```
   AWS_REGION | HIGHEST_LSN_WRITTEN | DURABILITY_LAG_IN_MILLISECONDS |
    RPO_LAG_IN_MILLISECONDS | LAST_LAG_CALCULATION_TIMESTAMP | OLDEST_READ_VIEW_TRX_ID
   -----------+---------------------+---------------------------------
   +------------------------+---------------------------------
   +------------------------
   us-east-1  |           183537946 |                              0   |
         0 |  1970-01-01 00:00:00.000000     |               0
   us-west-2  |           183537944 |                              428 |
         0 |  2023-02-18 01:26:41.925000     |               20806982
   (2 rows)
   ```

   The output includes a row for each DB cluster of the global database containing the following columns:

   - **AWS_REGION** – The AWS Region that this DB cluster is in. For tables listing AWS Regions by engine, see Regions and Availability Zones.

- **HIGHEST_LSN_WRITTEN** – The highest log sequence number (LSN) currently written on this DB cluster.

  A *log sequence number (LSN)* is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
- **DURABILITY_LAG_IN_MILLISECONDS** – The difference in the timestamp values between the HIGHEST_LSN_WRITTEN on a secondary DB cluster and the HIGHEST_LSN_WRITTEN on the primary DB cluster. This value is always 0 on the primary DB cluster of the Aurora global database.
- **RPO_LAG_IN_MILLISECONDS** – The recovery point objective (RPO) lag. The RPO lag is the time it takes for the most recent user transaction COMMIT to be stored on a secondary DB cluster after it's been stored on the primary DB cluster of the Aurora global database. This value is always 0 on the primary DB cluster of the Aurora global database.

  In simple terms, this metric calculates the recovery point objective for each Aurora MySQL DB cluster in the Aurora global database, that is, how much data might be lost if there were an outage. As with lag, RPO is measured in time.
- **LAST_LAG_CALCULATION_TIMESTAMP** – The timestamp that specifies when values were last calculated for DURABILITY_LAG_IN_MILLISECONDS and RPO_LAG_IN_MILLISECONDS. A time value such as 1970-01-01 00:00:00+00 means this is the primary DB cluster.
- **OLDEST_READ_VIEW_TRX_ID** – The ID of the oldest transaction that the writer DB instance can purge to.

3. Query the information_schema.aurora_global_db_instance_status table to list all secondary DB instances for both the primary DB cluster and the secondary DB clusters.

```
mysql> select * from information_schema.aurora_global_db_instance_status;
```

```
SERVER_ID            |              SESSION_ID              | AWS_REGION
 | DURABLE_LSN | HIGHEST_LSN_RECEIVED | OLDEST_READ_VIEW_TRX_ID |
 OLDEST_READ_VIEW_LSN | VISIBILITY_LAG_IN_MSEC
---------------------+--------------------------------------+------------
+-------------+----------------------+------------------------
+----------------------+------------------------
```

```
ams-gdb-primary-i2   | MASTER_SESSION_ID                    | us-east-1  |
 183537698   |                    0 |                       0 |
 0 |                      0
ams-gdb-secondary-i1 | cc43165b-bdc6-4651-abbf-4f74f08bf931 | us-west-2  |
 183537689   |            183537692 |                20806928 |
 183537682 |                      0
ams-gdb-secondary-i2 | 53303ff0-70b5-411f-bc86-28d7a53f8c19 | us-west-2  |
 183537689   |            183537692 |                20806928 |
 183537682 |                    677
ams-gdb-primary-i1   | 5af1e20f-43db-421f-9f0d-2b92774c7d02 | us-east-1  |
 183537697   |            183537698 |                20806930 |
 183537691 |                     21
(4 rows)
```

The output includes a row for each DB instance of the global database containing the following columns:

- **SERVER_ID** – The server identifier for the DB instance.
- **SESSION_ID** – A unique identifier for the current session. A value of `MASTER_SESSION_ID` identifies the Writer (primary) DB instance.
- **AWS_REGION** – The AWS Region that this DB instance is in. For tables listing AWS Regions by engine, see [Regions and Availability Zones](#).
- **DURABLE_LSN** – The LSN made durable in storage.
- **HIGHEST_LSN_RECEIVED** – The highest LSN received by the DB instance from the writer DB instance.
- **OLDEST_READ_VIEW_TRX_ID** – The ID of the oldest transaction that the writer DB instance can purge to.
- **OLDEST_READ_VIEW_LSN** – The oldest LSN used by the DB instance to read from storage.
- **VISIBILITY_LAG_IN_MSEC** – For readers in the primary DB cluster, how far this DB instance is lagging behind the writer DB instance in milliseconds. For readers in a secondary DB cluster, how far this DB instance is lagging behind the secondary volume in milliseconds.

To see how these values change over time, consider the following transaction block where a table insert takes an hour.

```
mysql> BEGIN;
mysql> INSERT INTO table1 SELECT Large_Data_That_Takes_1_Hr_To_Insert;
mysql> COMMIT;
```

In some cases, there might be a network disconnect between the primary DB cluster and the secondary DB cluster after the BEGIN statement. If so, the secondary DB cluster's **DURABILITY_LAG_IN_MILLISECONDS** value starts increasing. At the end of the INSERT statement, the **DURABILITY_LAG_IN_MILLISECONDS** value is 1 hour. However, the **RPO_LAG_IN_MILLISECONDS** value is 0 because all the user data committed between the primary DB cluster and secondary DB cluster are still the same. As soon as the COMMIT statement completes, the **RPO_LAG_IN_MILLISECONDS** value increases.

Monitoring Aurora PostgreSQL-based global databases

To view the status of an Aurora PostgreSQL-based global database, use the aurora_global_db_status and aurora_global_db_instance_status functions.

Note

Only Aurora PostgreSQL supports the aurora_global_db_status and aurora_global_db_instance_status functions.

To monitor an Aurora PostgreSQL-based global database

1. Connect to the global database primary cluster endpoint using a PostgreSQL utility such as psql. For more information about how to connect, see Connecting to an Amazon Aurora global database.
2. Use the aurora_global_db_status function in a psql command to list the primary and secondary volumes. This shows the lag times of the global database secondary DB clusters.

```
postgres=> select * from aurora_global_db_status();
```

```
aws_region | highest_lsn_written | durability_lag_in_msec | rpo_lag_in_msec |
 last_lag_calculation_time  | feedback_epoch | feedback_xmin
------------+---------------------+------------------------+-----------------
+----------------------------+----------------+---------------
us-east-1  |         93763984222 |                     -1 |              -1 |
 1970-01-01 00:00:00+00     |              0 |             0
us-west-2  |         93763984222 |                    900 |            1090 |
 2020-05-12 22:49:14.328+00 |              2 |    3315479243
```

```
(2 rows)
```

The output includes a row for each DB cluster of the global database containing the following columns:

- **aws_region** – The AWS Region that this DB cluster is in. For tables listing AWS Regions by engine, see Regions and Availability Zones.
- **highest_lsn_written** – The highest log sequence number (LSN) currently written on this DB cluster.

  A *log sequence number (LSN)* is a unique sequential number that identifies a record in the database transaction log. LSNs are ordered such that a larger LSN represents a later transaction.
- **durability_lag_in_msec** – The timestamp difference between the highest log sequence number written on a secondary DB cluster (`highest_lsn_written`) and the `highest_lsn_written` on the primary DB cluster.
- **rpo_lag_in_msec** – The recovery point objective (RPO) lag. This lag is the time difference between the most recent user transaction commit stored on a secondary DB cluster and the most recent user transaction commit stored on the primary DB cluster.
- **last_lag_calculation_time** – The timestamp when values were last calculated for `durability_lag_in_msec` and `rpo_lag_in_msec`.
- **feedback_epoch** – The epoch a secondary DB cluster uses when it generates hot standby information.

  Hot standby is when a DB cluster can connect and query while the server is in recovery or standby mode. Hot standby feedback is information about the DB cluster when it's in hot standby. For more information, see Hot standby in the PostgreSQL documentation.
- **feedback_xmin** – The minimum (oldest) active transaction ID used by a secondary DB cluster.

3. Use the `aurora_global_db_instance_status` function to list all secondary DB instances for both the primary DB cluster and secondary DB clusters.

```
postgres=> select * from aurora_global_db_instance_status();
```

```
server_id                                   |              session_id
  | aws_region | durable_lsn | highest_lsn_rcvd | feedback_epoch | feedback_xmin |
 oldest_read_view_lsn | visibility_lag_in_msec
--------------------------------------------+--------------------------------------
+------------+-------------+------------------+----------------+--------------
+----------------------+-----------------------
apg-global-db-rpo-mammothrw-elephantro-1-n1 | MASTER_SESSION_ID
 | us-east-1  | 93763985102 |                  |                |               |
                    |
apg-global-db-rpo-mammothrw-elephantro-1-n2 | f38430cf-6576-479a-b296-dc06b1b1964a
 | us-east-1  | 93763985099 |      93763985102 |              2 |    3315479243 |
        93763985095 |                     10
apg-global-db-rpo-elephantro-mammothrw-n1   | 0d9f1d98-04ad-4aa4-8fdd-e08674cbbbfe
 | us-west-2  | 93763985095 |      93763985099 |              2 |    3315479243 |
        93763985089 |                   1017
(3 rows)
```

The output includes a row for each DB instance of the global database containing the following columns:

- **server_id** – The server identifier for the DB instance.
- **session_id** – A unique identifier for the current session.
- **aws_region** – The AWS Region that this DB instance is in. For tables listing AWS Regions by engine, see Regions and Availability Zones.
- **durable_lsn** – The LSN made durable in storage.
- **highest_lsn_rcvd** – The highest LSN received by the DB instance from the writer DB instance.
- **feedback_epoch** – The epoch the DB instance uses when it generates hot standby information.

  Hot standby is when a DB instance can connect and query while the server is in recovery or standby mode. Hot standby feedback is information about the DB instance when it's in hot standby. For more information, see the PostgreSQL documentation on Hot standby.
- **feedback_xmin** – The minimum (oldest) active transaction ID used by the DB instance.
- **oldest_read_view_lsn** – The oldest LSN used by the DB instance to read from storage.
- **visibility_lag_in_msec** – How far this DB instance is lagging behind the writer DB instance.

To see how these values change over time, consider the following transaction block where a table insert takes an hour.

```
psql> BEGIN;
psql> INSERT INTO table1 SELECT Large_Data_That_Takes_1_Hr_To_Insert;
psql> COMMIT;
```

In some cases, there might be a network disconnect between the primary DB cluster and the secondary DB cluster after the `BEGIN` statement. If so, the secondary DB cluster's `durability_lag_in_msec` value starts increasing. At the end of the `INSERT` statement, the `durability_lag_in_msec` value is 1 hour. However, the `rpo_lag_in_msec` value is 0 because all the user data committed between the primary DB cluster and secondary DB cluster are still the same. As soon as the `COMMIT` statement completes, the `rpo_lag_in_msec` value increases.

Using Amazon Aurora global databases with other AWS services

You can use your Aurora global databases with other AWS services, such as Amazon S3 and AWS Lambda. Doing so requires that all Aurora DB clusters in your global database have the same privileges, external functions, and so on in the respective AWS Regions. Because a read-only Aurora secondary DB cluster in an Aurora global database can be promoted to the role of primary, we recommend that you set up write privileges ahead of time, on all Aurora DB clusters for any services you plan to use with your Aurora global database.

The following procedures summarize the actions to take for each AWS service.

To invoke AWS Lambda functions from an Aurora global database

1. For all the Aurora clusters that make up the Aurora global database, perform the procedures in Invoking a Lambda function from an Amazon Aurora MySQL DB cluster.
2. For each cluster in the Aurora global database, set the (ARN) of the new IAM (IAM) role.
3. To permit database users in an Aurora global database to invoke Lambda functions, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with each cluster in the Aurora global database.
4. Configure each cluster in the Aurora global database to allow outbound connections to Lambda. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

To load data from Amazon S3

1. For all the Aurora clusters that make up the Aurora global database, perform the procedures in Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.
2. For each Aurora cluster in the global database, set either the `aurora_load_from_s3_role` or `aws_default_s3_role` DB cluster parameter to the Amazon Resource Name (ARN) of the new IAM role. If an IAM role isn't specified for `aurora_load_from_s3_role`, Aurora uses the IAM role specified in `aws_default_s3_role`.
3. To permit database users in an Aurora global database to access S3, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with each Aurora cluster in the global database.
4. Configure each Aurora cluster in the global database to allow outbound connections to S3. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

To save queried data to Amazon S3

1. For all the Aurora clusters that make up the Aurora global database, perform the procedures in Saving data from an Amazon Aurora MySQL DB cluster into text files in an Amazon S3 bucket.
2. For each Aurora cluster in the global database, set either the `aurora_select_into_s3_role` or `aws_default_s3_role` DB cluster parameter to the Amazon Resource Name (ARN) of the new IAM role. If an IAM role isn't specified for `aurora_select_into_s3_role`, Aurora uses the IAM role specified in `aws_default_s3_role`.
3. To permit database users in an Aurora global database to access S3, associate the role that you created in Creating an IAM role to allow Amazon Aurora to access AWS services with each Aurora cluster in the global database.
4. Configure each Aurora cluster in the global database to allow outbound connections to S3. For instructions, see Enabling network communication from Amazon Aurora MySQL to other AWS services.

Upgrading an Amazon Aurora global database

Upgrading an Aurora global database follows the same procedures as upgrading Aurora DB clusters. However, following are some important differences to take note of before you start the process.

We recommend that you upgrade the primary and secondary DB clusters to the same version. You can only perform a managed cross-Region database failover on an Aurora global database if the primary and secondary DB clusters have the same major, minor, and patch level engine versions. However, the patch levels can be different, depending on the minor engine version. For more information, see Patch level compatibility for managed cross-Region switchovers and failovers.

Major version upgrades

When you perform a major version upgrade of an Amazon Aurora global database, you upgrade the global database cluster instead the individual clusters that it contains.

To learn how to upgrade an Aurora PostgreSQL global database to a higher major version, see Major upgrades for global databases.

Note

With an Aurora global database based on Aurora PostgreSQL, you can't perform a major version upgrade of the Aurora DB engine if the recovery point objective (RPO) feature is turned on. For information about the RPO feature, see Managing RPOs for Aurora PostgreSQL–based global databases.

To learn how to upgrade an Aurora MySQL global database to a higher major version, see In-place major upgrades for global databases.

Note

With an Aurora global database based on Aurora MySQL, you can't perform an in-place upgrade from Aurora MySQL version 2 to version 3 if the `lower_case_table_names` parameter is turned on.
To perform a major version upgrade to Aurora MySQL version 3 when using `lower_case_table_names`, use the following process:

1. Remove all secondary Regions from the global cluster. Follow the steps in Removing a cluster from an Amazon Aurora global database.
2. Upgrade the engine version of the primary Region to Aurora MySQL version 3. Follow the steps in How to perform an in-place upgrade.
3. Add secondary Regions to the global cluster. Follow the steps in Adding an AWS Region to an Amazon Aurora global database.

You can also use the snapshot restore method instead. For more information, see Restoring from a DB cluster snapshot.

Minor version upgrades

For a minor upgrade on an Aurora global database, you upgrade all of the secondary clusters before you upgrade the primary cluster.

To learn how to upgrade an Aurora PostgreSQL global database to a higher minor version, see How to perform minor version upgrades and apply patches. To learn how to upgrade an Aurora MySQL global database to a higher minor version, see Upgrading Aurora MySQL by modifying the engine version.

Before you perform the upgrade, review the following considerations:

- Upgrading the minor version of a secondary cluster doesn't affect availability or usage of the primary cluster in any way.
- A secondary cluster must have at least one DB instance to perform a minor upgrade.
- If you upgrade an Aurora MySQL global database to version 2.11.*, you must upgrade your primary and secondary DB clusters to the exact same version, including the patch level.
- To support managed cross-Region switchovers or failovers, you must upgrade your primary and secondary DB clusters to the exact same version, including the patch level, depending on the engine version. For more information, see Patch level compatibility for managed cross-Region switchovers and failovers.

Patch level compatibility for managed cross-Region switchovers and failovers

When you upgrade your Aurora global database to one of the following minor engine versions, you can perform managed cross-Region switchovers or failovers even if the patch levels of your primary and secondary DB clusters don't match. For minor engine versions lower than the ones on this list, you must upgrade your primary and secondary DB clusters to the same major, minor, and patch levels to perform managed cross-Region switchovers or failovers. Make sure to review the version information and the notes in the following table.

> **Note**
>
> For manual cross-Region failovers, you can perform the failover process as long as the target secondary DB cluster is running the same major and minor engine version as the primary DB cluster. In this case, the patch levels don't need to match.

Database engine	Minor engine versions	Notes
Aurora MySQL	No minor versions	With all minor versions, you can perform managed cross-Region switchovers or failovers only if the patch levels of the primary and secondary DB clusters match.
Aurora PostgreSQL	• Version 14.5 or higher minor version • Version 13.8 or higher minor version • Version 12.12 or higher minor version • Version 11.17 or higher minor version	With the minor engine versions listed in the previous column, you can perform managed cross-Reg ion switchovers or failovers from a primary DB cluster with one patch level to a secondary DB cluster with a different patch level. With minor versions lower than these, you can perform managed cross-Region switchovers or failovers only if the patch levels of the

Database engine	Minor engine versions	Notes
		primary and secondary DB clusters match.

Using Amazon RDS Proxy for Aurora

By using Amazon RDS Proxy, you can allow your applications to pool and share database connections to improve their ability to scale. RDS Proxy makes applications more resilient to database failures by automatically connecting to a standby DB instance while preserving application connections. By using RDS Proxy, you can also enforce AWS Identity and Access Management (IAM) authentication for databases, and securely store credentials in AWS Secrets Manager.

Using RDS Proxy, you can handle unpredictable surges in database traffic. Otherwise, these surges might cause issues due to oversubscribing connections or new connections being created at a fast rate. RDS Proxy establishes a database connection pool and reuses connections in this pool. This approach avoids the memory and CPU overhead of opening a new database connection each time. To protect a database against oversubscription, you can control the number of database connections that are created.

RDS Proxy queues or throttles application connections that can't be served immediately from the connection pool. Although latencies might increase, your application can continue to scale without abruptly failing or overwhelming the database. If connection requests exceed the limits you specify, RDS Proxy rejects application connections (that is, it sheds load). At the same time, it maintains predictable performance for the load that RDS can serve with the available capacity.

You can reduce the overhead to process credentials and establish a secure connection for each new connection. RDS Proxy can handle some of that work on behalf of the database.

RDS Proxy is fully compatible with the engine versions that it supports. You can enable RDS Proxy for most applications with no code changes. For a list of supported engine versions, see Supported Regions and Aurora DB engines for Amazon RDS Proxy.

Topics

- Working with Amazon RDS Proxy endpoints
- Monitoring RDS Proxy metrics with Amazon CloudWatch
- Working with RDS Proxy events
- RDS Proxy command-line examples
- Troubleshooting for RDS Proxy
- Using RDS Proxy with AWS CloudFormation
- Using RDS Proxy with Aurora global databases

Region and version availability

For information about database engine version support and availability of RDS Proxy in a given AWS Region, see Supported Regions and Aurora DB engines for Amazon RDS Proxy.

Quotas and limitations for RDS Proxy

The following quotas and limitations apply to RDS Proxy:

- Each AWS account ID is limited to 20 proxies. If your application requires more proxies, request an increase via the **Service Quotas** page within the AWS Management Console. In the **Service Quotas** page, select **Amazon Relational Database Service (Amazon RDS)** and locate **Proxies** to request a quota increase. AWS can automatically increase your quota or pending review of your request by AWS Support.
- Each proxy can have up to 200 associated Secrets Manager secrets. Thus, each proxy can connect to with up to 200 different user accounts at any given time.
- Each proxy has a default endpoint. You can also add up to 20 proxy endpoints for each proxy. You can create, view, modify, and delete these endpoints.
- In an Aurora cluster, all of the connections using the default proxy endpoint are handled by the Aurora writer instance. To perform load balancing for read-intensive workloads, you can create a read-only endpoint for a proxy. That endpoint passes connections to the reader endpoint of the cluster. That way, your proxy connections can take advantage of Aurora read scalability. For more information, see Overview of proxy endpoints.
- You can use RDS Proxy with Aurora Serverless v2 clusters, but not with Aurora Serverless v1 clusters.

- Your RDS Proxy must be in the same virtual private cloud (VPC) as the database. The proxy can't be publicly accessible, although the database can be. For example, if you're prototyping your database on a local host, you can't connect to your proxy unless you set up the necessary network requirements to allow connection to the proxy. This is because your local host is outside of the proxy's VPC.

  > **Note**
  >
  > For Aurora DB clusters, you can turn on cross-VPC access. To do this, create an additional endpoint for a proxy and specify a different VPC, subnets, and security groups with that endpoint. For more information, see Accessing Aurora databases across VPCs.

- You can't use RDS Proxy with a VPC that has its tenancy set to `dedicated`.
- If you use RDS Proxy with an Aurora DB cluster that has IAM authentication enabled, check user authentication. Users who connect through a proxy must authenticate through sign-in credentials. For details about Secrets Manager and IAM support in RDS Proxy, see Setting up database credentials in AWS Secrets Manager and Setting up AWS Identity and Access Management (IAM) policies.
- You can't use RDS Proxy with custom DNS when using SSL hostname validation.
- Each proxy can be associated with a single target DB cluster. However, you can associate multiple proxies with the same DB cluster.
- Any statement with a text size greater than 16 KB causes the proxy to pin the session to the current connection.
- Certain Regions have Availability-Zone (AZ) restrictions to consider while creating your proxy. US East (N. Virginia) Region does not support RDS Proxy in the `use1-az3` Availability Zone. US West (N. California) Region does not support RDS Proxy in the `usw1-az2` Availability Zone. When selecting subnets while creating your proxy, make sure that you don't select subnets in the Availability Zones mentioned above.
- Currently, RDS Proxy doesn't support any global condition context keys.

  For more information about global condition context keys, see AWS global condition context keys in the *IAM User Guide*.

For additional limitations for each DB engine, see the following sections:

- Additional limitations for Aurora MySQL

- Additional limitations for Aurora PostgreSQL

Additional limitations for Aurora MySQL

The following additional limitations apply to RDS Proxy with Aurora MySQL databases:

- RDS Proxy doesn't support the MySQL `sha256_password` and `caching_sha2_password` authentication plugins. These plugins implement SHA-256 hashing for user account passwords.
- Currently, all proxies listen on port 3306 for MySQL. The proxies still connect to your database using the port that you specified in the database settings.
- You can't use RDS Proxy with self-managed MySQL databases in EC2 instances.
- You can't use RDS Proxy with an RDS for MySQL DB instance that has the `read_only` parameter in its DB parameter group set to `1`.
- RDS Proxy doesn't support MySQL compressed mode. For example, it doesn't support the compression used by the `--compress` or `-C` options of the `mysql` command.
- Database connections processing a `GET DIAGNOSTIC` command might return inaccurate information when RDS Proxy reuses the same database connection to run another query. This can happen when RDS Proxy multiplexes database connections.
- Some SQL statements and functions such as `SET LOCAL` can change the connection state without causing pinning. For the most current pinning behavior, see Avoiding pinning.
- Using the `ROW_COUNT()` function in a multi-statement query is not supported.
- RDS Proxy does not support client applications that can't handle multiple response messages in one TLS record.

> ⚠ **Important**
>
> For proxies associated with MySQL databases, don't set the configuration parameter `sql_auto_is_null` to `true` or a nonzero value in the initialization query. Doing so might cause incorrect application behavior.

Additional limitations for Aurora PostgreSQL

The following additional limitations apply to RDS Proxy with Aurora PostgreSQL databases:

- RDS Proxy doesn't support session pinning filters for PostgreSQL.
- Currently, all proxies listen on port 5432 for PostgreSQL.
- For PostgreSQL, RDS Proxy doesn't currently support canceling a query from a client by issuing a `CancelRequest`. This is the case, for example, when you cancel a long-running query in an interactive psql session by using Ctrl+C.
- The results of the PostgreSQL function lastval aren't always accurate. As a work-around, use the INSERT statement with the `RETURNING` clause.
- RDS Proxy currently doesn't support streaming replication mode.

> ⚠ **Important**
>
> For existing proxies with PostgreSQL databases, if you modify the database authentication to use `SCRAM` only, the proxy becomes unavailable for up to 60 seconds. To avoid the issue, do one of the following:
>
> - Ensure that the database allows both `SCRAM` and `MD5` authentication.
> - To use only `SCRAM` authentication, create a new proxy, migrate your application traffic to the new proxy, then delete the proxy previously associated with the database.

Planning where to use RDS Proxy

You can determine which of your DB instances, clusters, and applications might benefit the most from using RDS Proxy. To do so, consider these factors:

- Any DB cluster that encounters "too many connections" errors is a good candidate for associating with a proxy. This is often characterized by a high value of the `ConnectionAttempts` CloudWatch metric. The proxy enables applications to open many client connections, while the proxy manages a smaller number of long-lived connections to the DB cluster.
- For DB clusters that use smaller AWS instance classes, such as T2 or T3, using a proxy can help avoid out-of-memory conditions. It can also help reduce the CPU overhead for establishing connections. These conditions can occur when dealing with large numbers of connections.
- You can monitor certain Amazon CloudWatch metrics to determine whether a DB cluster is approaching certain types of limit. These limits are for the number of connections and the memory associated with connection management. You can also monitor certain CloudWatch

metrics to determine whether a DB cluster is handling many short-lived connections. Opening and closing such connections can impose performance overhead on your database. For information about the metrics to monitor, see Monitoring RDS Proxy metrics with Amazon CloudWatch.

- AWS Lambda functions can also be good candidates for using a proxy. These functions make frequent short database connections that benefit from connection pooling offered by RDS Proxy. You can take advantage of any IAM authentication you already have for Lambda functions, instead of managing database credentials in your Lambda application code.
- Applications that typically open and close large numbers of database connections and don't have built-in connection pooling mechanisms are good candidates for using a proxy.
- Applications that keep a large number of connections open for long periods are typically good candidates for using a proxy. Applications in industries such as software as a service (SaaS) or ecommerce often minimize the latency for database requests by leaving connections open. With RDS Proxy, an application can keep more connections open than it can when connecting directly to the DB cluster.
- You might not have adopted IAM authentication and Secrets Manager due to the complexity of setting up such authentication for all DB clusters. If so, you can leave the existing authentication methods in place and delegate the authentication to a proxy. The proxy can enforce the authentication policies for client connections for particular applications. You can take advantage of any IAM authentication you already have for Lambda functions, instead of managing database credentials in your Lambda application code.
- RDS Proxy can help make applications more resilient and transparent to database failures. RDS Proxy bypasses Domain Name System (DNS) caches to reduce failover times by up to 66% for Aurora Multi-AZ databases. RDS Proxy also automatically routes traffic to a new database instance while preserving application connections. This makes failovers more transparent for applications.

RDS Proxy concepts and terminology

You can simplify connection management for your Amazon Aurora DB clusters by using RDS Proxy.

RDS Proxy handles the network traffic between the client application and the database. It does so in an active way first by understanding the database protocol. It then adjusts its behavior based on the SQL operations from your application and the result sets from the database.

RDS Proxy reduces the memory and CPU overhead for connection management on your database. The database needs less memory and CPU resources when applications open many simultaneous connections. It also doesn't require logic in your applications to close and reopen connections that stay idle for a long time. Similarly, it requires less application logic to reestablish connections in case of a database problem.

The infrastructure for RDS Proxy is highly available and deployed over multiple Availability Zones (AZs). The computation, memory, and storage for RDS Proxy are independent of your Aurora DB cluster. This separation helps lower overhead on your database servers, so that they can devote their resources to serving database workloads. The RDS Proxy compute resources are serverless, automatically scaling based on your database workload.

Topics

Overview of RDS Proxy concepts

RDS Proxy handles the infrastructure to perform connection pooling and the other features described in the sections that follow. You see the proxies represented in the RDS console on the **Proxies** page.

Each proxy handles connections to a single Aurora DB cluster. The proxy automatically determines the current writer instance for Aurora provisioned clusters.

The connections that a proxy keeps open and available for your database applications to use make up the *connection pool*.

By default, RDS Proxy can reuse a connection after each transaction in your session. This transaction-level reuse is called *multiplexing*. When RDS Proxy temporarily removes a connection from the connection pool to reuse it, that operation is called *borrowing* the connection. When it's safe to do so, RDS Proxy returns that connection to the connection pool.

In some cases, RDS Proxy can't be sure that it's safe to reuse a database connection outside of the current session. In these cases, it keeps the session on the same connection until the session ends. This fallback behavior is called *pinning*.

A proxy has a default endpoint. You connect to this endpoint when you work with an Amazon Aurora DB cluster. You do so instead of connecting to the read/write endpoint that connects directly to the cluster. The special-purpose endpoints for an Aurora cluster remain available for you to use. For Aurora DB clusters, you can also create additional read/write and read-only endpoints. For more information, see Overview of proxy endpoints.

For example, you can still connect to the cluster endpoint for read/write connections without connection pooling. You can still connect to the reader endpoint for load-balanced read-only connections. You can still connect to the instance endpoints for diagnosis and troubleshooting of specific DB instances within a cluster. If you use other AWS services such as AWS Lambda to connect to RDS databases, change their connection settings to use the proxy endpoint. For example, you specify the proxy endpoint to allow Lambda functions to access your database while taking advantage of RDS Proxy functionality.

Each proxy contains a target group. This *target group* embodies the Aurora DB cluster that the proxy can connect to. For an Aurora cluster, by default the target group is associated with all the DB instances in that cluster. That way, the proxy can connect to whichever Aurora DB instance is promoted to be the writer instance in the cluster. The Aurora DB cluster associated with a proxy are called the *targets* of that proxy. For convenience, when you create a proxy through the console, RDS Proxy also creates the corresponding target group and registers the associated targets automatically.

An *engine family* is a related set of database engines that use the same DB protocol. You choose the engine family for each proxy that you create.

Connection pooling

Each proxy performs connection pooling for the writer instance of its associated Aurora DB. *Connection pooling* is an optimization that reduces the overhead associated with opening and closing connections and with keeping many connections open simultaneously. This overhead includes memory needed to handle each new connection. It also involves CPU overhead to close each connection and open a new one. Examples include Transport Layer Security/Secure Sockets Layer (TLS/SSL) handshaking, authentication, negotiating capabilities, and so on. Connection pooling simplifies your application logic. You don't need to write application code to minimize the number of simultaneous open connections.

Each proxy also performs connection multiplexing, also known as connection reuse. With *multiplexing*, RDS Proxy performs all the operations for a transaction using one underlying database connection. RDS then can use a different connection for the next transaction. You can open many simultaneous connections to the proxy, and the proxy keeps a smaller number of connections open to the DB instance or cluster. Doing so further minimizes the memory overhead for connections on the database server. This technique also reduces the chance of "too many connections" errors.

RDS Proxy security

RDS Proxy uses the existing RDS security mechanisms such as TLS/SSL and AWS Identity and Access Management (IAM). For general information about those security features, see Security in Amazon Aurora. Also, make sure to familiarize yourself with how Aurora work with authentication, authorization, and other areas of security.

RDS Proxy can act as an additional layer of security between client applications and the underlying database. For example, you can connect to the proxy using TLS 1.3, even if the underlying DB instance supports an older version of TLS. You can connect to the proxy using an IAM role. This is so even if the proxy connects to the database using the native user and password authentication method. By using this technique, you can enforce strong authentication requirements for database applications without a costly migration effort for the DB instances themselves.

You store the database credentials used by RDS Proxy in AWS Secrets Manager. Each database user for the Aurora DB cluster accessed by a proxy must have a corresponding secret in Secrets Manager. You can also set up IAM authentication for users of RDS Proxy. By doing so, you can enforce IAM authentication for database access even if the databases use native password authentication. We recommend using these security features instead of embedding database credentials in your application code.

Using TLS/SSL with RDS Proxy

You can connect to RDS Proxy using the TLS/SSL protocol.

> **Note**
>
> RDS Proxy uses certificates from the AWS Certificate Manager (ACM). If you are using RDS Proxy, you don't need to download Amazon RDS certificates or update applications that use RDS Proxy connections.

To enforce TLS for all connections between the proxy and your database, you can specify a setting **Require Transport Layer Security** when you create or modify a proxy in the AWS Management Console.

RDS Proxy can also ensure that your session uses TLS/SSL between your client and the RDS Proxy endpoint. To have RDS Proxy do so, specify the requirement on the client side. SSL session variables are not set for SSL connections to a database using RDS Proxy.

- For Aurora MySQL, specify the requirement on the client side with the `--ssl-mode` parameter when you run the `mysql` command.
- For and Aurora PostgreSQL, specify `sslmode=require` as part of the `conninfo` string when you run the `psql` command.

RDS Proxy supports TLS protocol version 1.0, 1.1, 1.2, and 1.3. You can connect to the proxy using a higher version of TLS than you use in the underlying database.

By default, client programs establish an encrypted connection with RDS Proxy, with further control available through the `--ssl-mode` option. From the client side, RDS Proxy supports all SSL modes.

For the client, the SSL modes are the following:

PREFERRED

SSL is the first choice, but it isn't required.

DISABLED

No SSL is allowed.

REQUIRED

Enforce SSL.

VERIFY_CA

Enforce SSL and verify the certificate authority (CA).

VERIFY_IDENTITY

Enforce SSL and verify the CA and CA hostname.

When using a client with `--ssl-mode` `VERIFY_CA` or `VERIFY_IDENTITY`, specify the `--ssl-ca` option pointing to a CA in `.pem` format. For the `.pem` file to use, download all root CA PEMs from Amazon Trust Services and place them into a single `.pem` file.

RDS Proxy uses wildcard certificates, which apply to both a domain and its subdomains. If you use the `mysql` client to connect with SSL mode `VERIFY_IDENTITY`, currently you must use the MySQL 8.0-compatible `mysql` command.

Failover

Failover is a high-availability feature that replaces a database instance with another one when the original instance becomes unavailable. A failover might happen because of a problem with a database instance. It might also be part of normal maintenance procedures, such as during a database upgrade. Failover applies to Aurora DB clusters with one or more reader instances in addition to the writer instance.

Connecting through a proxy makes your applications more resilient to database failovers. When the original DB instance becomes unavailable, RDS Proxy connects to the standby database without dropping idle application connections. This helps speed up and simplify the failover process. This is less disruptive to your application than a typical reboot or database problem.

Without RDS Proxy, a failover involves a brief outage. During the outage, you can't perform write operations on the database in failover. Any existing database connections are disrupted, and your application must reopen them. The database becomes available for new connections and write operations when a read-only DB instance is promoted in place of one that's unavailable.

During DB failovers, RDS Proxy continues to accept connections at the same IP address and automatically directs connections to the new primary DB instance. Clients connecting through RDS Proxy are not susceptible to the following:

- Domain Name System (DNS) propagation delays on failover.
- Local DNS caching.
- Connection timeouts.
- Uncertainty about which DB instance is the current writer.
- Waiting for a query response from a former writer that became unavailable without closing connections.

For applications that maintain their own connection pool, going through RDS Proxy means that most connections stay alive during failovers or other disruptions. Only connections that are in the middle of a transaction or SQL statement are canceled. RDS Proxy immediately accepts new connections. When the database writer is unavailable, RDS Proxy queues up incoming requests.

For applications that don't maintain their own connection pools, RDS Proxy offers faster connection rates and more open connections. It offloads the expensive overhead of frequent reconnects from the database. It does so by reusing database connections maintained in the RDS Proxy connection pool. This approach is particularly important for TLS connections, where setup costs are significant.

Transactions

All the statements within a single transaction always use the same underlying database connection. The connection becomes available for use by a different session when the transaction ends. Using the transaction as the unit of granularity has the following consequences:

- Connection reuse can happen after each individual statement when the Aurora MySQL `autocommit` setting is turned on.
- Conversely, when the `autocommit` setting is turned off, the first statement you issue in a session begins a new transaction. For example, suppose that you enter a sequence of `SELECT`, `INSERT`, `UPDATE`, and other data manipulation language (DML) statements. In this case, connection reuse doesn't happen until you issue a `COMMIT`, `ROLLBACK`, or otherwise end the transaction.
- Entering a data definition language (DDL) statement causes the transaction to end after that statement completes.

RDS Proxy detects when a transaction ends through the network protocol used by the database client application. Transaction detection doesn't rely on keywords such as `COMMIT` or `ROLLBACK` appearing in the text of the SQL statement.

In some cases, RDS Proxy might detect a database request that makes it impractical to move your session to a different connection. In these cases, it turns off multiplexing for that connection the remainder of your session. The same rule applies if RDS Proxy can't be certain that multiplexing is practical for the session. This operation is called *pinning*. For ways to detect and minimize pinning, see Avoiding pinning.

Getting started with RDS Proxy

Use the information in the following pages to set up and manage Using Amazon RDS Proxy for Auroraand set related security options. The security options control who can access each proxy and how each proxy connects to DB instances.

If you're new to RDS Proxy, we recommend following the pages in the order that we present them.

Topics

- Setting up network prerequisites
- Setting up database credentials in AWS Secrets Manager
- Setting up AWS Identity and Access Management (IAM) policies
- Creating an RDS Proxy
- Viewing an RDS Proxy
- Connecting to a database through RDS Proxy

Setting up network prerequisites

Using RDS Proxy requires you to have a common virtual private cloud (VPC) between your Aurora DB cluster and RDS Proxy. This VPC should have a minimum of two subnets that are in different Availability Zones. Your account can either own these subnets or share them with other accounts. For information about VPC sharing, see Work with shared VPCs.

Your client application resources such as Amazon EC2, Lambda, or Amazon ECS can be in the same VPC as the proxy. Or they can be in a separate VPC from the proxy. If you successfully connected to any Aurora DB clusters, you already have the required network resources.

Topics

- Getting information about your subnets
- Planning for IP address capacity

Getting information about your subnets

If you're just getting started with Aurora, you can learn the basics of connecting to a database by following the procedures in Setting up your environment for Amazon Aurora. You can also follow the tutorial in Getting started with Amazon Aurora.

To create a proxy, you must provide the subnets and the VPC that the proxy operates within. The following Linux example shows AWS CLI commands that examine the VPCs and subnets owned by your AWS account. In particular, you pass subnet IDs as parameters when you create a proxy using the CLI.

```
aws ec2 describe-vpcs
aws ec2 describe-internet-gateways
aws ec2 describe-subnets --query '*[].[VpcId,SubnetId]' --output text | sort
```

The following Linux example shows AWS CLI commands to determine the subnet IDs corresponding to a specific Aurora DB cluster.

For an Aurora cluster, first you find the ID for one of the associated DB instances. You can extract the subnet IDs used by that DB instance. To do so, examine the nested fields within the `DBSubnetGroup` and `Subnets` attributes in the describe output for the DB instance. You specify some or all of those subnet IDs when setting up a proxy for that database server.

```
$ # Find the ID of any DB instance in the cluster.
$  aws rds describe-db-clusters --db-cluster-identifier my_cluster_id --query '*[].
[DBClusterMembers]|[0]|[0][*].DBInstanceIdentifier' --output text
```

```
my_instance_id
instance_id_2
instance_id_3
```

After finding the DB instance identifier, examine the associated VPC to find its subnets. The following Linux example shows how.

```
$ #From the DB instance, trace through the DBSubnetGroup and Subnets to find the subnet
 IDs.
$ aws rds describe-db-instances --db-instance-identifier my_instance_id --query '*[].
[DBSubnetGroup]|[0]|[0]|[Subnets]|[0]|[*].SubnetIdentifier' --output text
```

```
subnet_id_1
subnet_id_2
subnet_id_3
...
```

```
$ #From the DB instance, find the VPC.
$ aws rds describe-db-instances --db-instance-identifier my_instance_id --query '*[].
[DBSubnetGroup]|[0]|[0].VpcId' --output text
```

```
my_vpc_id
```

```
$ aws ec2 describe-subnets --filters Name=vpc-id,Values=my_vpc_id --query '*[].
[SubnetId]' --output text
```

```
subnet_id_1
subnet_id_2
subnet_id_3
subnet_id_4
subnet_id_5
subnet_id_6
```

Planning for IP address capacity

An RDS Proxy automatically adjusts its capacity as needed based on the size and number of DB instances registered with it. Certain operations might also require additional proxy capacity such as increasing the size of a registered database or internal RDS Proxy maintenance operations. During these operations, your proxy might need more IP addresses to provision the extra capacity. These additional addresses allow your proxy to scale without affecting your workload. A lack of free IP addresses in your subnets prevents a proxy from scaling up. This can lead to higher query latencies or client connection failures. RDS notifies you through event `RDS-EVENT-0243` when there aren't enough free IP addresses in your subnets. For information about this event, see Working with RDS Proxy events.

Following are the recommended minimum numbers of IP addresses to leave free in your subnets for your proxy based on DB instance class sizes.

DB instance class	Minimum free IP addresses
db.*.xlarge or smaller	10
db.*.2xlarge	15
db.*.4xlarge	25
db.*.8xlarge	45

DB instance class	Minimum free IP addresses
db.*.12xlarge	60
db.*.16xlarge	75
db.*.24xlarge	110

These recommended numbers of IP addresses are estimates for a proxy with only the default endpoint. A proxy with additional endpoints or read replicas might need more free IP addresses. For each additional endpoint, we recommend that you reserve three more IP addresses. For each read replica, we recommend that you reserve additional IP addresses as specified in the table based on that read replica's size.

> **Note**
>
> RDS Proxy doesn't support more than 215 IP addresses in a VPC.

For example, suppose that you want to estimate the required IP addresses for a proxy that's associated with an Aurora DB cluster.

In this case, assume the following:

- Your Aurora DB cluster has 1 writer instance of size db.r5.8xlarge and 1 reader instance of size db.r5.2xlarge.
- The proxy that's attached to this DB cluster has the default endpoint and 1 custom endpoint with the read-only role.

In this case, the proxy needs approximately 63 free IP addresses (45 for the writer instance, 15 for reader instance, and 3 for the additional custom endpoint).

Setting up database credentials in AWS Secrets Manager

For each proxy that you create, you first use the Secrets Manager service to store sets of user name and password credentials. You create a separate Secrets Manager secret for each database user account that the proxy connects to on the Aurora DB cluster.

In Secrets Manager console, you create these secrets with values for the `username` and `password` fields. Doing so allows the proxy to connect to the corresponding database users on a Aurora DB cluster that you associate with the proxy. To do this, you can use the setting **Credentials for other database**, **Credentials for RDS database**, or **Other type of secrets**. Fill in the appropriate values for the **User name** and **Password** fields, and values for any other required fields. The proxy ignores other fields such as **Host** and **Port** if they're present in the secret. Those details are automatically supplied by the proxy.

You can also choose **Other type of secrets**. In this case, you create the secret with keys named `username` and `password`.

To connect through the proxy as a specific database user, make sure that the password associated with a secret matches the database password for that user. If there's a mismatch, you can update the associated secret in Secrets Manager. In this case, you can still connect to other accounts where the secret credentials and the database passwords do match.

When you create a proxy through the AWS CLI or RDS API, you specify the Amazon Resource Names (ARNs) of the corresponding secrets. You do so for all the DB user accounts that the proxy can access. In the AWS Management Console, you choose the secrets by their descriptive names.

For instructions about creating secrets in Secrets Manager, see the [Creating a secret]() page in the Secrets Manager documentation. Use one of the following techniques:

- Use [Secrets Manager]() in the console.
- To use the CLI to create a Secrets Manager secret for use with RDS Proxy, use a command such as the following.

  ```
  aws secretsmanager create-secret
    --name "secret_name"
    --description "secret_description"
    --region region_name
    --secret-string '{"username":"db_user","password":"db_user_password"}'
  ```

- You can also create a custom key to encrypt your Secrets Manager secret. The following command creates an example key.

  ```
  PREFIX=my_identifier
  aws kms create-key --description "$PREFIX-test-key" --policy '{
    "Id":"$PREFIX-kms-policy",
    "Version":"2012-10-17",
  ```

```
  "Statement":
    [
      {
        "Sid":"Enable IAM User Permissions",
        "Effect":"Allow",
        "Principal":{"AWS":"arn:aws:iam::account_id:root"},
        "Action":"kms:*","Resource":"*"
      },
      {
        "Sid":"Allow access for Key Administrators",
        "Effect":"Allow",
        "Principal":
          {
            "AWS":
              ["$USER_ARN","arn:aws:iam:account_id::role/Admin"]
          },
        "Action":
          [
            "kms:Create*",
            "kms:Describe*",
            "kms:Enable*",
            "kms:List*",
            "kms:Put*",
            "kms:Update*",
            "kms:Revoke*",
            "kms:Disable*",
            "kms:Get*",
            "kms:Delete*",
            "kms:TagResource",
            "kms:UntagResource",
            "kms:ScheduleKeyDeletion",
            "kms:CancelKeyDeletion"
          ],
        "Resource":"*"
      },
      {
        "Sid":"Allow use of the key",
        "Effect":"Allow",
        "Principal":{"AWS":"$ROLE_ARN"},
        "Action":["kms:Decrypt","kms:DescribeKey"],
        "Resource":"*"
      }
    ]
```

```
}'
```

For example, the following commands create Secrets Manager secrets for two database users:

```
aws secretsmanager create-secret \
  --name secret_name_1 --description "db admin user" \
  --secret-string '{"username":"admin","password":"choose_your_own_password"}'

aws secretsmanager create-secret \
  --name secret_name_2 --description "application user" \
  --secret-string '{"username":"app-user","password":"choose_your_own_password"}'
```

To create these secrets encrypted with your custom AWS KMS key, use the following commands:

```
aws secretsmanager create-secret \
  --name secret_name_1 --description "db admin user" \
  --secret-string '{"username":"admin","password":"choose_your_own_password"}'
  --kms-key-id arn:aws:kms:us-east-2:account_id:key/key_id

aws secretsmanager create-secret \
  --name secret_name_2 --description "application user" \
  --secret-string '{"username":"app-user","password":"choose_your_own_password"}'
  --kms-key-id arn:aws:kms:us-east-2:account_id:key/key_id
```

To see the secrets owned by your AWS account, use a command such as the following.

```
aws secretsmanager list-secrets
```

When you create a proxy using the CLI, you pass the Amazon Resource Names (ARNs) of one or more secrets to the `--auth` parameter. The following Linux example shows how to prepare a report with only the name and ARN of each secret owned by your AWS account. This example uses the `--output table` parameter that is available in AWS CLI version 2. If you are using AWS CLI version 1, use `--output text` instead.

```
aws secretsmanager list-secrets --query '*[].[Name,ARN]' --output table
```

To verify that you stored the correct credentials and in the right format in a secret, use a command such as the following. Substitute the short name or the ARN of the secret for *`your_secret_name`*.

```
aws secretsmanager get-secret-value --secret-id your_secret_name
```

The output should include a line displaying a JSON-encoded value like the following.

```
"SecretString": "{\"username\":\"your_username\",\"password\":\"your_password\"}",
```

Setting up AWS Identity and Access Management (IAM) policies

After you create the secrets in Secrets Manager, you create an IAM policy that can access those secrets. For general information about using IAM, see Identity and access management for Amazon Aurora.

> **Tip**
>
> The following procedure applies if you use the IAM console. If you use the AWS Management Console for RDS, RDS can create the IAM policy for you automatically. In that case, you can skip the following procedure.

To create an IAM policy that accesses your Secrets Manager secrets for use with your proxy

1. Sign in to the IAM console. Follow the **Create role** process, as described in Creating IAM roles, choosing Creating a role to delegate permissions to an AWS service.

   Choose **AWS service** for the **Trusted entity type**. Under **Use case**, select **RDS** from **Use cases for other AWS services** dropdown. Select **RDS - Add Role to Database**.

2. For the new role, perform the **Add inline policy** step. Use the same general procedures as in Editing IAM policies. Paste the following JSON into the JSON text box. Substitute your own account ID. Substitute your AWS Region for `us-east-2`. Substitute the Amazon Resource Names (ARNs) for the secrets that you created, see Specifying KMS keys in IAM policy statements. For the `kms:Decrypt` action, substitute the ARN of the default AWS KMS key or your own KMS key. Which one you use depends on which one you used to encrypt the Secrets Manager secrets.

   ```
   {
       "Version": "2012-10-17",
       "Statement": [
           {
   ```

```
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": "secretsmanager:GetSecretValue",
      "Resource": [
        "arn:aws:secretsmanager:us-east-2:account_id:secret:secret_name_1",
        "arn:aws:secretsmanager:us-east-2:account_id:secret:secret_name_2"
      ]
    },
    {
      "Sid": "VisualEditor1",
      "Effect": "Allow",
      "Action": "kms:Decrypt",
      "Resource": "arn:aws:kms:us-east-2:account_id:key/key_id",
      "Condition": {
        "StringEquals": {
          "kms:ViaService": "secretsmanager.us-east-2.amazonaws.com"
        }
      }
    }
  ]
}
```

3. Edit the trust policy for this IAM role. Paste the following JSON into the JSON text box.

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Sid": "",
   "Effect": "Allow",
   "Principal": {
    "Service": "rds.amazonaws.com"
   },
   "Action": "sts:AssumeRole"
  }
 ]
}
```

The following commands perform the same operation through the AWS CLI.

```
PREFIX=my_identifier
USER_ARN=$(aws sts get-caller-identity --query "Arn" --output text)
```

```
aws iam create-role --role-name my_role_name \
  --assume-role-policy-document '{"Version":"2012-10-17","Statement":
[{"Effect":"Allow","Principal":{"Service":
["rds.amazonaws.com"]},"Action":"sts:AssumeRole"}]}'

ROLE_ARN=arn:aws:iam::account_id:role/my_role_name

aws iam put-role-policy --role-name my_role_name \
  --policy-name $PREFIX-secret-reader-policy --policy-document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": [
                "arn:aws:secretsmanager:us-east-2:account_id:secret:secret_name_1",
                "arn:aws:secretsmanager:us-east-2:account_id:secret:secret_name_2"
            ]
        },
        {
            "Sid": "VisualEditor1",
            "Effect": "Allow",
            "Action": "kms:Decrypt",
            "Resource": "arn:aws:kms:us-east-2:account_id:key/key_id",
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "secretsmanager.us-east-2.amazonaws.com"
                }
            }
        }
    ]
}
```

Creating an RDS Proxy

To manage connections for a DB cluster, create a proxy. You can associate a proxy with an Aurora MySQL or Aurora PostgreSQL DB cluster.

AWS Management Console

To create a proxy

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. Choose **Create proxy**.
4. Choose all the settings for your proxy.

   For **Proxy configuration**, provide information for the following:

   - **Engine family**. This setting determines which database network protocol the proxy recognizes when it interprets network traffic to and from the database. For Aurora MySQL, choose **MariaDB and MySQL**. For Aurora PostgreSQL, choose **PostgreSQL**.
   - **Proxy identifier**. Specify a name of that is unique within your AWS account ID and current AWS Region.
   - **Idle client connection timeout**. Choose a time period that a client connection can be idle before the proxy closes it. The default is 1,800 seconds (30 minutes). A client connection is considered idle when the application doesn't submit a new request within the specified time after the previous request completed. The underlying database connection stays open and is returned to the connection pool. Thus, it's available to be reused for new client connections.

     To have the proxy proactively remove stale connections, lower the idle client connection timeout. When the workload is spiking, to save the cost of establishing connections, increase the idle client connection timeout."

   For **Target group configuration**, provide information for the following:

   - **Database**. Choose one Aurora DB cluster to access through this proxy. The list only includes DB instances and clusters with compatible database engines, engine versions, and other settings. If the list is empty, create a new DB instance or cluster that's compatible with RDS Proxy. To do so, follow the procedure in Creating an Amazon Aurora DB cluster. Then try creating the proxy again.
   - **Connection pool maximum connections**. Specify a value from 1 through 100. This setting represents the percentage of the `max_connections` value that RDS Proxy can use for its connections. If you only intend to use one proxy with this DB instance or cluster,

you can set this value to 100. For details about how RDS Proxy uses this setting, see MaxConnectionsPercent.

- **Session pinning filters**. (Optional) This option allows you to force RDS Proxy to not pin for certain types of detected session states. This circumvents the default safety measures for multiplexing database connections across client connections. Currently, the setting isn't supported for PostgreSQL. The only choice is `EXCLUDE_VARIABLE_SETS`.

  Enabling this setting can cause session variables of one connection to impact other connections. This can cause errors or correctness issues if your queries depend on session variable values set outside of the current transaction. Consider using this option after verifying it is safe for your applications to share database connections across client connections.

  The following patterns can be considered safe:

  - `SET` statements where there is no change to the effective session variable value, i.e., there is no change to the session variable.
  - You change the session variable value and execute a statement in the same transaction.

  For more information, see Avoiding pinning.
- **Connection borrow timeout**. In some cases, you might expect the proxy to sometimes use all available database connections. In such cases, you can specify how long the proxy waits for a database connection to become available before returning a timeout error. You can specify a period up to a maximum of five minutes. This setting only applies when the proxy has the maximum number of connections open and all connections are already in use.
- **Initialization query**. (Optional) You can specify one or more SQL statements for the proxy to run when opening each new database connection. The setting is typically used with `SET` statements to make sure that each connection has identical settings, such as time zone and character sets. For multiple statements, use semicolons as the separator. You can also include multiple variables in a single `SET` statement, such as `SET x=1, y=2`.

For **Authentication**, provide information for the following:

- **IAM role**. Choose an IAM role that has permission to access the Secrets Manager secrets that you chose earlier. Or, you can create a new IAM role from the AWS Management Console.
- **Secrets Manager secrets**. Choose at least one Secrets Manager secret that contains database user credentials that allow the proxy to access the Aurora DB cluster.

- **Client authentication type**. Choose the type of authentication the proxy uses for connections from clients. Your choice applies to all Secrets Manager secrets that you associate with this proxy. If you need to specify a different client authentication type for each secret, then create your proxy by using the AWS CLI or the API instead.
- **IAM authentication**. Choose whether to require or disallow IAM authentication for connections to your proxy. Your choice applies to all Secrets Manager secrets that you associate with this proxy. If you need to specify a different IAM authentication for each secret, create your proxy by using the AWS CLI or the API instead.

For **Connectivity**, provide information for the following:

- **Require Transport Layer Security**. Choose this setting if you want the proxy to enforce TLS/SSL for all client connections. For an encrypted or unencrypted connection to a proxy, the proxy uses the same encryption setting when it makes a connection to the underlying database.
- **Subnets**. This field is prepopulated with all the subnets associated with your VPC. You can remove any subnets that you don't need for this proxy. You must leave at least two subnets.

Provide additional connectivity configuration:

- **VPC security group**. Choose an existing VPC security group. Or, you can create a new security group from the AWS Management Console. You must configure the **Inbound rules** to allow your applications to access the proxy. You must also configure the **Outbound rules** to allow traffic from your DB targets.

  > **Note**
  >
  > This security group must allow connections from the proxy to the database. The same security group is used for ingress from your applications to the proxy, and for egress from the proxy to the database. For example, suppose that you use the same security group for your database and your proxy. In this case, make sure that you specify that resources in that security group can communicate with other resources in the same security group.
  > When using a shared VPC, you can't use the default security group for the VPC, or one that belongs to another account. Choose a security group that belongs to your

> account. If one doesn't exist, create one. For more information about this limitation, see Work with shared VPCs.

RDS deploys a proxy over multiple Availability Zones to ensure high availability. To enable cross-AZ communication for such a proxy, the network access control list (ACL) for your proxy subnet must allow engine port specific egress and all ports to ingress. For more information about network ACLs, see Control traffic to subnets using network ACLs. If the network ACL for your proxy and target are identical, you must add a **TCP** protocol ingress rule where the **Source** is set to the VPC CIDR. You must also add an engine port specific **TCP** protocol egress rule where the **Destination** is set to the VPC CIDR.

(Optional) Provide advanced configuration:

- **Enable enhanced logging**. You can enable this setting to troubleshoot proxy compatibility or performance issues.

  When this setting is enabled, RDS Proxy includes detailed information about proxy performance in its logs. This information helps you to debug issues involving SQL behavior or the performance and scalability of the proxy connections. Thus, only enable this setting for debugging and when you have security measures in place to safeguard any sensitive information that appears in the logs.

  To minimize overhead associated with your proxy, RDS Proxy automatically turns this setting off 24 hours after you enable it. Enable it temporarily to troubleshoot a specific issue.

5. Choose **Create Proxy**.

AWS CLI

To create a proxy by using the AWS CLI, call the create-db-proxy command with the following required parameters:

- `--db-proxy-name`
- `--engine-family`
- `--role-arn`
- `--auth`

- `--vpc-subnet-ids`

The `--engine-family` value is case-sensitive.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-proxy \
    --db-proxy-name proxy_name \
    --engine-family { MYSQL | POSTGRESQL | SQLSERVER } \
    --auth ProxyAuthenticationConfig_JSON_string \
    --role-arn iam_role \
    --vpc-subnet-ids space_separated_list \
    [--vpc-security-group-ids space_separated_list] \
    [--require-tls | --no-require-tls] \
    [--idle-client-timeout value] \
    [--debug-logging | --no-debug-logging] \
    [--tags comma_separated_list]
```

For Windows:

```
aws rds create-db-proxy ^
    --db-proxy-name proxy_name ^
    --engine-family { MYSQL | POSTGRESQL | SQLSERVER } ^
    --auth ProxyAuthenticationConfig_JSON_string ^
    --role-arn iam_role ^
    --vpc-subnet-ids space_separated_list ^
    [--vpc-security-group-ids space_separated_list] ^
    [--require-tls | --no-require-tls] ^
    [--idle-client-timeout value] ^
    [--debug-logging | --no-debug-logging] ^
    [--tags comma_separated_list]
```

The following is an example of the JSON value for the `--auth` option. This example applies a different client authentication type to each secret.

```
[
  {
    "Description": "proxy description 1",
    "AuthScheme": "SECRETS",
    "SecretArn": "arn:aws:secretsmanager:us-
west-2:123456789123:secret/1234abcd-12ab-34cd-56ef-1234567890ab",
    "IAMAuth": "DISABLED",
    "ClientPasswordAuthType": "POSTGRES_SCRAM_SHA_256"
  },

  {
    "Description": "proxy description 2",
    "AuthScheme": "SECRETS",
    "SecretArn": "arn:aws:secretsmanager:us-
west-2:111122223333:secret/1234abcd-12ab-34cd-56ef-1234567890cd",
    "IAMAuth": "DISABLED",
    "ClientPasswordAuthType": "POSTGRES_MD5"

  },

  {
    "Description": "proxy description 3",
    "AuthScheme": "SECRETS",
    "SecretArn": "arn:aws:secretsmanager:us-
west-2:111122221111:secret/1234abcd-12ab-34cd-56ef-1234567890ef",
    "IAMAuth": "REQUIRED"
  }

]
```

> **Tip**
>
> If you don't already know the subnet IDs to use for the `--vpc-subnet-ids` parameter, see Setting up network prerequisites for examples of how to find them.

> **Note**
>
> The security group must allow access to the database the proxy connects to. The same security group is used for ingress from your applications to the proxy, and for egress from the proxy to the database. For example, suppose that you use the same security group for your database and your proxy. In this case, make sure that you specify that resources in that security group can communicate with other resources in the same security group.
> When using a shared VPC, you can't use the default security group for the VPC, or one that belongs to another account. Choose a security group that belongs to your account. If one

doesn't exist, create one. For more information about this limitation, see Work with shared VPCs.

To create the right associations for the proxy, you also use the register-db-proxy-targets command. Specify the target group name default. RDS Proxy automatically creates a target group with this name when you create each proxy.

```
aws rds register-db-proxy-targets
    --db-proxy-name value
    [--target-group-name target_group_name]
    [--db-instance-identifiers space_separated_list]  # rds db instances, or
    [--db-cluster-identifiers cluster_id]        # rds db cluster (all instances)
```

RDS API

To create an RDS proxy, call the Amazon RDS API operation CreateDBProxy. You pass a parameter with the AuthConfig data structure.

RDS Proxy automatically creates a target group named default when you create each proxy. You associate an Aurora DB cluster with the target group by calling the function RegisterDBProxyTargets.

Viewing an RDS Proxy

After you create one or more RDS proxies, you can view them all. Doing so makes it possible to examine their configuration details and choose which ones to modify, delete, and so on.

In order for database applications to use a proxy, you must provide the proxy endpoint in the connection string.

AWS Management Console

To view your proxy

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you created the RDS Proxy.

3. In the navigation pane, choose **Proxies**.
4. Choose the name of an RDS proxy to display its details.
5. On the details page, the **Target groups** section shows how the proxy is associated with a specific Aurora DB cluster. You can follow the link to the **default** target group page to see more details about the association between the proxy and the database. This page is where you see settings that you specified when creating the proxy. These include maximum connection percentage, connection borrow timeout, engine family, and session pinning filters.

CLI

To view your proxy using the CLI, use the describe-db-proxies command. By default, it displays all proxies owned by your AWS account. To see details for a single proxy, specify its name with the `--db-proxy-name` parameter.

```
aws rds describe-db-proxies [--db-proxy-name proxy_name]
```

To view the other information associated with the proxy, use the following commands.

```
aws rds describe-db-proxy-target-groups  --db-proxy-name proxy_name

aws rds describe-db-proxy-targets --db-proxy-name proxy_name
```

Use the following sequence of commands to see more detail about the things that are associated with the proxy:

1. To get a list of proxies, run describe-db-proxies.
2. To show connection parameters such as the maximum percentage of connections that the proxy can use, run describe-db-proxy-target-groups `--db-proxy-name`. Use the name of the proxy as the parameter value.
3. To see the details of the Aurora DB cluster associated with the returned target group, run describe-db-proxy-targets.

RDS API

To view your proxies using the RDS API, use the DescribeDBProxies operation. It returns values of the DBProxy data type.

To see details of the connection settings for the proxy, use the proxy identifiers from this return value with the DescribeDBProxyTargetGroups operation. It returns values of the DBProxyTargetGroup data type.

To see the RDS instance or Aurora DB cluster associated with the proxy, use the DescribeDBProxyTargets operation. It returns values of the DBProxyTarget data type.

Connecting to a database through RDS Proxy

You connect to an Aurora DB cluster or cluster that uses Aurora Serverless v2 through a proxy in generally the same way as you connect directly to the database. The main difference is that you specify the proxy endpoint instead of the cluster endpoint. By default all proxy connections have read/write capability and use the writer instance. If you normally use the reader endpoint for read-only connections, you can create an additional read-only endpoint for the proxy. You can use that endpoint the same way. For more information, see Overview of proxy endpoints.

Topics

- Connecting to a proxy using native authentication
- Connecting to a proxy using IAM authentication
- Considerations for connecting to a proxy with PostgreSQL

Connecting to a proxy using native authentication

Use the following steps to connect to a proxy using native authentication:

1. Find the proxy endpoint. In the AWS Management Console, you can find the endpoint on the details page for the corresponding proxy. With the AWS CLI, you can use the describe-db-proxies command. The following example shows how.

```
# Add --output text to get output as a simple tab-separated list.
$ aws rds describe-db-proxies --query '*[*].
{DBProxyName:DBProxyName,Endpoint:Endpoint}'
[
    [
        {
            "Endpoint": "the-proxy.proxy-demo.us-east-1.rds.amazonaws.com",
            "DBProxyName": "the-proxy"
        },
        {
            "Endpoint": "the-proxy-other-secret.proxy-demo.us-
east-1.rds.amazonaws.com",
            "DBProxyName": "the-proxy-other-secret"
        },
        {
            "Endpoint": "the-proxy-rds-secret.proxy-demo.us-
east-1.rds.amazonaws.com",
            "DBProxyName": "the-proxy-rds-secret"
        },
        {
            "Endpoint": "the-proxy-t3.proxy-demo.us-east-1.rds.amazonaws.com",
            "DBProxyName": "the-proxy-t3"
        }
    ]
]
```

2. Specify the endpoint as the host parameter in the connection string for your client application. For example, specify the proxy endpoint as the value for the `mysql -h` option or `psql -h` option.

3. Supply the same database user name and password as you usually do.

Connecting to a proxy using IAM authentication

When you use IAM authentication with RDS Proxy, set up your database users to authenticate with regular user names and passwords. The IAM authentication applies to RDS Proxy retrieving the user name and password credentials from Secrets Manager. The connection from RDS Proxy to the underlying database doesn't go through IAM.

To connect to RDS Proxy using IAM authentication, use the same general connection procedure as for IAM authentication with an Aurora DB cluster. For general information about using IAM, see Security in Amazon Aurora.

The major differences in IAM usage for RDS Proxy include the following:

- You don't configure each individual database user with an authorization plugin. The database users still have regular user names and passwords within the database. You set up Secrets Manager secrets containing these user names and passwords, and authorize RDS Proxy to retrieve the credentials from Secrets Manager.

The IAM authentication applies to the connection between your client program and the proxy. The proxy then authenticates to the database using the user name and password credentials retrieved from Secrets Manager.

- Instead of the instance, cluster, or reader endpoint, you specify the proxy endpoint. For details about the proxy endpoint, see Connecting to your DB cluster using IAM authentication.
- In the direct database IAM authentication case, you selectively choose database users and configure them to be identified with a special authentication plugin. You can then connect to those users using IAM authentication.

  In the proxy use case, you provide the proxy with Secrets that contain some user's user name and password (native authentication). You then connect to the proxy using IAM authentication. Here, you do this by generating an authentication token with the proxy endpoint, not the database endpoint. You also use a user name that matches one of the user names for the secrets that you provided.
- Make sure that you use Transport Layer Security (TLS)/Secure Sockets Layer (SSL) when connecting to a proxy using IAM authentication.

You can grant a specific user access to the proxy by modifying the IAM policy. An example follows.

```
"Resource": "arn:aws:rds-db:us-east-2:1234567890:dbuser:prx-ABCDEFGHIJKL01234/db_user"
```

Considerations for connecting to a proxy with PostgreSQL

For PostgreSQL, when a client starts a connection to a PostgreSQL database, it sends a startup message. This message includes pairs of parameter name and value strings. For details, see the `StartupMessage` in PostgreSQL message formats in the PostgreSQL documentation.

When connecting through an RDS proxy, the startup message can include the following currently recognized parameters:

- `user`
- `database`

The startup message can also include the following additional runtime parameters:

- `application_name`

- `client_encoding`
- `DateStyle`
- `TimeZone`
- `extra_float_digits`
- `search_path`

For more information about PostgreSQL messaging, see the Frontend/Backend protocol in the PostgreSQL documentation.

For PostgreSQL, if you use JDBC, we recommend the following to avoid pinning:

- Set the JDBC connection parameter `assumeMinServerVersion` to at least `9.0` to avoid pinning. This prevents the JDBC driver from performing an extra round trip during connection startup when it runs `SET extra_float_digits = 3`.
- Set the JDBC connection parameter `ApplicationName` to *`any/your-application-name`* to avoid pinning. Doing this prevents the JDBC driver from performing an extra round trip during connection startup when it runs `SET application_name = "PostgreSQL JDBC Driver"`. Note the JDBC parameter is `ApplicationName` but the PostgreSQL `StartupMessage` parameter is `application_name`.

For more information, see Avoiding pinning. For more information about connecting using JDBC, see Connecting to the database in the PostgreSQL documentation.

Managing an RDS Proxy

This section provides information on how to manage RDS Proxy operation and configuration. These procedures help your application make the most efficient use of database connections and achieve maximum connection reuse. The more that you can take advantage of connection reuse, the more CPU and memory overhead that you can save. This in turn reduces latency for your application and enables the database to devote more of its resources to processing application requests.

Topics

- Client and database connections
- Configuring connection settings
- Avoiding pinning
- Deleting an RDS Proxy

Modifying an RDS Proxy

You can change specific settings associated with a proxy after you create the proxy. You do so by modifying the proxy itself, its associated target group, or both. Each proxy has an associated target group.

AWS Management Console

> ⚠ **Important**
>
> The values in the **Client authentication type** and **IAM authentication** fields apply to all Secrets Manager secrets that are associated with this proxy. To specify different values for each secret, modify your proxy by using the AWS CLI or the API instead.

To modify the settings for a proxy

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. In the list of proxies, choose the proxy whose settings you want to modify or go to its details page.
4. For **Actions**, choose **Modify**.
5. Enter or choose the properties to modify. You can modify the following:

   - **Proxy identifier** – Rename the proxy by entering a new identifier.
   - **Idle client connection timeout** – Enter a time period for the idle client connection timeout.
   - **IAM role** – Change the IAM role used to retrieve the secrets from Secrets Manager.
   - **Secrets Manager secrets** – Add or remove Secrets Manager secrets. These secrets correspond to database user names and passwords.

- **Client authentication type** – (PostgreSQL only) Change the type of authentication for client connections to the proxy.
- **IAM authentication –** Require or disallow IAM authentication for connections to the proxy.
- **Require Transport Layer Security –** Turn the requirement for Transport layer Security (TLS) on or off.
- **VPC security group** – Add or remove VPC security groups for the proxy to use.
- **Enable enhanced logging –** Enable or disable enhanced logging.

6. Choose **Modify**.

If you didn't find the settings listed that you want to change, use the following procedure to update the target group for the proxy. The *target group* associated with a proxy controls the settings related to the physical database connections. Each proxy has one associated target group named `default`, which is created automatically along with the proxy.

You can only modify the target group from the proxy details page, not from the list on the **Proxies** page.

To modify the settings for a proxy target group

1. On the **Proxies** page, go to the details page for a proxy.
2. For **Target groups**, choose the `default` link. Currently, all proxies have a single target group named `default`.
3. On the details page for the **default** target group, choose **Modify**.
4. Choose new settings for the properties that you can modify:

   - **Database –** Choose a different Aurora cluster.
   - **Connection pool maximum connections –** Adjust what percentage of the maximum available connections the proxy can use.
   - **Session pinning filters –** (Optional) Choose a session pinning filter. This circumvents the default safety measures for multiplexing database connections across client connections. Currently, the setting isn't supported for PostgreSQL. The only choice is `EXCLUDE_VARIABLE_SETS`.

     Enabling this setting can cause session variables of one connection to impact other connections. This can cause errors or correctness issues if your queries depend on session variable values set outside of the current transaction. Consider using this option after verifying it is safe for your applications to share database connections across client connections.

The following patterns can be considered safe:

  - SET statements where there is no change to the effective session variable value, i.e., there is no change to the session variable.
  - You change the session variable value and execute a statement in the same transaction.

For more information, see Avoiding pinning.

- **Connection borrow timeout** – Adjust the connection borrow timeout interval. This setting applies when the maximum number of connections is already being used for the proxy. The setting determines how long the proxy waits for a connection to become available before returning a timeout error.
- **Initialization query** – (Optional) Add an initialization query, or modify the current one. You can specify one or more SQL statements for the proxy to run when opening each new database connection. The setting is typically used with SET statements to make sure that each connection has identical settings such as time zone and character set. For multiple statements, use semicolons as the separator. You can also include multiple variables in a single SET statement, such as `SET x=1, y=2`.

You can't change certain properties, such as the target group identifier and the database engine.

5. Choose **Modify target group**.

AWS CLI

To modify a proxy using the AWS CLI, use the commands modify-db-proxy, modify-db-proxy-target-group, deregister-db-proxy-targets, and register-db-proxy-targets.

With the `modify-db-proxy` command, you can change properties such as the following:

- The set of Secrets Manager secrets used by the proxy.
- Whether TLS is required.
- The idle client timeout.
- Whether to log additional information from SQL statements for debugging.
- The IAM role used to retrieve Secrets Manager secrets.

- The security groups used by the proxy.

The following example shows how to rename an existing proxy.

```
aws rds modify-db-proxy --db-proxy-name the-proxy --new-db-proxy-name the_new_name
```

To modify connection-related settings or rename the target group, use the `modify-db-proxy-target-group` command. Currently, all proxies have a single target group named `default`. When working with this target group, you specify the name of the proxy and `default` for the name of the target group.

The following example shows how to first check the `MaxIdleConnectionsPercent` setting for a proxy and then change it, using the target group.

```
aws rds describe-db-proxy-target-groups --db-proxy-name the-proxy

{
    "TargetGroups": [
        {
            "Status": "available",
            "UpdatedDate": "2019-11-30T16:49:30.342Z",
            "ConnectionPoolConfig": {
                "MaxIdleConnectionsPercent": 50,
                "ConnectionBorrowTimeout": 120,
                "MaxConnectionsPercent": 100,
                "SessionPinningFilters": []
            },
            "TargetGroupName": "default",
            "CreatedDate": "2019-11-30T16:49:27.940Z",
            "DBProxyName": "the-proxy",
            "IsDefault": true
        }
    ]
}

aws rds modify-db-proxy-target-group --db-proxy-name the-proxy --target-group-name
 default --connection-pool-config '
{ "MaxIdleConnectionsPercent": 75 }'

{
    "DBProxyTargetGroup": {
        "Status": "available",
        "UpdatedDate": "2019-12-02T04:09:50.420Z",
        "ConnectionPoolConfig": {
            "MaxIdleConnectionsPercent": 75,
            "ConnectionBorrowTimeout": 120,
            "MaxConnectionsPercent": 100,
            "SessionPinningFilters": []
        },
        "TargetGroupName": "default",
        "CreatedDate": "2019-11-30T16:49:27.940Z",
        "DBProxyName": "the-proxy",
        "IsDefault": true
    }
}
```

With the `deregister-db-proxy-targets` and `register-db-proxy-targets` commands, you change which Aurora DB clusters the proxy is associated with through its target group. Currently, each proxy can connect to one Aurora DB cluster. The target group tracks the connection details for all the all the DB instances in an Aurora cluster.

The following example starts with a proxy that is associated with an Aurora MySQL cluster named `cluster-56-2020-02-25-1399`. The example shows how to change the proxy so that it can connect to a different cluster named `provisioned-cluster`.

When you work with an Aurora DB cluster, you specify the `--db-cluster-identifier` option.

The following example modifies an Aurora MySQL proxy. An Aurora PostgreSQL proxy has port 5432.

```
aws rds describe-db-proxy-targets --db-proxy-name the-proxy

{
    "Targets": [
        {
            "Endpoint": "instance-9814.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "instance-9814"
        },
        {
            "Endpoint": "instance-8898.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "instance-8898"
        },
        {
            "Endpoint": "instance-1018.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "instance-1018"
        },
        {
            "Type": "TRACKED_CLUSTER",
            "Port": 0,
            "RdsResourceId": "cluster-56-2020-02-25-1399"
        },
        {
            "Endpoint": "instance-4330.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "instance-4330"
        }
    ]
}

aws rds deregister-db-proxy-targets --db-proxy-name the-proxy --db-cluster-identifier
 cluster-56-2020-02-25-1399

aws rds describe-db-proxy-targets --db-proxy-name the-proxy

{
    "Targets": []
}

aws rds register-db-proxy-targets --db-proxy-name the-proxy --db-cluster-identifier
 provisioned-cluster

{
    "DBProxyTargets": [
        {
            "Type": "TRACKED_CLUSTER",
            "Port": 0,
            "RdsResourceId": "provisioned-cluster"
        },
        {
            "Endpoint": "gkldje.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "gkldje"
        },
        {
            "Endpoint": "provisioned-1.demo.us-east-1.rds.amazonaws.com",
            "Type": "RDS_INSTANCE",
            "Port": 3306,
            "RdsResourceId": "provisioned-1"
        }
    ]
}
```

RDS API

To modify a proxy using the RDS API, you use the operations ModifyDBProxy, ModifyDBProxyTargetGroup, DeregisterDBProxyTargets, and RegisterDBProxyTargets operations.

With `ModifyDBProxy`, you can change properties such as the following:

- The set of Secrets Manager secrets used by the proxy.
- Whether TLS is required.
- The idle client timeout.
- Whether to log additional information from SQL statements for debugging.
- The IAM role used to retrieve Secrets Manager secrets.
- The security groups used by the proxy.

With `ModifyDBProxyTargetGroup`, you can modify connection-related settings or rename the target group. Currently, all proxies have a single target group named `default`. When working with this target group, you specify the name of the proxy and `default` for the name of the target group.

With `DeregisterDBProxyTargets` and `RegisterDBProxyTargets`, you change which Aurora cluster the proxy is associated with through its target group. Currently, each proxy can connect to one Aurora cluster. The target group tracks the connection details for the DB instances in an Aurora cluster.

Adding a new database user

In some cases, you might add a new database user to an Aurora cluster that's associated with a proxy. If so, add or repurpose a Secrets Manager secret to store the credentials for that user. To do this, choose one of the following options:

1. Create a new Secrets Manager secret, using the procedure described in Setting up database credentials in AWS Secrets Manager.
2. Update the IAM role to give RDS Proxy access to the new Secrets Manager secret. To do so, update the resources section of the IAM role policy.
3. Modify the RDS Proxy to add the new Secrets Manager secret under **Secrets Manager secrets**.
4. If the new user takes the place of an existing one, update the credentials stored in the proxy's Secrets Manager secret for the existing user.

Adding a new database user to a PostgreSQL database

When adding a new user to your PostgreSQL database, if you have run the following command:

```
REVOKE CONNECT ON DATABASE postgres FROM PUBLIC;
```

Grant the `rdsproxyadmin` user the `CONNECT` privilege so the user can monitor connections on the target database.

```
GRANT CONNECT ON DATABASE postgres TO rdsproxyadmin;
```

You can also allow other target database users to perform health checks by changing `rdsproxyadmin` to the database user in the command above.

Changing the password for a database user

In some cases, you might change the password for a database user in an Aurora cluster that's associated with a proxy. If so, update the corresponding Secrets Manager secret with the new password.

Client and database connections

Connections from your application to RDS Proxy are known as client connections. Connections from a proxy to the database are database connections. When using RDS Proxy, client connections terminate at the proxy while database connections are managed within RDS Proxy.

Application-side connection pooling can provide the benefit of reducing recurring connection establishment between your application and RDS Proxy.

Consider the following configuration aspects before implementing an application-side connection pool:

- Client connection max life: RDS Proxy enforces a maximum life of client connections of 24 hours. This value is not configurable. Configure your pool with a maximum connection life less than 24 hours to avoid unexpected client connection drops.
- Client connection idle timeout: RDS Proxy enforces a maximum idle time for client connections. Configure your pool with an idle connection timeout of a value lower than your client connection idle timeout setting for RDS Proxy to avoid unexpected connection drops.

The maximum number of client connections configured in your application-side connection pool does not have to be limited to the **max_connections** setting for RDS Proxy.

Client connection pooling results in a longer client connection life. If your connections experience pinning, then pooling client connections might reduce multiplexing efficiency. Client connections that are pinned but idle in the application-side connection pool continue to hold on to a database connection and prevent the database connection to be reused by other client connections. Review your proxy logs to check whether your connections experience pinning.

> **Note**
>
> RDS Proxy closes database connections some time after 24 hours when they are no longer in use. The proxy performs this action regardless of the value of the maximum idle connections setting.

Configuring connection settings

To adjust RDS Proxy's connection pooling, you can modify the following settings:

- IdleClientTimeout
- MaxConnectionsPercent
- MaxIdleConnectionsPercent
- ConnectionBorrowTimeout

IdleClientTimeout

You can specify how long a client connection can be idle before the proxy closes it. The default is 1,800 seconds (30 minutes).

A client connection is considered *idle* when the application doesn't submit a new request within the specified time after the previous request completed. The underlying database connection stays open and is returned to the connection pool. Thus, it's available to be reused for new client connections. If you want the proxy to proactively remove stale connections, then lowering the idle client connection timeout. If your workload establishes frequent connections with the proxy, then raising the idle client connection timeout to save the cost of establishing connections.

This setting is represented by the **Idle client connection timeout** field in the RDS console and the `IdleClientTimeout` setting in the AWS CLI and the API. To learn how to change the value of the **Idle client connection timeout** field in the RDS console, see AWS Management Console. To learn how to change the value of the `IdleClientTimeout` setting, see the CLI command modify-db-proxy or the API operation ModifyDBProxy.

MaxConnectionsPercent

You can limit the number of connections that an RDS Proxy can establish with the target database. You specify the limit as a percentage of the maximum connections available for your database. This setting is represented by the **Connection pool maximum connections** field in the RDS console and the `MaxConnectionsPercent` setting in the AWS CLI and the API.

The `MaxConnectionsPercent` value is expressed as a percentage of the `max_connections` setting for the Aurora DB cluster used by the target group. The proxy doesn't create all of these connections in advance. This setting allows the proxy to establish these connections as the workload needs them.

For example, for a registered database target with `max_connections` set to 1000, and `MaxConnectionsPercent` set to 95, RDS Proxy sets 950 connections as the upper limit for concurrent connections to that database target.

A common side-effect of your workload reaching the maximum number of allowed database connections is an increase in overall query latency, along with an increase in the `DatabaseConnectionsBorrowLatency` metric. You can monitor currently used and total allowed database connections by comparing the `DatabaseConnections` and `MaxDatabaseConnectionsAllowed` metrics.

When setting this parameter, note the following best practices:

- Allow sufficient connection headroom for changes in workload pattern. It is recommended to set the parameter at least 30% above your maximum recent monitored usage. As RDS Proxy redistributes database connection quotas across multiple nodes, internal capacity changes might require at least 30% headroom for additional connections to avoid increased borrow latencies.
- RDS Proxy reserves a certain number of connections for active monitoring to support fast failover, traffic routing and internal operations. The `MaxDatabaseConnectionsAllowed` metric does not include these reserved connections. It represents the number of connections available to serve the workload, and can be lower than the value derived from the `MaxConnectionsPercent` setting.

  Minimal recommended `MaxConnectionsPercent` values are as follows:

  - db.t3.small: 100
  - db.t3.medium: 55
  - db.t3.large: 35
  - db.r3.large or above: 20

  If multiple target instances are registered with RDS Proxy like an Aurora cluster with reader nodes, set the minimum value based on the smallest registered instance.

To learn how to change the value of the **Connection pool maximum connections** field in the RDS console, see AWS Management Console. To learn how to change the value of the `MaxConnectionsPercent` setting, see the CLI command modify-db-proxy-target-group or the API operation ModifyDBProxyTargetGroup.

> ⚠ **Important**
>
> If the DB cluster is part of a global database with write forwarding turned on, reduce your proxy's `MaxConnectionsPercent` value by the quota that's allotted for write forwarding. The write forwarding quota is set in the DB cluster parameter

aurora_fwd_writer_max_connections_pct. For information about write forwarding, see Using write forwarding in an Amazon Aurora global database.

For information on database connection limits, see Maximum connections to an Aurora MySQL DB instance and Maximum connections to an Aurora PostgreSQL DB instance.

MaxIdleConnectionsPercent

You can control the number of idle database connections that RDS Proxy can keep in the connection pool. By default, RDS Proxy considers a database connection in its pool to be *idle* when there's been no activity on the connection for five minutes.

The MaxIdleConnectionsPercent value is expressed as a percentage of the max_connections setting for the RDS DB instance target group. The default value is 50 percent of MaxConnectionsPercent, and the upper limit is the value of MaxConnectionsPercent. For example, if MaxConnectionsPercent, is 80, then the default value of MaxIdleConnectionsPercent is 40.

With a high value, the proxy leaves a high percentage of idle database connections open. With a low value, the proxy closes a high percentage of idle database connections. If your workloads are unpredictable, consider setting a high value for MaxIdleConnectionsPercent. Doing so means that RDS Proxy can accommodate surges in activity without opening a lot of new database connections.

This setting is represented by the MaxIdleConnectionsPercent setting of DBProxyTargetGroup in the AWS CLI and the API. To learn how to change the value of the MaxIdleConnectionsPercent setting, see the CLI command modify-db-proxy-target-group or the API operation ModifyDBProxyTargetGroup.

For information on database connection limits, see Maximum connections to an Aurora MySQL DB instance and Maximum connections to an Aurora PostgreSQL DB instance.

ConnectionBorrowTimeout

You can choose how long RDS Proxy waits for a database connection in the connection pool to become available for use before returning a timeout error. The default is 120 seconds. This setting applies when the number of connections is at the maximum, and so no connections are available in the connection pool. It also applies when no appropriate database instance is available to handle the request, such as when a failover operation is in process. Using this setting, you can set the best wait period for your application without changing the query timeout in your application code.

This setting is represented by the **Connection borrow timeout** field in the RDS console or the `ConnectionBorrowTimeout` setting of `DBProxyTargetGroup` in the AWS CLI or API. To learn how to change the value of the **Connection borrow timeout** field in the RDS console, see AWS Management Console. To learn how to change the value of the `ConnectionBorrowTimeout` setting, see the CLI command modify-db-proxy-target-group or the API operation ModifyDBProxyTargetGroup.

Avoiding pinning

Multiplexing is more efficient when database requests don't rely on state information from previous requests. In that case, RDS Proxy can reuse a connection at the conclusion of each transaction. Examples of such state information include most variables and configuration parameters that you can change through `SET` or `SELECT` statements. SQL transactions on a client connection can multiplex between underlying database connections by default.

Your connections to the proxy can enter a state known as *pinning*. When a connection is pinned, each later transaction uses the same underlying database connection until the session ends. Other client connections also can't reuse that database connection until the session ends. The session ends when the client connection is dropped.

RDS Proxy automatically pins a client connection to a specific DB connection when it detects a session state change that isn't appropriate for other sessions. Pinning reduces the effectiveness of connection reuse. If all or almost all of your connections experience pinning, consider modifying your application code or workload to reduce the conditions that cause the pinning.

For example, your application changes a session variable or configuration parameter. In this case, later statements can rely on the new variable or parameter to be in effect. Thus, when RDS Proxy processes requests to change session variables or configuration settings, it pins that session to the DB connection. That way, the session state remains in effect for all later transactions in the same session.

For some database engines, this rule doesn't apply to all parameters that you can set. RDS Proxy tracks certain statements and variables. Thus, RDS Proxy doesn't pin the session when you modify them. In this case, RDS Proxy only reuses the connection for other sessions that have the same values for those settings. For the lists of tracked statements and variables for Aurora MySQL, see What RDS Proxy tracks for Aurora MySQL databases.

What RDS Proxy tracks for Aurora MySQL databases

Following are the MySQL statements that RDS Proxy tracks:

- DROP DATABASE
- DROP SCHEMA
- USE

Following are the MySQL variables that RDS Proxy tracks:

- `AUTOCOMMIT`
- `AUTO_INCREMENT_INCREMENT`
- `CHARACTER SET (or CHAR SET)`
- `CHARACTER_SET_CLIENT`
- `CHARACTER_SET_DATABASE`
- `CHARACTER_SET_FILESYSTEM`
- `CHARACTER_SET_CONNECTION`
- `CHARACTER_SET_RESULTS`
- `CHARACTER_SET_SERVER`
- `COLLATION_CONNECTION`
- `COLLATION_DATABASE`
- `COLLATION_SERVER`
- `INTERACTIVE_TIMEOUT`
- `NAMES`
- `NET_WRITE_TIMEOUT`
- `QUERY_CACHE_TYPE`
- `SESSION_TRACK_SCHEMA`
- `SQL_MODE`
- `TIME_ZONE`
- `TRANSACTION_ISOLATION (or TX_ISOLATION)`
- `TRANSACTION_READ_ONLY (or TX_READ_ONLY)`

- WAIT_TIMEOUT

Minimizing pinning

Performance tuning for RDS Proxy involves trying to maximize transaction-level connection reuse (multiplexing) by minimizing pinning.

You can minimize pinning by doing the following:

- Avoid unnecessary database requests that might cause pinning.
- Set variables and configuration settings consistently across all connections. That way, later sessions are more likely to reuse connections that have those particular settings.

  However, for PostgreSQL setting a variable leads to session pinning.
- For a MySQL engine family database, apply a session pinning filter to the proxy. You can exempt certain kinds of operations from pinning the session if you know that doing so doesn't affect the correct operation of your application.
- See how frequently pinning occurs by monitoring the Amazon CloudWatch metric `DatabaseConnectionsCurrentlySessionPinned`. For information about this and other CloudWatch metrics, see Monitoring RDS Proxy metrics with Amazon CloudWatch.
- If you use `SET` statements to perform identical initialization for each client connection, you can do so while preserving transaction-level multiplexing. In this case, you move the statements that set up the initial session state into the initialization query used by a proxy. This property is a string containing one or more SQL statements, separated by semicolons.

  For example, you can define an initialization query for a proxy that sets certain configuration parameters. Then, RDS Proxy applies those settings whenever it sets up a new connection for that proxy. You can remove the corresponding `SET` statements from your application code, so that they don't interfere with transaction-level multiplexing.

  For metrics about how often pinning occurs for a proxy, see Monitoring RDS Proxy metrics with Amazon CloudWatch.

Conditions that cause pinning for all engine families

The proxy pins the session to the current connection in the following situations where multiplexing might cause unexpected behavior:

- Any statement with a text size greater than 16 KB causes the proxy to pin the session.

Conditions that cause pinning for Aurora MySQL

For MySQL, the following interactions also cause pinning:

- Explicit table lock statements `LOCK TABLE`, `LOCK TABLES`, or `FLUSH TABLES WITH READ LOCK` cause the proxy to pin the session.
- Creating named locks by using `GET_LOCK` causes the proxy to pin the session.
- Setting a user variable or a system variable (with some exceptions) causes the proxy to pin the session. If this situation reduces your connection reuse too much, then choose for `SET` operations to not cause pinning. For information about how to do so by setting the session pinning filters property, see [Creating an RDS Proxy]() and [Modifying an RDS Proxy]().
- Creating a temporary table causes the proxy to pin the session. That way, the contents of the temporary table are preserved throughout the session regardless of transaction boundaries.
- Calling the functions `ROW_COUNT`, `FOUND_ROWS`, and `LAST_INSERT_ID` sometimes causes pinning.

  The exact circumstances where these functions cause pinning might differ among Aurora MySQL versions that are compatible with MySQL 5.7.
- Prepared statements cause the proxy to pin the session. This rule applies whether the prepared statement uses SQL text or the binary protocol.
- RDS Proxy does not pin connections when you use SET LOCAL.
- Calling stored procedures and stored functions doesn't cause pinning. RDS Proxy doesn't detect any session state changes resulting from such calls. Make sure that your application doesn't change session state inside stored routines if you rely on that session state to persist across transactions. For example, RDS Proxy isn't currently compatible with a stored procedure that creates a temporary table that persists across all transactions.

If you have expert knowledge about your application behavior, you can skip the pinning behavior for certain application statements. To do so, choose the **Session pinning filters** option when creating the proxy. Currently, you can opt out of session pinning for setting session variables and configuration settings.

Conditions that cause pinning for Aurora PostgreSQL

For PostgreSQL, the following interactions also cause pinning:

- Using `SET` commands.
- Using `PREPARE`, `DISCARD`, `DEALLOCATE`, or `EXECUTE` commands to manage prepared statements.
- Creating temporary sequences, tables, or views.
- Declaring cursors.
- Discarding the session state.
- Listening on a notification channel.
- Loading a library module such as `auto_explain`.
- Manipulating sequences using functions such as `nextval` and `setval`.
- Interacting with locks using functions such as `pg_advisory_lock` and `pg_try_advisory_lock`.

> **Note**
>
> RDS Proxy does not pin on transaction level advisory locks, specifically `pg_advisory_xact_lock`, `pg_advisory_xact_lock_shared`, `pg_try_advisory_xact_lock`, and `pg_try_advisory_xact_lock_shared`.

- Setting a parameter, or resetting a parameter to its default. Specifically, using `SET` and `set_config` commands to assign default values to session variables.
- Calling stored procedures and stored functions doesn't cause pinning. RDS Proxy doesn't detect any session state changes resulting from such calls. Make sure that your application doesn't change session state inside stored routines if you rely on that session state to persist across transactions. For example, RDS Proxy isn't currently compatible with a stored procedure that creates a temporary table that persists across all transactions.

Deleting an RDS Proxy

You can delete a proxy when you no longer need it. Or, you might delete a proxy if you take the DB instance or cluster associated with it out of service.

AWS Management Console

To delete a proxy

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. Choose the proxy to delete from the list.
4. Choose **Delete Proxy**.

AWS CLI

To delete a DB proxy, use the AWS CLI command delete-db-proxy. To remove related associations, also use the deregister-db-proxy-targets command.

```
aws rds delete-db-proxy --name proxy_name
```

```
aws rds deregister-db-proxy-targets
    --db-proxy-name proxy_name
    [--target-group-name target_group_name]
    [--target-ids comma_separated_list]       # or
    [--db-instance-identifiers instance_id]       # or
    [--db-cluster-identifiers cluster_id]
```

RDS API

To delete a DB proxy, call the Amazon RDS API function DeleteDBProxy. To delete related items and associations, you also call the functions DeleteDBProxyTargetGroup and DeregisterDBProxyTargets.

Working with Amazon RDS Proxy endpoints

Learn about endpoints for RDS Proxy and how to use them. By using proxy endpoints, you can take advantage of the following capabilities:

- You can use multiple endpoints with a proxy to monitor and troubleshoot connections from different applications independently.

- You can use reader endpoints with Aurora DB clusters to improve read scalability and high availability for your query-intensive applications.
- You can use a cross-VPC endpoint to allow access to databases in one VPC from resources such as Amazon EC2 instances in a different VPC.

Topics

Overview of proxy endpoints

Working with RDS Proxy endpoints involves the same kinds of procedures as with Aurora DB cluster and reader endpoints. If you aren't familiar with Aurora endpoints, find more information in Amazon Aurora connection management.

By default, the endpoint that you connect to when you use RDS Proxy with an Aurora cluster has read/write capability. As a result, this endpoint sends all requests to the writer instance of the cluster. All of those connections count against the `max_connections` value for the writer instance. If your proxy is associated with an Aurora DB cluster, you can create additional read/write or read-only endpoints for that proxy.

You can use a read-only endpoint with your proxy for read-only queries. You do this the same way that you use the reader endpoint for an Aurora provisioned cluster. Doing so helps you to take advantage of the read scalability of an Aurora cluster with one or more reader DB instances. You can run more simultaneous queries and make more simultaneous connections by using a read-only endpoint and adding more reader DB instances to your Aurora cluster as needed.

Tip

When you create a proxy for an Aurora cluster using the AWS Management Console, you can have RDS Proxy automatically create a reader endpoint. For information about the benefits of a reader endpoint, see Using reader endpoints with Aurora clusters.

For a proxy endpoint that you create, you can also associate the endpoint with a different virtual private cloud (VPC) than the proxy itself uses. By doing so, you can connect to the proxy from a different VPC, for example a VPC used by a different application within your organization.

For information about limits associated with proxy endpoints, see Limitations for proxy endpoints.

In the RDS Proxy logs, each entry is prefixed with the name of the associated proxy endpoint. This name can be the name you specified for a user-defined endpoint. Or, it can be the special name `default` for the default endpoint of a proxy that performs read/write requests.

Each proxy endpoint has its own set of CloudWatch metrics. You can monitor the metrics for all endpoints of a proxy. You can also monitor metrics for a specific endpoint, or for all the read/write or read-only endpoints of a proxy. For more information, see Monitoring RDS Proxy metrics with Amazon CloudWatch.

A proxy endpoint uses the same authentication mechanism as its associated proxy. RDS Proxy automatically sets up permissions and authorizations for the user-defined endpoint, consistent with the properties of the associated proxy.

To learn how proxy endpoints work for DB clusters in an Aurora global database, see How RDS Proxy endpoints work with global databases.

Using reader endpoints with Aurora clusters

You can create and connect to read-only endpoints called *reader endpoints* when you use RDS Proxy with Aurora clusters. These reader endpoints help to improve the read scalability of your query-intensive applications. Reader endpoints also help to improve the availability of your connections if a reader DB instance in your cluster becomes unavailable.

Note

When you specify that a new endpoint is read-only, RDS Proxy requires that the Aurora cluster has one or more reader DB instances. In some cases, you might change the target of the proxy to an Aurora cluster containing only a single writer or a multi-writer Aurora cluster. If you do, any requests to the reader endpoint fail with an error. Requests also fail if the target of the proxy is an RDS instance instead of an Aurora cluster.

If an Aurora cluster has reader instances but those instances aren't available, RDS Proxy waits to send the request instead of returning an error immediately. If no reader instance becomes available within the connection borrow timeout period, the request fails with an error.

How reader endpoints help application availability

In some cases, one or more reader instances in your cluster might become unavailable. If so, connections that use a reader endpoint of a DB proxy can recover more quickly than ones that use the Aurora reader endpoint. RDS Proxy routes connections to only the available reader instances in the cluster. There isn't a delay due to DNS caching when an instance becomes unavailable.

If the connection is multiplexed, RDS Proxy directs subsequent queries to a different reader DB instance without any interruption to your application. During the automatic switchover to a new reader instance, RDS Proxy checks the replication lag of the old and new reader instances. RDS Proxy makes sure that the new reader instance is up to date with the same changes as the previous reader instance. That way, your application never sees stale data when RDS Proxy switches from one reader DB instance to another.

If the connection is pinned, the next query on the connection returns an error. However, your application can immediately reconnect to the same endpoint. RDS Proxy routes the connection to a different reader DB instance that's in `available` state. When you manually reconnect, RDS Proxy doesn't check the replication lag between the old and new reader instances.

If your Aurora cluster doesn't have any available reader instances, RDS Proxy checks whether this condition is temporary or permanent. The behavior in each case is as follows:

- Suppose that your cluster has one or more reader DB instances, but none of them are in the `Available` state. For example, all reader instances might be rebooting or encountering problems. In that case, attempts to connect to a reader endpoint wait for a reader instance to become available. If no reader instance becomes available within the connection borrow timeout period, the connection attempt fails. If a reader instance does become available, the connection attempt succeeds.

- Suppose that your cluster has no reader DB instances. In that case, RDS Proxy returns an error immediately if you try to connect to a reader endpoint. To resolve this problem, add one or more reader instances to your cluster before you connect to the reader endpoint.

How reader endpoints help query scalability

Reader endpoints for a proxy help with Aurora query scalability in the following ways:

- As you add reader instances to your Aurora cluster, RDS Proxy can route new connections to any reader endpoints to the different reader instances. That way, queries performed using one reader endpoint connection don't slow down queries performed using another reader endpoint connection. The queries run on separate DB instances. Each DB instance has its own compute resources, buffer cache, and so on.
- Where practical, RDS Proxy uses the same reader DB instance for all the queries issue using a particular reader endpoint connection. That way, a set of related queries on the same tables can take advantage of caching, plan optimization, and so on, on a particular DB instance.
- If a reader DB instance becomes unavailable, the effect on your application depends on whether the session is multiplexed or pinned. If the session is multiplexed, RDS Proxy routes any subsequent queries to a different reader DB instance without any action on your part. If the session is pinned, your application gets an error and must reconnect. You can reconnect to the reader endpoint immediately and RDS Proxy routes the connection to an available reader DB instance. For more information about multiplexing and pinning for proxy sessions, see Overview of RDS Proxy concepts.
- The more reader DB instances that you have in the cluster, the more simultaneous connections you can make using reader endpoints. For example, suppose that your cluster has four reader DB instances, each configured to support 200 simultaneous connections. Suppose also that your proxy is configured to use 50% of the maximum connections. Here, the maximum number of connections that you can make through the reader endpoints in the proxy is 100 (50% of 200) for reader 1. It's also 100 for reader 2, and so on, for a total of 400. If you double the number of cluster reader DB instances to eight, then the maximum number of connections through the reader endpoints also doubles, to 800.

Examples of using reader endpoints

The following Linux example shows how you can confirm that you're connected to an Aurora MySQL cluster through a reader endpoint. The `innodb_read_only` configuration setting is enabled. Attempts to perform write operations such as `CREATE DATABASE` statements fail with an error. And you can confirm that you're connected to a reader DB instance by checking the DB instance name using the `aurora_server_id` variable.

> **Tip**
>
> Don't rely only on checking the DB instance name to determine whether the connection is read/write or read-only. Remember that DB instances in an Aurora cluster can change roles between writer and reader when failovers happen.

```
$ mysql -h endpoint-demo-reader.endpoint.proxy-demo.us-east-1.rds.amazonaws.com -u
 admin -p
...
mysql> select @@innodb_read_only;
+--------------------+
| @@innodb_read_only |
+--------------------+
|                  1 |
+--------------------+
mysql> create database shouldnt_work;
ERROR 1290 (HY000): The MySQL server is running with the --read-only option so it
 cannot execute this statement

mysql> select @@aurora_server_id;
+---------------------------------------+
| @@aurora_server_id                    |
+---------------------------------------+
| proxy-reader-endpoint-demo-instance-3 |
+---------------------------------------+
```

The following example shows how your connection to a proxy reader endpoint can keep working even when the reader DB instance is deleted. In this example, the Aurora cluster has two reader instances, `instance-5507` and `instance-7448`. The connection to the reader endpoint begins using one of the reader instances. During the example, this reader instance is deleted by a `delete-db-instance` command. RDS Proxy switches to a different reader instance for subsequent queries.

```
$ mysql -h reader-demo.endpoint.proxy-demo.us-east-1.rds.amazonaws.com
  -u my_user -p
```

```
...
mysql> select @@aurora_server_id;
+--------------------+
| @@aurora_server_id |
+--------------------+
| instance-5507      |
+--------------------+

mysql> select @@innodb_read_only;
+--------------------+
| @@innodb_read_only |
+--------------------+
|                  1 |
+--------------------+

mysql> select count(*) from information_schema.tables;
+----------+
| count(*) |
+----------+
|      328 |
+----------+
```

While the `mysql` session is still running, the following command deletes the reader instance that the reader endpoint is connected to.

```
aws rds delete-db-instance --db-instance-identifier instance-5507 --skip-final-snapshot
```

Queries in the `mysql` session continue working without the need to reconnect. RDS Proxy automatically switches to a different reader DB instance.

```
mysql> select @@aurora_server_id;
+--------------------+
| @@aurora_server_id |
+--------------------+
| instance-7448      |
+--------------------+

mysql> select count(*) from information_schema.TABLES;
+----------+
| count(*) |
+----------+
|      328 |
+----------+
```

Accessing Aurora databases across VPCs

By default, the components of your Aurora technology stack are all in the same Amazon VPC. For example, suppose that an application running on an Amazon EC2 instance connects to an Aurora DB cluster. In this case, the application server and database must both be within the same VPC.

With RDS Proxy, you can set up access to an Aurora DB cluster in one VPC from resources in another VPC, such as EC2 instances. For example, your organization might have multiple applications that access the same database resources. Each application might be in its own VPC.

To enable cross-VPC access, you create a new endpoint for the proxy. The proxy itself resides in the same VPC as the Aurora DB cluster. However, the cross-VPC endpoint resides in the other VPC, along with the other resources such as the EC2 instances. The cross-VPC endpoint is associated with subnets and security groups from the same VPC as the EC2 and other resources. These associations let you connect to the endpoint from the applications that otherwise can't access the database due to the VPC restrictions.

The following steps explain how to create and access a cross-VPC endpoint through RDS Proxy:

1. Create two VPCs, or choose two VPCs that you already use for Aurora work. Each VPC should have its own associated network resources such as an internet gateway, route tables, subnets, and security groups. If you only have one VPC, you can consult Getting started with Amazon Aurora for the steps to set up another VPC to use Aurora successfully. You can also examine your existing VPC in the Amazon EC2 console to see the kinds of resources to connect together.
2. Create a DB proxy associated with the Aurora DB cluster that you want to connect to. Follow the procedure in Creating an RDS Proxy.
3. On the **Details** page for your proxy in the RDS console, under the **Proxy endpoints** section, choose **Create endpoint**. Follow the procedure in Creating a proxy endpoint.
4. Choose whether to make the cross-VPC endpoint read/write or read-only.
5. Instead of accepting the default of the same VPC as the Aurora DB cluster, choose a different VPC. This VPC must be in the same AWS Region as the VPC where the proxy resides.
6. Now instead of accepting the defaults for subnets and security groups from the same VPC as the Aurora DB cluster, make new selections. Make these based on the subnets and security groups from the VPC that you chose.

7. You don't need to change any of the settings for the Secrets Manager secrets. The same credentials work for all endpoints for your proxy, regardless of which VPC each endpoint is in.
8. Wait for the new endpoint to reach the **Available** state.
9. Make a note of the full endpoint name. This is the value ending in *`Region_name`*`.rds.amazonaws.com` that you supply as part of the connection string for your database application.
10. Access the new endpoint from a resource in the same VPC as the endpoint. A simple way to test this process is to create a new EC2 instance in this VPC. Then, log into the EC2 instance and run the `mysql` or `psql` commands to connect by using the endpoint value in your connection string.

Creating a proxy endpoint

Console

To create a proxy endpoint

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. Click the name of the proxy that you want to create a new endpoint for.

   The details page for that proxy appears.
4. In the **Proxy endpoints** section, choose **Create proxy endpoint**.

   The **Create proxy endpoint** window appears.
5. For **Proxy endpoint name**, enter a descriptive name of your choice.
6. For **Target role**, choose whether to make the endpoint read/write or read-only.

   Connections that use read/write endpoints can perform any kind of operations, such as data definition language (DDL) statements, data manipulation language (DML) statements, and queries. These endpoints always connect to the primary instance of the Aurora cluster. You can use read/write endpoints for general database operations when you only use a single endpoint in your application. You can also use read/write endpoints for administrative operations, online transaction processing (OLTP) applications, and extract-transform-load (ETL) jobs.

Connections that use a read-only endpoint can only perform queries. When there are multiple reader instances in the Aurora cluster, RDS Proxy can use a different reader instance for each connection to the endpoint. That way, a query-intensive application can take advantage of Aurora's clustering capability. You can add more query capacity to the cluster by adding more reader DB instances. These read-only connections don't impose any overhead on the primary instance of the cluster. That way, your reporting and analysis queries don't slow down the write operations of your OLTP applications.

7. For **Virtual Private Cloud (VPC)**, choose the default to access the endpoint from the same EC2 instances or other resources that normally use to access the proxy or its associated database. To set up cross-VPC access for this proxy, choose a VPC other than the default. For more information about cross-VPC access, see Accessing Aurora databases across VPCs.
8. For **Subnets**, RDS Proxy fills in the same subnets as the associated proxy by default. To restrict access to the endpoint to only a portion of the VPC's address range being able to connect to it, remove one or more subnets.
9. For **VPC security group**, you can choose an existing security group or create a new one. RDS Proxy fills in the same security group or groups as the associated proxy by default. If the inbound and outbound rules for the proxy are appropriate for this endpoint, then keep the default choice.

   If you choose to create a new security group, specify a name for the security group on this page. Then edit the security group settings from the EC2 console later.
10. Choose **Create proxy endpoint**.

AWS CLI

To create a proxy endpoint, use the AWS CLI create-db-proxy-endpoint command.

Include the following required parameters:

- `--db-proxy-name` *`value`*
- `--db-proxy-endpoint-name` *`value`*
- `--vpc-subnet-ids` *`list_of_ids`*. Separate the subnet IDs with spaces. You don't specify the ID of the VPC itself.

You can also include the following optional parameters:

- `--target-role { READ_WRITE | READ_ONLY }`. This parameter defaults to `READ_WRITE`. The `READ_ONLY` value affects only Aurora provisioned clusters that contain one or more reader DB instances. When the proxy is associated with an Aurora cluster that only contains a writer DB instance, you can't specify `READ_ONLY`. For more information about the intended use of read-only endpoints with Aurora clusters, see Using reader endpoints with Aurora clusters .
- `--vpc-security-group-ids` *`value`*. Separate the security group IDs with spaces. If you omit this parameter, RDS Proxy uses the default security group for the VPC. RDS Proxy determines the VPC based on the subnet IDs that you specify for the `--vpc-subnet-ids` parameter.

Example

The following example creates a proxy endpoint named `my-endpoint`.

For Linux, macOS, or Unix:

```
aws rds create-db-proxy-endpoint \
  --db-proxy-name my-proxy \
  --db-proxy-endpoint-name my-endpoint \
  --vpc-subnet-ids subnet_id subnet_id subnet_id ... \
  --target-role READ_ONLY \
  --vpc-security-group-ids security_group_id ]
```

For Windows:

```
aws rds create-db-proxy-endpoint ^
  --db-proxy-name my-proxy ^
  --db-proxy-endpoint-name my-endpoint ^
  --vpc-subnet-ids subnet_id_1 subnet_id_2 subnet_id_3 ... ^
  --target-role READ_ONLY ^
  --vpc-security-group-ids security_group_id
```

RDS API

To create a proxy endpoint, use the RDS API CreateDBProxyEndpoint action.

Viewing proxy endpoints

Console

To view the details for a proxy endpoint

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. In the list, choose the proxy whose endpoint you want to view. Click the proxy name to view its details page.
4. In the **Proxy endpoints** section, choose the endpoint that you want to view. Click its name to view the details page.
5. Examine the parameters whose values you're interested in. You can check properties such as the following:

   - Whether the endpoint is read/write or read-only.
   - The endpoint address that you use in a database connection string.
   - The VPC, subnets, and security groups associated with the endpoint.

AWS CLI

To view one or more proxy endpoints, use the AWS CLI describe-db-proxy-endpoints command.

You can include the following optional parameters:

- `--db-proxy-endpoint-name`
- `--db-proxy-name`

The following example describes the `my-endpoint` proxy endpoint.

Example

For Linux, macOS, or Unix:

```
aws rds describe-db-proxy-endpoints \
  --db-proxy-endpoint-name my-endpoint
```

For Windows:

```
aws rds describe-db-proxy-endpoints ^
  --db-proxy-endpoint-name my-endpoint
```

RDS API

To describe one or more proxy endpoints, use the RDS API DescribeDBProxyEndpoints operation.

Modifying a proxy endpoint

Console

To modify one or more proxy endpoints

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Proxies**.
3. In the list, choose the proxy whose endpoint you want to modify. Click the proxy name to view its
4. In the **Proxy endpoints** section, choose the endpoint that you want to modify. You can select it in the list, or click its name to view the details page.
5. On the proxy details page, under the **Proxy endpoints** section, choose **Edit**. Or, on the proxy endpoint details page, for **Actions**, choose **Edit**.
6. Change the values of the parameters that you want to modify.
7. Choose **Save changes**.

AWS CLI

To modify a proxy endpoint, use the AWS CLI modify-db-proxy-endpoint command with the following required parameters:

- `--db-proxy-endpoint-name`

Specify changes to the endpoint properties by using one or more of the following parameters:

- `--new-db-proxy-endpoint-name`
- `--vpc-security-group-ids`. Separate the security group IDs with spaces.

The following example renames the `my-endpoint` proxy endpoint to `new-endpoint-name`.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-proxy-endpoint \
  --db-proxy-endpoint-name my-endpoint \
  --new-db-proxy-endpoint-name new-endpoint-name
```

For Windows:

```
aws rds modify-db-proxy-endpoint ^
  --db-proxy-endpoint-name my-endpoint ^
  --new-db-proxy-endpoint-name new-endpoint-name
```

RDS API

To modify a proxy endpoint, use the RDS API ModifyDBProxyEndpoint operation.

Deleting a proxy endpoint

You can delete an endpoint for your proxy using the console as described following.

> **Note**
>
> You can't delete the default proxy endpoint that RDS Proxy automatically creates for each proxy.
> When you delete a proxy, RDS Proxy automatically deletes all the associated endpoints.

Console

To delete a proxy endpoint using the AWS Management Console

1. In the navigation pane, choose **Proxies**.
2. In the list, choose the proxy whose endpoint you want to endpoint. Click the proxy name to view its details page.
3. In the **Proxy endpoints** section, choose the endpoint that you want to delete. You can select one or more endpoints in the list, or click the name of a single endpoint to view the details page.

4. On the proxy details page, under the **Proxy endpoints** section, choose **Delete**. Or, on the proxy endpoint details page, for **Actions**, choose **Delete**.

AWS CLI

To delete a proxy endpoint, run the delete-db-proxy-endpoint command with the following required parameters:

- `--db-proxy-endpoint-name`

The following command deletes the proxy endpoint named `my-endpoint`.

For Linux, macOS, or Unix:

```
aws rds delete-db-proxy-endpoint \
  --db-proxy-endpoint-name my-endpoint
```

For Windows:

```
aws rds delete-db-proxy-endpoint ^
  --db-proxy-endpoint-name my-endpoint
```

RDS API

To delete a proxy endpoint with the RDS API, run the DeleteDBProxyEndpoint operation. Specify the name of the proxy endpoint for the `DBProxyEndpointName` parameter.

Limitations for proxy endpoints

RDS Proxy endpoints have the following limitations:

- Each proxy has a default endpoint that you can modify but not create or delete.
- The maximum number of user-defined endpoints for a proxy is 20. Thus, a proxy can have up to 21 endpoints: the default endpoint, plus 20 that you create.
- When you associate additional endpoints with a proxy, RDS Proxy automatically determines which DB instances in your cluster to use for each endpoint. You can't choose specific instances the way that you can with Aurora custom endpoints.

- Reader endpoints aren't available for Aurora multi-writer clusters.

Monitoring RDS Proxy metrics with Amazon CloudWatch

You can monitor RDS Proxy by using Amazon CloudWatch. CloudWatch collects and processes raw data from the proxies into readable, near-real-time metrics. To find these metrics in the CloudWatch console, choose **Metrics**, then choose **RDS**, and choose **Per-Proxy Metrics**. For more information, see Using Amazon CloudWatch metrics in the Amazon CloudWatch User Guide.

> **Note**
>
> RDS publishes these metrics for each underlying Amazon EC2 instance associated with a proxy. A single proxy might be served by more than one EC2 instance. Use CloudWatch statistics to aggregate the values for a proxy across all the associated instances.
> Some of these metrics might not be visible until after the first successful connection by a proxy.

In the RDS Proxy logs, each entry is prefixed with the name of the associated proxy endpoint. This name can be the name you specified for a user-defined endpoint, or the special name `default` for the default endpoint of a proxy that performs read/write requests.

All RDS Proxy metrics are in the group `proxy`.

Each proxy endpoint has its own CloudWatch metrics. You can monitor the usage of each proxy endpoint independently. For more information about proxy endpoints, see Working with Amazon RDS Proxy endpoints.

You can aggregate the values for each metric using one of the following dimension sets. For example, by using the `ProxyName` dimension set, you can analyze all the traffic for a particular proxy. By using the other dimension sets, you can split the metrics in different ways. You can split the metrics based on the different endpoints or target databases of each proxy, or the read/write and read-only traffic to each database.

- Dimension set 1: `ProxyName`
- Dimension set 2: `ProxyName`, `EndpointName`
- Dimension set 3: `ProxyName`, `TargetGroup`, `Target`

- Dimension set 4: `ProxyName`, `TargetGroup`, `TargetRole`

Metric	Description	Valid period	CloudWatch dimension set
`AvailabilityPercentage`	The percentage of time for which the target group was available in the role indicated by the dimension. This metric is reported every minute. The most useful statistic for this metric is `Average`.	1 minute	Dimension set 4
`ClientConnections`	The current number of client connections. This metric is reported every minute. The most useful statistic for this metric is `Sum`.	1 minute	Dimension set 1, Dimension set 2
`ClientConnectionsClosed`	The number of client connections closed. The most useful statistic for this metric is `Sum`.	1 minute and above	Dimension set 1, Dimension set 2
`ClientConnectionsNoTLS`	The current number of client connections without Transport Layer Security (TLS). This metric is reported every	1 minute and above	Dimension set 1, Dimension set 2

Metric	Description	Valid period	CloudWatch dimension set
	minute. The most useful statistic for this metric is Sum.		
ClientConnectionsReceived	The number of client connection requests received. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2
ClientConnectionsSetupFailedAuth	The number of client connection attempts that failed due to misconfigured authentication or TLS. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2
ClientConnectionsSetupSucceeded	The number of client connections successfully established with any authentication mechanism with or without TLS. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2
ClientConnectionsTLS	The current number of client connections with TLS. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2

Metric	Description	Valid period	CloudWatch dimension set
`DatabaseConnectionRequests`	The number of requests to create a database connection. The most useful statistic for this metric is `Sum`.	1 minute and above	Dimension set 1, Dimension set 3, Dimension set 4
`DatabaseConnectionRequestsWithTLS`	The number of requests to create a database connection with TLS. The most useful statistic for this metric is `Sum`.	1 minute and above	Dimension set 1, Dimension set 3, Dimension set 4
`DatabaseConnections`	The current number of database connections. This metric is reported every minute. The most useful statistic for this metric is `Sum`.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4
`DatabaseConnectionsBorrowLatency`	The time in microseconds that it takes for the proxy being monitored to get a database connection. The most useful statistic for this metric is `Average`.	1 minute and above	Dimension set 1, Dimension set 2

Metric	Description	Valid period	CloudWatch dimension set
DatabaseConnectionsCurrentlyBorrowed	The current number of database connections in the borrow state. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4
DatabaseConnectionsCurrentlyInTransaction	The current number of database connections in a transaction. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4
DatabaseConnectionsCurrentlySessionPinned	The current number of database connections currently pinned because of operations in client requests that change session state. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4

Metric	Description	Valid period	CloudWatch dimension set
`DatabaseConnectionsSetupFailed`	The number of database connection requests that failed. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 3, Dimension set 4
`DatabaseConnectionsSetupSucceeded`	The number of database connections successfully established with or without TLS. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 3, Dimension set 4
`DatabaseConnectionsWithTLS`	The current number of database connections with TLS. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4
`MaxDatabaseConnectionsAllowed`	The maximum number of database connections allowed. This metric is reported every minute. The most useful statistic for this metric is Sum.	1 minute	Dimension set 1, Dimension set 3, Dimension set 4

Metric	Description	Valid period	CloudWatch dimension set
QueryDatabaseResponseLatency	The time in microseconds that the database took to respond to the query. The most useful statistic for this metric is Average.	1 minute and above	Dimension set 1, Dimension set 2, Dimension set 3, Dimension set 4
QueryRequests	The number of queries received. A query including multiple statements is counted as one query. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2
QueryRequestsNoTLS	The number of queries received from non-TLS connections. A query including multiple statements is counted as one query. The most useful statistic for this metric is Sum.	1 minute and above	Dimension set 1, Dimension set 2

Metric	Description	Valid period	CloudWatch dimension set
`QueryRequestsTLS`	The number of queries received from TLS connections. A query including multiple statements is counted as one query. The most useful statistic for this metric is `Sum`.	1 minute and above	Dimension set 1, Dimension set 2
`QueryResponseLatency`	The time in microseconds between getting a query request and the proxy responding to it. The most useful statistic for this metric is `Average`.	1 minute and above	Dimension set 1, Dimension set 2

You can find logs of RDS Proxy activity under CloudWatch in the AWS Management Console. Each proxy has an entry in the **Log groups** page.

⚠ **Important**

These logs are intended for human consumption for troubleshooting purposes and not for programmatic access. The format and content of the logs is subject to change.
In particular, older logs don't contain any prefixes indicating the endpoint for each request. In newer logs, each entry is prefixed with the name of the associated proxy endpoint. This name can be the name that you specified for a user-defined endpoint, or the special name `default` for requests using the default endpoint of a proxy.

Working with RDS Proxy events

An *event* indicates a change in an environment such as an AWS environment or a service or application from a software as a service (SaaS) partner. Or, it can be one of your own custom applications or services. For example, Amazon Aurora generates an event when you create or modify an RDS Proxy. Amazon Aurora delivers events to Amazon EventBridge in near-real time. Following, you can find a list of RDS Proxy events that you can subscribe to and an example of an RDS Proxy event.

For more information about working with events, see the following:

- For instructions on how to view events by using the AWS Management Console, AWS CLI, or RDS API, see Viewing Amazon RDS events.
- To learn how to configure Amazon Aurora to send events to EventBridge, see Creating a rule that triggers on an Amazon Aurora event.

RDS Proxy events

The following table shows the event category and a list of events when an RDS Proxy is the source type.

Category	RDS event ID	Message	Notes
configuration change	RDS-EVENT-0204	RDS modified DB proxy *name*.	
configuration change	RDS-EVENT-0207	RDS modified the end point of the DB proxy *name*.	
configuration change	RDS-EVENT-0213	RDS detected the addition of the DB instance and automatically added it to the target group of the DB proxy *name*.	
configuration change	RDS-EVENT-0213	RDS detected creation of DB instance *name* and	

Category	RDS event ID	Message	Notes
		automatically added it to target group *name* of DB proxy *name*.	
configuration change	RDS-EVENT-0214	RDS detected deletion of DB instance *name* and automatically removed it from target group *name* of DB proxy *name*.	
configuration change	RDS-EVENT-0215	RDS detected deletion of DB cluster *name* and automatically removed it from target group *name* of DB proxy *name*.	
creation	RDS-EVENT-0203	RDS created DB proxy *name*.	
creation	RDS-EVENT-0206	RDS created endpoint *name* for DB proxy *name*.	
deletion	RDS-EVENT-0205	RDS deleted DB proxy *name*.	
deletion	RDS-EVENT-0208	RDS deleted endpoint *name* for DB proxy *name*.	

Category	RDS event ID	Message	Notes
failure	RDS-EVENT-0243	RDS failed to provision capacity for proxy *name* because there aren't enough IP addresses available in your subnets: *name*. To fix the issue, make sure that your subnets have the minimum number of unused IP addresses as recommended in the RDS Proxy documentation.	To determine the recommended number for your instance class, see Planning for IP address capacity.
failure	RDS-EVENT-0275	RDS throttled some connections to DB proxy *name*. The number of simultaneous connection requests from the client to the proxy has exceeded the limit.	

The following is an example of an RDS Proxy event in JSON format. The event shows that RDS modified the endpoint named `my-endpoint` of the RDS Proxy named `my-rds-proxy`. The event ID is RDS-EVENT-0207.

```
{
  "version": "0",
  "id": "68f6e973-1a0c-d37b-f2f2-94a7f62ffd4e",
  "detail-type": "RDS DB Proxy Event",
  "source": "aws.rds",
  "account": "123456789012",
  "time": "2018-09-27T22:36:43Z",
  "region": "us-east-1",
  "resources": [
     "arn:aws:rds:us-east-1:123456789012:db-proxy:my-rds-proxy"
  ],
  "detail": {
    "EventCategories": [
      "configuration change"
    ],
    "SourceType": "DB_PROXY",
    "SourceArn": "arn:aws:rds:us-east-1:123456789012:db-proxy:my-rds-proxy",
    "Date": "2018-09-27T22:36:43.292Z",
    "Message": "RDS modified endpoint my-endpoint of DB Proxy my-rds-proxy.",
    "SourceIdentifier": "my-endpoint",
    "EventID": "RDS-EVENT-0207"
  }
}
```

RDS Proxy command-line examples

To see how combinations of connection commands and SQL statements interact with RDS Proxy, look at the following examples.

Examples

- Preserving Connections to a MySQL Database Across a Failover
- Adjusting the max_connections Setting for an Aurora DB Cluster

Example Preserving connections to a MySQL database across a failover

This MySQL example demonstrates how open connections continue working during a failover. An example is when you reboot a database or it becomes unavailable due to a problem. This example uses a proxy named `the-proxy` and an Aurora DB cluster with DB instances `instance-8898` and `instance-9814`. When you run the `failover-db-cluster` command from the Linux command line, the writer instance that the proxy is connected to changes to a different DB instance. You can see that the DB instance associated with the proxy changes while the connection remains open.

```
$ mysql -h the-proxy.proxy-demo.us-east-1.rds.amazonaws.com -u admin_user -p
Enter password:
...

mysql> select @@aurora_server_id;
+--------------------+
| @@aurora_server_id |
+--------------------+
| instance-9814       |
+---------------------+
1 row in set (0.01 sec)

mysql>
[1]+  Stopped                 mysql -h the-proxy.proxy-demo.us-east-1.rds.amazonaws.com
 -u admin_user -p
$ # Initially, instance-9814 is the writer.
$ aws rds failover-db-cluster --db-cluster-identifier cluster-56-2019-11-14-1399
JSON output
$ # After a short time, the console shows that the failover operation is complete.
$ # Now instance-8898 is the writer.
$ fg
mysql -h the-proxy.proxy-demo.us.us-east-1.rds.amazonaws.com -u admin_user -p

mysql> select @@aurora_server_id;
+---------------------+
| @@aurora_server_id |
+---------------------+
| instance-8898       |
+---------------------+
1 row in set (0.01 sec)

mysql>
[1]+  Stopped                 mysql -h the-proxy.proxy-demo.us-east-1.rds.amazonaws.com
 -u admin_user -p
$ aws rds failover-db-cluster --db-cluster-identifier cluster-56-2019-11-14-1399
JSON output
$ # After a short time, the console shows that the failover operation is complete.
$ # Now instance-9814 is the writer again.
$ fg
mysql -h the-proxy.proxy-demo.us-east-1.rds.amazonaws.com -u admin_user -p

mysql> select @@aurora_server_id;
+---------------------+
| @@aurora_server_id |
+---------------------+
| instance-9814       |
+---------------------+
1 row in set (0.01 sec)
+---------------+---------------+
| Variable_name | Value         |
+---------------+---------------+
| hostname      | ip-10-1-3-178 |
+---------------+---------------+
1 row in set (0.02 sec)
```

Example Adjusting the max_connections setting for an Aurora DB cluster

This example demonstrates how you can adjust the `max_connections` setting for an Aurora MySQL DB cluster. To do so, you create your own DB cluster parameter group based on the default parameter settings for clusters that are compatible with MySQL 5.7. You specify a value for the `max_connections` setting, overriding the formula that sets the default value. You associate the DB cluster parameter group with your DB cluster.

```
export REGION=us-east-1
export CLUSTER_PARAM_GROUP=rds-proxy-mysql-57-max-connections-demo
export CLUSTER_NAME=rds-proxy-mysql-57

aws rds create-db-parameter-group --region $REGION \
  --db-parameter-group-family aurora-mysql5.7 \
  --db-parameter-group-name $CLUSTER_PARAM_GROUP \
  --description "Aurora MySQL 5.7 cluster parameter group for RDS Proxy demo."

aws rds modify-db-cluster --region $REGION \
  --db-cluster-identifier $CLUSTER_NAME \
  --db-cluster-parameter-group-name $CLUSTER_PARAM_GROUP

echo "New cluster param group is assigned to cluster:"
aws rds describe-db-clusters --region $REGION \
  --db-cluster-identifier $CLUSTER_NAME \
  --query '*[*].{DBClusterParameterGroup:DBClusterParameterGroup}'

echo "Current value for max_connections:"
aws rds describe-db-cluster-parameters --region $REGION \
  --db-cluster-parameter-group-name $CLUSTER_PARAM_GROUP \
  --query '*[*].{ParameterName:ParameterName,ParameterValue:ParameterValue}' \
  --output text | grep "^max_connections"

echo -n "Enter number for max_connections setting: "
read answer

aws rds modify-db-cluster-parameter-group --region $REGION --db-cluster-parameter-
group-name $CLUSTER_PARAM_GROUP \
  --parameters "ParameterName=max_connections,ParameterValue=$
$answer,ApplyMethod=immediate"
```

```
echo "Updated value for max_connections:"
aws rds describe-db-cluster-parameters --region $REGION \
  --db-cluster-parameter-group-name $CLUSTER_PARAM_GROUP \
  --query '*[*].{ParameterName:ParameterName,ParameterValue:ParameterValue}' \
  --output text | grep "^max_connections"
```

Troubleshooting for RDS Proxy

Following, you can find troubleshooting ideas for some common RDS Proxy issues and information on CloudWatch logs for RDS Proxy.

In the RDS Proxy logs, each entry is prefixed with the name of the associated proxy endpoint. This name can be the name that you specified for a user-defined endpoint. Or, it can be the special name `default` for the default endpoint of a proxy that performs read/write requests. For more information about proxy endpoints, see Working with Amazon RDS Proxy endpoints.

Topics

Verifying connectivity for a proxy

You can use the following commands to verify that all components such as the proxy, database, and compute instances in the connection can communicate with the each other.

Examine the proxy itself using the describe-db-proxies command. Also examine the associated target group using the describe-db-proxy-target-groups command. Check that the details of the targets match the Aurora cluster that you intend to associate with the proxy. Use commands such as the following.

```
aws rds describe-db-proxies --db-proxy-name $DB_PROXY_NAME
aws rds describe-db-proxy-target-groups --db-proxy-name $DB_PROXY_NAME
```

To confirm that the proxy can connect to the underlying database, examine the targets specified in the target groups using the describe-db-proxy-targets command. Use a command such as the following.

```
aws rds describe-db-proxy-targets --db-proxy-name $DB_PROXY_NAME
```

The output of the describe-db-proxy-targets command includes a TargetHealth field. You can examine the fields State, Reason, and Description inside TargetHealth to check if the proxy can communicate with the underlying DB instance.

- A State value of AVAILABLE indicates that the proxy can connect to the DB instance.
- A State value of UNAVAILABLE indicates a temporary or permanent connection problem. In this case, examine the Reason and Description fields. For example, if Reason has a value of PENDING_PROXY_CAPACITY, try connecting again after the proxy finishes its scaling operation. If Reason has a value of UNREACHABLE, CONNECTION_FAILED, or AUTH_FAILURE, use the explanation from the Description field to help you diagnose the issue.
- The State field might have a value of REGISTERING for a brief time before changing to AVAILABLE or UNAVAILABLE.

If the following Netcat command (nc) is successful, you can access the proxy endpoint from the EC2 instance or other system where you're logged in. This command reports failure if you're not in the same VPC as the proxy and the associated database. You might be able to log directly in to the database without being in the same VPC. However, you can't log into the proxy unless you're in the same VPC.

```
nc -zx MySQL_proxy_endpoint 3306

nc -zx PostgreSQL_proxy_endpoint 5432
```

You can use the following commands to make sure that your EC2 instance has the required properties. In particular, the VPC for the EC2 instance must be the same as the VPC for the RDS DB instance Aurora cluster that the proxy connects to.

```
aws ec2 describe-instances --instance-ids your_ec2_instance_id
```

Examine the Secrets Manager secrets used for the proxy.

```
aws secretsmanager list-secrets
aws secretsmanager get-secret-value --secret-id your_secret_id
```

Make sure that the SecretString field displayed by get-secret-value is encoded as a JSON string that includes the username and password fields. The following example shows the format of the SecretString field.

```
{
  "ARN": "some_arn",
  "Name": "some_name",
  "VersionId": "some_version_id",
  "SecretString": '{"username":"some_username","password":"some_password"}',
  "VersionStages": [ "some_stage" ],
  "CreatedDate": some_timestamp
}
```

Common issues and solutions

This section describes some common issues and potential solutions when using RDS Proxy.

After running the `aws rds describe-db-proxy-targets` CLI command, if the `TargetHealth` description states `Proxy does not have any registered credentials`, verify the following:

- There are credentials registered for the user to access the proxy.
- The IAM role to access the Secrets Manager secret used by the proxy is valid.

You might encounter the following RDS events while creating or connecting to a DB proxy.

Category	RDS event ID	Description
failure	RDS-EVENT-0243	RDS couldn't provision capacity for the proxy because there aren't enough IP addresses available in your subnets. To fix the issue, make sure that your subnets have the minimum number of unused IP addresses. To determine the recommend ed number for your instance class, see Planning for IP address capacity.

Category	RDS event ID	Description
failure	RDS-EVENT-0275	RDS throttled some connections to DB proxy *name*. The number of simultaneous connection requests from the client to the proxy has exceeded the limit.

You might encounter the following issues while creating a new proxy or connecting to a proxy.

Error	Causes or workarounds
`403: The security token included in the request is invalid`	Select an existing IAM role instead of choosing to create a new one.

You might encounter the following issues while connecting to a MySQL proxy.

Error	Causes or workarounds
`ERROR 1040 (HY000): Connections rate limit exceeded (`*`limit_value`*` )`	The rate of connection requests from the client to the proxy has exceeded the limit.
`ERROR 1040 (HY000): IAM authentication rate limit exceeded`	The number of simultaneous requests with IAM authentication from the client to the proxy has exceeded the limit.

Error	Causes or workarounds
ERROR 1040 (HY000): Number simultaneous connections exceeded (*limit_value*)	The number of simultaneous connection requests from the client to the proxy exceeded the limit.
ERROR 1045 (28000): Access denied for user '*DB_USER*'@'%' (usi password: YES)	The Secrets Manager secret used by the proxy doesn't match the user name and password of an existing database user. Either update the credentials in the Secrets Manager secret, or make sure the database user exists and has the same password as in the secret.
ERROR 1105 (HY000): Unknown error	An unknown error occurred.
ERROR 1231 (42000): Variable ''character_set_client'' can't be set to the value of *value*	The value set for the `character_set_client` parameter is not valid. For example, the value `ucs2` is not valid because it can crash the MySQL server.
ERROR 3159 (HY000): This RDS Proxy requires TLS connections.	You enabled the setting **Require Transport Layer Security** in the proxy but your connection included the parameter `ssl-mode=DISABLED` in the MySQL client. Do either of the following: • Disable the setting **Require Transport Layer Security** for the proxy. • Connect to the database using the minimum setting of `ssl-mode=REQUIRED` in the MySQL client.

Error	Causes or workarounds
`ERROR 2026 (HY000): SSL connection error: Internal Server Error`	The TLS handshake to the proxy failed. Some possible reasons include the following: - SSL is required but the server doesn't support it. - An internal server error occurred. - A bad handshake occurred.
`ERROR 9501 (HY000): Timed-out waiting to acquire database connection`	The proxy timed-out waiting to acquire a database connection. Some possible reasons include the following: - The proxy is unable to establish a database connection because the maximum connections have been reached - The proxy is unable to establish a database connection because the database is unavailable.

You might encounter the following issues while connecting to a PostgreSQL proxy.

Error	Cause	Solution
`IAM authentication is allowed only with SSL connections.`	The user tried to connect to the database using IAM authentication with the setting `sslmode=disable` in the PostgreSQL client.	The user needs to connect to the database using the minimum setting of `sslmode=require` in the PostgreSQL client. For more information, see the PostgreSQL SSL support documentation.
`This RDS Proxy requires TLS connections.`	The user enabled the option **Require Transport Layer Security** but tried to connect with `sslmode=disable` in the PostgreSQL client.	To fix this error, do one of the following: - Disable the proxy's **Require Transport Layer Security** option.

Error	Cause	Solution
		- Connect to the database using the minimum setting of `sslmode=allow` in the PostgreSQL client.
`IAM authentication failed for user` *`user_name`* `. Check the IAM token for this user and try again.`	This error might be due to the following reasons: - The client supplied the incorrect IAM user name. - The client supplied an incorrect IAM authorization token for the user. - The client is using an IAM policy that does not have the necessary permissions. - The client supplied an expired IAM authorization token for the user.	To fix this error, do the following: 1. Confirm that the provided IAM user exists. 2. Confirm that the IAM authorization token belongs to the provided IAM user. 3. Confirm that the IAM policy has adequate permissions for RDS. 4. Check the validity of the IAM authorization token used.
`This RDS proxy has no credentials for the role` *`role_name`* `. Check the credentials for this role and try again.`	There is no Secrets Manager secret for this role.	Add a Secrets Manager secret for this role. For more information, see Setting up AWS Identity and Access Management (IAM) policies.
`RDS supports only IAM, MD5, or SCRAM authentication.`	The database client being used to connect to the proxy is using an authentication mechanism not currently supported by the proxy.	If you're not using IAM authentication, use the MD5 or SCRAM password authentic ation.

Error	Cause	Solution
`A user name is missing from the connection startup packet. Provide a user name for this connection.`	The database client being used to connect to the proxy isn't sending a user name when trying to establish a connection.	Make sure to define a user name when setting up a connection to the proxy using the PostgreSQL client of your choice.
`Feature not supported : RDS Proxy supports only version 3.0 of the PostgreSQL messaging protocol.`	The PostgreSQL client used to connect to the proxy uses a protocol older than 3.0.	Use a newer PostgreSQL client that supports the 3.0 messaging protocol. If you're using the PostgreSQL `psql` CLI, use a version greater than or equal to 7.4.
`Feature not supported : RDS Proxy currently doesn't support streaming replication mode.`	The PostgreSQL client used to connect to the proxy is trying to use the streaming replication mode, which isn't currently supported by RDS Proxy.	Turn off the streaming replication mode in the PostgreSQL client being used to connect.
`Feature not supported : RDS Proxy currently doesn't support the option` *`option_name`* `.`	Through the startup message, the PostgreSQL client used to connect to the proxy is requesting an option that isn't currently supported by RDS Proxy.	Turn off the option being shown as not supported from the message above in the PostgreSQL client being used to connect.
`The IAM authentication failed because of too many competing requests.`	The number of simultaneous requests with IAM authentication from the client to the proxy has exceeded the limit.	Reduce the rate in which connections using IAM authentication from a PostgreSQL client are established.

Error	Cause	Solution
`The maximum number of client connections to the proxy exceeded` *number_value* `.`	The number of simultaneous connection requests from the client to the proxy exceeded the limit.	Reduce the number of active connections from PostgreSQL clients to this RDS proxy.
`Rate of connection to proxy exceeded` *number_value* `.`	The rate of connection requests from the client to the proxy has exceeded the limit.	Reduce the rate in which connections from a PostgreSQL client are established.
`The password that was provided for the role` *role_name* `is wrong.`	The password for this role doesn't match the Secrets Manager secret.	Check the secret for this role in Secrets Manager to see if the password is the same as what's being used in your PostgreSQL client.
`The IAM authentication failed for the role` *role_name* `. Check the IAM token for this role and try again.`	There is a problem with the IAM token used for IAM authentication.	Generate a new authentication token and use it in a new connection.
`IAM is allowed only with SSL connections.`	A client tried to connect using IAM authentication, but SSL wasn't enabled.	Enable SSL in the PostgreSQL client.
`Unknown error.`	An unknown error occurred.	Reach out to AWS Support to investigate the issue.

Error	Cause	Solution
`Timed-out waiting to acquire database connection.`	The proxy timed-out waiting to acquire a database connection. Some possible reasons include the following : - The proxy can't establish a database connection because the maximum connections have been reached. - The proxy can't establish a database connection because the database is unavailable.	Possible solutions are the following: - Check the target of the RDS DB instance Aurora cluster status to see if it's unavailable. - Check if there are long-running transactions and/or queries being executed. They can use database connections from the connection pool for a long time.
`Request returned an error: ` *`database_error`* `.`	The database connection established from the proxy returned an error.	The solution depends on the specific database error. One example is: `Request returned an error: database "your-database-name" does not exist`. This means that the specified database name doesn't exist on the database server. Or it means that the user name used as a database name (if a database name isn't specified) doesn't exist on the server.

Using RDS Proxy with AWS CloudFormation

You can use RDS Proxy with AWS CloudFormation. This helps you to create groups of related resources. Such a group can include a proxy that can connect to a newly created Aurora DB cluster. RDS Proxy support in AWS CloudFormation involves two new registry types: `DBProxy` and `DBProxyTargetGroup`.

The following listing shows a sample AWS CloudFormation template for RDS Proxy.

```
Resources:
 DBProxy:
   Type: AWS::RDS::DBProxy
   Properties:
     DBProxyName: CanaryProxy
     EngineFamily: MYSQL
     RoleArn:
      Fn::ImportValue: SecretReaderRoleArn
     Auth:
       - {AuthScheme: SECRETS, SecretArn: !ImportValue ProxySecret, IAMAuth: DISABLED}
     VpcSubnetIds:
       Fn::Split: [",", "Fn::ImportValue": SubnetIds]

 ProxyTargetGroup:
   Type: AWS::RDS::DBProxyTargetGroup
   Properties:
     DBProxyName: CanaryProxy
     TargetGroupName: default
     DBInstanceIdentifiers:
       - Fn::ImportValue: DBInstanceName
   DependsOn: DBProxy
```

For more information about the resources in this sample, see `DBProxy` and `DBProxyTargetGroup`.

For more information about resources that you can create using AWS CloudFormation, see RDS resource type reference.

Using RDS Proxy with Aurora global databases

An *Aurora global database* is a single database that spans multiple AWS Regions, allowing for low-latency global reads and disaster recovery from any Region-wide outage. It provides built-in fault tolerance for your deployment because the DB instance relies not on a single AWS Region, but upon multiple Regions and different Availability Zones. For more information, see Using Amazon Aurora global databases.

You can use RDS Proxy with any DB cluster in an Aurora global database. Before you begin to use these features together, make sure that you familiarize yourself with the following information.

> ⚠ **Important**
>
> If the DB cluster is part of a global database with write forwarding turned on, reduce your proxy's `MaxConnectionsPercent` value by the quota that's allotted for write forwarding. The write forwarding quota is set in the DB cluster parameter `aurora_fwd_writer_max_connections_pct`. For information about write forwarding, see Using write forwarding in an Amazon Aurora global database.

Limitations for RDS Proxy with global databases

When the Aurora DB cluster has write forwarding turned on, RDS Proxy doesn't support the `SESSION` value for the `aurora_replica_read_consistency` variable. Setting this value can cause unexpected behavior.

How RDS Proxy endpoints work with global databases

When you understand how RDS Proxy endpoints work with global databases, you can better manage your applications that use Aurora databases with both of these features.

For a proxy with a global database's primary cluster as the registered target, the proxy endpoints work the same way as with any Aurora DB cluster. The proxy's read/write endpoints send all requests to the writer instance of the cluster. The proxy's read-only endpoints send all requests to the reader instances. If a reader becomes unavailable while a connection is open, RDS Proxy redirects subsequent queries on the connection to another reader instance. For a proxy with a secondary cluster as the registered target, requests that are sent to the proxy's read-only endpoints are also sent to the reader instances. Because the cluster has no writer instances, requests that are sent to read/write endpoints fail with the error "`The target group doesn't have any associated read/write instances`".

Global database switchover and failover operations both involve a role switch between the primary and one of the secondary DB clusters. When the selected secondary cluster becomes the new primary, one of its reader instances is promoted to a writer. This DB instance is now the new writer instance for the global cluster. Make sure to redirect your application's write operations to the appropriate read/write endpoint of the proxy that's associated with the new primary cluster. This proxy endpoint might be the default endpoint or a custom read/write endpoint.

RDS Proxy queues all requests through read/write endpoints and sends them to the writer instance of the new primary cluster as soon as it's available. It does so regardless of whether the switchover or failover operation has completed. During switchover or failover, the default endpoint of the proxy for the old primary cluster still accepts write operations. However, as soon as that cluster becomes a secondary cluster, all of the write operations fail. To learn how and when to perform specific global switchover or failover tasks, see the following topics:

- Global database switchover – Performing switchovers for Amazon Aurora global databases
- Global database failover – Recovering an Amazon Aurora global database from an unplanned outage

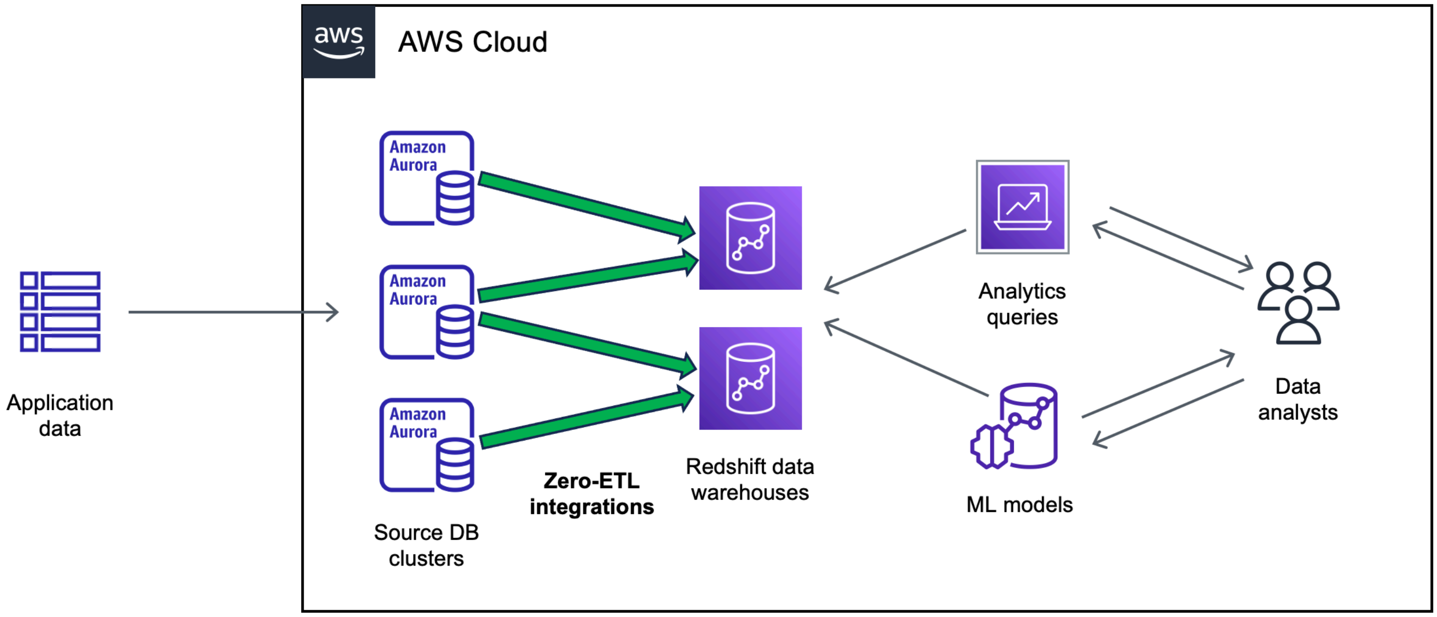
Working with Aurora zero-ETL integrations with Amazon Redshift

An Aurora zero-ETL integration with Amazon Redshift enables near real-time analytics and machine learning (ML) using Amazon Redshift on petabytes of transactional data from Aurora. It's a fully managed solution for making transactional data available in Amazon Redshift after it is written to an Aurora DB cluster. *Extract, transform,* and *load* (ETL) is the process of combining data from multiple sources into a large, central data warehouse.

A zero-ETL integration makes the data in your Aurora DB cluster available in Amazon Redshift in near real-time. Once that data is in Amazon Redshift, you can power your analytics, ML, and AI workloads using the built-in capabilities of Amazon Redshift, such as machine learning, materialized views, data sharing, federated access to multiple data stores and data lakes, and integrations with Amazon SageMaker, Amazon QuickSight, and other AWS services.

To create a zero-ETL integration, you specify an Aurora DB cluster as the *source*, and an Amazon Redshift data warehouse as the *target*. The integration replicates data from the source database into the target data warehouse.

The following diagram illustrates this functionality:



The integration monitors the health of the data pipeline and recovers from issues when possible. You can create integrations from multiple Aurora DB clusters into a single Amazon Redshift namespace, enabling you to derive insights across multiple applications.

For information about pricing for zero-ETL integrations, see Amazon Aurora pricing and Amazon Redshift pricing.

Topics

Benefits

Aurora zero-ETL integrations with Amazon Redshift have the following benefits:

- Help you derive holistic insights from multiple data sources.
- Eliminate the need to build and maintain complex data pipelines that perform extract, transform, and load (ETL) operations. Zero-ETL integrations remove the challenges that come with building and managing pipelines by provisioning and managing them for you.
- Reduce operational burden and cost, and let you focus on improving your applications.
- Let you leverage Amazon Redshift's analytics and ML capabilities to derive insights from transactional and other data, to respond effectively to critical, time-sensitive events.

Key concepts

As you get started with zero-ETL integrations, consider the following concepts:

Integration

A fully managed data pipeline that automatically replicates transactional data and schemas from an Aurora DB cluster to an Amazon Redshift data warehouse.

Source DB cluster

The Aurora DB cluster where data is replicated from. For Aurora MySQL, you can specify a DB cluster that uses provisioned DB instances or Aurora Serverless v2 DB instances as the source. For the Aurora PostgreSQL preview, you can only specify a cluster that uses provisioned DB instances.

Target data warehouse

The Amazon Redshift data warehouse where the data is replicated to. There are two types of data warehouse: a provisioned cluster data warehouse and a serverless data warehouse. A provisioned cluster data warehouse is a collection of computing resources called nodes, which are organized into a group called a *cluster*. A serverless data warehouse is comprised of a workgroup that stores compute resources, and a namespace that houses the database objects and users. Both data warehouses run an Amazon Redshift engine and contain one or more databases.

Multiple source DB clusters can write to the same target.

For more information, see Data warehouse system architecture in the *Amazon Redshift Developer Guide*.

Limitations

The following limitations apply to Aurora zero-ETL integrations with Amazon Redshift.

Topics

General limitations

- The source DB cluster must be in the same Region as the target Amazon Redshift data warehouse.
- You can't rename a DB cluster or any of its instances if it has existing integrations.
- You can't delete a DB cluster that has existing integrations. You must delete all associated integrations first.
- If you stop the source DB cluster, the last few transactions might not be replicated to the target data warehouse until you resume the cluster.
- If your cluster is the source of a blue/green deployment, the blue and green environments can't have existing zero-ETL integrations during switchover. You must delete the integration first and switch over, then recreate it.
- A DB cluster must contain at least one DB instance in order to be the source of an integration.
- If your source cluster is the primary DB cluster in an Aurora global database and it fails over to one of its secondary clusters, the integration becomes inactive. You must delete and recreate the integration.
- You can't create an integration for a source database that has another integration being actively created.
- When you initially create an integration, or when a table is being resynchronized, data seeding from the source to the target can take 20-25 minutes or more depending on the size of the source database. This delay can lead to increased replica lag.
- Some data types aren't supported. For more information, see the section called "Data type differences".
- Foreign key references with predefined table updates aren't supported. Specifically, `ON DELETE` and `ON UPDATE` rules aren't supported with `CASCADE`, `SET NULL`, and `SET DEFAULT` actions. Attempting to create or update a table with such references to another table will put the table into a failed state.
- `ALTER TABLE` partition operations cause your table to resynchronize in order to reload data from Aurora to Amazon Redshift. The table will be unavailable for querying while it's resynchronizing. For more information, see the section called "One or more of my Amazon Redshift tables requires a resync".
- XA transaction aren't supported.
- Object identifiers (including database name, table name, column names, and others) can contain only alphanumeric characters, numbers, $, and _ (underscore).

Aurora MySQL limitations

- Your source DB cluster must be running Aurora MySQL version 3.05 (compatible with MySQL 8.0.32) or higher.
- Zero-ETL integrations rely on MySQL binary logging (binlog) to capture ongoing data changes. Don't use binlog-based data filtering, as it can cause data inconsistencies between the source and target databases.
- Aurora MySQL system tables, temporary tables, and views aren't replicated to Amazon Redshift.
- Zero-ETL integrations are supported only for databases configured to use the InnoDB storage engine.

Aurora PostgreSQL preview limitations

> ⚠ **Important**
>
> The zero-ETL integrations with Amazon Redshift feature for Aurora PostgreSQL is in preview release. The documentation and the feature are both subject to change. You can use this feature only in test environments, not in production environments. For preview terms and conditions, see *Betas and Previews* in AWS Service Terms.

- Your source DB cluster must be running **Aurora PostgreSQL (compatible with PostgreSQL 15.4 and Zero-ETL Support)**.
- You can create and manage zero-ETL integrations for Aurora PostgreSQL only in the Amazon RDS Database Preview Environment, in the US East (Ohio) (us-east-2) AWS Region. You can use the preview environment to test beta, release candidate, and early production versions of PostgreSQL database engine software.
- You can create and manage integrations for Aurora PostgreSQL only using the AWS Management Console. You can't use the AWS Command Line Interface (AWS CLI), the Amazon RDS API, or any of the AWS SDKs.
- When you create a source DB cluster, the parameter group that you choose must already have the required DB cluster parameter values configured. You can't create a new parameter group afterwards and then associate it with the cluster. For a list of required parameters, see the section called "Step 1: Create a custom DB cluster parameter group".

- You can't modify an integration after you create it. If you need to change certain settings, you must delete and recreate the integration.
- Currently, Aurora PostgreSQL DB clusters that are the source of an integration don't perform garbage collection of logical replication data.
- All databases created within the source Aurora PostgreSQL DB cluster must use UTF-8 encoding.
- Column names can't contain any of the following characters: commas (,), semicolons (;), parentheses ( ), curly brackets { }, newlines (\n), tabs (\t), equal signs (=), and spaces.
- Zero-ETL integrations with Aurora PostgreSQL don't support the following:
  - Aurora Serverless v2 DB instances. Your source DB cluster must use provisioned DB instances.
  - Custom data types or data types created by extensions.
  - [Subtransactions](#) on the source DB cluster.
  - Renaming of schemas or databases within a source DB cluster.
  - Restoring from a DB cluster snapshot or using Aurora cloning to create a source DB cluster. If you want to bring existing data into a preview cluster, then you must use the `pg_dump` or `pg_restore` utilities.
  - Creation of logical replication slots on the writer instance of the source DB cluster.
  - Large field values that require The Oversized-Attribute Storage Technique (TOAST).
  - `ALTER TABLE` partition operations. These operations can cause your table to resynchronize and eventually enter a `Failed` state. If a table fails, then you must drop and recreate it.

Amazon Redshift limitations

For a list of Amazon Redshift limitations related to zero-ETL integrations, see Considerations in the *Amazon Redshift Management Guide*.

Quotas

Your account has the following quotas related to Aurora zero-ETL integrations with Amazon Redshift. Each quota is per-Region unless otherwise specified.

Name	Default	Description
Integrations	100	The total number of integrations within an AWS account.

Name	Default	Description
Integrations per target data warehouse	50	The number of integrations sending data to a single target Amazon Redshift data warehouse.
Integrations per source cluster	5 for Aurora MySQL, 1 for Aurora PostgreSQL	The number of integrations sending data from a single source DB cluster.

In addition, Amazon Redshift places certain limits on the number of tables allowed in each DB instance or cluster node. For more information, see Quotas and limits in Amazon Redshift in the *Amazon Redshift Management Guide*.

Supported Regions

Aurora zero-ETL integrations with Amazon Redshift are available in a subset of AWS Regions. For a list of supported Regions, see the section called “Zero-ETL integrations ”.

Getting started with Aurora zero-ETL integrations with Amazon Redshift

Before you create a zero-ETL integration with Amazon Redshift, configure your Aurora DB cluster and your Amazon Redshift data warehouse with the required parameters and permissions. During setup, you'll complete the following steps:

1. Create a custom DB cluster parameter group.
2. Create a source DB cluster.
3. Create a target Amazon Redshift data warehouse.

After you complete these tasks, continue to the section called “Creating zero-ETL integrations”.

You can use the AWS SDKs to automate the setup process for you. For more information, see the section called “Set up an integration using the AWS SDKs (Aurora MySQL only)”.

Step 1: Create a custom DB cluster parameter group

Aurora zero-ETL integrations with Amazon Redshift require specific values for the DB cluster parameters that control replication. Specifically, Aurora MySQL requires *enhanced binlog* (`aurora_enhanced_binlog`), and Aurora PostgreSQL requires *enhanced logical replication* (`aurora.enhanced_logical_replication`).

To configure binary logging or logical replication, you must first create a custom DB cluster parameter group, and then associate it with the source DB cluster.

Create a custom DB cluster parameter group with the following settings depending on your source DB engine. For instructions to create a parameter group, see the section called “Working with DB cluster parameter groups”.

Aurora MySQL (aurora-mysql8.0 family):

- `aurora_enhanced_binlog=1`
- `binlog_backup=0`
- `binlog_format=ROW`
- `binlog_replication_globaldb=0`
- `binlog_row_image=full`
- `binlog_row_metadata=full`

In addition, make sure that the `binlog_transaction_compression` parameter is *not* set to `ON`, and that the `binlog_row_value_options` parameter is *not* set to `PARTIAL_JSON`.

For more information about Aurora MySQL enhanced binlog, see the section called “Setting up enhanced binlog”.

Aurora PostgreSQL (aurora-postgresql15 family):

> **Note**
>
> For Aurora PostgreSQL DB clusters, you must create the custom parameter group within the Amazon RDS Database Preview Environment, in the US East (Ohio) (us-east-2) AWS Region.

- `rds.logical_replication=1`
- `aurora.enhanced_logical_replication=1`
- `aurora.logical_replication_backup=0`
- `aurora.logical_replication_globaldb=0`

Enabling enhanced logical replication (`aurora.enhanced_logical_replication`) automatically sets the `REPLICA IDENTITY` parameter to `FULL`, which means that all column values are written to the write ahead log (WAL). This will increase the IOPS for your source DB cluster.

Step 2: Select or create a source DB cluster

After you create a custom DB cluster parameter group, choose or create an Aurora MySQL or Aurora PostgreSQL DB cluster. This cluster will be the source of data replication to Amazon Redshift.

The cluster must be running Aurora MySQL version 3.05 (compatible with MySQL 8.0.32) or higher, or Aurora PostgreSQL (compatible with PostgreSQL 15.4 and Zero-ETL Support). For instructions to create a DB cluster, see the section called "Creating a DB cluster".

> **Note**
>
> You must create Aurora PostgreSQL DB clusters within the Amazon RDS Database Preview Environment, in the US East (Ohio) (us-east-2) AWS Region.

Under **Additional configuration**, change the default **DB cluster parameter group** to the custom parameter group that you created in the previous step.

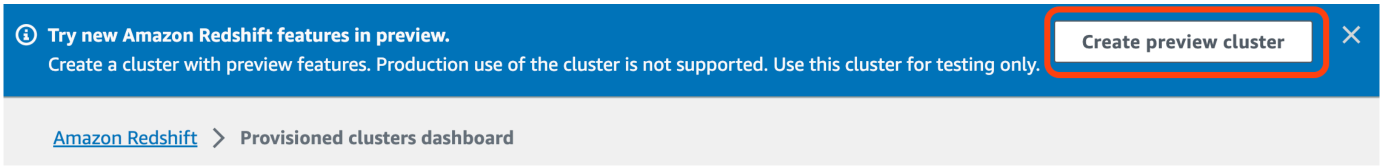
> **Note**
>
> For Aurora MySQL, if you associate the parameter group with the DB cluster *after* the cluster is already created, you must reboot the primary DB instance in the cluster to apply the changes before you can create a zero-ETL integration. For instructions, see the section called "Rebooting an Aurora DB cluster or instance".
> During the preview release of Aurora PostgreSQL zero-ETL integrations with Amazon Redshift, you must associate the cluster with the custom DB cluster parameter group *while*

creating the cluster. You can't perform this action after the source DB cluster is already created, otherwise you need to delete and recreate the cluster.

Step 3: Create a target Amazon Redshift data warehouse

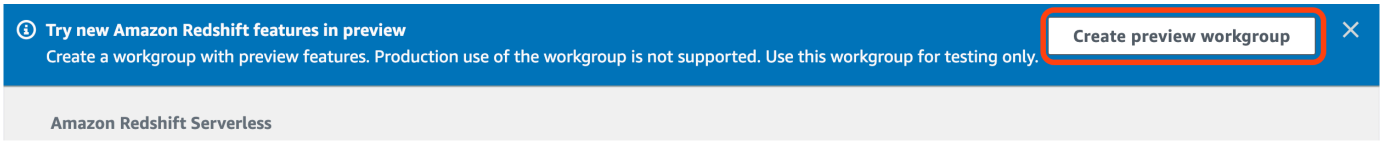
After you create your source DB cluster, you must create and configure a target data warehouse in Amazon Redshift. The data warehouse must meet the following requirements:

- Created in *preview* (for Aurora PostgreSQL sources only). For Aurora MySQL sources, you must create production clusters and workgroups.
  - To create a provisioned cluster in preview, choose **Create preview cluster** from the banner on the provisioned clusters dashboard. For more information, see Creating a preview cluster.

    Try new Amazon Redshift features in preview.
    Create a cluster with preview features. Production use of the cluster is not supported. Use this cluster for testing only.

    When creating the cluster, set the **Preview track** to `preview_2023`.
  - To create a Redshift Serverless workgroup in preview, choose **Create preview workgroup** from the banner on the Serverless dashboard. For more information, see Creating a preview workgroup.

    Try new Amazon Redshift features in preview
    Create a workgroup with preview features. Production use of the workgroup is not supported. Use this workgroup for testing only.

- Using an RA3 node type (`ra3.xlplus`, `ra3.4xlarge`, or `ra3.16xlarge`) , or Redshift Serverless.
- Encrypted (if using a provisioned cluster). For more information, see Amazon Redshift database encryption.

For instructions to create a data warehouse, see Creating a cluster for provisioned clusters, or Creating a workgroup with a namespace for Redshift Serverless.

Enable case sensitivity on the data warehouse

For the integration to be successful, the case sensitivity parameter (`enable_case_sensitive_identifier`) must be enabled for the data warehouse. By default, case sensitivity is disabled on all provisioned clusters and Redshift Serverless workgroups.

To enable case sensitivity, perform the following steps depending on your data warehouse type:

- **Provisioned cluster** – To enable case sensitivity on a provisioned cluster, create a custom parameter group with the `enable_case_sensitive_identifier` parameter enabled. Then, associate the parameter group with the cluster. For instructions, see Managing parameter groups using the console or Configuring parameter values using the AWS CLI.

  > **Note**
  >
  > Remember to reboot the cluster after you associate the custom parameter group with it.

- **Serverless workgroup** – To enable case sensitivity on a Redshift Serverless workgroup, you must use the AWS CLI. The Amazon Redshift console doesn't currently support modifying Redshift Serverless parameter values. Send the following update-workgroup request:

  ```
  aws redshift-serverless update-workgroup \
    --workgroup-name target-workgroup \
    --config-parameters
   parameterKey=enable_case_sensitive_identifier,parameterValue=true
  ```

  You don't need to reboot a workgroup after you modify its parameter values.

Configure authorization for the data warehouse

After you create a data warehouse, you must configure the source Aurora DB cluster as an authorized integration source. For instructions, see Configure authorization for your Amazon Redshift data warehouse.

Set up an integration using the AWS SDKs (Aurora MySQL only)

Rather than setting up each resource manually, you can run the following Python script to automatically set up the required resources for you. The code example uses the AWS SDK for Python (Boto3) to create a source Aurora MySQL DB cluster and target Amazon Redshift data warehouse, each with the required parameter values. It then waits for the clusters to be available before creating a zero-ETL integration between them. You can comment out different functions depending on which resources you need to set up.

To install the required dependencies, run the following commands:

```
pip install boto3
pip install time
```

Within the script, optionally modify the names of the source, target, and parameter groups. The final function creates an integration named `my-integration` after the resources are set up.

Python code example

```
import boto3
import time

# Build the client using the default credential configuration.
# You can use the CLI and run 'aws configure' to set access key, secret
# key, and default Region.

rds = boto3.client('rds')
redshift = boto3.client('redshift')
sts = boto3.client('sts')

source_cluster_name = 'my-source-cluster' # A name for the source cluster
source_param_group_name = 'my-source-param-group' # A name for the source parameter
 group
target_cluster_name = 'my-target-cluster' # A name for the target cluster
target_param_group_name = 'my-target-param-group' # A name for the target parameter
 group

def create_source_cluster(*args):
    """Creates a source Aurora MySQL DB cluster"""

    response = rds.create_db_cluster_parameter_group(
        DBClusterParameterGroupName=source_param_group_name,
        DBParameterGroupFamily='aurora-mysql8.0',
        Description='For Aurora MySQL zero-ETL integrations'
    )
    print('Created source parameter group: ' + response['DBClusterParameterGroup']
['DBClusterParameterGroupName'])
```

```
    response = rds.modify_db_cluster_parameter_group(
        DBClusterParameterGroupName=source_param_group_name,
        Parameters=[
            {
                'ParameterName': 'aurora_enhanced_binlog',
                'ParameterValue': '1',
                'ApplyMethod': 'pending-reboot'
            },
            {
                'ParameterName': 'binlog_backup',
                'ParameterValue': '0',
                'ApplyMethod': 'pending-reboot'
            },
            {
                'ParameterName': 'binlog_format',
                'ParameterValue': 'ROW',
                'ApplyMethod': 'pending-reboot'
            },
            {
                'ParameterName': 'binlog_replication_globaldb',
                'ParameterValue': '0',
                'ApplyMethod': 'pending-reboot'
            },
            {
                'ParameterName': 'binlog_row_image',
                'ParameterValue': 'full',
                'ApplyMethod': 'pending-reboot'
            },
            {
                'ParameterName': 'binlog_row_metadata',
                'ParameterValue': 'full',
                'ApplyMethod': 'pending-reboot'
            }
        ]
    )
    print('Modified source parameter group: ' +
 response['DBClusterParameterGroupName'])

    response = rds.create_db_cluster(
        DBClusterIdentifier=source_cluster_name,
        DBClusterParameterGroupName=source_param_group_name,
        Engine='aurora-mysql',
        EngineVersion='8.0.mysql_aurora.3.05.2',
        DatabaseName='myauroradb',
        MasterUsername='username',
        MasterUserPassword='Password01**'
    )
    print('Creating source cluster: ' + response['DBCluster']['DBClusterIdentifier'])
    source_arn = (response['DBCluster']['DBClusterArn'])
    create_target_cluster(target_cluster_name, source_arn, target_param_group_name)

    response = rds.create_db_instance(
        DBInstanceClass='db.r6g.2xlarge',
        DBClusterIdentifier=source_cluster_name,
        DBInstanceIdentifier=source_cluster_name + '-instance',
        Engine='aurora-mysql'
    )
    return(response)

def create_target_cluster(target_cluster_name, source_arn, target_param_group_name):
    """Creates a target Redshift cluster"""

    response = redshift.create_cluster_parameter_group(
        ParameterGroupName=target_param_group_name,
        ParameterGroupFamily='redshift-1.0',
        Description='For Aurora MySQL zero-ETL integrations'
    )
    print('Created target parameter group: ' + response['ClusterParameterGroup']
['ParameterGroupName'])

    response = redshift.modify_cluster_parameter_group(
        ParameterGroupName=target_param_group_name,
        Parameters=[
            {
                'ParameterName': 'enable_case_sensitive_identifier',
                'ParameterValue': 'true'
            }
        ]
    )
    print('Modified target parameter group: ' + response['ParameterGroupName'])

    response = redshift.create_cluster(
        ClusterIdentifier=target_cluster_name,
        NodeType='ra3.4xlarge',
        NumberOfNodes=2,
        Encrypted=True,
        MasterUsername='username',
        MasterUserPassword='Password01**',
        ClusterParameterGroupName=target_param_group_name
    )
    print('Creating target cluster: ' + response['Cluster']['ClusterIdentifier'])

    # Retrieve the target cluster ARN
    response = redshift.describe_clusters(
        ClusterIdentifier=target_cluster_name
    )
    target_arn = response['Clusters'][0]['ClusterNamespaceArn']

    # Retrieve the current user's account ID
    response = sts.get_caller_identity()
    account_id = response['Account']

    # Create a resource policy specifying cluster ARN and account ID
    response = redshift.put_resource_policy(
        ResourceArn=target_arn,
        Policy='''
        {
            \"Version\":\"2012-10-17\",
            \"Statement\":[
                {\"Effect\":\"Allow\",
                \"Principal\":{
                    \"Service\":\"redshift.amazonaws.com\"
                },
                \"Action\":[\"redshift:AuthorizeInboundIntegration\"],
                \"Condition\":{
                    \"StringEquals\":{
                        \"aws:SourceArn\":\"%s\"}
                    }
                },
                {\"Effect\":\"Allow\",
                \"Principal\":{
                    \"AWS\":\"arn:aws:iam::%s:root\"},
                \"Action\":\"redshift:CreateInboundIntegration\"}
            ]
        }
        ''' % (source_arn, account_id)
    )
    return(response)

def wait_for_cluster_availability(*args):
    """Waits for both clusters to be available"""
    print('Waiting for clusters to be available...')

    response = rds.describe_db_clusters(
        DBClusterIdentifier=source_cluster_name
    )
    source_status = response['DBClusters'][0]['Status']
    source_arn = response['DBClusters'][0]['DBClusterArn']

    response = rds.describe_db_instances(
        DBInstanceIdentifier=source_cluster_name + '-instance'
    )
    source_instance_status = response['DBInstances'][0]['DBInstanceStatus']

    response = redshift.describe_clusters(
        ClusterIdentifier=target_cluster_name
    )
    target_status = response['Clusters'][0]['ClusterStatus']
    target_arn = response['Clusters'][0]['ClusterNamespaceArn']

    # Every 60 seconds, check whether the clusters are available.
    if source_status != 'available' or target_status != 'available' or
 source_instance_status != 'available':
        time.sleep(60)
        response = wait_for_cluster_availability(
            source_cluster_name, target_cluster_name)
    else:
        print('Clusters available. Ready to create zero-ETL integration.')
        create_integration(source_arn, target_arn)
        return

def create_integration(source_arn, target_arn):
    """Creates a zero-ETL integration using the source and target clusters"""

    response = rds.create_integration(
        SourceArn=source_arn,
        TargetArn=target_arn,
        IntegrationName='my-integration'
    )
    print('Creating integration: ' + response['IntegrationName'])

def main():
    """main function"""
    create_source_cluster(source_cluster_name, source_param_group_name)
    wait_for_cluster_availability(source_cluster_name, target_cluster_name)
```

```
if __name__ == "__main__":
    main()
```

Next steps

With a source Aurora DB cluster and an Amazon Redshift target data warehouse, you can now create a zero-ETL integration and replicate data. For instructions, see the section called “Creating zero-ETL integrations”.

Creating Aurora zero-ETL integrations with Amazon Redshift

When you create an Aurora zero-ETL integration, you specify the source Aurora DB cluster and the target Amazon Redshift data warehouse. You can also customize encryption settings and add tags. Aurora creates an integration between the source DB cluster and its target. Once the integration is active, any data that you insert into the source DB cluster will be replicated into the configured Amazon Redshift target.

Topics

Prerequisites

Before you create a zero-ETL integration, you must create a source DB cluster and a target Amazon Redshift data warehouse. You also must allow replication into the data warehouse by adding the DB cluster as an authorized integration source.

For instructions to complete each of these steps, see the section called “Getting started with zero-ETL integrations”.

Required permissions

Certain IAM permissions are required to create a zero-ETL integration. At minimum, you need permissions to perform the following actions:

- Create zero-ETL integrations for the source Aurora DB cluster.
- View and delete all zero-ETL integrations.
- Create inbound integrations into the target data warehouse. You don't need this permission if the same account owns the Amazon Redshift data warehouse and this account is an authorized principal for that data warehouse. For information about adding authorized principals, see Configure authorization for your Amazon Redshift data warehouse.

The following sample policy demonstrates the least privilege permissions required to create and manage integrations. You might not need these exact permissions if your user or role has broader permissions, such as an `AdministratorAccess` managed policy.

Note

Redshift Amazon Resource Names (ARNs) have the following format. Note the use of a forward slash (`/`) rather than a colon (`:`) before the serverless namespace UUID.

- Provisioned cluster – arn:aws:**redshift**:{region}:{account-id}:namespace**:***namespace-uuid*
- Serverless – arn:aws:**redshift-serverless**:{region}:{account-id}:namespace**/***namespace-uuid*

Sample policy

Important

For the Aurora PostgreSQL preview, all ARNs and actions within the Amazon RDS Database Preview Environment have `-preview` appended to the service namespace. For example, rds-*preview*:CreateIntegration and arn:aws:rds-*preview*:....

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "rds:CreateIntegration"
        ],
        "Resource": [
            "arn:aws:rds:{region}:{account-id}:cluster:source-db",
            "arn:aws:rds:{region}:{account-id}:integration:*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "rds:DescribeIntegrations"
        ],
        "Resource": ["*"]
    },
    {
        "Effect": "Allow",
        "Action": [
            "rds:DeleteIntegration",
            "rds:ModifyIntegration"
        ],
        "Resource": [
            "arn:aws:rds:{region}:{account-id}:integration:*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "redshift:CreateInboundIntegration"
        ],
        "Resource": [
            "arn:aws:redshift:{region}:{account-id}:namespace:namespace-uuid"
        ]
    }]
}
```

Choosing a target data warehouse in a different account

If you plan to specify a target Amazon Redshift data warehouse that's in another AWS account, you must create a role that allows users in the current account to access resources in the target account. For more information, see Providing access to an IAM user in another AWS account that you own.

The role must have the following permissions, which allow the user to view available Amazon Redshift provisioned clusters and Redshift Serverless namespaces in the target account.

Required permissions and trust policy

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "redshift:DescribeClusters",
            "redshift-serverless:ListNamespaces"
         ],
         "Resource":[
            "*"
         ]
      }
   ]
}
```

The role must have the following trust policy, which specifies the target account ID.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "AWS": "arn:aws:iam::{external-account-id}:root"
         },
         "Action":"sts:AssumeRole"
      }
   ]
}
```

For instructions to create the role, see Creating a role using custom trust policies.

Creating zero-ETL integrations

You can create an Aurora MySQL zero-ETL integration using the AWS Management Console, the AWS CLI, or the RDS API. To create an Aurora PostgreSQL integration, you must use the AWS Management Console.

RDS console

To create a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   If you're using an Aurora PostgreSQL DB cluster as the source of the integration, you must sign in to the Amazon RDS Database Preview Environment at https://us-east-2.console.aws.amazon.com/rds-preview/home?region=us-east-2#databases.
2. In the left navigation pane, choose **Zero-ETL integrations**.
3. Choose **Create zero-ETL integration**.
4. For **Integration identifier**, enter a name for the integration. The name can have up to 63 alphanumeric characters and can include hyphens.
5. Choose **Next**.
6. For **Source**, select the Aurora DB cluster where the data will originate from. The cluster must be running Aurora MySQL version 3.05 or higher, or Aurora PostgreSQL (compatible with PostgreSQL 15.4 and Zero-ETL Support).

   > **Note**
   >
   > For MySQL sources, RDS notifies you if the DB cluster parameters aren't configured correctly. If you receive this message, you can either choose **Fix it for me**, or configure them manually. For instructions to fix them manually, see [the section called "Step 1: Create a custom DB cluster parameter group"](#).
   > Modifying DB cluster parameters requires a reboot. Before you can create the integration, the reboot must be complete and the new parameter values must be successfully applied to the cluster.
7. If you selected an Aurora PostgreSQL source cluster, under **Named database**, specify the named database to use as the source for your integration. The PostgreSQL resource model allows the creation of multiple databases within a single DB cluster, but only one can be used for each zero-ETL integration.

   The named database must be created from `template1`. For more information, see [Template Databases](#) in the PostgreSQL documentation.

8. (Optional) If you selected an Aurora MySQL source DB cluster, select **Customize data filtering options** and add data filters to your integration. You can use data filters to define the scope of replication to the target data warehouse. For more information, see the section called "Data filtering for zero-ETL integrations".
9. Once your source DB cluster is successfully configured, choose **Next**.
10. For **Target**, do the following:

    1. (Optional) To use a different AWS account for the Amazon Redshift target, choose **Specify a different account**. Then, enter the ARN of an IAM role with permissions to display your data warehouses. For instructions to create the IAM role, see the section called "Choosing a target data warehouse in a different account".
    2. For **Amazon Redshift data warehouse**, select the target for replicated data from the source DB cluster. You can choose a provisioned Amazon Redshift *cluster* or a Redshift Serverless *namespace* as the target.

    > **Note**
    >
    > RDS notifies you if the resource policy or case sensitivity settings for the specified data warehouse aren't configured correctly. If you receive this message, you can either choose **Fix it for me**, or configure them manually. For instructions to fix them manually, see Turn on case sensitivity for your data warehouse and Configure authorization for your data warehouse in the *Amazon Redshift Management Guide*.
    > Modifying case sensitivity for a *provisioned* Redshift cluster requires a reboot. Before you can create the integration, the reboot must be complete and the new parameter value must be successfully applied to the cluster.
    > If your selected source and target are in different AWS accounts, then Amazon RDS cannot fix these settings for you. You must navigate to the other account and fix them manually in Amazon Redshift.

11. Once your target data warehouse is configured correctly, choose **Next**.
12. (Optional) For **Tags**, add one or more tags to the integration. For more information, see the section called "Tagging RDS resources".
13. For **Encryption**, specify how you want your integration to be encrypted. By default, RDS encrypts all integrations with an AWS owned key. To choose a customer managed key instead, enable **Customize encryption settings** and choose a KMS key to use for encryption. For more information, see the section called "Encrypting Amazon Aurora resources".

> **Note**
>
> If you specify a custom KMS key, the key policy must allow the `kms:CreateGrant` action for the Amazon Redshift service principal (`redshift.amazonaws.com`). For more information, see Creating a key policy in the *AWS Key Management Service Developer Guide*.

Optionally, add an encryption context. For more information, see Encryption context in the *AWS Key Management Service Developer Guide*.

14. Choose **Next**.
15. Review your integration settings and choose **Create zero-ETL integration**.

    If creation fails, see the section called “I can't create a zero-ETL integration” for troubleshooting steps.

The integration has a status of `Creating` while it's being created, and the target Amazon Redshift data warehouse has a status of `Modifying`. During this time, you can't query the data warehouse or make any configuration changes on it.

When the integration is successfully created, the status of the integration and the target Amazon Redshift data warehouse both change to `Active`.

AWS CLI

> **Note**
>
> During the preview of Aurora PostgreSQL zero-ETL integrations, you can only create integrations through the AWS Management Console. You can't use the AWS CLI, the Amazon RDS API, or any of the SDKs.

To create a zero-ETL integration using the AWS CLI, use the create-integration command with the following options:

- `--integration-name` – Specify a name for the integration.

- `--source-arn` – Specify the ARN of the Aurora DB cluster that will be the source for the integration.
- `--target-arn` – Specify the ARN of the Amazon Redshift data warehouse that will be the target for the integration.

Example

For Linux, macOS, or Unix:

```
aws rds create-integration \
    --integration-name my-integration \
    --source-arn arn:aws:rds:{region}:{account-id}:my-db \
    --target-arn arn:aws:redshift:{region}:{account-id}:namespace:namespace-uuid
```

For Windows:

```
aws rds create-integration ^
    --integration-name my-integration ^
    --source-arn arn:aws:rds:{region}:{account-id}:my-db ^
    --target-arn arn:aws:redshift:{region}:{account-id}:namespace:namespace-uuid
```

RDS API

> **Note**
>
> During the preview of Aurora PostgreSQL zero-ETL integrations, you can only create integrations through the AWS Management Console. You can't use the AWS CLI, the Amazon RDS API, or any of the SDKs.

To create a zero-ETL integration by using the Amazon RDS API, use the `CreateIntegration` operation with the following parameters:

- `IntegrationName` – Specify a name for the integration.
- `SourceArn` – Specify the ARN of the Aurora DB cluster that will be the source for the integration.
- `TargetArn` – Specify the ARN of the Amazon Redshift data warehouse that will be the target for the integration.

Next steps

After you successfully create a zero-ETL integration, you must create a destination database within your target Amazon Redshift cluster or workgroup. Then, you can start adding data to the source Aurora DB cluster and querying it in Amazon Redshift. For instructions, see Creating destination databases in Amazon Redshift.

Data filtering for Aurora zero-ETL integrations with Amazon Redshift

You can use data filtering for Aurora zero-ETL integrations to define the scope of replication from the source Aurora DB cluster to the target Amazon Redshift data warehouse. Rather than replicating *all* data to the target, you can define one or more filters that selectively include or exclude certain tables from being replicated. Only filtering at the database and table level is available for zero-ETL integrations. You can't filter by columns or rows.

Data filtering can be useful when you want to:

- Join certain tables from two or more different source clusters and you don't need complete data from either cluster.
- Save costs by performing analytics using only a subset of tables rather than an entire fleet of databases.
- Filter out sensitive information—such as phone numbers, addresses, or credit card details—from certain tables.

You can add data filters to a zero-ETL integration using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the Amazon RDS API.

If the integration has a provisioned Amazon Redshift cluster as its target, the cluster must be on patch 180 or higher.

> **Note**
>
> Currently, you can perform data filtering only on integrations that have Aurora MySQL sources. The preview release of Aurora PostgreSQL zero-ETL integrations with Amazon Redshift doesn't support data filtering.

Topics

Format of a data filter

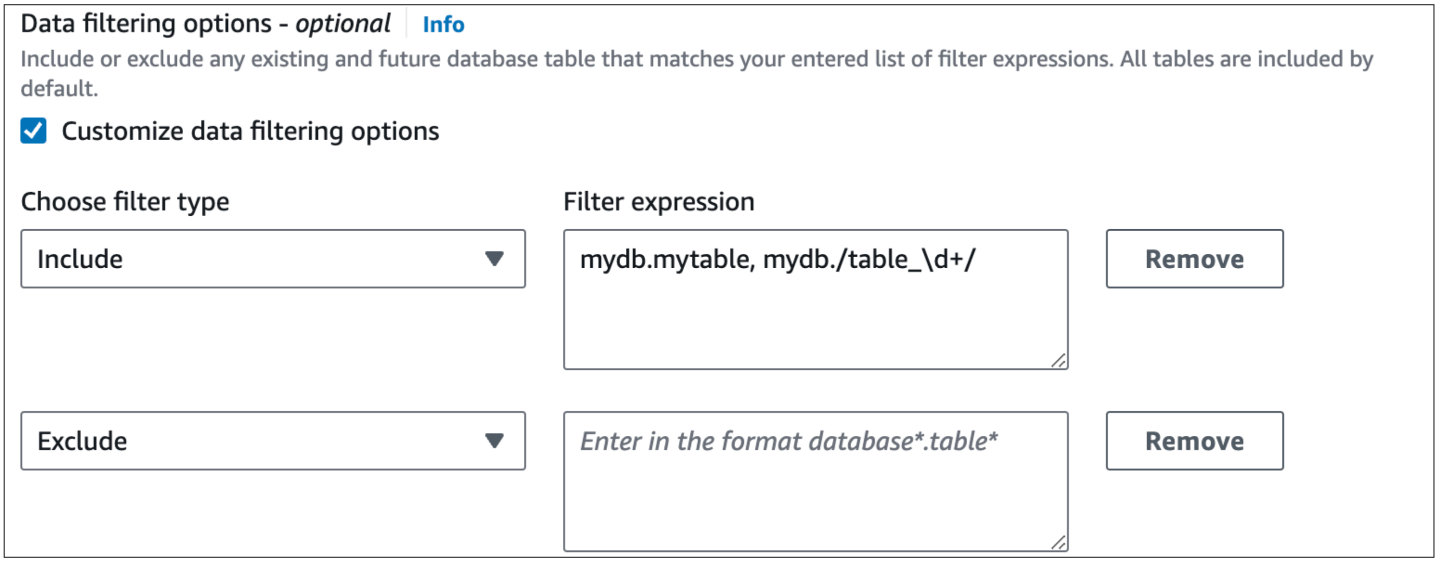
You can define multiple filters for a single integration. Each filter either includes or excludes any existing and future database tables that match one of the patterns in the filter expression. Aurora zero-ETL integrations use Maxwell filter syntax for data filtering.

Each filter has the following elements:

Element	Description
Filter type	An `Include` filter type *includes* all tables that match one of the patterns in the filter expression. An `Exclude` filter type *excludes* all tables that match one of the patterns.
Filter expression	A comma-separated list of patterns. Expressions must use Maxwell filter syntax.
Pattern	A filter pattern in the format *`database`*`.`*`table`*. You can specify literal database and table names (such as `mydb.mytable` , or use wildcards (*). You can also define regular expressions in the database and table name. Aurora supports filtering only at the database and table level. You can't include column-level filters (`database.`

Element	Description
	`table.column`) or blacklists (`blacklist: bad_db.*`). A single integration can have a maximum of 99 total patterns. In the console, you can contain patterns within a single filter expression, or spread them out among multiple expressions. A single pattern can't exceed 256 characters in length.

The following image shows the structure of data filters in the console:



Data filtering options - *optional* Info

Include or exclude any existing and future database table that matches your entered list of filter expressions. All tables are included by default.

☑ Customize data filtering options

Choose filter type	Filter expression
Include	mydb.mytable, mydb./table_\d+/
Exclude	*Enter in the format database*.table**

> **Important**
>
> Do not include personally identifying, confidential, or sensitive information in your filter patterns.

Data filters in the AWS CLI

When using the AWS CLI to add a data filter, the syntax differs slightly compared to the console. Each individual pattern must be associated with its own filter type (`Include` or `Exclude`). You can't group multiple patterns with a single filter type.

For example, in the console you can group the following comma-separated patterns within a single `Include` statement:

```
mydb.mytable, mydb./table_\d+/
```

However, when using the AWS CLI, the same data filter must be in the following format:

```
'include: mydb.mytable, include: mydb./table_\d+/'
```

Filter logic

If you don't specify any data filters in your integration, Aurora assumes a default filter of `include:*.*` and replicates all tables to the target data warehouse. However, if you specify at least one filter, the logic starts with an assumed `exclude:*.*`, meaning that all tables are automatically *excluded* from replication. This allows you to directly define which tables and databases to include.

For example, if you define the following filter:

```
'include: db.table1, include: db.table2'
```

Aurora evaluates the filter as follows:

```
'exclude:*.*, include: db.table1, include: db.table2'
```

Therefore, only `table1` and `table2` from the database named `db` are replicated to the target data warehouse.

Filter precedence

Aurora evaluates data filters in the order in which they're specified. In the AWS Management Console, this means that Aurora evaluates filter expressions from left to right and from top to bottom. If you specify a certain pattern for the first filter, then a second filter or even an individual pattern specified immediately after it can override it.

For example, your first filter might be `Include books.stephenking`, which includes a single table named `stephenking` from within the `books` database. However, if you add a second filter of Exclude books.*, it overrides the Include filter defined before it. Thus, no tables from the books index are replicated to Amazon Redshift.

If you specify at least one filter, the logic starts with an assumed exclude:*.*, meaning that all tables are automatically *excluded* from replication. Therefore, as a general best practice, define your filters from most broad to least broad. For example, use one or more Include statements to define all of the data that you want to replicate. Then, begin adding Exclude filters to selectively exclude certain tables from being replicated.

The same principle applies to filters that you define using the AWS CLI. Aurora evaluates these filter patterns in the order that they're specified, so a pattern might override one specified before it.

Examples

The following examples demonstrate how data filtering works for zero-ETL integrations:

- Include all databases and all tables:

  ```
  'include: *.*'
  ```

- Include all tables within the books database:

  ```
  'include: books.*'
  ```

- Exclude any tables named mystery:

  ```
  'include: *.*, exclude: *.mystery'
  ```

- Include two specific tables within the books database:

  ```
  'include: books.stephen_king, include: books.carolyn_keene'
  ```

- Include all tables in the books database, except for those containing the substring mystery:

  ```
  'include: books.*, exclude: books./.*mystery.*/
  ```

- Include all tables in the books database, except those starting with mystery:

  ```
  'include: books.*, exclude: books./mystery.*/'
  ```

- Include all tables in the books database, except those ending with mystery:

```
'include: books.*, exclude: books./.*mystery/'
```

- Include all tables in the books database that start with table_, except for the one named table_stephen_king. For example, table_movies or table_books would be replicated, but not table_stephen_king.

```
'include: books./table_.*/, exclude: books.table_stephen_king'
```

Adding data filters to an integration

You can configure data filtering using the AWS Management Console, the AWS CLI, or the Amazon RDS API.

> ⚠ **Important**
>
> If you add a filter after creating an integration, then Aurora reevaluates the filter as if it always existed. It removes any data that is currently in the target Amazon Redshift data warehouse that doesn't match the new filtering criteria. This action causes all affected tables to resynchronize.

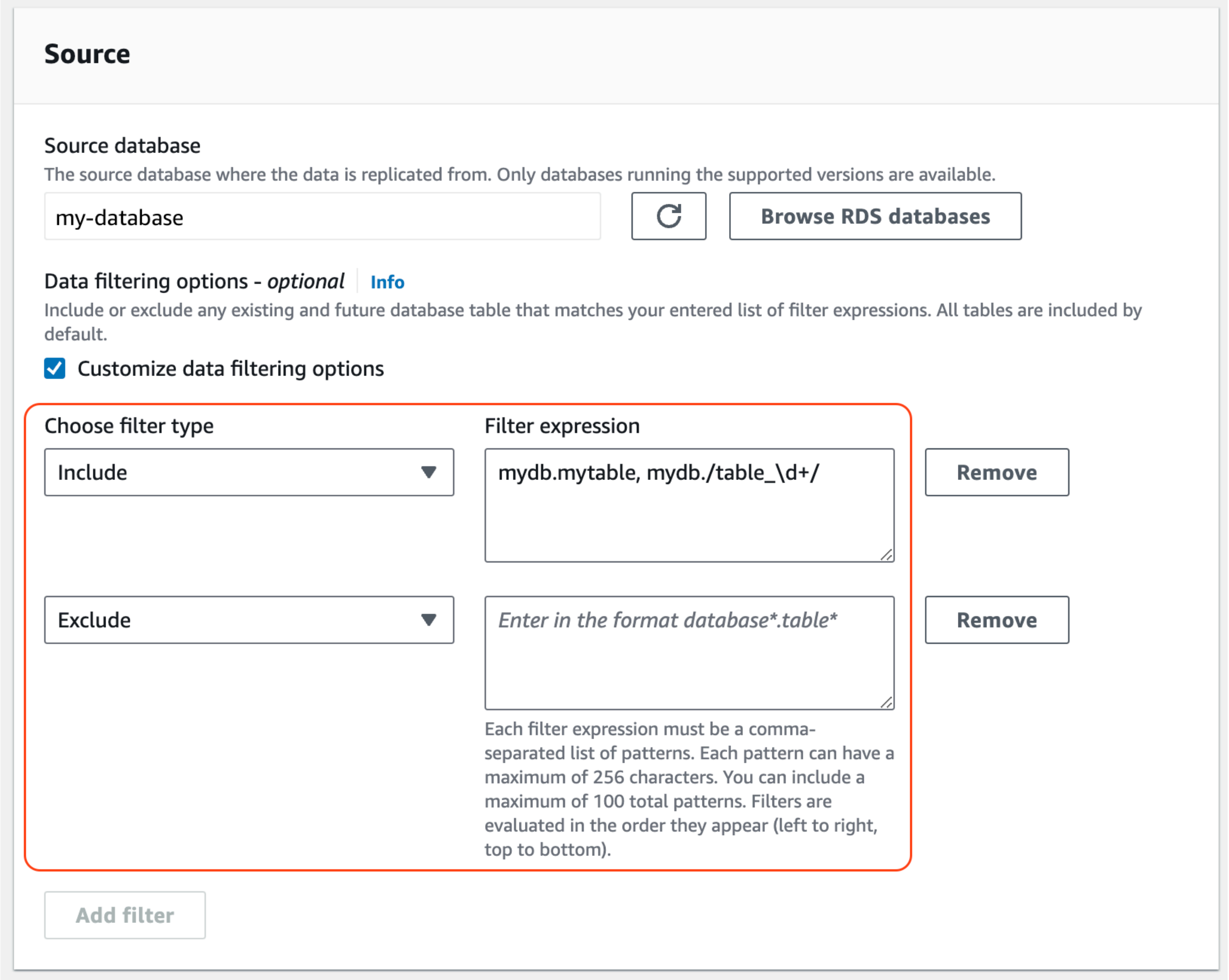
Currently, you can only perform data filtering on integrations that have Aurora MySQL sources. The preview release of Aurora PostgreSQL zero-ETL integrations with Amazon Redshift doesn't support data filtering.

RDS console

To add data filters to a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Zero-ETL integrations**. Select the integration that you want to add data filters to, and then choose **Modify**.
3. Under **Source**, add one or more Include and Exclude statements.

   The following image shows an example of data filters for an integration:



Source

Source database

The source database where the data is replicated from. Only databases running the supported versions are available.

my-database

Data filtering options - *optional* Info

Include or exclude any existing and future database table that matches your entered list of filter expressions. All tables are included by default.

☑ Customize data filtering options

Choose filter type	Filter expression	
Include	mydb.mytable, mydb./table_\d+/	Remove
Exclude	*Enter in the format database*.table**	Remove

Each filter expression must be a comma-separated list of patterns. Each pattern can have a maximum of 256 characters. You can include a maximum of 100 total patterns. Filters are evaluated in the order they appear (left to right, top to bottom).

4. When all the changes are as you want them, choose **Continue** and **Save changes**.

AWS CLI

To add data filters to a zero-ETL integration using the AWS CLI, call the modify-integration command. In addition to the integration identifier, specify the `--data-filter` parameter with a comma-separated list of `Include` and `Exclude` Maxwell filters.

Example

The following example adds filter patterns to `my-integration`.

For Linux, macOS, or Unix:

```
aws rds modify-integration \
    --integration-identifier my-integration \
    --data-filter 'include: foodb.*, exclude: foodb.tbl, exclude: foodb./table_\d+/'
```

For Windows:

```
aws rds modify-integration ^
    --integration-identifier my-integration ^
    --data-filter 'include: foodb.*, exclude: foodb.tbl, exclude: foodb./table_\d+/'
```

RDS API

To modify a zero-ETL integration using the RDS API, call the ModifyIntegration operation. Specify the integration identifier and provide a comma-separated list of filter patterns.

Removing data filters from an integration

When you remove a data filter from an integration, Aurora reevaluates the remaining filters as if the removed filter never existed. Aurora then replicates any data that previously didn't match the filtering criteria (but now does) into the target Amazon Redshift data warehouse.

Removing one or more data filters causes all affected tables to resynchronize.

Adding data to a source Aurora DB cluster and querying it in Amazon Redshift

To finish creating a zero-ETL integration that replicates data from Amazon Aurora into Amazon Redshift, you must create a destination database in Amazon Redshift.

First, connect to your Amazon Redshift cluster or workgroup and create a database with a reference to your integration identifier. Then, you can add data to your source Aurora DB cluster and see it replicated in Amazon Redshift.

Topics

Creating a destination database in Amazon Redshift

Before you can start replicating data into Amazon Redshift, after you create an integration, you must create a destination database in your target data warehouse. This destination database must include a reference to the integration identifier. You can use the Amazon Redshift console or the Query editor v2 to create the database.

For instructions to create a destination database, see Create a destination database in Amazon Redshift.

Adding data to the source DB cluster

After you configure your integration, you can add some data to the Aurora DB cluster that you want to replicate into your Amazon Redshift data warehouse.

> **Note**
>
> There are differences between data types in Amazon Aurora and Amazon Redshift. For a table of data type mappings, see the section called “Data type differences”.

First, connect to the source DB cluster using the MySQL or PostgreSQL client of your choice. For instructions, see the section called “Connecting to a DB cluster”.

Then, create a table and insert a row of sample data.

> **Important**
>
> Make sure that the table has a primary key. Otherwise, it can't be replicated to the target data warehouse.

The pg_dump and pg_restore PostgreSQL utilities initially create tables without a primary key and then add it afterwards. If you're using one of these utilities, we recommend first creating a schema and then loading data in a separate command.

MySQL

The following example uses the MySQL Workbench utility.

```
CREATE DATABASE my_db;
```

```
USE my_db;

CREATE TABLE books_table (ID int NOT NULL, Title VARCHAR(50) NOT NULL, Author
 VARCHAR(50) NOT NULL,
Copyright INT NOT NULL, Genre VARCHAR(50) NOT NULL, PRIMARY KEY (ID));

INSERT INTO books_table VALUES (1, 'The Shining', 'Stephen King', 1977, 'Supernatural
 fiction');
```

PostgreSQL

The following example uses the psql PostgreSQL interactive terminal. When connecting to the cluster, include the named database that you specified when creating the integration.

```
psql -h mycluster.cluster-123456789012.us-east-2.rds.amazonaws.com -p 5432 -U username
 -d named_db;

named_db=> CREATE TABLE books_table (ID int NOT NULL, Title VARCHAR(50) NOT NULL,
 Author VARCHAR(50) NOT NULL,
Copyright INT NOT NULL, Genre VARCHAR(50) NOT NULL, PRIMARY KEY (ID));

named_db=> INSERT INTO books_table VALUES (1, "The Shining", "Stephen King", 1977,
 "Supernatural fiction");
```

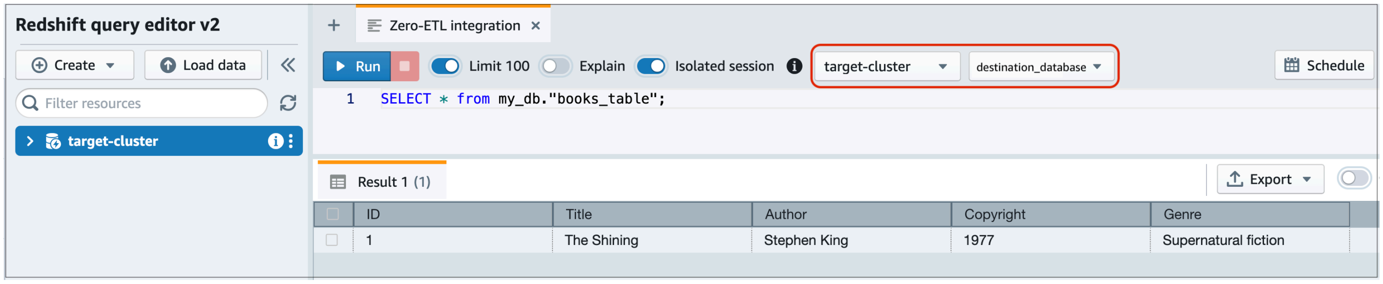
Querying your Aurora data in Amazon Redshift

After you add data to the Aurora DB cluster, it's replicated into Amazon Redshift and is ready to be queried.

To query the replicated data

1. Navigate to the Amazon Redshift console and choose **Query editor v2** from the left navigation pane.
2. Connect to your cluster or workgroup and choose your destination database (which you created from the integration) from the dropdown menu (**destination_database** in this example). For instructions to create a destination database, see Create a destination database in Amazon Redshift.
3. Use a SELECT statement to query your data. In this example, you can run the following command to select all data from the table that you created in the source Aurora DB cluster:

```
SELECT * from my_db."books_table";
```



```
SELECT * from my_db."books_table";
```

ID	Title	Author	Copyright	Genre
1	The Shining	Stephen King	1977	Supernatural fiction

- *my_db* is the Aurora database schema name. This option is only needed for MySQL databases.
- *books_table* is the Aurora table name.

You can also query the data using the a command line client. For example:

```
destination_database=# select * from my_db."books_table";

 ID |       Title |         Author |   Copyright |                  Genre |  txn_seq |
 txn_id
----+-------------+----------------+-------------+------------------------+----------
+--------+
  1 | The Shining |  Stephen King |        1977 |   Supernatural fiction |        2 |
 12192
```

Note

For case-sensitivity, use double quotes (" ") for schema, table, and column names. For more information, see enable_case_sensitive_identifier.

Data type differences between Aurora and Amazon Redshift databases

The following tables show the mappings of an Aurora MySQL or Aurora PostgreSQL data type to a corresponding Amazon Redshift data type. *Amazon Aurora currently supports only these data types for zero-ETL integrations.*

If a table in your source DB cluster includes an unsupported data type, the table goes out of sync and isn't consumable by the Amazon Redshift target. Streaming from the source to the target continues, but the table with the unsupported data type isn't available. To fix the table and make it available in Amazon Redshift, you must manually revert the breaking change and then refresh the integration by running `ALTER DATABASE...INTEGRATION REFRESH`.

Topics

Aurora MySQL

Aurora MySQL data type	Amazon Redshift data type	Description	Limitations
INT	INTEGER	Signed four-byte integer	
SMALLINT	SMALLINT	Signed two-byte integer	
TINYINT	SMALLINT	Signed two-byte integer	
MEDIUMINT	INTEGER	Signed four-byte integer	
BIGINT	BIGINT	Signed eight-byte integer	
INT UNSIGNED	BIGINT	Signed eight-byte integer	
TINYINT UNSIGNED	SMALLINT	Signed two-byte integer	
MEDIUMINT UNSIGNED	INTEGER	Signed four-byte integer	

Aurora MySQL data type	Amazon Redshift data type	Description	Limitations
BIGINT UNSIGNED	DECIMAL(20,0)	Exact numeric of selectable precision	
DECIMAL(p,s) = NUMERIC(p,s)	DECIMAL(p,s)	Exact numeric of selectable precision	Precision greater than 38 and scale greater than 37 not supported
DECIMAL(p,s) UNSIGNED = NUMERIC(p,s) UNSIGNED	DECIMAL(p,s)	Exact numeric of selectable precision	Precision greater than 38 and scale greater than 37 not supported
FLOAT4/REAL	REAL	Single precision floating-point number	
FLOAT4/REAL UNSIGNED	REAL	Single precision floating-point number	
DOUBLE/REAL/FLOAT8	DOUBLE PRECISION	Double precision floating-point number	
DOUBLE/REAL/FLOAT8 UNSIGNED	DOUBLE PRECISION	Double precision floating-point number	
BIT(n)	VARBYTE(8)	Variable-length binary value	

Aurora MySQL data type	Amazon Redshift data type	Description	Limitations
BINARY(n)	VARBYTE(n)	Variable-length binary value	
VARBINARY(n)	VARBYTE(n)	Variable-length binary value	
CHAR(n)	VARCHAR(n)	Variable-length string value	
VARCHAR(n)	VARCHAR(n)	Variable-length string value	
TEXT	VARCHAR(65535)	Variable-length string value up to 65535 bytes	
TINYTEXT	VARCHAR(255)	Variable-length string value up to 255 bytes	
MEDIUMTEXT	VARCHAR(65535)	Variable-length string value up to 65535 bytes	
LONGTEXT	VARCHAR(65535)	Variable-length string value up to 65535 bytes	
ENUM	VARCHAR(1020)	Variable-length string value up to 1020 bytes	
SET	VARCHAR(1020)	Variable-length string value up to 1020 bytes	

Aurora MySQL data type	Amazon Redshift data type	Description	Limitations
DATE	DATE	Calendar date (year, month, day)	
DATETIME	TIMESTAMP	Date and time (without time zone)	
TIMESTAMP(p)	TIMESTAMP	Date and time (without time zone)	
TIME	VARCHAR(18)	Variable-length string value up to 18 bytes	
YEAR	VARCHAR(4)	Variable-length string value up to 4 bytes	
JSON	SUPER	Semistruc tured data or documents as values	

Aurora PostgreSQL

Zero-ETL integrations for Aurora PostgreSQL don't support custom data types or data types created by extensions.

⚠ **Important**

The zero-ETL integrations with Amazon Redshift feature for Aurora PostgreSQL is in preview release. The documentation and the feature are both subject to change. You can use this feature only in test environments, not in production environments. For preview terms and conditions, see *Betas and Previews* in AWS Service Terms.

Aurora PostgreSQL data type	Amazon Redshift data type	Description	Limitations
bigint	BIGINT	Signed eight-byte integer	
bigserial	BIGINT	Signed eight-byte integer	
bit(n)	VARBYTE(n)	Variable-length binary value	
bit varying(n)	VARBYTE(n)	Variable-length binary value	
bit	VARBYTE(1024000)	Variable-length string value up to 1,024,000 bytes	
boolean	BOOLEAN	Logical boolean (true/false)	
bytea	VARBYTE(1024000)	Variable-length string value up to 1,024,000 bytes	
character(n)	CHAR(n)	Fixed-length character string	
character varying(n)	VARCHAR(65535)	Variable-length string value	

Aurora PostgreSQL data type	Amazon Redshift data type	Description	Limitations
date	DATE	Calendar date (year, month, day)	• Values greater than 9999-12-31 not supported • B.C. values not supported
double precision	DOUBLE PRECISION	Double precision floating-point numbers	Subnormal values not supported
integer	INTEGER	Signed four-byte integer	
money	DECIMAL(20,3)	Currency amount	
numeric(p,s)	DECIMAL(p,s)	Variable-length string value	• NaN values not supported • Precision greater than 38 and scale greater than 37 not supported • Negative scale not supported
real	REAL	Single precision floating-point number	
smallint	SMALLINT	Signed two-byte integer	

Aurora PostgreSQL data type	Amazon Redshift data type	Description	Limitations
smallserial	SMALLINT	Signed two-byte integer	
serial	INTEGER	Signed four-byte integer	
text	VARCHAR(65535)	Variable-length string value up to 65,535 bytes	
time [(p)] [without time zone]	VARCHAR(19)	Variable-length string value up to 19 bytes	`Infinity` and `-Infinity` values not supported
time [(p)] with time zone	VARCHAR(22)	Variable-length string value up to 22 bytes	• `Infinity` and -`Infinity` values not supported
timestamp [(p)] [without timezone]	TIMESTAMP	Date and time (without time zone)	• `Infinity` and -`Infinity` values not supported • Values greater than `9999-12-31` not supported • B.C. values not supported

Aurora PostgreSQL data type	Amazon Redshift data type	Description	Limitations
timestamp [(p)] with time zone	TIMESTAMPTZ	Date and time (with time zone)	• `Infinity` and - `Infinity` values not supported • Values greater than `9999-12-31` not supported • B.C. values not supported

Viewing and monitoring Aurora zero-ETL integrations with Amazon Redshift

You can view the details of an Amazon Aurora zero-ETL integration to see its configuration information and current status. You can also monitor the status of your integration by querying specific system views in Amazon Redshift. In addition, Amazon Redshift publishes certain integration-related metrics to Amazon CloudWatch, which you can view within the Amazon Redshift console.

Topics

- Viewing integrations
- Monitoring integrations using system tables
- Monitoring integrations with Amazon EventBridge

Viewing integrations

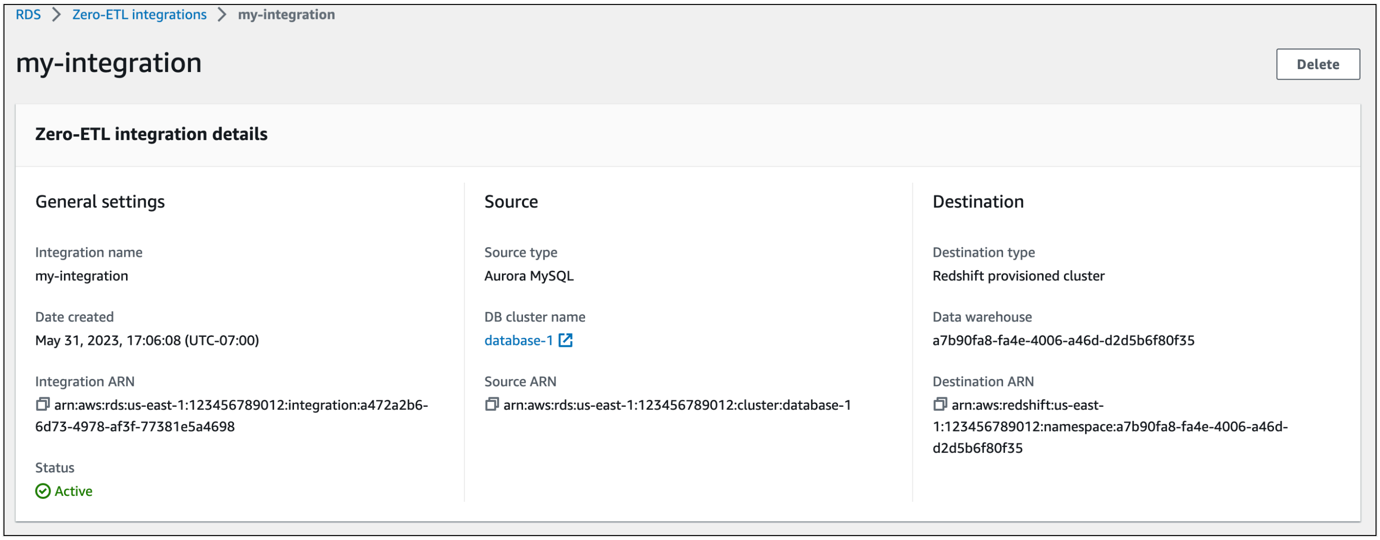
You can view Aurora zero-ETL integrations with Amazon Redshift using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To view the details of a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   If the integration has an Aurora PostgreSQL source DB cluster, you must sign in to the Amazon RDS Database Preview Environment at https://us-east-2.console.aws.amazon.com/rds-preview/home?region=us-east-2#databases.
2. From the left navigation pane, choose **Zero-ETL integrations**.
3. Select an integration to view more details about it, such as its source DB cluster and target data warehouse.



my-integration

Zero-ETL integration details

General settings

Integration name
my-integration

Date created
May 31, 2023, 17:06:08 (UTC-07:00)

Integration ARN
arn:aws:rds:us-east-1:123456789012:integration:a472a2b6-6d73-4978-af3f-77381e5a4698

Status
Active

Source

Source type
Aurora MySQL

DB cluster name
database-1

Source ARN
arn:aws:rds:us-east-1:123456789012:cluster:database-1

Destination

Destination type
Redshift provisioned cluster

Data warehouse
a7b90fa8-fa4e-4006-a46d-d2d5b6f80f35

Destination ARN
arn:aws:redshift:us-east-1:123456789012:namespace:a7b90fa8-fa4e-4006-a46d-d2d5b6f80f35

An integration can have the following statuses:

- `Creating` – The integration is being created.
- `Active` – The integration is sending transactional data to the target data warehouse.
- `Syncing` – The integration has encountered a recoverable error and is reseeding data. Affected tables aren't available for querying in Amazon Redshift until they finish resyncing.
- `Needs attention` – The integration encountered an event or error that requires manual intervention to resolve it. To fix the issue, follow the instructions in the error message on the integration details page.
- `Failed` – The integration encountered an unrecoverable event or error that can't be fixed. You must delete and recreate the integration.

- Deleting – The integration is being deleted.

AWS CLI

To view all zero-ETL integrations in the current account using the AWS CLI, use the describe-integrations command and specify the --integration-identifier option.

Example

For Linux, macOS, or Unix:

```
aws rds describe-integrations \
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374
```

For Windows:

```
aws rds describe-integrations ^
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374
```

RDS API

To view zero-ETL integration using the Amazon RDS API, use the DescribeIntegrations operation with the IntegrationIdentifier parameter.

Monitoring integrations using system tables

Amazon Redshift has system tables and views that contain information about how the system is functioning. You can query these system tables and views the same way that you would query any other database table. For more information about system tables and views in Amazon Redshift, see System tables reference in the *Amazon Redshift Database Developer Guide*.

You can query the following system views and tables to get information about your Aurora zero-ETL integrations with Amazon Redshift:

- SVV_INTEGRATION – Provides configuration details for your integrations.
- SVV_INTEGRATION_TABLE_STATE – Describes the state of each table within an integration.
- SYS_INTEGRATION_TABLE_STATE_CHANGE – Displays table state change logs for an integration.
- SYS_INTEGRATION_ACTIVITY – Provides information about completed integration runs.

All integration-related Amazon CloudWatch metrics originate from Amazon Redshift. For more information, see Monitoring zero-ETL integrations in the *Amazon Redshift Management Guide*. Currently, Amazon Aurora doesn't publish any integration metrics to CloudWatch.

Monitoring integrations with Amazon EventBridge

Amazon Redshift sends integration-related events to Amazon EventBridge. For a list of events and their corresponding event IDs, see Zero-ETL integration event notifications with Amazon EventBridge in the *Amazon Redshift Management Guide*.

Modifying Aurora zero-ETL integrations with Amazon Redshift

You can modify only the name, description, and data filtering options for a zero-ETL integration with Amazon Redshift. You can't modify the AWS KMS key used to encrypt the integration, or the source or target databases.

If you add a data filter to an existing integration, Aurora reevaluates the filter as if it always existed. It removes any data that is currently in the target Amazon Redshift data warehouse that doesn't match the new filtering criteria. If you *remove* a data filter from an integration, it replicates any data that previously didn't match the filtering criteria (but now does) into the target data warehouse. For more information, see the section called “Data filtering for zero-ETL integrations”.

You can modify a zero-ETL integration using the AWS Management Console, the AWS CLI, or the Amazon RDS API.

> **Note**
>
> Currently, you can only modify integrations that have Aurora MySQL source DB clusters. Modifying integrations isn't supported for the preview release of Aurora PostgreSQL zero-ETL integrations with Amazon Redshift.

RDS console

To modify a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

2. In the navigation pane, choose **Zero-ETL integrations**, and then choose the integration that you want to modify.
3. Choose **Modify** and make modifications to any available settings.
4. When all the changes are as you want them, choose **Modify**.

AWS CLI

To modify a zero-ETL integration using the AWS CLI, call the modify-integration command. Along with the `--integration-identifier`, specify any of the following options:

- `--integration-name` – Specify a new name for the integration.
- `--description` – Specify a new description for the integration.
- `--data-filter` – Specify data filtering options for the integration. For more information, see the section called "Data filtering for zero-ETL integrations".

Example

The following request modifies an existing integration.

For Linux, macOS, or Unix:

```
aws rds modify-integration \
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374 \
    --integration-name my-renamed-integration
```

For Windows:

```
aws rds modify-integration ^
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374 ^
    --integration-name my-renamed-integration
```

RDS API

To modify a zero-ETL integration using the RDS API, call the ModifyIntegration operation. Specify the integration identifier, and the parameters that you want to modify.

Deleting Aurora zero-ETL integrations with Amazon Redshift

When you delete a zero-ETL integration, Amazon Aurora removes it from the source Aurora DB cluster. Your transactional data isn't deleted from Amazon Aurora or Amazon Redshift, but Aurora doesn't send new data to Amazon Redshift.

You can only delete an integration when it has a status of `Active`, `Failed`, `Syncing`, or `Needs attention`.

You can delete zero-ETL integrations using the AWS Management Console, the AWS CLI, or the RDS API.

Console

To delete a zero-ETL integration

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   If the integration has an Aurora PostgreSQL source DB cluster, you must sign in to the Amazon RDS Database Preview Environment at https://us-east-2.console.aws.amazon.com/rds-preview/home?region=us-east-2#databases.
2. From the left navigation pane, choose **Zero-ETL integrations**.
3. Select the zero-ETL integration that you want to delete.
4. Choose **Actions**, **Delete**, and confirm deletion.

AWS CLI

> **Note**
>
> During the preview of Aurora PostgreSQL zero-ETL integrations, you can only delete integrations through the AWS Management Console. You can't use the AWS CLI, the Amazon RDS API, or any of the SDKs.

To delete a zero-ETL integration, use the delete-integration command and specify the `--integration-identifier` option.

Example

For Linux, macOS, or Unix:

```
aws rds delete-integration \
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374
```

For Windows:

```
aws rds delete-integration ^
    --integration-identifier ee605691-6c47-48e8-8622-83f99b1af374
```

RDS API

Note

During the preview of Aurora PostgreSQL zero-ETL integrations, you can only delete integrations through the AWS Management Console. You can't use the AWS CLI, the Amazon RDS API, or any of the SDKs.

To delete a zero-ETL integration using the Amazon RDS API, use the `DeleteIntegration` operation with the `IntegrationIdentifier` parameter.

Troubleshooting Aurora zero-ETL integrations with Amazon Redshift

You can check the state of a zero-ETL integration by querying the SVV_INTEGRATION system table in Amazon Redshift. If the `state` column has a value of `ErrorState`, it means something's wrong. For more information, see the section called “Monitoring using system tables”.

Use the following information to troubleshoot common issues with Aurora zero-ETL integrations with Amazon Redshift.

Topics

- I can't create a zero-ETL integration
- My integration is stuck in a state of Syncing

- My tables aren't replicating to Amazon Redshift
- One or more of my Amazon Redshift tables requires a resync

I can't create a zero-ETL integration

If you can't create a zero-ETL integration, make sure that the following are correct for your source DB cluster:

- Your source DB cluster is running Aurora MySQL version 3.05 (compatible with MySQL 8.0.32) or higher, or Aurora PostgreSQL (compatible with PostgreSQL 15.4 and Zero-ETL Support). To validate the engine version, choose the **Configuration** tab for the DB cluster and check the **Engine version**.
- You correctly configured DB cluster parameters. If the required parameters are set incorrectly or not associated with the cluster, creation fails. See the section called "Step 1: Create a custom DB cluster parameter group".

In addition, make sure the following are correct for your target data warehouse:

- Case sensitivity is enabled. See Turn on case sensitivity for your data warehouse.
- You added the correct authorized principal and integration source. See Configure authorization for your Amazon Redshift data warehouse.
- The data warehouse is encrypted (if it's a provisioned cluster). See Amazon Redshift database encryption.

My integration is stuck in a state of `Syncing`

Your integration might consistently show a status of `Syncing` if you change the value of one of the required DB parameters.

To fix this issue, check the values of the parameters in the parameter group associated with the source DB cluster, and make sure that they match the required values. For more information, see the section called "Step 1: Create a custom DB cluster parameter group".

If you modify any parameters, make sure to reboot the DB cluster to apply the changes.

My tables aren't replicating to Amazon Redshift

Your data might not be replicating because one or more of your source tables doesn't have a primary key. The monitoring dashboard in Amazon Redshift displays the status of these tables as `Failed`, and the status of the overall zero-ETL integration changes to `Needs attention`.

To resolve this issue, you can identify an existing key in your table that can become a primary key, or you can add a synthetic primary key. For detailed solutions, see the following resources:

- Handle tables without primary keys while creating Amazon Aurora MySQL or Amazon RDS for MySQL zero-ETL integrations with Amazon Redshift
- Handle tables without primary keys while creating Amazon Aurora PostgreSQL zero-ETL integrations with Amazon Redshift

One or more of my Amazon Redshift tables requires a resync

Running certain commands on your source DB cluster might require your tables to be resynchronized. In these cases, the SVV_INTEGRATION_TABLE_STATE system view shows a `table_state` of `ResyncRequired`, which means that the integration must completely reload data for that specific table from MySQL to Amazon Redshift.

When the table starts to resynchronize, it enters a state of `Syncing`. You don't need to take any manual action to resynchronize a table. While table data is resynchronizing, you can't access it in Amazon Redshift.

The following are some example operations that can put a table into a `ResyncRequired` state, and possible alternatives to consider.

Operation	Example	Alternative

Adding a column into a specific position	`ALTER TABLE table_name`
	`  ADD COLUMN column_name  INTEGER`
	`  NOT NULL first;`
		Amazon Redshift doesn't support adding columns into specific positions

Operation	Example	Alternative
		using `first` or `after` keywords. If the order of columns in the target table isn't critical, add the column to the end of the table using a simpler command: `ALTER TABLE table_name ADD COLUMN column_name column_type ;`

Operation	Example	Alternative

Adding a timestamp column with the default `CURRENT_TIMESTAMP`	`ALTER TABLE table_name`
	`  ADD COLUMN column_name  TIMESTAMP`
	`  NOT NULL DEFAULT CURRENT_TIMESTAMP;`
		The `CURRENT_TIMESTAMP` value for existing table rows is calculated by Aurora MySQL and can't be simulated in Amazon Redshift without full table data resynchronization.
		If possible, switch the default value to a literal constant like `2023-01-01 00:00:15` to avoid latency in table availability.

Operation	Example	Alternative

Performing multiple column operations within a single command	`ALTER TABLE table_name`
	`  ADD COLUMN column_1,`
	`  RENAME COLUMN column_2 TO column_3;`
		Consider splitting the command into two separate operations, `ADD` and `RENAME`, which won't require resynchronization.

Using Aurora Serverless v2

Aurora Serverless v2 is an on-demand, autoscaling configuration for Amazon Aurora. Aurora Serverless v2 helps to automate the processes of monitoring the workload and adjusting the capacity for your databases. Capacity is adjusted automatically based on application demand. You're charged only for the resources that your DB clusters consume. Thus, Aurora Serverless v2 can help you to stay within budget and avoid paying for computer resources that you don't use.

This type of automation is especially valuable for multitenant databases, distributed databases, development and test systems, and other environments with highly variable and unpredictable workloads.

Topics

Aurora Serverless v2 use cases

Aurora Serverless v2 supports many types of database workloads. These range from development and testing environments, to websites and applications that have unpredictable workloads, to the most demanding, business-critical applications that require high scale and availability.

Aurora Serverless v2 is especially useful for the following use cases:

- **Variable workloads** – You're running workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. Another is an e-commerce site with increased traffic when you offer sales or special promotions. With Aurora Serverless v2, your database automatically scales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over. With Aurora Serverless v2, you no longer need to provision for peak or average capacity. You can specify an upper capacity limit to handle the worst-case situation, and that capacity isn't used unless it's needed.

The granularity of scaling in Aurora Serverless v2 helps you to match capacity closely to your database's needs. For a provisioned cluster, scaling up requires adding a whole new DB instance. For an Aurora Serverless v1 cluster, scaling up requires doubling the number of Aurora capacity units (ACUs) for the cluster, such as from 16 to 32 or 32 to 64. In contrast, Aurora Serverless v2 can add half an ACU when only a little more capacity is needed. It can add 0.5, 1, 1.5, 2, or additional half-ACUs based on the additional capacity needed to handle an increase in workload. And it can remove 0.5, 1, 1.5, 2, or additional half-ACUs when the workload decreases and that capacity is no longer needed.

- **Multi-tenant applications –** With Aurora Serverless v2, you don't have to individually manage database capacity for each application in your fleet. Aurora Serverless v2 manages individual database capacity for you.

  You can create a cluster for each tenant. That way, you can use features such as cloning, snapshot restore, and Aurora global databases to enhance high availability and disaster recovery as appropriate for each tenant.

  Each tenant might have specific busy and idle periods depending on the time of day, time of year, promotional events, and so on. Each cluster can have a wide capacity range. That way, clusters with low activity incur minimal DB instance charges. Any cluster can quickly scale up to handle periods of high activity.
- **New applications –** You're deploying a new application and you're unsure about the DB instance size you need. By using Aurora Serverless v2, you can set up a cluster with one or many DB instances and have the database autoscale to the capacity requirements of your application.
- **Mixed-use applications –** Suppose that you have an online transaction processing (OLTP) application, but you periodically experience spikes in query traffic. By specifying promotion tiers for the Aurora Serverless v2 DB instances in a cluster, you can configure your cluster so that the reader DB instances can scale independently of the writer DB instance to handle the additional load. When the usage spike subsides, the reader DB instances scale back down to match the capacity of the writer DB instance.
- **Capacity planning –** Suppose that you usually adjust your database capacity, or verify the optimal database capacity for your workload, by modifying the DB instance classes of all the DB instances in a cluster. With Aurora Serverless v2, you can avoid this administrative overhead. You

can determine the appropriate minimum and maximum capacity by running the workload and checking how much the DB instances actually scale.

You can modify existing DB instances from provisioned to Aurora Serverless v2 or from Aurora Serverless v2 to provisioned. You don't need to create a new cluster or a new DB instance in such cases.

With an Aurora global database, you might not need as much capacity for the secondary clusters as in the primary cluster. You can use Aurora Serverless v2 DB instances in the secondary clusters. That way, the cluster capacity can scale up if a secondary region is promoted and takes over your application's workload.

- **Development and testing** – In addition to running your most demanding applications, you can also use Aurora Serverless v2 for development and testing environments. With Aurora Serverless v2, you can create DB instances with a low minimum capacity instead of using burstable db.t* DB instance classes. You can set the maximum capacity high enough that those DB instances can still run substantial workloads without running low on memory. When the database isn't in use, all of the DB instances scale down to avoid unnecessary charges.

> **Tip**
>
> To make it convenient to use Aurora Serverless v2 in development and test environments, the AWS Management Console provides the **Easy create** shortcut when you create a new cluster. If you choose the **Dev/Test** option, Aurora creates a cluster with an Aurora Serverless v2 DB instance and a capacity range that's typical for a development and test system.

Using Aurora Serverless v2 for existing provisioned workloads

Suppose that you already have an Aurora application running on a provisioned cluster. You can check how the application would work with Aurora Serverless v2 by adding one or more Aurora Serverless v2 DB instances to the existing cluster as reader DB instances. You can check how often the reader DB instances scale up and down. You can use the Aurora failover mechanism to promote an Aurora Serverless v2 DB instance to be the writer and check how it handles the read/write workload. That way, you can switch over with minimal downtime and without changing the endpoint that your client applications use. For details on the procedure to convert existing clusters to Aurora Serverless v2, see Migrating to Aurora Serverless v2.

Advantages of Aurora Serverless v2

Aurora Serverless v2 is intended for variable or "spiky" workloads. With such unpredictable workloads, you might have difficulty planning when to change your database capacity. You might also have trouble making capacity changes quickly enough using the familiar mechanisms such as adding DB instances or changing DB instance classes. Aurora Serverless v2 provides the following advantages to help with such use cases:

- **Simpler capacity management than provisioned –** Aurora Serverless v2 reduces the effort for planning DB instance sizes and resizing DB instances as the workload changes. It also reduces the effort for maintaining consistent capacity for all the DB instances in a cluster.
- **Faster and easier scaling during periods of high activity –** Aurora Serverless v2 scales compute and memory capacity as needed, with no disruption to client transactions or your overall workload. The ability to use reader DB instances with Aurora Serverless v2 helps you to take advantage of horizontal scaling in addition to vertical scaling. The ability to use Aurora global databases means that you can spread your Aurora Serverless v2 read workload across multiple AWS Regions. This capability is more convenient than the scaling mechanisms for provisioned clusters. It's also faster and more granular than the scaling capabilities in Aurora Serverless v1.
- **Cost-effective during periods of low activity –** Aurora Serverless v2 helps you to avoid overprovisioning your DB instances. Aurora Serverless v2 adds resources in granular increments when DB instances scale up. You pay only for the database resources that you consume. Aurora Serverless v2 resource usage is measured on a per-second basis. That way, when a DB instance scales down, the reduced resource usage is registered right away.
- **Greater feature parity with provisioned –** You can use many Aurora features with Aurora Serverless v2 that aren't available for Aurora Serverless v1. For example, with Aurora Serverless v2 you can use reader DB instances, global databases, AWS Identity and Access Management (IAM) database authentication, and Performance Insights. You can also use many more configuration parameters than with Aurora Serverless v1.

  In particular, with Aurora Serverless v2 you can take advantage of the following features from provisioned clusters:

  - **Reader DB instances –** Aurora Serverless v2 can take advantage of reader DB instances to scale horizontally. When a cluster contains one or more reader DB instances, the cluster can fail over immediately in case of problems with the writer DB instance. This is a capability that isn't available with Aurora Serverless v1.

  - **Multi-AZ clusters –** You can distribute the Aurora Serverless v2 DB instances of a cluster across multiple Availability Zones (AZs). Setting up a Multi-AZ cluster helps to ensure business continuity even in the rare case of issues that affect an entire AZ. This is a capability that isn't available with Aurora Serverless v1.
  - **Global databases –** You can use Aurora Serverless v2 in combination with Aurora global databases to create additional read-only copies of your cluster in other AWS Regions for disaster recovery purposes.
  - **RDS Proxy –** You can use Amazon RDS Proxy to allow your applications to pool and share database connections to improve their ability to scale.
- **Faster, more granular, less disruptive scaling than Aurora Serverless v1 –** Aurora Serverless v2 can scale up and down faster. Scaling can change capacity by as little as 0.5 ACUs, instead of doubling or halving the number of ACUs. Scaling typically happens with no pause in processing at all. Scaling doesn't involve an event that you have to be aware of, as with Aurora Serverless v1. Scaling can happen while SQL statements are running and transactions are open, without the need to wait for a quiet point.

How Aurora Serverless v2 works

The following overview describes how Aurora Serverless v2 works.

Topics

Aurora Serverless v2 overview

Amazon Aurora Serverless v2 is suitable for the most demanding, highly variable workloads. For example, your database usage might be heavy for a short period of time, followed by long periods of light activity or no activity at all. Some examples are retail, gaming, or sports websites with periodic promotional events, and databases that produce reports when needed. Others are development and testing environments, and new applications where usage might ramp up quickly. For cases such as these and many others, configuring capacity correctly in advance isn't always possible with the provisioned model. It can also result in higher costs if you overprovision and have capacity that you don't use.

In contrast, *Aurora provisioned clusters* are suitable for steady workloads. With provisioned clusters, you choose a DB instance class that has a predefined amount of memory, CPU power, I/O bandwidth, and so on. If your workload changes, you manually modify the instance class of your writer and readers. The provisioned model works well when you can adjust capacity in advance of expected consumption patterns and it's acceptable to have brief outages while you change the instance class of the writer and readers in your cluster.

Aurora Serverless v2 is architected from the ground up to support serverless DB clusters that are instantly scalable. Aurora Serverless v2 is engineered to provide the same degree of security and isolation as with provisioned writers and readers. These aspects are crucial in multitenant serverless cloud environments. The dynamic scaling mechanism has very little overhead so that it can respond quickly to changes in the database workload. It's also powerful enough to meet dramatic increases in processing demand.

By using Aurora Serverless v2, you can create an Aurora DB cluster without being locked into a specific database capacity for each writer and reader. You specify the minimum and maximum capacity range. Aurora scales each Aurora Serverless v2 writer or reader in the cluster within that capacity range. By using a Multi-AZ cluster where each writer or reader can scale dynamically, you can take advantage of dynamic scaling and high availability.

Aurora Serverless v2 scales the database resources automatically based on your minimum and maximum capacity specifications. Scaling is fast because most scaling events operations keep the writer or reader on the same host. In the rare cases that an Aurora Serverless v2 writer or reader is moved from one host to another, Aurora Serverless v2 manages the connections automatically. You don't need to change your database client application code or your database connection strings.

With Aurora Serverless v2, as with provisioned clusters, storage capacity and compute capacity are separate. When we refer to Aurora Serverless v2 capacity and scaling, it's always compute capacity that's increasing or decreasing. Thus, your cluster can contain many terabytes of data even when the CPU and memory capacity scale down to low levels.

Instead of provisioning and managing database servers, you specify database capacity. For details about Aurora Serverless v2 capacity, see Aurora Serverless v2 capacity. The actual capacity of each

Aurora Serverless v2 writer or reader varies over time, depending on your workload. For details about that mechanism, see Aurora Serverless v2 scaling.

⚠ **Important**

With Aurora Serverless v1, your cluster has a single measure of compute capacity that can scale between the minimum and maximum capacity values. With Aurora Serverless v2, your cluster can contain readers in addition to the writer. Each Aurora Serverless v2 writer and reader can scale between the minimum and maximum capacity values. Thus, the total capacity of your Aurora Serverless v2 cluster depends on both the capacity range that you define for your DB cluster and the number of writers and readers in the cluster. At any specific time, you are only charged for the Aurora Serverless v2 capacity that is being actively used in your Aurora DB cluster.

Configurations for Aurora DB clusters

For each of your Aurora DB clusters, you can choose any combination of Aurora Serverless v2 capacity, provisioned capacity, or both.

You can set up a cluster that contains both Aurora Serverless v2 and provisioned capacity, called a *mixed-configuration cluster*. For example, suppose that you need more read/write capacity than is available for an Aurora Serverless v2 writer. In this case, you can set up the cluster with a very large provisioned writer. In that case, you can still use Aurora Serverless v2 for the readers. Or suppose that the write workload for your cluster varies but the read workload is steady. In this case, you can set up your cluster with an Aurora Serverless v2 writer and one or more provisioned readers.

You can also set up a DB cluster where all the capacity is managed by Aurora Serverless v2. To do this, you can create a new cluster and use Aurora Serverless v2 from the start. Or you can replace all the provisioned capacity in an existing cluster with Aurora Serverless v2. For example, some of the upgrade paths from older engine versions require starting with a provisioned writer and replacing it with a Aurora Serverless v2 writer. For the procedures to create a new DB cluster with Aurora Serverless v2 or to switch an existing DB cluster to Aurora Serverless v2, see Creating an Aurora Serverless v2 DB cluster and Switching from a provisioned cluster to Aurora Serverless v2.

If you don't use Aurora Serverless v2 at all in a DB cluster, all the writers and readers in the DB cluster are *provisioned*. This is the oldest and most common kind of DB cluster that most users are familiar with. In fact, before Aurora Serverless, there wasn't a special name for this kind of Aurora DB cluster. Provisioned capacity is constant. The charges are relatively easy to forecast. However, you have to predict in advance how much capacity you need. In some cases, your predictions might be inaccurate or your capacity needs might change. In these cases, your DB cluster can become underprovisioned (slower than you want) or overprovisioned (more expensive than you want).

Aurora Serverless v2 capacity

The unit of measure for Aurora Serverless v2 is the *Aurora capacity unit (ACU)*. Aurora Serverless v2 capacity isn't tied to the DB instance classes that you use for provisioned clusters.

Each ACU is a combination of approximately 2 gibibytes (GiB) of memory, corresponding CPU, and networking. You specify the database capacity range using this unit of measure. The `ServerlessDatabaseCapacity` and `ACUUtilization` metrics help you to determine how much capacity your database is actually using and where that capacity falls within the specified range.

At any moment in time, each Aurora Serverless v2 DB writer or reader has a *capacity*. The capacity is represented as a floating-point number representing ACUs. The capacity increases or decreases whenever the writer or reader scales. This value is measured every second. For each DB cluster where you intend to use Aurora Serverless v2, you define a *capacity range*: the minimum and maximum capacity values that each Aurora Serverless v2 writer or reader can scale between. The capacity range is the same for each Aurora Serverless v2 writer or reader in a DB cluster. Each Aurora Serverless v2 writer or reader has its own capacity, falling somewhere in that range.

The largest Aurora Serverless v2 capacity that you can define is 128 ACUs. For all the considerations when choosing the maximum capacity value, see Choosing the maximum Aurora Serverless v2 capacity setting for a cluster.

The smallest Aurora Serverless v2 capacity that you can define is 0.5 ACUs. You can specify a higher number if it's less than or equal to the maximum capacity value. Setting the minimum capacity to a small number lets lightly loaded DB clusters consume minimal compute resources. At the same time, they stay ready to accept connections immediately and scale up when they become busy.

We recommend setting the minimum to a value that allows each DB writer or reader to hold the working set of the application in the buffer pool. That way, the contents of the buffer pool aren't discarded during idle periods. For all the considerations when choosing the minimum capacity value, see Choosing the minimum Aurora Serverless v2 capacity setting for a cluster.

Depending on how you configure the readers in a Multi-AZ DB cluster, their capacities can be tied to the capacity of the writer or independently. For details about how to do that, see Aurora Serverless v2 scaling.

Monitoring Aurora Serverless v2 involves measuring the capacity values for the writer and readers in your DB cluster over time. If your database doesn't scale down to the minimum capacity, you can take actions such as adjusting the minimum and optimizing your database application. If your database consistently reaches its maximum capacity, you can take actions such as increasing the maximum. You can also optimize your database application and spread the query load across more readers.

The charges for Aurora Serverless v2 capacity are measured in terms of ACU-hours. For information about how Aurora Serverless v2 charges are calculated, see the Aurora pricing page.

Suppose that the total number of writers and readers in your cluster is N. In that case, the cluster consumes approximately `n x minimum ACUs` when you aren't running any database operations. Aurora itself might run monitoring or maintenance operations that cause some small amount of load. That cluster consumes no more than `n x maximum ACUs` when the database is running at full capacity.

For more details about choosing appropriate minimum and maximum ACU values, see Choosing the Aurora Serverless v2 capacity range for an Aurora cluster. The minimum and maximum ACU values that you specify also affect the way some of the Aurora configuration parameters work for Aurora Serverless v2. For details about the interaction between the capacity range and configuration parameters, see Working with parameter groups for Aurora Serverless v2.

Aurora Serverless v2 scaling

For each Aurora Serverless v2 writer or reader, Aurora continuously tracks utilization of resources such as CPU, memory, and network. These measurements collectively are called the *load*. The load includes the database operations performed by your application. It also includes background processing for the database server and Aurora administrative tasks. When capacity is constrained by any of these, Aurora Serverless v2 scales up. Aurora Serverless v2 also scales up when it detects performance issues that it can resolve by doing so. You can monitor resource utilization and how it affects Aurora Serverless v2 scaling by using the procedures in Important Amazon CloudWatch metrics for Aurora Serverless v2 and Monitoring Aurora Serverless v2 performance with Performance Insights.

The load can vary across the writer and readers in your DB cluster. The writer handles all data definition language (DDL) statements, such as `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`. The writer also handles all data manipulation language (DML) statements, such as `INSERT` and `UPDATE`. Readers can process read-only statements, such as `SELECT` queries.

Scaling is the operation that increases or decreases Aurora Serverless v2 capacity for your database. With Aurora Serverless v2, each writer and reader has its own current capacity value, measured in ACUs. Aurora Serverless v2 scales a writer or reader up to a higher capacity when its current capacity is too low to handle the load. It scales the writer or reader down to a lower capacity when its current capacity is higher than needed.

Unlike Aurora Serverless v1, which scales by doubling the capacity each time the DB cluster reaches a threshold, Aurora Serverless v2 can increase capacity incrementally. When your workload demand begins to reach the current database capacity of a writer or reader, Aurora Serverless v2 increases the number of ACUs for that writer or reader. Aurora Serverless v2 scales capacity in the increments required to provide the best performance for the resources consumed. Scaling happens in increments as small as 0.5 ACUs. The larger the current capacity, the larger the scaling increment and thus the faster scaling can happen.

Because Aurora Serverless v2 scaling is so frequent, granular, and nondisruptive, it doesn't cause discrete events in the AWS Management Console the way that Aurora Serverless v1 does. Instead, you can measure the Amazon CloudWatch metrics such as `ServerlessDatabaseCapacity` and `ACUUtilization` and track their minimum, maximum, and average values over time. To learn more about Aurora metrics, see Monitoring metrics in an Amazon Aurora cluster. For tips about monitoring Aurora Serverless v2, see Important Amazon CloudWatch metrics for Aurora Serverless v2.

You can choose to make a reader scale at the same time as the associated writer, or independently from the writer. You do so by specifying the promotion tier for that reader.

- Readers in promotion tiers 0 and 1 scale at the same time as the writer. That scaling behavior makes readers in priority tiers 0 and 1 ideal for availability. That's because they are always sized to the right capacity to take over the workload from the writer in case of failover.
- Readers in promotion tiers 2–15 scale independently from the writer. Each reader remains within the minimum and maximum ACU values that you specified for your cluster. When a reader scales independently of the associated writer DB, it can become idle and scale down while the writer continues to process a high volume of transactions. It's still available as a failover target, if no other readers are available in lower promotion tiers. However, if it's promoted to be the writer, it might need to scale up to handle the full workload of the writer.

For details about promotion tiers, see Choosing the promotion tier for an Aurora Serverless v2 reader.

The notions of scaling points and associated timeout periods from Aurora Serverless v1 don't apply in Aurora Serverless v2. Aurora Serverless v2 scaling can happen while database connections are open, while SQL transactions are in process, while tables are locked, and while temporary tables are in use. Aurora Serverless v2 doesn't wait for a quiet point to begin scaling. Scaling doesn't disrupt any database operations that are underway.

If your workload requires more read capacity than is available with a single writer and a single reader, you can add multiple Aurora Serverless v2 readers to the cluster. Each Aurora Serverless v2 reader can scale within the range of minimum and maximum capacity values that you specified for your DB cluster. You can use the cluster's reader endpoint to direct read-only sessions to the readers and reduce the load on the writer.

Whether Aurora Serverless v2 performs scaling, and how fast scaling occurs once it starts, also depends on the minimum and maximum ACU settings for the cluster. In addition, it depends on whether a reader is configured to scale along with the writer or independently from it. For details about the factors that affect Aurora Serverless v2 scaling, see Performance and scaling for Aurora Serverless v2.

Note

Currently, Aurora Serverless v2 writers and readers don't scale all the way down to zero ACUs. Idle Aurora Serverless v2 writers and readers can scale down to the minimum ACU value that you specified for the cluster.
That behavior is different than Aurora Serverless v1, which can pause after a period of idleness, but then takes some time to resume when you open a new connection. When your DB cluster with Aurora Serverless v2 capacity isn't needed for some time, you can stop and start clusters as with provisioned DB clusters. For details about stopping and starting clusters, see Stopping and starting an Amazon Aurora DB cluster.

Aurora Serverless v2 and high availability

The way to establish high availability for an Aurora DB cluster is to make it a Multi-AZ DB cluster. A *Multi-AZ Aurora DB cluster* has compute capacity available at all times in more than one Availability Zone (AZ). That configuration keeps your database up and running even in case of a significant outage. Aurora performs an automatic failover in case of an issue that affects the writer or even the entire AZ. With Aurora Serverless v2, you can choose for the standby compute capacity to scale up and down along with the capacity of the writer. That way, the compute capacity in the second

AZ is ready to take over the current workload at any time. At the same time, the compute capacity in all AZs can scale down when the database is idle. For details about how Aurora works with AWS Regions and Availability Zones, see High availability for Aurora DB instances.

The Aurora Serverless v2 Multi-AZ capability uses *readers* in addition to the writer. Support for readers is new for Aurora Serverless v2 compared to Aurora Serverless v1. You can add up to 15 Aurora Serverless v2 readers spread across 3 AZs to an Aurora DB cluster.

For business-critical applications that must remain available even in case of an issue that affects your entire cluster or the whole AWS Region, you can set up an Aurora global database. You can use Aurora Serverless v2 capacity in the secondary clusters so they're ready to take over during disaster recovery. They can also scale down when the database isn't busy. For details about Aurora global databases, see Using Amazon Aurora global databases.

Aurora Serverless v2 works like provisioned for failover and other high availability features. For more information, see High availability for Amazon Aurora.

Suppose that you want to ensure maximum availability for your Aurora Serverless v2 cluster. You can create a reader in addition to the writer. If you assign the reader to promotion tier 0 or 1, whatever scaling happens for the writer also happens for the reader. That way, a reader with identical capacity is always ready to take over for the writer in case of a failover.

Suppose that you want to run quarterly reports for your business at the same time as your cluster continues to process transactions. If you add an Aurora Serverless v2 reader to the cluster and assign it to a promotion tier from 2 through 15, you can connect directly to that reader to run the reports. Depending on how memory-intensive and CPU-intensive the reporting queries are, that reader can scale up to accommodate the workload. It can then scale down again when the reports are finished.

Aurora Serverless v2 and storage

The storage for each Aurora DB cluster consists of six copies of all your data, spread across three AZs. This built-in data replication applies regardless of whether your DB cluster includes any readers in addition to the writer. That way, your data is safe, even from issues that affect the compute capacity of the cluster.

Aurora Serverless v2 storage has the same reliability and durability characteristics as described in Amazon Aurora storage and reliability. That's because the storage for Aurora DB clusters works the same whether the compute capacity uses Aurora Serverless v2 or provisioned.

Configuration parameters for Aurora clusters

You can adjust all the same cluster and database configuration parameters for clusters with Aurora Serverless v2 capacity as for provisioned DB clusters. However, some capacity-related parameters are handled differently for Aurora Serverless v2. In a mixed-configuration cluster, the parameter values that you specify for those capacity-related parameters still apply to any provisioned writers and readers.

Almost all of the parameters work the same way for Aurora Serverless v2 writers and readers as for provisioned ones. The exceptions are some parameters that Aurora automatically adjusts during scaling, and some parameters that Aurora keeps at fixed values that depend on the maximum capacity setting.

For example, the amount of memory reserved for the buffer cache increases as a writer or reader scales up, and decreases as it scales down. That way, memory can be released when your database isn't busy. Conversely, Aurora automatically sets the maximum number of connections to a value that's appropriate based on the maximum capacity setting. That way, active connections aren't dropped if the load drops and Aurora Serverless v2 scales down. For information about how Aurora Serverless v2 handles specific parameters, see Working with parameter groups for Aurora Serverless v2.

Requirements and limitations for Aurora Serverless v2

When you create a cluster where you intend to use Aurora Serverless v2 DB instances, pay attention to the following requirements and limitations.

Topics

Region and version availability

Feature availability and support varies across specific versions of each Aurora database engine, and across AWS Regions. For more information on version and Region availability with Aurora and Aurora Serverless v2, see Supported Regions and Aurora DB engines for Aurora Serverless v2.

The following example shows the AWS CLI commands to confirm the exact DB engine values you can use with Aurora Serverless v2 for a specific AWS Region. The `--db-instance-class` parameter for Aurora Serverless v2 is always `db.serverless`. The `--engine` parameter can be `aurora-mysql` or `aurora-postgresql`. Substitute the appropriate `--region` and `--engine` values to confirm the `--engine-version` values that you can use. If the command doesn't produce any output, Aurora Serverless v2 isn't available for that combination of AWS Region and DB engine.

```
aws rds describe-orderable-db-instance-options --engine aurora-mysql --db-instance-
class db.serverless \
  --region my_region --query 'OrderableDBInstanceOptions[].[EngineVersion]' --output
 text

aws rds describe-orderable-db-instance-options --engine aurora-postgresql --db-
instance-class db.serverless \
  --region my_region --query 'OrderableDBInstanceOptions[].[EngineVersion]' --output
 text
```

Clusters that use Aurora Serverless v2 must have a capacity range specified

An Aurora cluster must have a `ServerlessV2ScalingConfiguration` attribute before you can add any DB instances that use the `db.serverless` DB instance class. This attribute specifies the capacity range. Aurora Serverless v2 capacity ranges from a minimum of 0.5 Aurora capacity units (ACU) through 128 ACUs, in increments of 0.5 ACU. Each ACU provides the equivalent of approximately 2 gibibytes (GiB) of RAM and associated CPU and networking. For details about how Aurora Serverless v2 uses the capacity range settings, see How Aurora Serverless v2 works.

You can specify the minimum and maximum ACU values in the AWS Management Console when you create a cluster and associated Aurora Serverless v2 DB instance. You can also specify the `--serverless-v2-scaling-configuration` option in the AWS CLI. Or you can specify the `ServerlessV2ScalingConfiguration` parameter with the Amazon RDS API. You can specify this attribute when you create a cluster or modify an existing cluster. For the procedures to set the capacity range, see Setting the Aurora Serverless v2 capacity range for a cluster. For a detailed discussion of how to pick minimum and maximum capacity values and how those settings affect some database parameters, see Choosing the Aurora Serverless v2 capacity range for an Aurora cluster.

Some provisioned features aren't supported in Aurora Serverless v2

The following features from Aurora provisioned DB instances currently aren't available for Amazon Aurora Serverless v2:

- Database activity streams (DAS).
- Cluster cache management for Aurora PostgreSQL. The `apg_ccm_enabled` configuration parameter doesn't apply to Aurora Serverless v2 DB instances.

Some Aurora features work with Aurora Serverless v2, but might cause issues if your capacity range is lower than needed for the memory requirements for those features with your specific workload. In that case, your database might not perform as well as usual, or might encounter out-of-memory errors. For recommendations about setting the appropriate capacity range, see Choosing the Aurora Serverless v2 capacity range for an Aurora cluster. For troubleshooting information if your database encounters out-of-memory errors due to a misconfigured capacity range, see Avoiding out-of-memory errors.

Aurora Auto Scaling isn't supported. This type of scaling adds new readers to handle additional read-intensive workload, based on CPU usage. However, scaling based on CPU usage isn't meaningful for Aurora Serverless v2. As an alternative, you can create Aurora Serverless v2 reader DB instances in advance and leave them scaled down to low capacity. That's a faster and less disruptive way to scale a cluster's read capacity than adding new DB instances dynamically.

Some Aurora Serverless v2 aspects are different from Aurora Serverless v1

If you are an Aurora Serverless v1 user and this is your first time using Aurora Serverless v2, consult the differences between Aurora Serverless v2 and Aurora Serverless v1 requirements to understand how requirements are different between Aurora Serverless v1 and Aurora Serverless v2.

Creating a DB cluster that uses Aurora Serverless v2

To create an Aurora cluster where you can add Aurora Serverless v2 DB instances, you follow the same procedure as in Creating an Amazon Aurora DB cluster. With Aurora Serverless v2, your clusters are interchangeable with provisioned clusters. You can have clusters where some DB instances use Aurora Serverless v2 and some DB instances are provisioned.

Topics

Settings for Aurora Serverless v2 DB clusters

Make sure that the cluster's initial settings meet the requirements listed in Requirements and limitations for Aurora Serverless v2. Specify the following settings to make sure that you can add Aurora Serverless v2 DB instances to the cluster:

AWS Region

Create the cluster in an AWS Region where Aurora Serverless v2 DB instances are available. For details about available Regions, see Supported Regions and Aurora DB engines for Aurora Serverless v2.

DB engine version

Choose an engine version that's compatible with Aurora Serverless v2. For information about the Aurora Serverless v2 version requirements, see Requirements and limitations for Aurora Serverless v2.

DB instance class

If you create a cluster using the AWS Management Console, you choose the DB instance class for the writer DB instance at the same time. Choose the **Serverless** DB instance class. When you choose that DB instance class, you also specify the capacity range for the writer DB instance. That same capacity range applies to all other Aurora Serverless v2 DB instances that you add to that cluster.

If you don't see the **Serverless** choice for the DB instance class, make sure that you chose a DB engine version that's supported for Supported Regions and Aurora DB engines for Aurora Serverless v2.

When you use the AWS CLI or the Amazon RDS API, the parameter that you specify for the DB instance class is `db.serverless`.

Capacity range

> Fill in the minimum and maximum Aurora capacity unit (ACU) values that apply to all the DB instances in the cluster. This option is available on both the **Create cluster** and **Add reader** console pages when you choose **Serverless** for the DB instance class.
>
> If you don't see the minimum and maximum ACU fields, make sure that you chose the **Serverless** DB instance class for the writer DB instance.

If you initially create the cluster with a provisioned DB instance, you don't specify the minimum and maximum ACUs. In that case you can modify the cluster afterward to add that setting. You can also add an Aurora Serverless v2 reader DB instance to the cluster. You specify the capacity range as part of that process.

Until you specify the capacity range for your cluster, you can't add any Aurora Serverless v2 DB instances to the cluster using the AWS CLI or RDS API. If you try to add a Aurora Serverless v2 DB instance, you get an error. In the AWS CLI or the RDS API procedures, the capacity range is represented by the `ServerlessV2ScalingConfiguration` attribute.

For clusters containing more than one reader DB instance, the failover priority of each Aurora Serverless v2 reader DB instance plays an important part in how that DB instance scales up and down. You can't specify the priority when you initially create the cluster. Keep this property in mind when you add a second or later reader DB instance to your cluster. For more information, see Choosing the promotion tier for an Aurora Serverless v2 reader.

Creating an Aurora Serverless v2 DB cluster

You can use the AWS Management Console, AWS CLI, or RDS API to create an Aurora Serverless v2 DB cluster.

Console

To create a cluster with an Aurora Serverless v2 writer

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose **Create database**. On the page that appears, choose the following options:

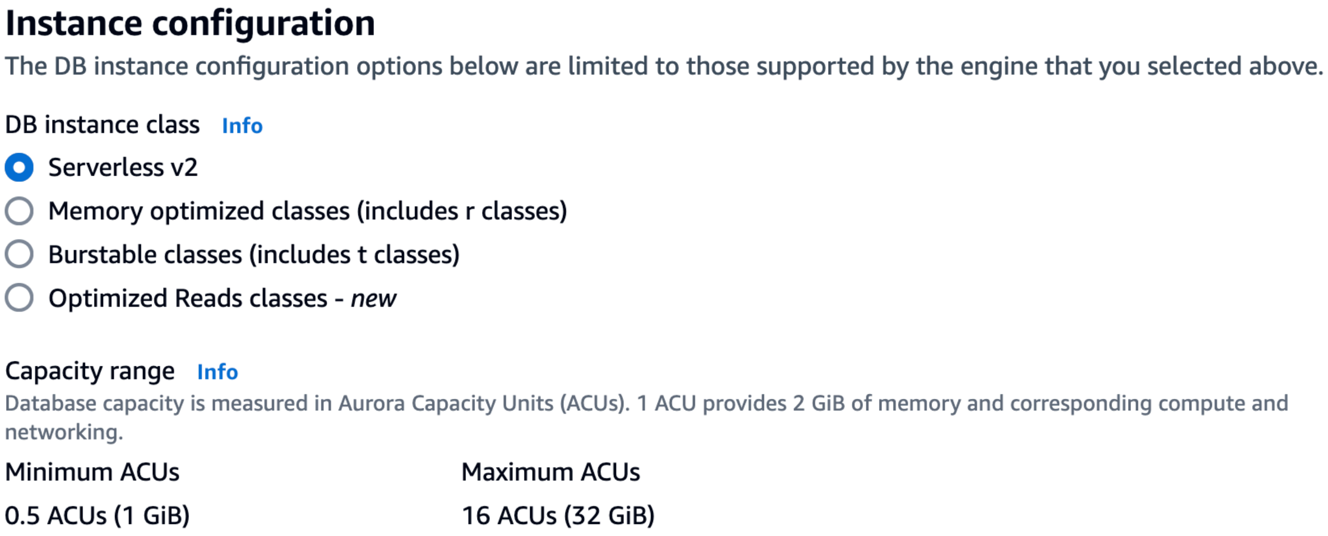
    - For **Engine type**, choose **Aurora (MySQL Compatible)** or **Aurora (PostgreSQL Compatible)**.

   - For **Version**, choose one of the supported versions for Supported Regions and Aurora DB engines for Aurora Serverless v2.

4. For **DB instance class**, select **Serverless v2**.

5. For **Capacity range**, you can accept the default range. Or you can choose other values for minimum and maximum capacity units. You can choose from 0.5 ACUs minimum through 128 ACUs maximum, in increments of 0.5 ACU.

   For more information about Aurora Serverless v2 capacity units, see Aurora Serverless v2 capacity and Performance and scaling for Aurora Serverless v2.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2
○ Memory optimized classes (includes r classes)
○ Burstable classes (includes t classes)
○ Optimized Reads classes - *new*

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
0.5 ACUs (1 GiB)	16 ACUs (32 GiB)

6. Choose any other DB cluster settings, as described in Settings for Aurora DB clusters.

7. Choose **Create database** to create your Aurora DB cluster with an Aurora Serverless v2 DB instance as the writer instance, also known as the primary DB instance.

CLI

To create a DB cluster that's compatible with Aurora Serverless v2 DB instances using the AWS CLI, you follow the CLI procedure in Creating an Amazon Aurora DB cluster. Include the following parameters in your `create-db-cluster` command:

- --region *AWS_Region_where_Aurora Serverless v2_instances_are_available*
- --engine-version *serverless_v2_compatible_engine_version*
- --serverless-v2-scaling-configuration MinCapacity=*minimum_capacity*,MaxCapacity=*maximum_capacity*

The following example shows the creation of an Aurora Serverless v2 DB cluster.

```
aws rds create-db-cluster \
    --db-cluster-identifier my-serverless-v2-cluster \
    --region eu-central-1 \
    --engine aurora-mysql \
    --engine-version 8.0.mysql_aurora.3.04.1 \
    --serverless-v2-scaling-configuration MinCapacity=1,MaxCapacity=4 \
    --master-username myuser \
    --manage-master-user-password
```

Note

When you create an Aurora Serverless v2 DB cluster using the AWS CLI, the engine mode appears in the output as `provisioned` rather than `serverless`. The `serverless` engine mode refers to Aurora Serverless v1.

This example specifies the `--manage-master-user-password` option to generate the administrative password and manage it in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager. Alternatively, you can use the `--master-password` option to specify and manage the password yourself.

For information about the Aurora Serverless v2 version requirements, see Requirements and limitations for Aurora Serverless v2. For information about the allowed numbers for the capacity range and what those numbers represent, see Aurora Serverless v2 capacity and Performance and scaling for Aurora Serverless v2.

To verify whether an existing cluster has the capacity settings specified, check the output of the `describe-db-clusters` command for the `ServerlessV2ScalingConfiguration` attribute. That attribute looks similar to the following.

```
"ServerlessV2ScalingConfiguration": {
    "MinCapacity": 1.5,
    "MaxCapacity": 24.0
}
```

> **Tip**
>
> If you don't specify the minimum and maximum ACUs when you create the cluster, you can use the `modify-db-cluster` command afterward to add that setting. Until you do, you can't add any Aurora Serverless v2 DB instances to the cluster. If you try to add a `db.serverless` DB instance, you get an error.

API

To create a DB cluster that's compatible with Aurora Serverless v2 DB instances using the RDS API, you follow the API procedure in Creating an Amazon Aurora DB cluster. Choose the following settings. Make sure that your `CreateDBCluster` operation includes the following parameters:

```
EngineVersion serverless_v2_compatible_engine_version
ServerlessV2ScalingConfiguration with MinCapacity=minimum_capacity and
 MaxCapacity=maximum_capacity
```

For information about the Aurora Serverless v2 version requirements, see Requirements and limitations for Aurora Serverless v2. For information about the allowed numbers for the capacity range and what those numbers represent, see Aurora Serverless v2 capacity and Performance and scaling for Aurora Serverless v2.

To check if an existing cluster has the capacity settings specified, check the output of the `DescribeDBClusters` operation for the `ServerlessV2ScalingConfiguration` attribute. That attribute looks similar to the following.

```
"ServerlessV2ScalingConfiguration": {
    "MinCapacity": 1.5,
    "MaxCapacity": 24.0
}
```

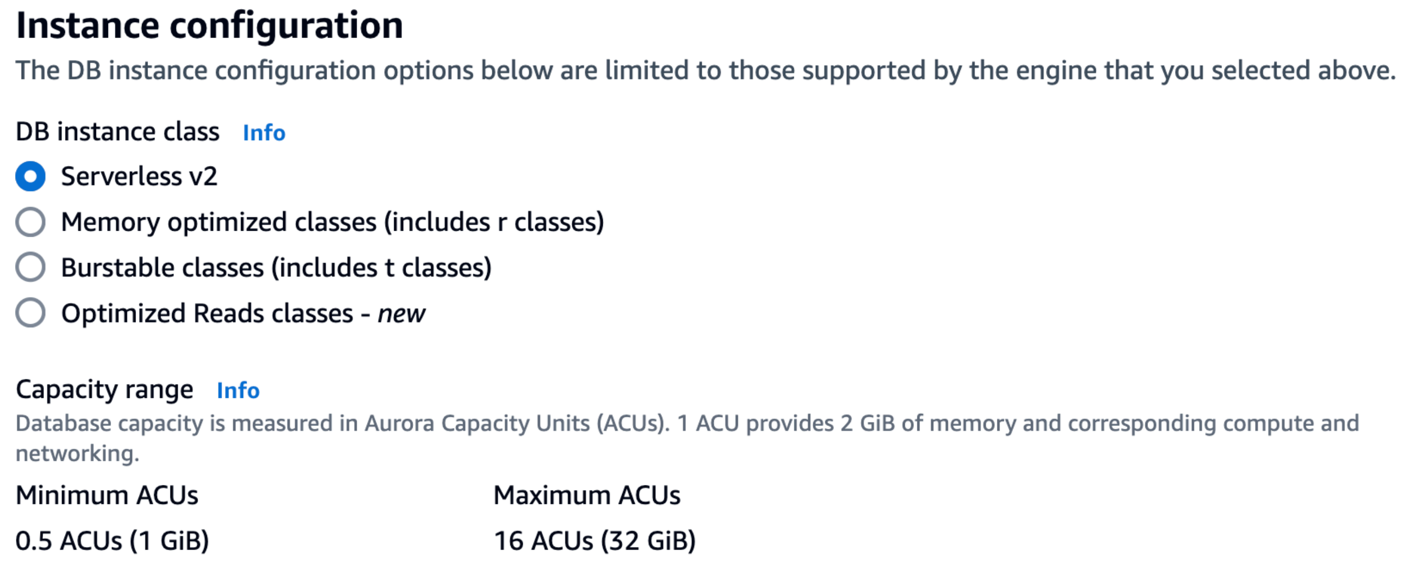
> **Tip**
>
> If you don't specify the minimum and maximum ACUs when you create the cluster, you can use the `ModifyDBCluster` operation afterward to add that setting. Until you do, you can't add any Aurora Serverless v2 DB instances to the cluster. If you try to add a `db.serverless` DB instance, you get an error.

Creating an Aurora Serverless v2 writer DB instance

Console

When you create a DB cluster using the AWS Management Console, you specify the properties of the writer DB instance at the same time. To make the writer DB instance use Aurora Serverless v2, choose the **Serverless** DB instance class.

Then you set the capacity range for the cluster by specifying the minimum and maximum Aurora capacity unit (ACU) values. These minimum and maximum values apply to each Aurora Serverless v2 DB instance in the cluster.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2
○ Memory optimized classes (includes r classes)
○ Burstable classes (includes t classes)
○ Optimized Reads classes - *new*

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
0.5 ACUs (1 GiB)	16 ACUs (32 GiB)

If you don't create an Aurora Serverless v2 DB instance when you first create the cluster, you can add one or more Aurora Serverless v2 DB instances later. To do so, follow the procedures in Adding an Aurora Serverless v2 reader and Converting a provisioned writer or reader to Aurora Serverless v2. You specify the capacity range at the time that you add the first Aurora Serverless v2 DB instance to the cluster. You can change the capacity range later by following the procedure in Setting the Aurora Serverless v2 capacity range for a cluster.

CLI

When you create a Aurora Serverless v2 DB cluster using the AWS CLI, you explicitly add the writer DB instance using the create-db-instance command. Include the following parameter:

- `--db-instance-class db.serverless`

The following example shows the creation of an Aurora Serverless v2 writer DB instance.

```
aws rds create-db-instance \
    --db-cluster-identifier my-serverless-v2-cluster \
    --db-instance-identifier my-serverless-v2-instance \
    --db-instance-class db.serverless \
    --engine aurora-mysql
```

Managing Aurora Serverless v2 DB clusters

With Aurora Serverless v2, your clusters are interchangeable with provisioned clusters. The Aurora Serverless v2 properties apply to one or more DB instances within a cluster. Thus, the procedures for creating clusters, modifying clusters, creating and restoring snapshots, and so on, are basically the same as for other kinds of Aurora clusters. For general procedures for managing Aurora clusters and DB instances, see Managing an Amazon Aurora DB cluster.

In the following topics, you can learn about management considerations for clusters that contain Aurora Serverless v2 DB instances.

Topics

Setting the Aurora Serverless v2 capacity range for a cluster

To modify configuration parameters or other settings for clusters containing Aurora Serverless v2 DB instances, or the DB instances themselves, follow the same general procedures as for provisioned clusters. For details, see Modifying an Amazon Aurora DB cluster.

The most important setting that's unique to Aurora Serverless v2 is the capacity range. After you set the minimum and maximum Aurora capacity unit (ACU) values for an Aurora cluster, you don't need to actively adjust the capacity of the Aurora Serverless v2 DB instances in the cluster. Aurora does that for you. This setting is managed at the cluster level. The same minimum and maximum ACU values apply to each Aurora Serverless v2 DB instance in the cluster.

You can set the following specific values:

- **Minimum ACUs** – The Aurora Serverless v2 DB instance can reduce capacity down to this number of ACUs.
- **Maximum ACUs** – The Aurora Serverless v2 DB instance can increase capacity up to this number of ACUs.

> **Note**
>
> When you modify the capacity range for an Aurora Serverless v2 DB cluster, the change takes place immediately, regardless of whether you choose to apply it immediately or during the next scheduled maintenance window.

For details about the effects of the capacity range and how to monitor and fine-tune it, see Important Amazon CloudWatch metrics for Aurora Serverless v2 and Performance and scaling for Aurora Serverless v2. Your goal is to make sure that the maximum capacity for the cluster is high enough to handle spikes in workload, and the minimum is low enough to minimize costs when the cluster isn't busy.

Suppose that you determine based on your monitoring that the ACU range for the cluster should be higher, lower, wider, or narrower. You can set the capacity of an Aurora cluster to a specific range of ACUs with the AWS Management Console, the AWS CLI, or the Amazon RDS API. This capacity range applies to every Aurora Serverless v2 DB instance in the cluster.

For example, suppose that your cluster has a capacity range of 1–16 ACUs and contains two Aurora Serverless v2 DB instances. Then the cluster as a whole consumes somewhere between 2 ACUs (when idle) and 32 ACUs (when fully utilized). If you change the capacity range from 8 to 20.5 ACUs, now the cluster consumes 16 ACUs when idle, and up to 41 ACUs when fully utilized.

Aurora automatically sets certain parameters for Aurora Serverless v2 DB instances to values that depend on the maximum ACU value in the capacity range. For the list of such parameters, see

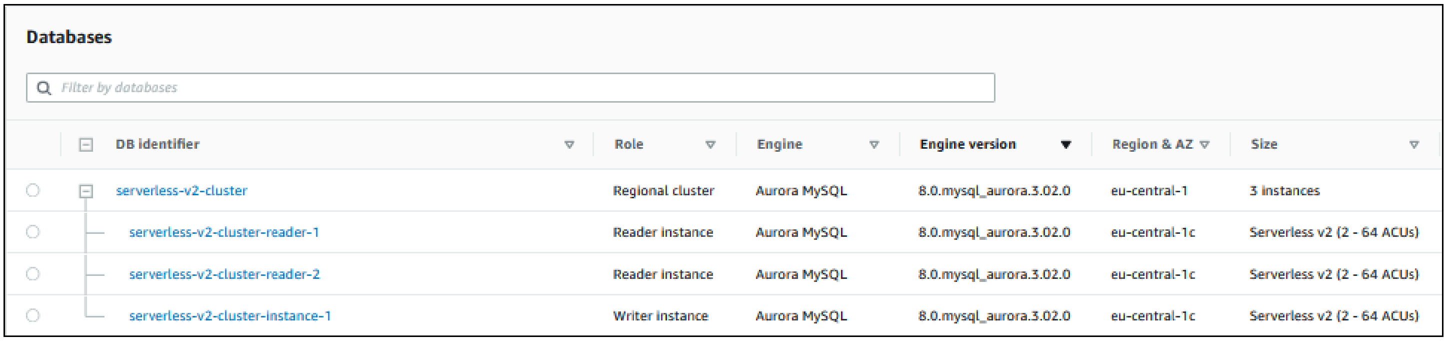
Maximum connections for Aurora Serverless v2. For static parameters that rely on this type of calculation, the value is evaluated again when you reboot the DB instance. Thus, you can update the value of such parameters by rebooting the DB instance after changing the capacity range. To check whether you need to reboot your DB instance to pick up such parameter changes, check the `ParameterApplyStatus` attribute of the DB instance. A value of `pending-reboot` indicates that rebooting will apply changes to some parameter values.

Console

You can set the capacity range of a cluster that contains Aurora Serverless v2 DB instances with the AWS Management Console.

When you use the console, you set the capacity range for the cluster at the time that you add the first Aurora Serverless v2 DB instance to that cluster. You might do so when you choose the **Serverless v2** DB instance class for the writer DB instance when you create the cluster. Or you might do so when you choose the **Serverless** DB instance class when you add an Aurora Serverless v2 reader DB instance to the cluster. Or you might do so when you convert an existing provisioned DB instance in the cluster to the **Serverless** DB instance class. For the full versions of those procedures, see Creating an Aurora Serverless v2 writer DB instance, Adding an Aurora Serverless v2 reader, and Converting a provisioned writer or reader to Aurora Serverless v2.

Whatever capacity range that you set at the cluster level applies to all Aurora Serverless v2 DB instances in your cluster. The following image shows a cluster with multiple Aurora Serverless v2 reader DB instances. Each has an identical capacity range of 2–64 ACUs.



Databases

DB identifier	Role	Engine	Engine version	Region & AZ	Size
serverless-v2-cluster	Regional cluster	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1	3 instances
serverless-v2-cluster-reader-1	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)
serverless-v2-cluster-reader-2	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)
serverless-v2-cluster-instance-1	Writer instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)

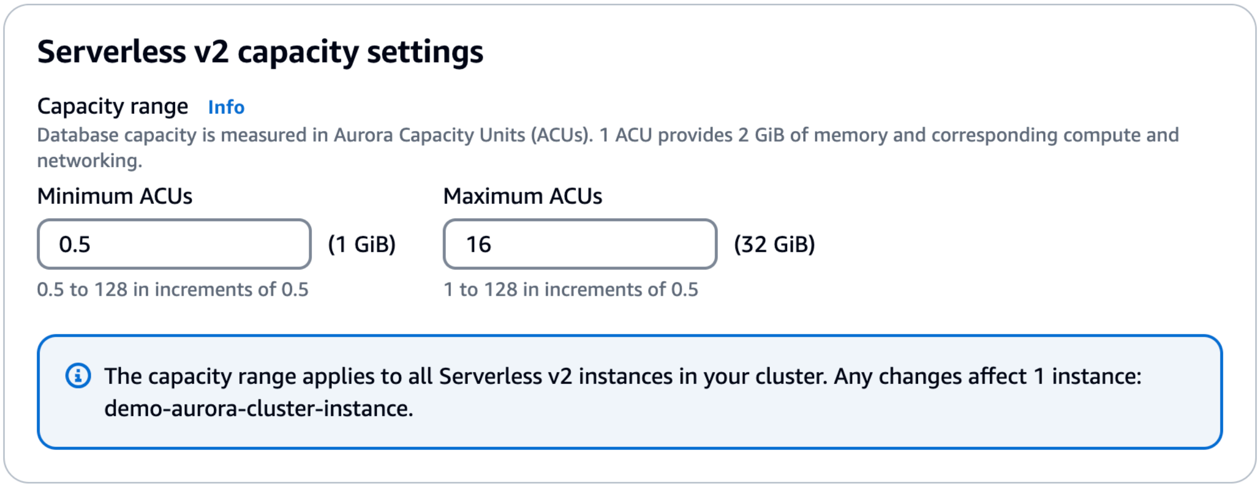
To modify the capacity range of an Aurora Serverless v2 cluster

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.

3. Choose the cluster containing your Aurora Serverless v2 DB instances from the list. The cluster must already contain at least one Aurora Serverless v2 DB instance. Otherwise, Aurora doesn't show the **Capacity range** section.
4. For **Actions**, choose **Modify**.
5. In the **Capacity range** section, choose the following:

   a. Enter a value for **Minimum ACUs**. The console shows the allowed range of values. You can choose a minimum capacity from 0.5 to 128 ACUs. You can choose a maximum capacity from 1 to 128 ACUs. You can adjust the capacity values in increments of 0.5 ACU.

   b. Enter a value for **Maximum ACUs**. This value must be greater than or equal to **Minimum ACUs**. The console shows the allowed range of values. The following figure shows that choice.



Serverless v2 capacity settings

Capacity range Info
Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs: 0.5 (1 GiB)
0.5 to 128 in increments of 0.5

Maximum ACUs: 16 (32 GiB)
1 to 128 in increments of 0.5

The capacity range applies to all Serverless v2 instances in your cluster. Any changes affect 1 instance: demo-aurora-cluster-instance.

6. Choose **Continue**. The **Summary of modifications** page appears.
7. Choose **Apply immediately**.

   The capacity modification takes place immediately, regardless of whether you choose to apply it immediately or during the next scheduled maintenance window.
8. Choose **Modify cluster** to accept the summary of modifications. You can also choose **Back** to modify your changes or **Cancel** to discard your changes.

AWS CLI

To set the capacity of a cluster where you intend to use Aurora Serverless v2 DB instances using the AWS CLI, run the modify-db-cluster AWS CLI command. Specify the `--serverless-v2-scaling-configuration` option. The cluster might already contain one or more

Aurora Serverless v2 DB instances, or you can add the DB instances later. Valid values for the `MinCapacity` and `MaxCapacity` fields include the following:

- `0.5`, 1, `1.5`, 2, and so on, in steps of 0.5, up to a maximum of 128.

In this example, you set the ACU range of an Aurora DB cluster named `sample-cluster` to a minimum of `48.5` and a maximum of 64.

```
aws rds modify-db-cluster --db-cluster-identifier sample-cluster \
--serverless-v2-scaling-configuration MinCapacity=48.5,MaxCapacity=64
```

The capacity modification takes place immediately, regardless of whether you choose to apply it immediately or during the next scheduled maintenance window.

After doing so, you can add Aurora Serverless v2 DB instances to the cluster and each new DB instance can scale between 48.5 and 64 ACUs. The new capacity range also applies to any Aurora Serverless v2 DB instances that were already in the cluster. The DB instances scale up or down if necessary to fall within the new capacity range.

For additional examples of setting the capacity range using the CLI, see Choosing the Aurora Serverless v2 capacity range for an Aurora cluster.

To modify the scaling configuration of an Aurora Serverless DB cluster using the AWS CLI, run the modify-db-cluster AWS CLI command. Specify the `--serverless-v2-scaling-configuration` option to configure the minimum capacity and maximum capacity. Valid capacity values include the following:

- Aurora MySQL: `0.5`, 1, `1.5`, 2, and so on, in increments of 0.5 ACUs up to a maximum of `128`.
- Aurora PostgreSQL: `0.5`, 1, `1.5`, 2, and so on, in increments of 0.5 ACUs up to a maximum of `128`.

In the following example, you modify the scaling configuration of an Aurora Serverless v2 DB instance named `sample-instance` that's part of a cluster named `sample-cluster`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster --db-cluster-identifier sample-cluster \
--serverless-v2-scaling-configuration MinCapacity=8,MaxCapacity=64
```

For Windows:

```
aws rds modify-db-cluster --db-cluster-identifier sample-cluster ^
--serverless-v2-scaling-configuration MinCapacity=8,MaxCapacity=64
```

RDS API

You can set the capacity of an Aurora DB instance with the ModifyDBCluster API operation. Specify the `ServerlessV2ScalingConfiguration` parameter. Valid values for the `MinCapacity` and `MaxCapacity` fields include the following:

- `0.5`, 1, `1.5`, 2, and so on, in steps of 0.5, up to a maximum of 128.

You can modify the scaling configuration of a cluster containing Aurora Serverless v2 DB instances with the ModifyDBCluster API operation. Specify the `ServerlessV2ScalingConfiguration` parameter to configure the minimum capacity and the maximum capacity. Valid capacity values include the following:

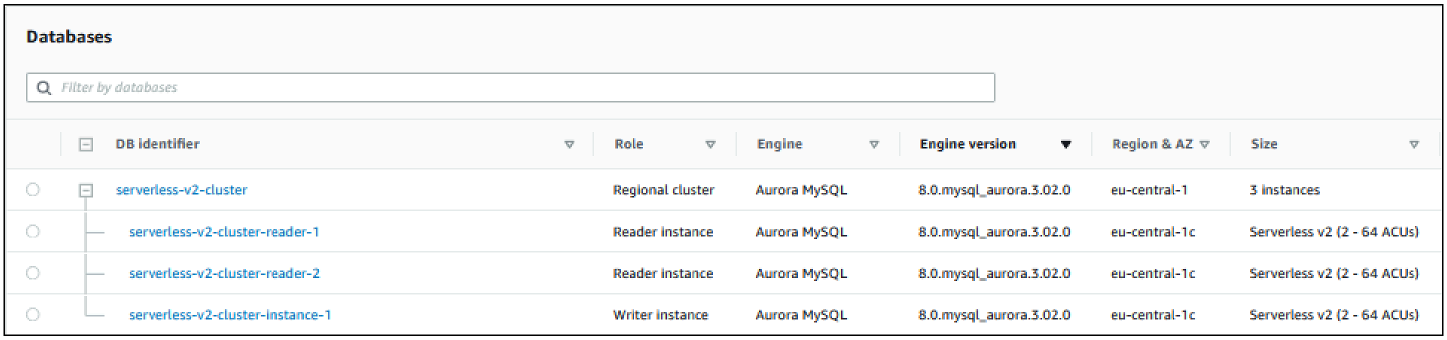
- Aurora MySQL: `0.5`, 1, `1.5`, 2, and so on, in increments of 0.5 ACUs up to a maximum of `128`.
- Aurora PostgreSQL: `0.5`, 1, `1.5`, 2, and so on, in increments of 0.5 ACUs up to a maximum of `128`.

The capacity modification takes place immediately, regardless of whether you choose to apply it immediately or during the next scheduled maintenance window.

Checking the capacity range for Aurora Serverless v2

The procedure to check the capacity range for your Aurora Serverless v2 cluster requires that you first set a capacity range. If you haven't done so, follow the procedure in Setting the Aurora Serverless v2 capacity range for a cluster.

Whatever capacity range you set at the cluster level applies to all Aurora Serverless v2 DB instances in your cluster. The following image shows a cluster with multiple Aurora Serverless v2 DB instances. Each has an identical capacity range.

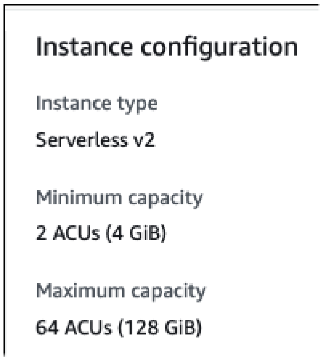


Databases

Filter by databases

DB identifier	Role	Engine	Engine version	Region & AZ	Size
serverless-v2-cluster	Regional cluster	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1	3 instances
serverless-v2-cluster-reader-1	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)
serverless-v2-cluster-reader-2	Reader instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)
serverless-v2-cluster-instance-1	Writer instance	Aurora MySQL	8.0.mysql_aurora.3.02.0	eu-central-1c	Serverless v2 (2 - 64 ACUs)

You can also view the details page for any Aurora Serverless v2 DB instance in the cluster. DB instances' capacity range appears on the **Configuration** tab.



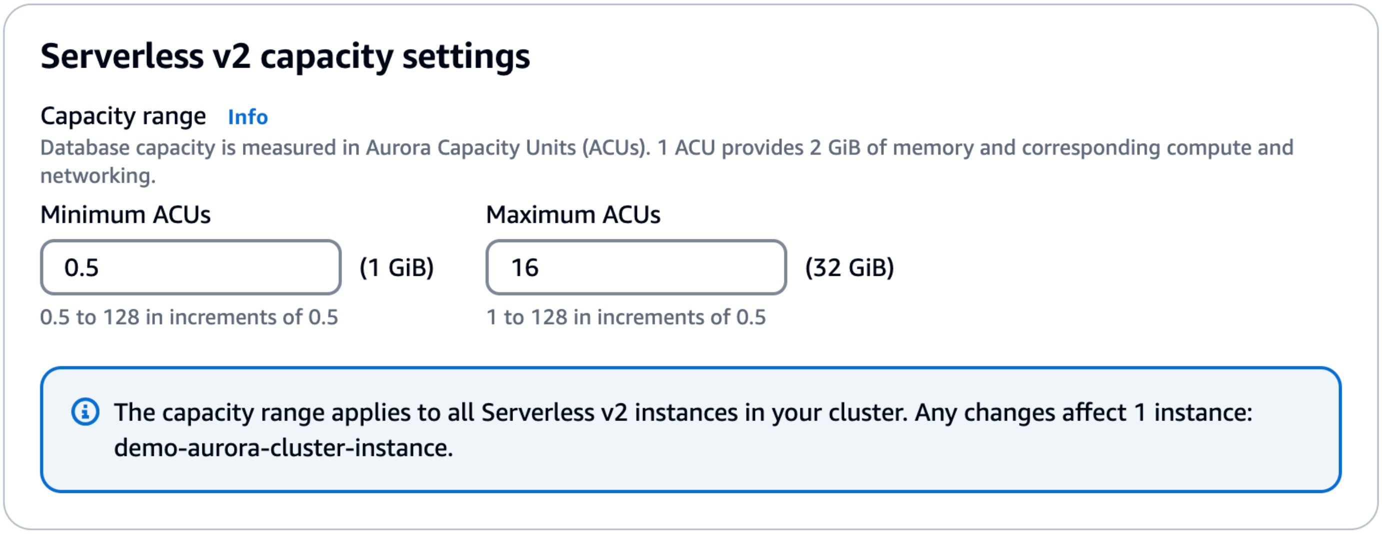
Instance configuration

Instance type
Serverless v2

Minimum capacity
2 ACUs (4 GiB)

Maximum capacity
64 ACUs (128 GiB)

You can also see the current capacity range for the cluster on the **Modify** page for the cluster. The following image shows how. At that point, you can change the capacity range. For all the ways that you can set or change the capacity range, see [Setting the Aurora Serverless v2 capacity range for a cluster](#).



Serverless v2 capacity settings

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs: 0.5 (1 GiB)
0.5 to 128 in increments of 0.5

Maximum ACUs: 16 (32 GiB)
1 to 128 in increments of 0.5

The capacity range applies to all Serverless v2 instances in your cluster. Any changes affect 1 instance: demo-aurora-cluster-instance.

Checking the current capacity range for an Aurora cluster

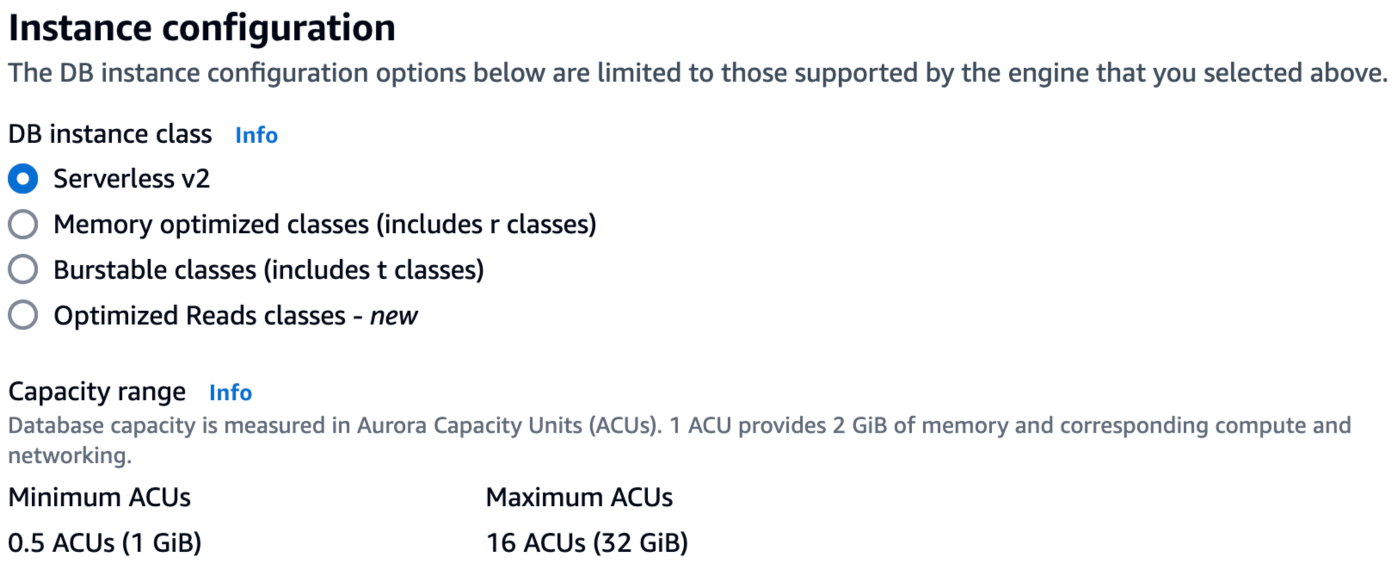
You can check the capacity range that's configured for Aurora Serverless v2 DB instances in a cluster by examining the `ServerlessV2ScalingConfiguration` attribute for the cluster. The following AWS CLI example shows a cluster with a minimum capacity of 0.5 Aurora capacity units (ACUs) and a maximum capacity of 16 ACUs.

```
$ aws rds describe-db-clusters --db-cluster-identifier serverless-v2-64-acu-cluster \
  --query 'DBClusters[*].[ServerlessV2ScalingConfiguration]'
[
    [
        {
            "MinCapacity": 0.5,
            "MaxCapacity": 16.0
        }
    ]
]
```

Adding an Aurora Serverless v2 reader

To add an Aurora Serverless v2 reader DB instance to your cluster, you follow the same general procedure as in [Adding Aurora Replicas to a DB cluster](). Choose the **Serverless v2** instance class for the new DB instance.

If the reader DB instance is the first Aurora Serverless v2 DB instance in the cluster, you also choose the capacity range. The following image shows the controls that you use to specify the minimum and maximum Aurora capacity units (ACUs). This setting applies to this reader DB instance and to any other Aurora Serverless v2 DB instances that you add to the cluster. Each Aurora Serverless v2 DB instance can scale between the minimum and maximum ACU values.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

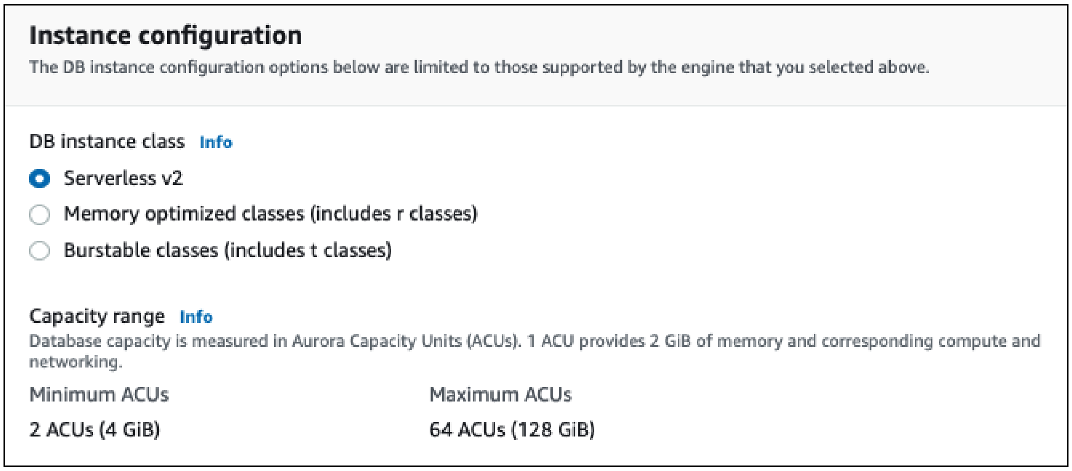
○ Optimized Reads classes - *new*

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
0.5 ACUs (1 GiB)	16 ACUs (32 GiB)

If you already added any Aurora Serverless v2 DB instances to the cluster, adding another Aurora Serverless v2 reader DB instance shows you the current capacity range. The following image shows those read-only controls.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
2 ACUs (4 GiB)	64 ACUs (128 GiB)

If you want to change the capacity range for the cluster, follow the procedure in Setting the Aurora Serverless v2 capacity range for a cluster.

For clusters containing more than one reader DB instance, the failover priority of each Aurora Serverless v2 reader DB instance plays an important part in how that DB instance scales up and down. You can't specify the priority when you initially create the cluster. Keep this property in mind when you add a second reader or later DB instance to your cluster. For more information, see Choosing the promotion tier for an Aurora Serverless v2 reader.

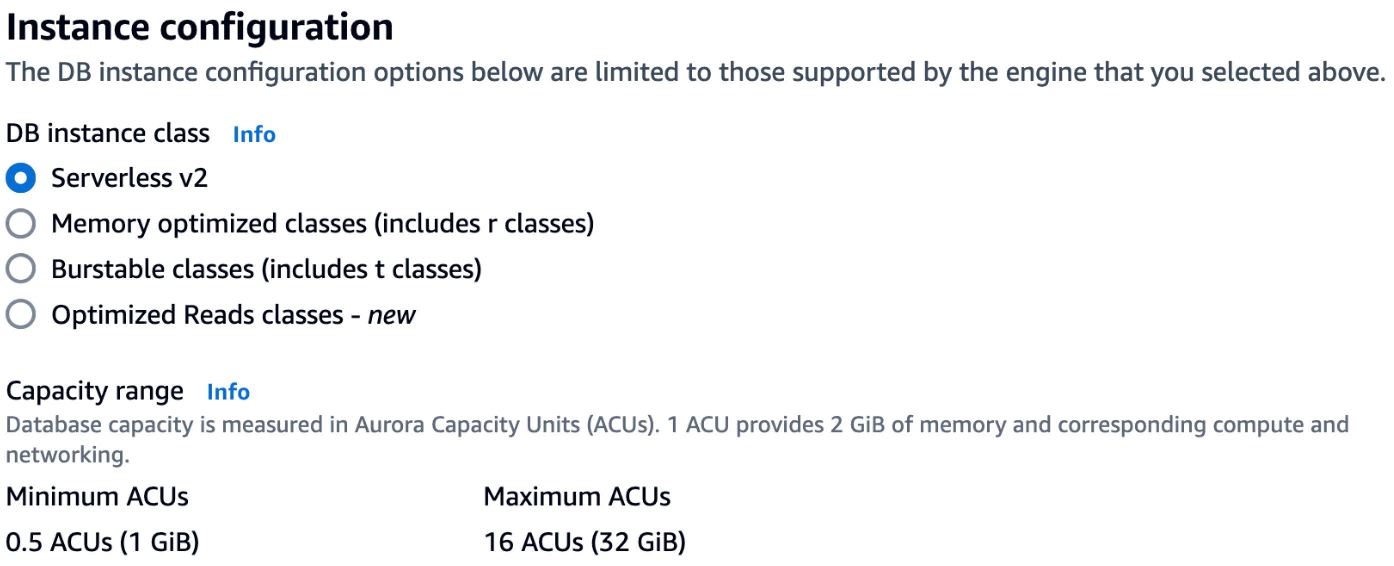
For other ways that you can see the current capacity range for a cluster, see Checking the capacity range for Aurora Serverless v2.

Converting a provisioned writer or reader to Aurora Serverless v2

You can convert a provisioned DB instance to use Aurora Serverless v2. To do so, you follow the procedure in Modifying a DB instance in a DB cluster. The cluster must meet the requirements in Requirements and limitations for Aurora Serverless v2. For example, Aurora Serverless v2 DB instances require that the cluster be running certain minimum engine versions.

Suppose that you are converting a running provisioned cluster to take advantage of Aurora Serverless v2. In that case, you can minimize downtime by converting a DB instance to Aurora Serverless v2 as the first step in the switchover process. For the full procedure, see Switching from a provisioned cluster to Aurora Serverless v2.

If the DB instance that you convert is the first Aurora Serverless v2 DB instance in the cluster, you choose the capacity range for the cluster as part of the **Modify** operation. This capacity range applies to each Aurora Serverless v2 DB instance that you add to the cluster. The following image shows the page where you specify the minimum and maximum Aurora capacity units (ACUs).



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

○ Optimized Reads classes - *new*

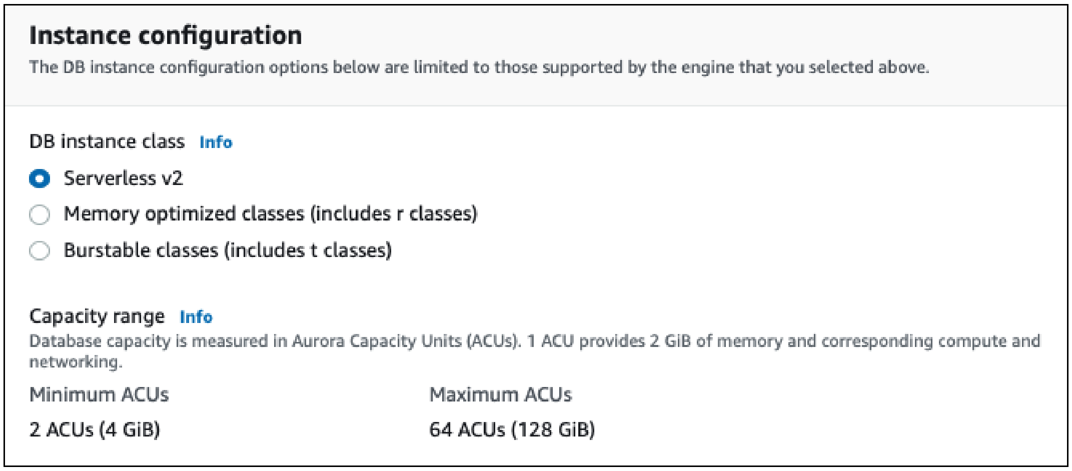
Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
0.5 ACUs (1 GiB)	16 ACUs (32 GiB)

For details about the significance of the capacity range, see Aurora Serverless v2 capacity.

If the cluster already contains one or more Aurora Serverless v2 DB instances, you see the existing capacity range during the **Modify** operation. The following image shows an example of that information panel.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless v2

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
2 ACUs (4 GiB)	64 ACUs (128 GiB)

If you want to change the capacity range for the cluster after you add more Aurora Serverless v2 DB instances, follow the procedure in Setting the Aurora Serverless v2 capacity range for a cluster.

Converting an Aurora Serverless v2 writer or reader to provisioned

You can convert an Aurora Serverless v2 DB instance to a provisioned DB instance. To do so, you follow the procedure in Modifying a DB instance in a DB cluster. Choose a DB instance class other than **Serverless**.

You might convert an Aurora Serverless v2 DB instance to provisioned if it needs a larger capacity than is available with the maximum Aurora capacity units (ACUs) of an Aurora Serverless v2 DB instance. For example, the largest db.r5 and db.r6g DB instance classes have a larger memory capacity than an Aurora Serverless v2 DB instance can scale to.

> **Tip**
>
> Some of the older DB instance classes such as db.r3 and db.t2 aren't available for the Aurora versions that you use with Aurora Serverless v2. To see which DB instance classes you can use when converting an Aurora Serverless v2 DB instance to a provisioned one, see Supported DB engines for DB instance classes.

If you are converting the writer DB instance of your cluster from Aurora Serverless v2 to provisioned, you can follow the procedure in Switching from a provisioned cluster to Aurora Serverless v2 but in reverse. Switch one of the reader DB instances in the cluster from Aurora Serverless v2 to provisioned. Then perform a failover to make that provisioned DB instance into the writer.

Any capacity range that you previously specified for the cluster remains in place, even if all Aurora Serverless v2 DB instances are removed from the cluster. If you want to change the capacity range, you can modify the cluster, as explained in Setting the Aurora Serverless v2 capacity range for a cluster.

Choosing the promotion tier for an Aurora Serverless v2 reader

For clusters containing multiple Aurora Serverless v2 DB instances or a mixture of provisioned and Aurora Serverless v2 DB instances, pay attention to the promotion tier setting for each Aurora Serverless v2 DB instance. This setting controls more behavior for Aurora Serverless v2 DB instances than for provisioned DB instances.

In the AWS Management Console, you specify this setting using the **Failover priority** choice under **Additional configuration** for the **Create database**, **Modify instance**, and **Add reader** pages. You see this property for existing DB instances in the optional **Priority tier** column on the **Databases** page. You can also see this property on the details page for a DB cluster or DB instance.

For provisioned DB instances, the choice of tier 0–15 determines only the order in which Aurora chooses which reader DB instance to promote to the writer during a failover operation. For Aurora Serverless v2 reader DB instances, the tier number also determines whether the DB instance scales up to match the capacity of the writer DB instance or scales independently based on its own workload. Aurora Serverless v2 reader DB instances in tier 0 or 1 are kept at a minimum capacity at least as high as the writer DB instance. That way, they are ready to take over from the writer DB instance in case of a failover. If the writer DB instance is a provisioned DB instance, Aurora estimates the equivalent Aurora Serverless v2 capacity. It uses that estimate as the minimum capacity for the Aurora Serverless v2 reader DB instance.

Aurora Serverless v2 reader DB instances in tiers 2–15 don't have the same constraint on their minimum capacity. When they are idle, they can scale down to the minimum Aurora capacity unit (ACU) value specified in the cluster's capacity range.

The following Linux AWS CLI example shows how to examine the promotion tiers of all the DB instances in your cluster. The final field includes a value of `True` for the writer DB instance and `False` for all the reader DB instances.

```
$ aws rds describe-db-clusters --db-cluster-identifier promotion-tier-demo \
  --query 'DBClusters[*].DBClusterMembers[*].
[PromotionTier,DBInstanceIdentifier,IsClusterWriter]' \
  --output text
```

```
1   instance-192  True
1   tier-01-4840  False
10  tier-10-7425  False
15  tier-15-6694  False
```

The following Linux AWS CLI example shows how to change the promotion tier of a specific DB instance in your cluster. The commands first modify the DB instance with a new promotion tier. Then they wait for the DB instance to become available again and confirm the new promotion tier for the DB instance.

```
$ aws rds modify-db-instance --db-instance-identifier instance-192 --promotion-tier 0
$ aws rds wait db-instance-available --db-instance-identifier instance-192
$ aws rds describe-db-instances --db-instance-identifier instance-192 \
  --query '*[].[PromotionTier]' --output text
0
```

For more guidance about specifying promotion tiers for different use cases, see Aurora Serverless v2 scaling.

Using TLS/SSL with Aurora Serverless v2

Aurora Serverless v2 can use the Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocol to encrypt communications between clients and your Aurora Serverless v2 DB instances. It supports TLS/SSL versions 1.0, 1.1, and 1.2. For general information about using TLS/SSL with Aurora, see Using TLS with Aurora MySQL DB clusters.

To learn more about connecting to Aurora MySQL database with the MySQL client, see Connecting to a DB instance running the MySQL database engine.

Aurora Serverless v2 supports all TLS/SSL modes available to the MySQL client (`mysql`) and PostgreSQL client (`psql`), including those listed in the following table.

Description of TLS/SSL mode	mysql	psql
Connect without using TLS/SSL.	DISABLED	disable

Description of TLS/SSL mode	mysql	psql
Try the connection using TLS/SSL first, but fall back to non-SSL if necessary.	PREFERRED	prefer (default)
Enforce using TLS/SSL.	REQUIRED	require
Enforce TLS/SSL and verify the certificate authority (CA).	VERIFY_CA	verify-ca
Enforce TLS/SSL, verify the CA, and verify the CA hostname.	VERIFY_IDENTITY	verify-full

Aurora Serverless v2 uses wildcard certificates. If you specify the "verify CA" or the "verify CA and CA hostname" option when using TLS/SSL, first download the Amazon root CA 1 trust store from Amazon Trust Services. After doing so, you can identify this PEM-formatted file in your client command. To do so using the PostgreSQL client, do the following.

For Linux, macOS, or Unix:

```
psql 'host=endpoint user=user sslmode=require sslrootcert=amazon-root-CA-1.pem
 dbname=db-name'
```

To learn more about working with the Aurora PostgreSQL database using the Postgres client, see Connecting to a DB instance running the PostgreSQL database engine.

For more information about connecting to Aurora DB clusters in general, see Connecting to an Amazon Aurora DB cluster.

Supported cipher suites for connections to Aurora Serverless v2 DB clusters

By using configurable cipher suites, you can have more control over the security of your database connections. You can specify a list of cipher suites that you want to allow to secure client TLS/SSL connections to your database. With configurable cipher suites, you can control the connection encryption that your database server accepts. Doing this prevents the use of ciphers that aren't secure or that are no longer used.

Aurora Serverless v2 DB clusters that are based on Aurora MySQL support the same cipher suites as Aurora MySQL provisioned DB clusters. For information about these cipher suites, see Configuring cipher suites for connections to Aurora MySQL DB clusters.

Aurora Serverless v2 DB clusters that are based on Aurora PostgreSQL support the same cipher suites as Aurora PostgreSQL provisioned DB clusters. For information about these cipher suites, see Configuring cipher suites for connections to Aurora PostgreSQL DB clusters.

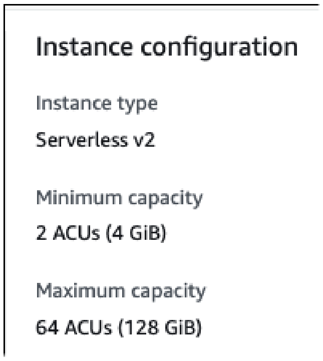
Viewing Aurora Serverless v2 writers and readers

You can view the details of Aurora Serverless v2 DB instances in the same way that you do for provisioned DB instances. To do so, follow the general procedure from Viewing an Amazon Aurora DB cluster. A cluster might contain all Aurora Serverless v2 DB instances, all provisioned DB instances, or some of each.

After you create one or more Aurora Serverless v2 DB instances, you can view which DB instances are type **Serverless** and which are type **Instance**. You can also view the minimum and maximum Aurora capacity units (ACUs) that the Aurora Serverless v2 DB instance can use. Each ACU is a combination of processing (CPU) and memory (RAM) capacity. This capacity range applies to each Aurora Serverless v2 DB instance in the cluster. For the procedure to check the capacity range of a cluster or any Aurora Serverless v2 DB instance in the cluster, see Checking the capacity range for Aurora Serverless v2.

In the AWS Management Console, Aurora Serverless v2 DB instances are marked under the **Size** column in the **Databases** page. Provisioned DB instances show the name of a DB instance class such as r6g.xlarge. The Aurora Serverless DB instances show **Serverless** for the DB instance class, along with the DB instance's minimum and maximum capacity. For example, you might see **Serverless v2 (4–64 ACUs)** or **Serverless v2 (1–40 ACUs)**.

You can find the same information on the **Configuration** tab for each Aurora Serverless v2 DB instance in the console. For example, you might see an **Instance type** section such as the following. Here, the **Instance type** value is **Serverless v2**, the **Minimum capacity** value is **2 ACUs (4 GiB)**, and the **Maximum capacity** value is **64 ACUs (128 GiB)**.



Instance configuration

Instance type
Serverless v2

Minimum capacity
2 ACUs (4 GiB)

Maximum capacity
64 ACUs (128 GiB)

You can monitor the capacity of each Aurora Serverless v2 DB instance over time. That way, you can check the minimum, maximum, and average ACUs consumed by each DB instance. You can also check how close the DB instance came to its minimum or maximum capacity. To see such details in the AWS Management Console, examine the graphs of Amazon CloudWatch metrics on the **Monitoring** tab for the DB instance. For information about the metrics to watch and how to interpret them, see Important Amazon CloudWatch metrics for Aurora Serverless v2.

Logging for Aurora Serverless v2

To turn on database logging, you specify the logs to enable using configuration parameters in your custom parameter group.

For Aurora MySQL, you can enable the following logs.

Aurora MySQL	Description
`general_log`	Creates the general log. Set to 1 to turn on. Default is off (0).
`log_queries_not_using_indexes`	Logs any queries to the slow query log that don't use an index. Default is off (0). Set to 1 to turn on this log.
`long_query_time`	Prevents fast-running queries from being logged in the slow query log. Can be set to a float between 0 and 31536000. Default is 0 (not active).
`server_audit_events`	The list of events to capture in the logs. Supported values are `CONNECT`, `QUERY`,

Aurora MySQL	Description
	QUERY_DCL , QUERY_DDL , QUERY_DML , and TABLE.
server_audit_logging	Set to 1 to turn on server audit logging. If you turn this on, you can specify the audit events to send to CloudWatch by listing them in the server_audit_events parameter.
slow_query_log	Creates a slow query log. Set to 1 to turn on the slow query log. Default is off (0).

For more information, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

For Aurora PostgreSQL, you can enable the following logs on your Aurora Serverless v2 DB instances.

Aurora PostgreSQL	Description
log_connections	Logs each successful connection.
log_disconnections	Logs end of a session including duration.
log_lock_waits	Default is 0 (off). Set to 1 to log lock waits.
log_min_duration_statement	The minimum duration (in milliseconds) for a statement to run before it's logged.
log_min_messages	Sets the message levels that are logged. Supported values are debug5, debug4, debug3, debug2, debug1, info, notice, warning, error, log, fatal, panic. To log performance data to the postgres log, set the value to debug1.
log_temp_files	Logs the use of temporary files that are above the specified kilobytes (kB).

Aurora PostgreSQL	Description
`log_statement`	Controls the specific SQL statements that get logged. Supported values are `none`, `ddl`, `mod`, and `all`. Default is `none`.

Topics

- Logging with Amazon CloudWatch
- Viewing Aurora Serverless v2 logs in Amazon CloudWatch
- Monitoring capacity with Amazon CloudWatch

Logging with Amazon CloudWatch

After you use the procedure in Logging for Aurora Serverless v2 to choose which database logs to turn on, you can choose which logs to upload ("publish") to Amazon CloudWatch.

You can use Amazon CloudWatch to analyze log data, create alarms, and view metrics. By default, error logs for Aurora Serverless v2 are enabled and automatically uploaded to CloudWatch. You can also upload other logs from Aurora Serverless v2 DB instances to CloudWatch.

Then you choose which of those logs to upload to CloudWatch, by using the **Log exports** settings in the AWS Management Console or the `--enable-cloudwatch-logs-exports` option in the AWS CLI.

You can choose which of your Aurora Serverless v2 logs to upload to CloudWatch. For more information, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

As with any type of Aurora DB cluster, you can't modify the default DB cluster parameter group. Instead, create your own DB cluster parameter group based on a default parameter for your DB cluster and engine type. We recommend that you create your custom DB cluster parameter group before creating your Aurora Serverless v2 DB cluster, so that it's available to choose when you create a database on the console.

Note

For Aurora Serverless v2, you can create both DB cluster and DB parameter groups. This contrasts with Aurora Serverless v1, where you can only create DB cluster parameter groups.

Viewing Aurora Serverless v2 logs in Amazon CloudWatch

After you use the procedure in Logging with Amazon CloudWatch to choose which database logs to turn on, you can view the contents of the logs.

For more information on using CloudWatch with Aurora MySQL and Aurora PostgreSQL logs, see Monitoring log events in Amazon CloudWatch and Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs.

To view logs for your Aurora Serverless v2 DB cluster

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose your AWS Region.
3. Choose **Log groups**.
4. Choose your Aurora Serverless v2 DB cluster log from the list. The log naming pattern is as follows.

```
/aws/rds/cluster/cluster-name/log_type
```

Note

For Aurora MySQL–compatible Aurora Serverless v2 DB clusters, the error log includes buffer pool scaling events even when there are no errors.

Monitoring capacity with Amazon CloudWatch

With Aurora Serverless v2, you can use CloudWatch to to monitor the capacity and utilization of all the Aurora Serverless v2 DB instances in your cluster. You can view instance-level metrics to check the capacity of each Aurora Serverless v2 DB instance as it scales up and down. You can also compare the capacity-related metrics to other metrics to see how changes in workloads affect resource consumption. For example, you can compare `ServerlessDatabaseCapacity` to `DatabaseUsedMemory`, `DatabaseConnections`, and `DMLThroughput` to assess how your DB cluster is responding during operations. For details about the capacity-related metrics that apply to Aurora Serverless v2, see Important Amazon CloudWatch metrics for Aurora Serverless v2.

To monitor your Aurora Serverless v2 DB cluster's capacity

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Metrics**. All available metrics appear as cards in the console, grouped by service name.
3. Choose **RDS**.
4. (Optional) Use the **Search** box to find the metrics that are especially important for Aurora Serverless v2: `ServerlessDatabaseCapacity`, `ACUUtilization`, `CPUUtilization`, and `FreeableMemory`.

We recommend that you set up a CloudWatch dashboard to monitor your Aurora Serverless v2 DB cluster capacity using the capacity-related metrics. To learn how, see Building dashboards with CloudWatch.

To learn more about using Amazon CloudWatch with Amazon Aurora, see Publishing Amazon Aurora MySQL logs to Amazon CloudWatch Logs.

Performance and scaling for Aurora Serverless v2

The following procedures and examples show how you can set the capacity range for Aurora Serverless v2 clusters and their associated DB instances. You can also use procedures following to monitor how busy your DB instances are. Then you can use your findings to determine if you need to adjust the capacity range upward or downward.

Before you use these procedures, make sure that you are familiar with how Aurora Serverless v2 scaling works. The scaling mechanism is different than in Aurora Serverless v1. For details, see Aurora Serverless v2 scaling.

Contents

Choosing the Aurora Serverless v2 capacity range for an Aurora cluster

With Aurora Serverless v2 DB instances, you set the capacity range that applies to all the DB instances in your DB cluster at the same time that you add the first Aurora Serverless v2 DB instance to the DB cluster. For the procedure to do so, see Setting the Aurora Serverless v2 capacity range for a cluster.

You can also change the capacity range for an existing cluster. The following sections discuss in more detail how to choose appropriate minimum and maximum values and what happens when you make a change to the capacity range. For example, changing the capacity range can modify the default values of some configuration parameters. Applying all the parameter changes can require rebooting each Aurora Serverless v2 DB instance.

Topics

Choosing the minimum Aurora Serverless v2 capacity setting for a cluster

It's tempting to always choose 0.5 for the minimum Aurora Serverless v2 capacity setting. That value allows the DB instance to scale down the most when it's completely idle. However, depending on how you use that cluster and the other settings that you configure, a different value might be the most effective. Consider the following factors when choosing the minimum capacity setting:

- The scaling rate for an Aurora Serverless v2 DB instance depends on its current capacity. The higher the current capacity, the faster it can scale up. If you need the DB instance to quickly scale up to a very high capacity, consider setting the minimum capacity to a value where the scaling rate meets your requirement.
- If you typically modify the DB instance class of your DB instances in anticipation of especially high or low workload, you can use that experience to make a rough estimate of the equivalent Aurora Serverless v2 capacity range. To determine the memory size to use in times of low traffic, consult Hardware specifications for DB instance classes for Aurora.

  For example, suppose that you use the db.r6g.xlarge DB instance class when your cluster has a low workload. That DB instance class has 32 GiB of memory. Thus, you can specify a minimum Aurora capacity unit (ACU) setting of 16 to set up an Aurora Serverless v2 DB instance that can scale down to approximately that same capacity. That's because each ACU corresponds to approximately 2 GiB of memory. You might specify a somewhat lower value to let the DB instance scale down further in case your db.r6g.xlarge DB instance was sometimes underutilized.
- If your application works most efficiently when the DB instances have a certain amount of data in the buffer cache, consider specifying a minimum ACU setting where the memory is large enough to hold the frequently accessed data. Otherwise, some data is evicted from the buffer cache when the Aurora Serverless v2 DB instances scale down to a lower memory size. Then when the DB instances scale back up, the information is read back into the buffer cache over time. If the amount of I/O to bring the data back into the buffer cache is substantial, it might be more effective to choose a higher minimum ACU value.
- If your Aurora Serverless v2 DB instances run most of the time at a particular capacity, consider specifying a minimum capacity setting that's lower than that baseline, but not too much lower. Aurora Serverless v2 DB instances can most effectively estimate how much and how fast to scale up when the current capacity isn't drastically lower than the required capacity.
- If your provisioned workload has memory requirements that are too high for small DB instance classes such as T3 or T4g, choose a minimum ACU setting that provides memory comparable to an R5 or R6g DB instance.

  In particular, we recommend the following minimum capacity for use with the specified features (these recommendations are subject to change):

  - Performance Insights – 2 ACUs
  - Aurora global databases – 8 ACUs (applies only to the primary AWS Region)

- In some cases, your cluster might contain Aurora Serverless v2 reader DB instances that scale independently from the writer. If so, choose a minimum capacity setting that's high enough that when the writer DB instance is busy with a write-intensive workload, the reader DB instances can apply the changes from the writer without falling behind. If you observe replica lag in readers that are in promotion tiers 2–15, consider increasing the minimum capacity setting for your cluster. For details on choosing whether reader DB instances scale along with the writer or independently, see Choosing the promotion tier for an Aurora Serverless v2 reader.
- If you have a DB cluster with Aurora Serverless v2 reader DB instances, the readers don't scale along with the writer DB instance when the promotion tier of the readers isn't 0 or 1. In that case, setting a low minimum capacity can result in excessive replication lag. That's because the readers might not have enough capacity to apply changes from the writer when the database is busy. We recommend that you set the minimum capacity to a value that represents a comparable amount of memory and CPU to the writer DB instance.
- The value of the `max_connections` parameter for Aurora Serverless v2DB instances is based on the memory size derived from the maximum ACUs. However, when you specify a minimum capacity of 0.5 ACUs on PostgreSQL-compatible DB instances, the maximum value of `max_connections` is capped at 2,000.

  If you intend to use the Aurora PostgreSQL cluster for a high-connection workload, consider using a minimum ACU setting of 1 or higher. For details about how Aurora Serverless v2 handles the `max_connections` configuration parameter, see Maximum connections for Aurora Serverless v2.
- The time it takes for an Aurora Serverless v2 DB instance to scale from its minimum capacity to its maximum capacity depends on the difference between its minimum and maximum ACU values. When the current capacity of the DB instance is large, Aurora Serverless v2 scales up in larger increments than when the DB instance starts from a small capacity. Thus, if you specify a relatively large maximum capacity and the DB instance spends most of its time near that capacity, consider increasing the minimum ACU setting. That way, an idle DB instance can scale back up to maximum capacity more quickly.

Choosing the maximum Aurora Serverless v2 capacity setting for a cluster

It's tempting to always choose some high value for the maximum Aurora Serverless v2 capacity setting. A large maximum capacity allows the DB instance to scale up the most when it's running an intensive workload. A low value avoids the possibility of unexpected charges. Depending on how you use that cluster and the other settings that you configure, the most effective value might be higher or lower than you originally thought. Consider the following factors when choosing the maximum capacity setting:

- The maximum capacity must be at least as high as the minimum capacity. You can set the minimum and maximum capacity to be identical. However, in that case the capacity never scales up or down. Thus, using identical values for minimum and maximum capacity isn't appropriate outside of testing situations.
- The maximum capacity must be higher than 0.5 ACUs. You can set the minimum and maximum capacity to be the same in most cases. However, you can't specify 0.5 for both the minimum and maximum. Use a value of 1 or higher for the maximum capacity.
- If you typically modify the DB instance class of your DB instances in anticipation of especially high or low workload, you can use that experience to estimate the equivalent Aurora Serverless v2 capacity range. To determine the memory size to use in times of high traffic, consult Hardware specifications for DB instance classes for Aurora.

  For example, suppose that you use the db.r6g.4xlarge DB instance class when your cluster has a high workload. That DB instance class has 128 GiB of memory. Thus, you can specify a maximum ACU setting of 64 to set up an Aurora Serverless v2 DB instance that can scale up to approximately that same capacity. That's because each ACU corresponds to approximately 2 GiB of memory. You might specify a somewhat higher value to let the DB instance scale up farther in case your db.r6g.4xlarge DB instance sometimes doesn't have enough capacity to handle the workload effectively.
- If you have a budgetary cap on your database usage, choose a value that stays within that cap even if all your Aurora Serverless v2 DB instances run at maximum capacity all the time. Remember that when you have *n* Aurora Serverless v2 DB instances in your cluster, the theoretical maximum Aurora Serverless v2 capacity that the cluster can consume at any moment is *n* times the maximum ACU setting for the cluster. (The actual amount consumed might be less, for example if some readers scale independently from the writer.)
- If you make use of Aurora Serverless v2 reader DB instances to offload some of the read-only workload from the writer DB instance, you might be able to choose a lower maximum capacity

setting. You do this to reflect that each reader DB instance doesn't need to scale as high as if the cluster contains only a single DB instance.

- Suppose that you want to protect against excessive usage due to misconfigured database parameters or inefficient queries in your application. In that case, you might avoid accidental overuse by choosing a maximum capacity setting that's lower than the absolute highest that you could set.
- If spikes due to real user activity are rare but do happen, you can take those occasions into account when choosing the maximum capacity setting. If the priority is for the application to keep running with full performance and scalability, you can specify a maximum capacity setting that's higher than you observe in normal usage. If it's OK for the application to run with reduced throughput during very extreme spikes in activity, you can choose a slightly lower maximum capacity setting. Make sure that you choose a setting that still has enough memory and CPU resources to keep the application running.
- If you turn on settings in your cluster that increase the memory usage for each DB instance, take that memory into account when deciding on the maximum ACU value. Such settings include those for Performance Insights, Aurora MySQL parallel queries, Aurora MySQL performance schema, and Aurora MySQL binary log replication. Make sure that the maximum ACU value allows the Aurora Serverless v2 DB instances to scale up enough to handle the workload when those feature are being used. For information about troubleshooting problems caused by the combination of a low maximum ACU setting and Aurora features that impose memory overhead, see Avoiding out-of-memory errors.

Example: Change the Aurora Serverless v2 capacity range of an Aurora MySQL cluster

The following AWS CLI example shows how to update the ACU range for Aurora Serverless v2 DB instances in an existing Aurora MySQL cluster. Initially, the capacity range for the cluster is 8–32 ACUs.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query 'DBClusters[*].ServerlessV2ScalingConfiguration|[0]'
{
    "MinCapacity": 8.0,
    "MaxCapacity": 32.0
}
```

The DB instance is idle and scaled down to 8 ACUs. The following capacity-related settings apply to the DB instance at this point. To represent the size of the buffer pool in easily readable units, we divide it by 2 to the power of 30, yielding a measurement in gibibytes (GiB). That's because memory-related measurements for Aurora use units based on powers of 2, not powers of 10.

```
mysql> select @@max_connections;
+-------------------+
| @@max_connections |
+-------------------+
|              3000 |
+-------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|                9294577664 |
+---------------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_buffer_pool_size / pow(2,30) as gibibytes;
+-----------+
| gibibytes |
+-----------+
|   8.65625 |
+-----------+
1 row in set (0.00 sec)
```

Next, we change the capacity range for the cluster. After the `modify-db-cluster` command finishes, the ACU range for the cluster is 12.5–80.

```
aws rds modify-db-cluster --db-cluster-identifier serverless-v2-cluster \
  --serverless-v2-scaling-configuration MinCapacity=12.5,MaxCapacity=80

aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query 'DBClusters[*].ServerlessV2ScalingConfiguration|[0]'
{
    "MinCapacity": 12.5,
    "MaxCapacity": 80.0
}
```

Changing the capacity range caused changes to the default values of some configuration parameters. Aurora can apply some of those new defaults immediately. However, some of the parameter changes take effect only after a reboot. The `pending-reboot` status indicates that a reboot is needed to apply some parameter changes.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query '*[].{DBClusterMembers:DBClusterMembers[*].
{DBInstanceIdentifier:DBInstanceIdentifier,DBClusterParameterGroupStatus:DBClusterParameterGrou
[0]'
{
    "DBClusterMembers": [
        {
            "DBInstanceIdentifier": "serverless-v2-instance-1",
            "DBClusterParameterGroupStatus": "pending-reboot"
        }
    ]
}
```

At this point, the cluster is idle and the DB instance `serverless-v2-instance-1` is consuming 12.5 ACUs. The `innodb_buffer_pool_size` parameter is already adjusted based on the current capacity of the DB instance. The `max_connections` parameter still reflects the value from the former maximum capacity. Resetting that value requires rebooting the DB instance.

Note

If you set the `max_connections` parameter directly in a custom DB parameter group, no reboot is required.

```
mysql> select @@max_connections;
+-------------------+
| @@max_connections |
+-------------------+
|              3000 |
+-------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|              15572402176 |
+--------------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_buffer_pool_size / pow(2,30) as gibibytes;
+---------------+
| gibibytes     |
+---------------+
| 14.5029296875 |
+---------------+
1 row in set (0.00 sec)
```

Now we reboot the DB instance and wait for it to become available again.

```
aws rds reboot-db-instance --db-instance-identifier serverless-v2-instance-1
{
  "DBInstanceIdentifier": "serverless-v2-instance-1",
  "DBInstanceStatus": "rebooting"
}

aws rds wait db-instance-available --db-instance-identifier serverless-v2-instance-1
```

The `pending-reboot` status is cleared. The value `in-sync` confirms that Aurora has applied all the pending parameter changes.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query '*[].{DBClusterMembers:DBClusterMembers[*].
{DBInstanceIdentifier:DBInstanceIdentifier,DBClusterParameterGroupStatus:DBClusterParameterGrou
[0]'
{
    "DBClusterMembers": [
        {
            "DBInstanceIdentifier": "serverless-v2-instance-1",
            "DBClusterParameterGroupStatus": "in-sync"
        }
    ]
}
```

The `innodb_buffer_pool_size` parameter has increased to its final size for an idle DB instance. The `max_connections` parameter has increased to reflect a value derived from the maximum ACU value. The formula that Aurora uses for `max_connections` causes an increase of 1,000 when the memory size doubles.

```
mysql> select @@innodb_buffer_pool_size;
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|               16139681792 |
+---------------------------+
1 row in set (0.00 sec)

mysql> select @@innodb_buffer_pool_size / pow(2,30) as gibibytes;
+-----------+
| gibibytes |
+-----------+
|  15.03125 |
+-----------+
1 row in set (0.00 sec)

mysql> select @@max_connections;
+-------------------+
| @@max_connections |
+-------------------+
|              4000 |
+-------------------+
1 row in set (0.00 sec)
```

We set the capacity range to 0.5–128 ACUs, and reboot the DB instance. Now the idle DB instance has a buffer cache size that's less than 1 GiB, so we measure it in mebibytes (MiB). The `max_connections` value of 5000 is derived from the memory size of the maximum capacity setting.

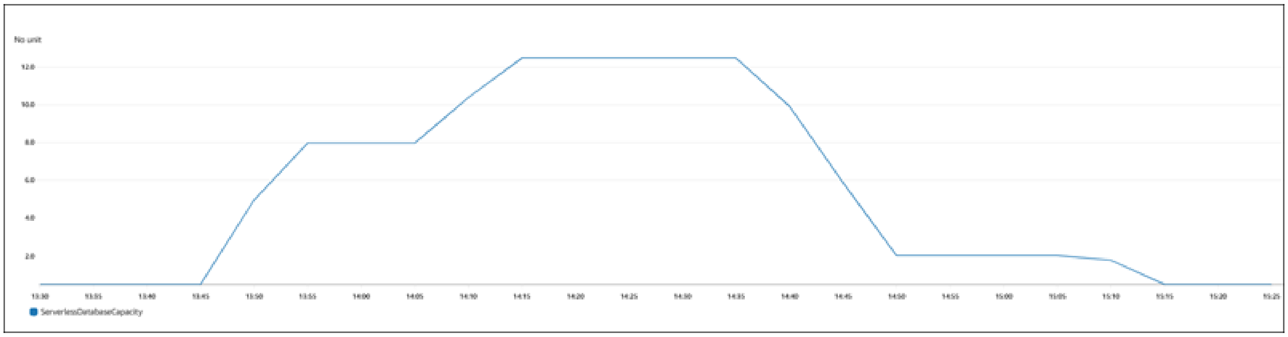
```
mysql> select @@innodb_buffer_pool_size / pow(2,20) as mebibytes, @@max_connections;
+-----------+-------------------+
| mebibytes | @@max_connections |
+-----------+-------------------+
|       672 |              5000 |
+-----------+-------------------+
1 row in set (0.00 sec)
```

Example: Change the Aurora Serverless v2 capacity range of an Aurora PostgreSQL cluster

The following CLI examples show how to update the ACU range for Aurora Serverless v2 DB instances in an existing Aurora PostgreSQL cluster.

1. The capacity range for the cluster starts at 0.5–1 ACU.
2. Change the capacity range to 8–32 ACUs.
3. Change the capacity range to 12.5–80 ACUs.
4. Change the capacity range to 0.5–128 ACUs.
5. Return the capacity to its initial range of 0.5–1 ACU.

The following figure shows the capacity changes in Amazon CloudWatch.



The DB instance is idle and scaled down to 0.5 ACUs. The following capacity-related settings apply to the DB instance at this point.

```
postgres=> show max_connections;
 max_connections
-----------------
 189
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 16384
(1 row)
```

Next, we change the capacity range for the cluster. After the `modify-db-cluster` command finishes, the ACU range for the cluster is 8.0–32.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query 'DBClusters[*].ServerlessV2ScalingConfiguration|[0]'
{
    "MinCapacity": 8.0,
    "MaxCapacity": 32.0
}
```

Changing the capacity range causes changes to the default values of some configuration parameters. Aurora can apply some of those new defaults immediately. However, some of the parameter changes take effect only after a reboot. The `pending-reboot` status indicates that you need a reboot to apply some parameter changes.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query '*[].{DBClusterMembers:DBClusterMembers[*].
{DBInstanceIdentifier:DBInstanceIdentifier,DBClusterParameterGroupStatus:DBClusterParameterGrou
[0]'
{
    "DBClusterMembers": [
        {
            "DBInstanceIdentifier": "serverless-v2-instance-1",
            "DBClusterParameterGroupStatus": "pending-reboot"
        }
    ]
}
```

At this point, the cluster is idle and the DB instance `serverless-v2-instance-1` is consuming 8.0 ACUs. The `shared_buffers` parameter is already adjusted based on the current capacity of the DB instance. The `max_connections` parameter still reflects the value from the former maximum capacity. Resetting that value requires rebooting the DB instance.

Note

If you set the `max_connections` parameter directly in a custom DB parameter group, no reboot is required.

```
postgres=> show max_connections;
 max_connections
-----------------
```

```
 189
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 1425408
(1 row)
```

We reboot the DB instance and wait for it to become available again.

```
aws rds reboot-db-instance --db-instance-identifier serverless-v2-instance-1
{
  "DBInstanceIdentifier": "serverless-v2-instance-1",
  "DBInstanceStatus": "rebooting"
}

aws rds wait db-instance-available --db-instance-identifier serverless-v2-instance-1
```

Now that the DB instance is rebooted, the `pending-reboot` status is cleared. The value `in-sync` confirms that Aurora has applied all the pending parameter changes.

```
aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query '*[].{DBClusterMembers:DBClusterMembers[*].
{DBInstanceIdentifier:DBInstanceIdentifier,DBClusterParameterGroupStatus:DBClusterParameterGrou
[0]'
{
    "DBClusterMembers": [
        {
            "DBInstanceIdentifier": "serverless-v2-instance-1",
            "DBClusterParameterGroupStatus": "in-sync"
        }
    ]
}
```

After rebooting, `max_connections` shows the value from the new maximum capacity.

```
postgres=> show max_connections;
 max_connections
-----------------
 5000
(1 row)
```

```

postgres=> show shared_buffers;
 shared_buffers
----------------
 1425408
(1 row)
```

Next, we change the capacity range for the cluster to 12.5–80 ACUs.

```
aws rds modify-db-cluster --db-cluster-identifier serverless-v2-cluster \
  --serverless-v2-scaling-configuration MinCapacity=12.5,MaxCapacity=80

aws rds describe-db-clusters --db-cluster-identifier serverless-v2-cluster \
  --query 'DBClusters[*].ServerlessV2ScalingConfiguration|[0]'
{
    "MinCapacity": 12.5,
    "MaxCapacity": 80.0
}
```

At this point, the cluster is idle and the DB instance `serverless-v2-instance-1` is consuming 12.5 ACUs. The `shared_buffers` parameter is already adjusted based on the current capacity of the DB instance. The `max_connections` value is still 5000.

```
postgres=> show max_connections;
 max_connections
-----------------
 5000
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 2211840
(1 row)
```

We reboot again, but the parameter values stay the same. This is because `max_connections` has a maximum value of 5000 for an Aurora Serverless v2 DB cluster running Aurora PostgreSQL.

```
postgres=> show max_connections;
 max_connections
-----------------
```

```
 5000
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 2211840
(1 row)
```

Now we set the capacity range from 0.5 to 128 ACUs. The DB cluster scales down to 10 ACUs, then to 2. We reboot the DB instance.

```
postgres=> show max_connections;
 max_connections
-----------------
 2000
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 16384
(1 row)
```

The `max_connections` value for Aurora Serverless v2 DB instances is based on the memory size derived from the maximum ACUs. However, when you specify a minimum capacity of 0.5 ACUs on PostgreSQL-compatible DB instances, the maximum value of `max_connections` is capped at 2,000.

Now we return the capacity to its initial range of 0.5–1 ACU and reboot the DB instance. The `max_connections` parameter has returned to its original value.

```
postgres=> show max_connections;
 max_connections
-----------------
 189
(1 row)

postgres=> show shared_buffers;
 shared_buffers
----------------
 16384
```

```
(1 row)
```

Working with parameter groups for Aurora Serverless v2

When you create your Aurora Serverless v2 DB cluster, you choose a specific Aurora DB engine and an associated DB cluster parameter group. If you aren't familiar with how Aurora uses parameter groups to apply configuration settings consistently across clusters, see Working with parameter groups. All of those procedures for creating, modifying, applying, and other actions for parameter groups apply to Aurora Serverless v2.

The parameter group feature works generally the same between provisioned clusters and clusters containing Aurora Serverless v2 DB instances:

- The default parameter values for all DB instances in the cluster are defined by the cluster parameter group.
- You can override some parameters for specific DB instances by specifying a custom DB parameter group for those DB instances. You might do so during debugging or performance tuning for specific DB instances. For example, suppose that you have a cluster containing some Aurora Serverless v2 DB instances and some provisioned DB instances. In this case, you might specify some different parameters for the provisioned DB instances by using a custom DB parameter group.
- For Aurora Serverless v2, you can use all the parameters that have the value `provisioned` in the `SupportedEngineModes` attribute in the parameter group. In Aurora Serverless v1, you can only use the subset of parameters that have `serverless` in the `SupportedEngineModes` attribute.

Topics

Default parameter values

The crucial difference between provisioned DB instances and Aurora Serverless v2 DB instances is that Aurora overrides any custom parameter values for certain parameters that are related to

DB instance capacity. The custom parameter values still apply to any provisioned DB instances in your cluster. For more details about how Aurora Serverless v2 DB instances interpret the parameters from Aurora parameter groups, see Configuration parameters for Aurora clusters. For the specific parameters that Aurora Serverless v2 overrides, see Parameters that Aurora adjusts as Aurora Serverless v2 scales up and down and Parameters that Aurora computes based on Aurora Serverless v2 maximum capacity.

You can get a list of default values for the default parameter groups for the various Aurora DB engines by using the describe-db-cluster-parameters CLI command and querying the AWS Region. The following are values that you can use for the `--db-parameter-group-family` and `-db-parameter-group-name` options for engine versions that are compatible with Aurora Serverless v2.

Database engine and version	Parameter group family	Default parameter group name
Aurora MySQL version 3	`aurora-mysql8.0`	`default.aurora-mysql8.0`
Aurora PostgreSQL version 13.x	`aurora-postgresql13`	`default.aurora-postgresql13`
Aurora PostgreSQL version 14.x	`aurora-postgresql14`	`default.aurora-postgresql14`
Aurora PostgreSQL version 15.x	`aurora-postgresql15`	`default.aurora-postgresql15`
Aurora PostgreSQL version 16.x	`aurora-postgresql16`	`default.aurora-postgresql16`

The following example gets a list of parameters from the default DB cluster group for Aurora MySQL version 3 and Aurora PostgreSQL 13. Those are the Aurora MySQL and Aurora PostgreSQL versions that you use with Aurora Serverless v2.

For Linux, macOS, or Unix:

```
aws rds describe-db-cluster-parameters \
  --db-cluster-parameter-group-name default.aurora-mysql8.0 \
  --query 'Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} |
    [?contains(SupportedEngineModes, `provisioned`) == `true`] | [*].[ParameterName]' \
  --output text

aws rds describe-db-cluster-parameters \
  --db-cluster-parameter-group-name default.aurora-postgresql13 \
  --query 'Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} |
    [?contains(SupportedEngineModes, `provisioned`) == `true`] | [*].[ParameterName]' \
  --output text
```

For Windows:

```
aws rds describe-db-cluster-parameters ^
  --db-cluster-parameter-group-name default.aurora-mysql8.0 ^
  --query 'Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} |
    [?contains(SupportedEngineModes, `provisioned`) == `true`] | [*].[ParameterName]' ^
  --output text

aws rds describe-db-cluster-parameters ^
  --db-cluster-parameter-group-name default.aurora-postgresql13 ^
  --query 'Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} |
    [?contains(SupportedEngineModes, `provisioned`) == `true`] | [*].[ParameterName]' ^
  --output text
```

Maximum connections for Aurora Serverless v2

For both Aurora MySQL and Aurora PostgreSQL, Aurora Serverless v2 DB instances hold the `max_connections` parameter constant so that connections aren't dropped when the DB instance scales down. The default value for this parameter is derived from a formula based on the memory size of the DB instance. For details about the formula and the default values for provisioned DB instance classes, see Maximum connections to an Aurora MySQL DB instance and Maximum connections to an Aurora PostgreSQL DB instance.

When Aurora Serverless v2 evaluates the formula, it uses the memory size based on the maximum Aurora capacity units (ACUs) for the DB instance, not the current ACU value. If you change the default value, we recommend using a variation of the formula instead of specifying a constant value. That way, Aurora Serverless v2 can use an appropriate setting based on the maximum capacity.

When you change the maximum capacity of an Aurora Serverless v2 DB cluster, you have to reboot the Aurora Serverless v2 DB instances to update the `max_connections` value. This is because `max_connections` is a static parameter for Aurora Serverless v2.

The following table shows the default values for `max_connections` for Aurora Serverless v2 based on the maximum ACU value.

Maximum ACUs	Default maximum connections on Aurora MySQL	Default maximum connections on Aurora PostgreSQL
1	90	189
4	135	823
8	1,000	1,669
16	2,000	3,360
32	3,000	5,000
64	4,000	5,000
128	5,000	5,000

Note

The `max_connections` value for Aurora Serverless v2DB instances is based on the memory size derived from the maximum ACUs. However, when you specify a minimum capacity of 0.5 ACUs on PostgreSQL-compatible DB instances, the maximum value of `max_connections` is capped at 2,000.

For specific examples showing how `max_connections` changes with the maximum ACU value, see Example: Change the Aurora Serverless v2 capacity range of an Aurora MySQL cluster and Example: Change the Aurora Serverless v2 capacity range of an Aurora PostgreSQL cluster.

Parameters that Aurora adjusts as Aurora Serverless v2 scales up and down

During autoscaling, Aurora Serverless v2 needs to be able to change parameters for each DB instance to work best for the increased or decreased capacity. Thus, you can't override some parameters related to capacity. For some parameters that you can override, avoid hardcoding fixed values. The following considerations apply to these settings that are related to capacity.

For Aurora MySQL, Aurora Serverless v2 resizes some parameters dynamically during scaling. For the following parameters, Aurora Serverless v2 doesn't use any custom parameter values that you specify:

- `innodb_buffer_pool_size`
- `innodb_purge_threads`
- `table_definition_cache`
- `table_open_cache`

For Aurora PostgreSQL, Aurora Serverless v2 resizes the following parameter dynamically during scaling. For the following parameters, Aurora Serverless v2 doesn't use any custom parameter values that you specify:

- `shared_buffers`

For all parameters other than those listed here, Aurora Serverless v2 DB instances work the same as provisioned DB instances. The default parameter value is inherited from the cluster parameter group. You can modify the default for the whole cluster by using a custom cluster parameter group. Or you can modify the default for certain DB instances by using a custom DB parameter group. Dynamic parameters are updated immediately. Changes to static parameters only take effect after you reboot the DB instance.

Parameters that Aurora computes based on Aurora Serverless v2 maximum capacity

For the following parameters, Aurora PostgreSQL uses default values that are derived from the memory size based on the maximum ACU setting, the same as with `max_connections`:

- `autovacuum_max_workers`
- `autovacuum_vacuum_cost_limit`

- `autovacuum_work_mem`
- `effective_cache_size`
- `maintenance_work_mem`

Avoiding out-of-memory errors

If one of your Aurora Serverless v2 DB instances consistently reaches the limit of its maximum capacity, Aurora indicates this condition by setting the DB instance to a status of `incompatible-parameters`. While the DB instance has the `incompatible-parameters` status, some operations are blocked. For example, you can't upgrade the engine version.

Typically, your DB instance goes into this status when it restarts frequently due to out-of-memory errors. Aurora records an event when this type of restart happens. You can view the event by following the procedure in Viewing Amazon RDS events. Unusually high memory usage can happen because of overhead from turning on settings such as Performance Insights and IAM authentication. It can also come from a heavy workload on your DB instance or from managing the metadata associated with a large number of schema objects.

If the memory pressure becomes lower so that the DB instance doesn't reach its maximum capacity very often, Aurora automatically changes the DB instance status back to `available`.

To recover from this condition, you can take some or all of the following actions:

- Increase the lower limit on capacity for Aurora Serverless v2 DB instances by changing the minimum Aurora capacity unit (ACU) value for the cluster. Doing so avoids issues where an idle database scales down to a capacity with less memory than is needed for the features that are turned on in your cluster. After changing the ACU settings for the cluster, reboot the Aurora Serverless v2 DB instance. Doing so evaluates whether Aurora can reset the status back to `available`.
- Increase the upper limit on capacity for Aurora Serverless v2 DB instances by changing the maximum ACU value for the cluster. Doing so avoids issues where a busy database can't scale up to a capacity with enough memory for the features that are turned on in your cluster and the database workload. After changing the ACU settings for the cluster, reboot the Aurora Serverless v2 DB instance. Doing so evaluates whether Aurora can reset the status back to `available`.
- Turn off configuration settings that require memory overhead. For example, suppose that you have features such as AWS Identity and Access Management (IAM), Performance Insights, or Aurora MySQL binary log replication turned on but don't use them. If so, you can turn them off.

Or you can adjust the minimum and maximum capacity values for the cluster higher to account for the memory used by those features. For guidelines about choosing minimum and maximum capacity settings, see Choosing the Aurora Serverless v2 capacity range for an Aurora cluster.

- Reduce the workload on the DB instance. For example, you can add reader DB instances to the cluster to spread the load from read-only queries across more DB instances.
- Tune the SQL code used by your application to use fewer resources. For example, you can examine your query plans, check the slow query log, or adjust the indexes on your tables. You can also perform other traditional kinds of SQL tuning.

Important Amazon CloudWatch metrics for Aurora Serverless v2

To get started with Amazon CloudWatch for your Aurora Serverless v2 DB instance, see Viewing Aurora Serverless v2 logs in Amazon CloudWatch. To learn more about how to monitor Aurora DB clusters through CloudWatch, see Monitoring log events in Amazon CloudWatch.

You can view your Aurora Serverless v2 DB instances in CloudWatch to monitor the capacity consumed by each DB instance with the `ServerlessDatabaseCapacity` metric. You can also monitor all of the standard Aurora CloudWatch metrics, such as `DatabaseConnections` and `Queries`. For the full list of CloudWatch metrics that you can monitor for Aurora, see Amazon CloudWatch metrics for Amazon Aurora. The metrics are divided into cluster-level and instance-level metrics, in Cluster-level metrics for Amazon Aurora and Instance-level metrics for Amazon Aurora.

The following CloudWatch instance-level metrics are important to monitor for you to understand how your Aurora Serverless v2 DB instances are scaling up and down. All of these metrics are calculated every second. That way, you can monitor the current status of your Aurora Serverless v2 DB instances. You can set alarms to notify you if any Aurora Serverless v2 DB instance approaches a threshold for metrics related to capacity. You can determine if the minimum and maximum capacity settings are appropriate, or if you need to adjust them. You can determine where to focus your efforts for optimizing the efficiency of your database.

- `ServerlessDatabaseCapacity`. As an instance-level metric, it reports the number of ACUs represented by the current DB instance capacity. As a cluster-level metric, it represents the average of the `ServerlessDatabaseCapacity` values of all the Aurora Serverless v2 DB instances in the cluster. This metric is only a cluster-level metric in Aurora Serverless v1. In Aurora Serverless v2, it's available at the DB instance level and at the cluster level.

- `ACUUtilization`. This metric is new in Aurora Serverless v2. This value is represented as a percentage. It's calculated as the value of the `ServerlessDatabaseCapacity` metric divided by the maximum ACU value of the DB cluster. Consider the following guidelines to interpret this metric and take action:
  - If this metric approaches a value of `100.0`, the DB instance has scaled up as high as it can. Consider increasing the maximum ACU setting for the cluster. That way, both writer and reader DB instances can scale to a higher capacity.
  - Suppose that a read-only workload causes a reader DB instance to approach an `ACUUtilization` of `100.0`, while the writer DB instance isn't close to its maximum capacity. In that case, consider adding additional reader DB instances to the cluster. That way, you can spread the read-only part of the workload spread across more DB instances, reducing the load on each reader DB instance.
  - Suppose that you are running a production application, where performance and scalability are the primary considerations. In that case, you can set the maximum ACU value for the cluster to a high number. Your goal is for the `ACUUtilization` metric to always be below `100.0`. With a high maximum ACU value, you can be confident that there's enough room in case there are unexpected spikes in database activity. You are only charged for the database capacity that's actually consumed.
- `CPUUtilization`. This metric is interpreted differently in Aurora Serverless v2 than in provisioned DB instances. For Aurora Serverless v2, this value is a percentage that's calculated as the amount of CPU currently being used divided by the CPU capacity that's available under the maximum ACU value of the DB cluster. Aurora monitors this value automatically and scales up your Aurora Serverless v2 DB instance when the DB instance consistently uses a high proportion of its CPU capacity.

  If this metric approaches a value of `100.0`, the DB instance has reached its maximum CPU capacity. Consider increasing the maximum ACU setting for the cluster. If this metric approaches a value of `100.0` on a reader DB instance, consider adding additional reader DB instances to the cluster. That way, you can spread the read-only part of the workload spread across more DB instances, reducing the load on each reader DB instance.
- `FreeableMemory`. This value represents the amount of unused memory that is available when the Aurora Serverless v2 DB instance is scaled to its maximum capacity. For every ACU that the current capacity is below the maximum capacity, this value increases by approximately 2 GiB. Thus, this metric doesn't approach zero until the DB instance is scaled up as high as it can.

If this metric approaches a value of `0`, the DB instance has scaled up as much as it can and is nearing the limit of its available memory. Consider increasing the maximum ACU setting for the cluster. If this metric approaches a value of `0` on a reader DB instance, consider adding additional reader DB instances to the cluster. That way, the read-only part of the workload can be spread across more DB instances, reducing the memory usage on each reader DB instance.

- `TempStorageIops`. The number of IOPS done on local storage attached to the DB instance. It includes the IOPS for both reads and writes. This metric represents a count and is measured once per second. This is a new metric for Aurora Serverless v2. For details, see Instance-level metrics for Amazon Aurora.
- `TempStorageThroughput`. The amount of data transferred to and from local storage associated with the DB instance. This metric represents bytes and is measured once per second. This is a new metric for Aurora Serverless v2. For details, see Instance-level metrics for Amazon Aurora.

Typically, most scaling up for Aurora Serverless v2 DB instances is caused by memory usage and CPU activity. The `TempStorageIops` and `TempStorageThroughput` metrics can help you to diagnose the rare cases where network activity for transfers between your DB instance and local storage devices is responsible for unexpected capacity increases. To monitor other network activity, you can use these existing metrics:

- `NetworkReceiveThroughput`
- `NetworkThroughput`
- `NetworkTransmitThroughput`
- `StorageNetworkReceiveThroughput`
- `StorageNetworkThroughput`
- `StorageNetworkTransmitThroughput`

You can have Aurora publish some or all database logs to CloudWatch. You select the logs to publish by turning on the configuration parameters such as `general_log` and `slow_query_log` in the DB cluster parameter group associated with the cluster that contains your Aurora Serverless v2 DB instances. When you turn off a log configuration parameter, publishing that log to CloudWatch stops. You can also delete the logs in CloudWatch if they are no longer needed.

How Aurora Serverless v2 metrics apply to your AWS bill

The Aurora Serverless v2 charges on your AWS bill are calculated based on the same `ServerlessDatabaseCapacity` metric that you can monitor. The billing mechanism can differ from the computed CloudWatch average for this metric in cases where you use Aurora Serverless v2 capacity for only part of an hour. It can also differ if system issues make the CloudWatch metric unavailable for brief periods. Thus, you might see a slightly different value of ACU-hours on your bill than if you compute the number yourself from the `ServerlessDatabaseCapacity` average value.

Examples of CloudWatch commands for Aurora Serverless v2 metrics

The following AWS CLI examples demonstrate how you can monitor the most important CloudWatch metrics related to Aurora Serverless v2. In each case, replace the `Value=` string for the `--dimensions` parameter with the identifier of your own Aurora Serverless v2 DB instance.

The following Linux example displays the minimum, maximum, and average capacity values for a DB instance, measured every 10 minutes over one hour. The Linux `date` command specifies the start and end times relative to the current date and time. The `sort_by` function in the `--query` parameter sorts the results chronologically based on the `Timestamp` field.

```
aws cloudwatch get-metric-statistics --metric-name "ServerlessDatabaseCapacity" \
  --start-time "$(date -d '1 hour ago')" --end-time "$(date -d 'now')" --period 600 \
  --namespace "AWS/RDS" --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=my_instance \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' --output table
```

The following Linux examples demonstrate monitoring the capacity of each DB instance in a cluster. They measure the minimum, maximum, and average capacity utilization of each DB instance. The measurements are taken once each hour over a three-hour period. These examples use the `ACUUtilization` metric representing a percentage of the upper limit on ACUs, instead of `ServerlessDatabaseCapacity` representing a fixed number of ACUs. That way, you don't need to know the actual numbers for the minimum and maximum ACU values in the capacity range. You can see percentages ranging from 0 to 100.

```
aws cloudwatch get-metric-statistics --metric-name "ACUUtilization" \
  --start-time "$(date -d '3 hours ago')" --end-time "$(date -d 'now')" --period 3600 \
  --namespace "AWS/RDS" --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=my_writer_instance \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' --output table

aws cloudwatch get-metric-statistics --metric-name "ACUUtilization" \
  --start-time "$(date -d '3 hours ago')" --end-time "$(date -d 'now')" --period 3600 \
  --namespace "AWS/RDS" --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=my_reader_instance \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' --output table
```

The following Linux example does similar measurements as the previous ones. In this case, the measurements are for the `CPUUtilization` metric. The measurements are taken every 10 minutes over a 1-hour period. The numbers represent the percentage of available CPU used, based on the CPU resources available to the maximum capacity setting for the DB instance.

```
aws cloudwatch get-metric-statistics --metric-name "CPUUtilization" \
  --start-time "$(date -d '1 hour ago')" --end-time "$(date -d 'now')" --period 600 \
  --namespace "AWS/RDS" --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=my_instance \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' --output table
```

The following Linux example does similar measurements as the previous ones. In this case, the measurements are for the `FreeableMemory` metric. The measurements are taken every 10 minutes over a 1-hour period.

```
aws cloudwatch get-metric-statistics --metric-name "FreeableMemory" \
  --start-time "$(date -d '1 hour ago')" --end-time "$(date -d 'now')" --period 600 \
  --namespace "AWS/RDS" --statistics Minimum Maximum Average \
  --dimensions Name=DBInstanceIdentifier,Value=my_instance \
  --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' --output table
```

Monitoring Aurora Serverless v2 performance with Performance Insights

You can use Performance Insights to monitor the performance of Aurora Serverless v2 DB instances. For Performance Insights procedures, see Monitoring DB load with Performance Insights on Amazon Aurora.

The following new Performance Insights counters apply to Aurora Serverless v2 DB instances:

- `os.general.serverlessDatabaseCapacity` – The current capacity of the DB instance in ACUs. The value corresponds to the `ServerlessDatabaseCapacity` CloudWatch metric for the DB instance.
- `os.general.acuUtilization` – The percentage of current capacity out of the maximum configured capacity. The value corresponds to the `ACUUtilization` CloudWatch metric for the DB instance.
- `os.general.maxConfiguredAcu` – The maximum capacity that you configured for this Aurora Serverless v2 DB instance. It's measured in ACUs.
- `os.general.minConfiguredAcu` – The minimum capacity that you configured for this Aurora Serverless v2 DB instance. It's measured in ACUs

For the full list of Performance Insights counters, see Performance Insights counter metrics.

When vCPU values are shown for an Aurora Serverless v2 DB instance in Performance Insights, those values represent estimates based on the ACU value for the DB instance. At the default interval of one minute, any fractional vCPU values are rounded up to the nearest whole number. For longer intervals, the vCPU value shown is the average of the integer vCPU values for each minute.

Troubleshooting Aurora Serverless v2 capacity issues

In some cases, Aurora Serverless v2 doesn't scale down to the minimum capacity, even with no load on the database. This can happen for the following reasons:

- Certain features can increase resource usage and prevent the database from scaling down to minimum capacity. These features include the following:
  - Aurora global databases
  - Exporting CloudWatch Logs
  - Enabling `pg_audit` on Aurora PostgreSQL–compatible DB clusters
  - Enhanced Monitoring
  - Performance Insights

  For more information, see Choosing the minimum Aurora Serverless v2 capacity setting for a cluster.

- If a reader instance isn't scaling down to the minimum and stays at the same or higher capacity than the writer instance, then check the priority tier of the reader instance. Aurora Serverless v2 reader DB instances in tier 0 or 1 are kept at a minimum capacity at least as high as the writer DB instance. Change the priority tier of the reader to 2 or higher so that it scales up and down independently of the writer. For more information, see Choosing the promotion tier for an Aurora Serverless v2 reader.
- Set any database parameters that impact the size of shared memory to their default values. Setting a value higher than the default increases the shared memory requirement and prevents the database from scaling down to the minimum capacity. Examples are `max_connections` and `max_locks_per_transaction`.

  Note

  Updating shared memory parameters requires a database restart for the changes to take effect.
- Heavy database workloads can increase resource usage.
- Large database volumes can increase resource usage.

  Amazon Aurora uses memory and CPU resources for DB cluster management. Aurora requires more CPU and memory to manage DB clusters with larger database volumes. If your cluster’s minimum capacity is less than the minimum required for cluster management, your cluster won't scale down to the minimum capacity.
- Background processes, such as purge, can also increase resource usage.

If the database still doesn't scale down to the minimum capacity configured, then stop and restart the database to reclaim any memory fragments that might have built up over time. Stopping and starting a database results in downtime, so we recommend doing this sparingly.

Migrating to Aurora Serverless v2

To convert an existing DB cluster to use Aurora Serverless v2, you can do the following:

- Upgrade from a provisioned Aurora DB cluster.
- Upgrade from an Aurora Serverless v1 cluster.
- Migrate from an on-premises database to an Aurora Serverless v2 cluster.

When your upgraded cluster is running the appropriate engine version as listed in Requirements and limitations for Aurora Serverless v2, you can begin adding Aurora Serverless v2 DB instances to it. The first DB instance that you add to the upgraded cluster must be a provisioned DB instance. Then you can switch over the processing for the write workload, the read workload, or both to the Aurora Serverless v2 DB instances.

Contents

Note

These topics describe how to convert an existing DB cluster. For information on creating a new Aurora Serverless v2 DB cluster, see Creating a DB cluster that uses Aurora Serverless v2.

Upgrading or switching existing clusters to use Aurora Serverless v2

If your provisioned cluster has an engine version that supports Aurora Serverless v2, switching to Aurora Serverless v2 doesn't require an upgrade. In that case, you can add Aurora Serverless v2 DB instances to your original cluster. You can switch the cluster to use all Aurora Serverless v2 DB instances. You can also use a combination of Aurora Serverless v2 and provisioned DB instances in the same DB cluster. For the Aurora engine versions that support Aurora Serverless v2, see Supported Regions and Aurora DB engines for Aurora Serverless v2.

If you're running a lower engine version that doesn't support Aurora Serverless v2, you take these general steps:

1. Upgrade the cluster.
2. Create a provisioned writer DB instance for the upgraded cluster.
3. Modify the cluster to use Aurora Serverless v2 DB instances.

> ⚠ **Important**
>
> When you perform a major version upgrade to an Aurora Serverless v2-compatible version by using snapshot restore or cloning, the first DB instance that you add to the new cluster must be a provisioned DB instance. This addition starts the final stage of the upgrade process.
> Until that final stage happens, the cluster doesn't have the infrastructure that's required for Aurora Serverless v2 support. Thus, these upgraded clusters always start with a provisioned writer DB instance. Then you can convert or fail over the provisioned DB instance to an Aurora Serverless v2 one.

Upgrading from Aurora Serverless v1 to Aurora Serverless v2 involves creating a provisioned cluster as an intermediate step. Then you perform the same upgrade steps as when you start with a provisioned cluster.

Upgrade paths for MySQL-compatible clusters to use Aurora Serverless v2

If your original cluster is running Aurora MySQL, choose the appropriate procedure depending on the engine version and engine mode of your cluster.

If your original Aurora MySQL cluster is this	Do this to switch to Aurora Serverless v2
Provisioned cluster running Aurora MySQL version 3, compatible with MySQL 8.0	This is the final stage for all conversions from existing Aurora MySQL clusters. If necessary, perform a minor version upgrade to version 3.02.0 or higher. Use a provisioned

If your original Aurora MySQL cluster is this	Do this to switch to Aurora Serverless v2
	DB instance for the writer DB instance. Add one Aurora Serverless v2 reader DB instance. Perform a failover to make that the writer DB instance. (Optional) Convert other provisioned DB instances in the cluster to Aurora Serverles s v2. Or add new Aurora Serverless v2 DB instances and remove the provisioned DB instances. For the full procedure and examples, see Switching from a provisioned cluster to Aurora Serverless v2.
Provisioned cluster running Aurora MySQL version 2, compatible with MySQL 5.7	Perform a major version upgrade to Aurora MySQL version 3.02.0 or higher. Then follow the procedure for Aurora MySQL version 3 to switch the cluster to use Aurora Serverless v2 DB instances.
Aurora Serverless v1 cluster running Aurora MySQL version 2, compatible with MySQL 5.7	To help plan your conversion from Aurora Serverless v1, consult Comparison of Aurora Serverless v2 and Aurora Serverless v1 first. Then follow the procedure in Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2.

Upgrade paths for PostgreSQL-compatible clusters to use Aurora Serverless v2

If your original cluster is running Aurora PostgreSQL, choose the appropriate procedure depending on the engine version and engine mode of your cluster.

If your original Aurora PostgreSQL cluster is this	Do this to switch to Aurora Serverless v2
Provisioned cluster running Aurora PostgreSQL version 13	This is the final stage for all conversions from existing Aurora PostgreSQL clusters. If necessary, perform a minor version upgrade to version 13.6 or higher. Add one provisioned DB instance for the writer DB instance. Add one Aurora Serverless v2 reader DB instance. Perform a failover to make that Aurora Serverless v2 instance the writer DB instance. (Optional) Convert other provisioned DB instances in the cluster to Aurora Serverless v2. Or add new Aurora Serverless v2 DB instances and remove the provisioned DB instances. For the full procedure and examples, see Switching from a provisioned cluster to Aurora Serverless v2.
Provisioned cluster running Aurora PostgreSQL version 11 or 12	Perform a major version upgrade to Aurora PostgreSQL version 13.6 or higher. Then follow the procedure for Aurora PostgreSQL version 13 to switch the cluster to use Aurora Serverless v2 DB instances.
Aurora Serverless v1 cluster running Aurora PostgreSQL version 11 or 13	To help plan your conversion from Aurora Serverless v1, consult Comparison of Aurora Serverless v2 and Aurora Serverless v1 first. Then follow the procedure in Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2.

Switching from a provisioned cluster to Aurora Serverless v2

To switch a provisioned cluster to use Aurora Serverless v2, follow these steps:

1. Check if the provisioned cluster needs to be upgraded to be used with Aurora Serverless v2 DB instances. For the Aurora versions that are compatible with Aurora Serverless v2, see Requirements and limitations for Aurora Serverless v2.

   If the provisioned cluster is running an engine version that isn't available for Aurora Serverless v2, upgrade the engine version of the cluster:

   - If you have a MySQL 5.7–compatible provisioned cluster, follow the upgrade instructions for Aurora MySQL version 3. Use the procedures in How to perform an in-place upgrade.
   - If you have a PostgreSQL-compatible provisioned cluster running PostgreSQL version 11 or 12, follow the upgrade instructions for Aurora PostgreSQL version 13. Use the procedures in How to perform a major version upgrade.
2. Configure any other cluster properties to match the Aurora Serverless v2 requirements from Requirements and limitations for Aurora Serverless v2.
3. Configure the scaling configuration for the cluster. Follow the procedure in Setting the Aurora Serverless v2 capacity range for a cluster.
4. Add one or more Aurora Serverless v2 DB instances to the cluster. Follow the general procedure in Adding Aurora Replicas to a DB cluster. For each new DB instance, specify the special DB instance class name **Serverless** in the AWS Management Console, or `db.serverless` in the AWS CLI or Amazon RDS API.

   In some cases, you might already have one or more provisioned reader DB instances in the cluster. If so, you can convert one of the readers to an Aurora Serverless v2 DB instance instead of creating a new DB instance. To do so, follow the procedure in Converting a provisioned writer or reader to Aurora Serverless v2.
5. Perform a failover operation to make one of the Aurora Serverless v2 DB instances the writer DB instance for the cluster.
6. (Optional) Convert any provisioned DB instances to Aurora Serverless v2, or remove them from the cluster. Follow the general procedure in Converting a provisioned writer or reader to Aurora Serverless v2 or Deleting a DB instance from an Aurora DB cluster.

Tip

Removing the provisioned DB instances isn't mandatory. You can set up a cluster containing both Aurora Serverless v2 and provisioned DB instances. However, until you are familiar with the performance and scaling characteristics of Aurora Serverless v2 DB instances, we recommend that you configure your clusters with DB instances all of the same type.

The following AWS CLI example shows the switchover process using a provisioned cluster that's running Aurora MySQL version 3.02.0. The cluster is named `mysql-80`. The cluster starts with two provisioned DB instances named `provisioned-instance-1` and `provisioned-instance-2`, a writer and a reader. They both use the `db.r6g.large` DB instance class.

```
$ aws rds describe-db-clusters --db-cluster-identifier mysql-80 \
  --query '*[].[DBClusterIdentifier,DBClusterMembers[*].
[DBInstanceIdentifier,IsClusterWriter]]' --output text
mysql-80
provisioned-instance-2     False
provisioned-instance-1     True

$ aws rds describe-db-instances --db-instance-identifier provisioned-instance-1 \
  --output text --query '*[].[DBInstanceIdentifier,DBInstanceClass]'
provisioned-instance-1     db.r6g.large

$ aws rds describe-db-instances --db-instance-identifier provisioned-instance-2 \
  --output text --query '*[].[DBInstanceIdentifier,DBInstanceClass]'
provisioned-instance-2     db.r6g.large
```

We create a table with some data. That way, we can confirm that the data and operation of the cluster are the same before and after the switchover.

```
mysql> create database serverless_v2_demo;
mysql> create table serverless_v2_demo.demo (s varchar(128));
mysql> insert into serverless_v2_demo.demo values ('This cluster started with a
 provisioned writer.');
Query OK, 1 row affected (0.02 sec)
```

First, we add a capacity range to the cluster. Otherwise, we get an error when adding any Aurora Serverless v2 DB instances to the cluster. If we use the AWS Management Console for this procedure, that step is automatic when we add the first Aurora Serverless v2 DB instance.

```
$ aws rds create-db-instance --db-instance-identifier serverless-v2-instance-1 \
  --db-cluster-identifier mysql-80 --db-instance-class db.serverless --engine aurora-
mysql

An error occurred (InvalidDBClusterStateFault) when calling the CreateDBInstance
 operation:
Set the Serverless v2 scaling configuration on the parent DB cluster before creating a
 Serverless v2 DB instance.

$ # The blank ServerlessV2ScalingConfiguration attribute confirms that the cluster
 doesn't have a capacity range set yet.
$ aws rds describe-db-clusters --db-cluster-identifier mysql-80 --query
 'DBClusters[*].ServerlessV2ScalingConfiguration'
[]

$ aws rds modify-db-cluster --db-cluster-identifier mysql-80 \
  --serverless-v2-scaling-configuration MinCapacity=0.5,MaxCapacity=16
{
  "DBClusterIdentifier": "mysql-80",
  "ServerlessV2ScalingConfiguration": {
    "MinCapacity": 0.5,
    "MaxCapacity": 16
  }
}
```

We create two Aurora Serverless v2 readers to take the place of the original DB instances. We do so by specifying the `db.serverless` DB instance class for the new DB instances.

```
$ aws rds create-db-instance --db-instance-identifier serverless-v2-instance-1 --db-
cluster-identifier mysql-80 --db-instance-class db.serverless --engine aurora-mysql
{
  "DBInstanceIdentifier": "serverless-v2-instance-1",
  "DBClusterIdentifier": "mysql-80",
  "DBInstanceClass": "db.serverless",
  "DBInstanceStatus": "creating"
}

$ aws rds create-db-instance --db-instance-identifier serverless-v2-instance-2 \
  --db-cluster-identifier mysql-80 --db-instance-class db.serverless --engine aurora-
mysql
{
  "DBInstanceIdentifier": "serverless-v2-instance-2",
  "DBClusterIdentifier": "mysql-80",
  "DBInstanceClass": "db.serverless",
  "DBInstanceStatus": "creating"
}

$ # Wait for both DB instances to finish being created before proceeding.
$ aws rds wait db-instance-available --db-instance-identifier serverless-v2-instance-1
 && \
  aws rds wait db-instance-available --db-instance-identifier serverless-v2-instance-2
```

We perform a failover to make one of the Aurora Serverless v2 DB instances the new writer for the cluster.

```
$ aws rds failover-db-cluster --db-cluster-identifier mysql-80 \
  --target-db-instance-identifier serverless-v2-instance-1
{
  "DBClusterIdentifier": "mysql-80",
  "DBClusterMembers": [
    {
      "DBInstanceIdentifier": "serverless-v2-instance-1",
      "IsClusterWriter": false,
      "DBClusterParameterGroupStatus": "in-sync",
      "PromotionTier": 1
    },
    {
      "DBInstanceIdentifier": "serverless-v2-instance-2",
      "IsClusterWriter": false,
      "DBClusterParameterGroupStatus": "in-sync",
      "PromotionTier": 1
    },
    {
      "DBInstanceIdentifier": "provisioned-instance-2",
      "IsClusterWriter": false,
      "DBClusterParameterGroupStatus": "in-sync",
      "PromotionTier": 1
    },
    {
      "DBInstanceIdentifier": "provisioned-instance-1",
      "IsClusterWriter": true,
      "DBClusterParameterGroupStatus": "in-sync",
      "PromotionTier": 1
    }
  ],
  "Status": "available"
}
```

It takes a few seconds for that change to take effect. At that point, we have an Aurora Serverless v2 writer and an Aurora Serverless v2 reader. Thus, we don't need either of the original provisioned DB instances.

```
$ aws rds describe-db-clusters --db-cluster-identifier mysql-80 \
  --query '*[].[DBClusterIdentifier,DBClusterMembers[*].
[DBInstanceIdentifier,IsClusterWriter]]' \
  --output text
mysql-80
serverless-v2-instance-1        True
serverless-v2-instance-2        False
provisioned-instance-2          False
provisioned-instance-1          False
```

The last step in the switchover procedure is to delete both of the provisioned DB instances.

```
$ aws rds delete-db-instance --db-instance-identifier provisioned-instance-2 --skip-
final-snapshot
{
  "DBInstanceIdentifier": "provisioned-instance-2",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql",
  "EngineVersion": "8.0.mysql_aurora.3.02.0",
  "DBInstanceClass": "db.r6g.large"
}

$ aws rds delete-db-instance --db-instance-identifier provisioned-instance-1 --skip-
final-snapshot
{
  "DBInstanceIdentifier": "provisioned-instance-1",
  "DBInstanceStatus": "deleting",
  "Engine": "aurora-mysql",
  "EngineVersion": "8.0.mysql_aurora.3.02.0",
  "DBInstanceClass": "db.r6g.large"
}
```

As a final check, we confirm that the original table is accessible and writeable from the Aurora Serverless v2 writer DB instance.

```
mysql> select * from serverless_v2_demo.demo;
+---------------------------------------------------+
| s                                                 |
+---------------------------------------------------+
| This cluster started with a provisioned writer.  |
+---------------------------------------------------+
1 row in set (0.00 sec)

mysql> insert into serverless_v2_demo.demo values ('And it finished with a Serverless
 v2 writer.');
Query OK, 1 row affected (0.01 sec)

mysql> select * from serverless_v2_demo.demo;
+---------------------------------------------------+
| s                                                 |
+---------------------------------------------------+
| This cluster started with a provisioned writer.  |
| And it finished with a Serverless v2 writer.      |
+---------------------------------------------------+
2 rows in set (0.01 sec)
```

We also connect to the Aurora Serverless v2 reader DB instance and confirm that the newly written data is available there too.

```
mysql> select * from serverless_v2_demo.demo;
+---------------------------------------------------+
| s                                                 |
+---------------------------------------------------+
| This cluster started with a provisioned writer.  |
| And it finished with a Serverless v2 writer.      |
+---------------------------------------------------+
2 rows in set (0.01 sec)
```

Comparison of Aurora Serverless v2 and Aurora Serverless v1

If you are already using Aurora Serverless v1, you can learn the major differences between Aurora Serverless v1 and Aurora Serverless v2. The architectural differences, such as support for reader DB instances, open up new types of use cases.

You can use the following tables to help understand the most important differences between Aurora Serverless v2 and Aurora Serverless v1.

Topics

Comparison of Aurora Serverless v2 and Aurora Serverless v1 requirements

The following table summarizes the different requirements to run your database using Aurora Serverless v2 or Aurora Serverless v1. Aurora Serverless v2 offers higher versions of the Aurora MySQL and Aurora PostgreSQL DB engines than Aurora Serverless v1 does.

Feature	Aurora Serverless v2 requirement	Aurora Serverless v1 requirement
DB engines	Aurora MySQL, Aurora PostgreSQL	Aurora MySQL, Aurora PostgreSQL
Supported Aurora MySQL versions	See Aurora Serverless v2 with Aurora MySQL.	See Aurora Serverless v1 with Aurora MySQL.
Supported Aurora PostgreSQL versions	See Aurora Serverless v2 with Aurora PostgreSQL.	See Aurora Serverless v1 with Aurora PostgreSQL.
Upgrading a DB cluster	Similarly to provisioned DB clusters, you can perform upgrades manually without waiting for Aurora to upgrade the DB cluster for you. For more information, see Modifying an Amazon Aurora DB cluster.	Minor version upgrades are applied automatically as they become available. For more information, see Aurora Serverless v1 and Aurora database engine versions. You can perform major version upgrades manually. For more information,

Feature	Aurora Serverless v2 requirement	Aurora Serverless v1 requirement
	Note To perform a major version upgrade from 13.x to 14.x or 15.x for an Aurora PostgreSQL–compatible DB cluster, the maximum capacity of your cluster must be at least 2 ACUs.	see Modifying an Aurora Serverless v1 DB cluster.

Feature	Aurora Serverless v2 requirement	Aurora Serverless v1 requirement
Converting from provisioned DB cluster	You can use the following methods: • Add one or more Aurora Serverless v2 reader DB instances to an existing provisioned cluster. To use Aurora Serverless v2 for the writer, perform a failover to one of the Aurora Serverles s v2 DB instances. For the entire cluster to use Aurora Serverless v2 DB instances , remove any provisioned writer DB instances after promoting the Aurora Serverless v2 DB instance to the writer. • Create a new cluster with the appropriate DB engine and engine version. Use any of the standard methods. For example, restore a cluster snapshot or create a clone of an existing cluster. Choose Aurora Serverless v2 for some or all of the DB instances in the new cluster. If you create the new cluster through cloning, you can't upgrade the engine version at the same time. Make sure that the	Restore snapshot of provision ed cluster to create new Aurora Serverless v1 cluster.

Feature	Aurora Serverless v2 requirement	Aurora Serverless v1 requirement
	original cluster is already running an engine version that's compatible with Aurora Serverless v2.	
Converting from Aurora Serverless v1 cluster	Follow the procedure in Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2.	Not applicable
Available DB instance classes	The special DB instance class `db.serverless` . In the AWS Management Console, it's labeled as **Serverless**.	Not applicable. Aurora Serverless v1 uses the `serverless` engine mode.
Port	Any port that's compatible with MySQL or PostgreSQL	Default MySQL or PostgreSQL port only
Public IP address allowed?	Yes	No
Virtual private cloud (VPC) required?	Yes	Yes. Each Aurora Serverless v1 cluster consumes 2 interface and Gateway Load Balancer endpoints allocated to your VPC.

Comparison of Aurora Serverless v2 and Aurora Serverless v1 scaling and availability

The following table summarizes differences between Aurora Serverless v2 and Aurora Serverless v1 for scalability and availability.

Aurora Serverless v2 scaling is more responsive, more granular, and less disruptive than the scaling in Aurora Serverless v1. Aurora Serverless v2 can scale both by changing the size of the DB instance and by adding more DB instances to the DB cluster. It can also scale by adding clusters in other AWS Regions to an Aurora global database. In contrast, Aurora Serverless v1 only scales by increasing or decreasing the capacity of the writer. All the compute for an Aurora Serverless v1 cluster runs in a single Availability Zone and a single AWS Region.

Scaling and high availability feature	Aurora Serverless v2 behavior	Aurora Serverless v1 behavior
Minimum Aurora capacity units (ACUs) (Aurora MySQL)	0.5	1 when the cluster is running, 0 when the cluster is paused.
Minimum ACUs (Aurora PostgreSQL)	0.5	2 when the cluster is running, 0 when the cluster is paused.
Maximum ACUs (Aurora MySQL)	128	256
Maximum ACUs (Aurora PostgreSQL)	128	384
Stopping a cluster	You can manually stop and start the cluster by using the same cluster stop and start feature as provisioned clusters.	The cluster pauses automatically after a timeout. It takes some time to become available when activity resumes.
Scaling for DB instances	Scale up and down with minimum increment of 0.5 ACUs.	Scale up and down by doubling or halving the ACUs.
Number of DB instances	Same as a provisioned cluster: 1 writer DB instance, up to 15 reader DB instances.	1 DB instance handling both reads and writes.
Scaling can happen while SQL statements are running?	Yes. Aurora Serverless v2 doesn't require waiting for a quiet point.	No. For example, scaling waits for completion of long-running transactions, temporary tables, and table locks.

Scaling and high availability feature	Aurora Serverless v2 behavior	Aurora Serverless v1 behavior
Reader DB instances scale along with writer	Optional.	Not applicable.
Maximum storage	128 TiB	128 TiB or 64 TiB, depending on database engine and version.
Buffer cache preserved when scaling	Yes. Buffer cache is resized dynamically.	No. Buffer cache is rewarmed after scaling.
Failover	Yes, same as for provisioned clusters.	Best effort only, subject to capacity availability. Slower than in Aurora Serverless v2.
Multi-AZ capability	Yes, same as for provisioned. A Multi-AZ cluster requires a reader DB instance in a second Availability Zone (AZ). For a Multi-AZ cluster, Aurora performs Multi-AZ failover in case of an AZ failure.	Aurora Serverless v1 clusters run all their compute in a single AZ. Recovery in case of AZ failure is best effort only and subject to capacity availability.
Aurora global databases	Yes	No
Scaling based on memory pressure	Yes	No
Scaling based on CPU load	Yes	Yes

Scaling and high availability feature	Aurora Serverless v2 behavior	Aurora Serverless v1 behavior
Scaling based on network traffic	Yes, based on memory and CPU overhead of network traffic. The `max_connections` parameter remains constant to avoid dropping connections when scaling down.	Yes, based on number of connections.
Timeout action for scaling events	No	Yes
Adding new DB instances to cluster through AWS Auto Scaling	Not applicable. You can create Aurora Serverless v2 reader DB instances in promotion tiers 2–15 and leave them scaled down to low capacity.	No. Reader DB instances aren't available.

Comparison of Aurora Serverless v2 and Aurora Serverless v1 feature support

The following table summarizes these:

- Features that are available in Aurora Serverless v2 but not Aurora Serverless v1
- Features that work differently between Aurora Serverless v1 and Aurora Serverless v2
- Features that aren't currently available in Aurora Serverless v2

Aurora Serverless v2 includes many features from provisioned clusters that aren't available for Aurora Serverless v1.

Feature	Aurora Serverless v2 support	Aurora Serverless v1 support
Cluster topology	Aurora Serverless v2 is a property of individual DB instances. A cluster can	Aurora Serverless v1 clusters don't use the notion of DB instances. You can't change

Feature	Aurora Serverless v2 support	Aurora Serverless v1 support
	contain multiple Aurora Serverless v2 DB instances, or a combination of Aurora Serverless v2 and provisioned DB instances.	the Aurora Serverless v1 property after you create the cluster.
Configuration parameters	Almost all the same parameters can be modified as in provisioned clusters. For details, see Working with parameter groups for Aurora Serverless v2.	Only a subset of parameters can be modified.
Parameter groups	Cluster parameter group and DB parameter groups. Parameters with `provisioned` value in `SupportedEngineModes` attribute are available. That's many more parameters than in Aurora Serverless v1.	Cluster parameter group only. Parameters with `serverless` value in `SupportedEngineModes` attribute are available.
Encryption for cluster volume	Optional	Required. The limitations in Limitations of Amazon Aurora encrypted DB clusters apply to all Aurora Serverless v1 clusters.
Cross-Region snapshots	Yes	Snapshot must be encrypted with your own AWS Key Management Service (AWS KMS) key.
Automated backups retained after DB cluster deletion	Yes	No

Feature	Aurora Serverless v2 support	Aurora Serverless v1 support
TLS/SSL	Yes. The support is the same as for provisioned clusters. For usage information, see Using TLS/SSL with Aurora Serverless v2.	Yes. There are some differenc es from TLS support for provisioned clusters. For usage information, see Using TLS/SSL with Aurora Serverless v1.
Cloning	Only from and to DB engine versions that are compatibl e with Aurora Serverless v2. You can't use cloning to upgrade from Aurora Serverless v1 or from an earlier version of a provision ed cluster.	Only from and to DB engine versions that are compatible with Aurora Serverless v1.
Integration with Amazon S3	Yes	Yes
Integration with AWS Secrets Manager	No	No
Exporting DB cluster snapshots to S3	Yes	No
Associating an IAM role	Yes	No
Uploading logs to Amazon CloudWatch	Optional. You choose which logs to turn on and which logs to upload to CloudWatc h.	All logs that are turned on are uploaded to CloudWatch automatically.
Data API available	Yes	Yes
Query editor available	Yes	Yes
Performance Insights	Yes	No

Feature	Aurora Serverless v2 support	Aurora Serverless v1 support
Amazon RDS Proxy available	Yes	No
Babelfish for Aurora PostgreSQL available	Yes. Supported for Aurora PostgreSQL versions that are compatible with both Babelfish and Aurora Serverless v2.	No

Adapting Aurora Serverless v1 use cases to Aurora Serverless v2

Depending on your use case for Aurora Serverless v1, you might adapt that approach to take advantage of Aurora Serverless v2 features as follows.

Suppose that you have an Aurora Serverless v1 cluster that is lightly loaded and your priority is maintaining continuous availability while minimizing costs. With Aurora Serverless v2, you can configure a smaller minimum ACU setting of 0.5, compared with a minimum of 1 ACU for Aurora Serverless v1. You can increase availability by creating a Multi-AZ configuration, with the reader DB instance also having a minimum of 0.5 ACUs.

Suppose that you have an Aurora Serverless v1 cluster that you use in a development and test scenario. In this case, cost is also a high priority but the cluster doesn't need to be available at all times. Currently, Aurora Serverless v2 doesn't automatically pause when the cluster is completely idle. Instead, you can manually stop the cluster when it's not needed, and start it when it's time for the next test or development cycle.

Suppose that you have an Aurora Serverless v1 cluster with a heavy workload. An equivalent cluster using Aurora Serverless v2 can scale with more granularity. For example, Aurora Serverless v1 scales by doubling the capacity, for example from 64 to 128 ACUs. In contrast, your Aurora Serverless v2 DB instance can scale to a value somewhere between those numbers.

Suppose that your workload requires a higher total capacity than is available in Aurora Serverless v1. You can use multiple Aurora Serverless v2 reader DB instances to offload the read-intensive parts of the workload from the writer DB instance. You can also divide the read-intensive workload among multiple reader DB instances.

For a write-intensive workload, you might configure the cluster with a large provisioned DB instance as the writer. You might do so alongside one or more Aurora Serverless v2 reader DB instances.

Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2

The process of upgrading a DB cluster from Aurora Serverless v1 to Aurora Serverless v2 has multiple steps. That's because you can't convert directly from Aurora Serverless v1 to Aurora Serverless v2. There's always an intermediate step that involves converting the Aurora Serverless v1 DB cluster to a provisioned cluster.

Aurora MySQL–compatible DB clusters

You can convert your Aurora Serverless v1 DB cluster to a provisioned DB cluster, then use a blue/green deployment to upgrade it and convert it to an Aurora Serverless v2 DB cluster. We recommend this procedure for production environments. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.

To use a blue/green deployment to upgrade an Aurora Serverless v1 cluster running Aurora MySQL version 2 (MySQL 5.7–compatible)

1. Convert the Aurora Serverless v1 DB cluster to a provisioned Aurora MySQL version 2 cluster. Follow the procedure in Converting from Aurora Serverless v1 to provisioned.
2. Create a blue/green deployment. Follow the procedure in Creating a blue/green deployment.
3. Choose an Aurora MySQL version for the green cluster that's compatible with Aurora Serverless v2, for example 3.04.1.

   For compatible versions, see Aurora Serverless v2 with Aurora MySQL.
4. Modify the writer DB instance of the green cluster to use the **Serverless v2** (db.serverless) DB instance class.

   For details, see Converting a provisioned writer or reader to Aurora Serverless v2.
5. When your upgraded Aurora Serverless v2 DB cluster is available, switch over from the blue cluster to the green cluster.

Aurora PostgreSQL–compatible DB clusters

You can convert your Aurora Serverless v1 DB cluster to a provisioned DB cluster, then use a blue/green deployment to upgrade it and convert it to an Aurora Serverless v2 DB cluster. We recommend this procedure for production environments. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.

To use a blue/green deployment to upgrade an Aurora Serverless v1 cluster running Aurora PostgreSQL version 11

1. Convert the Aurora Serverless v1 DB cluster to a provisioned Aurora PostgreSQL cluster. Follow the procedure in Converting from Aurora Serverless v1 to provisioned.
2. Create a blue/green deployment. Follow the procedure in Creating a blue/green deployment.
3. Choose an Aurora PostgreSQL version for the green cluster that's compatible with Aurora Serverless v2, for example 15.3.

   For compatible versions, see Aurora Serverless v2 with Aurora PostgreSQL.
4. Modify the writer DB instance of the green cluster to use the **Serverless v2** (db.serverless) DB instance class.

   For details, see Converting a provisioned writer or reader to Aurora Serverless v2.
5. When your upgraded Aurora Serverless v2 DB cluster is available, switch over from the blue cluster to the green cluster.

You can also upgrade your Aurora Serverless v1 DB cluster directly from Aurora PostgreSQL version 11 to version 13, convert it to a provisioned DB cluster, and then convert the provisioned cluster to an Aurora Serverless v2 DB cluster.

To upgrade, then convert an Aurora Serverless v1 cluster running Aurora PostgreSQL version 11

1. Upgrade the Aurora Serverless v1 cluster to an Aurora PostgreSQL version 13 version that's compatible with Aurora Serverless v2, for example, 13.12. Follow the procedure in Upgrading the major version.

   For compatible versions, see Aurora Serverless v2 with Aurora PostgreSQL.
2. Convert the Aurora Serverless v1 DB cluster to a provisioned Aurora PostgreSQL cluster. Follow the procedure in Converting from Aurora Serverless v1 to provisioned.

3. Add an Aurora Serverless v2 reader DB instance to the cluster. For more information, see Adding an Aurora Serverless v2 reader.
4. Fail over to the Aurora Serverless v2 DB instance:

   a. Select the writer DB instance of the DB cluster.

   b. For **Actions**, choose **Failover**.

   c. On the confirmation page, choose **Failover**.

For Aurora Serverless v1 DB clusters running Aurora PostgreSQL version 13, you convert the Aurora Serverless v1 cluster to a provisioned DB cluster, and then convert the provisioned cluster to an Aurora Serverless v2 DB cluster.

To upgrade an Aurora Serverless v1 cluster running Aurora PostgreSQL version 13

1. Convert the Aurora Serverless v1 DB cluster to a provisioned Aurora PostgreSQL cluster. Follow the procedure in Converting from Aurora Serverless v1 to provisioned.
2. Add an Aurora Serverless v2 reader DB instance to the cluster. For more information, see Adding an Aurora Serverless v2 reader.
3. Fail over to the Aurora Serverless v2 DB instance:

   a. Select the writer DB instance of the DB cluster.

   b. For **Actions**, choose **Failover**.

   c. On the confirmation page, choose **Failover**.

Migrating from an on-premises database to Aurora Serverless v2

You can migrate your on-premises databases to Aurora Serverless v2, just as with provisioned Aurora MySQL and Aurora PostgreSQL.

- For MySQL databases, you can use the `mysqldump` command. For more information, see Importing data to a MySQL or MariaDB DB instance with reduced downtime.
- For PostgreSQL databases, you can use the `pg_dump` and `pg_restore` commands. For more information, see the blog post Best practices for migrating PostgreSQL databases to Amazon RDS and Amazon Aurora.

Using Amazon Aurora Serverless v1

Amazon Aurora Serverless v1 (Amazon Aurora Serverless version 1) is an on-demand autoscaling configuration for Amazon Aurora. An *Aurora Serverless v1 DB cluster* is a DB cluster that scales compute capacity up and down based on your application's needs. This contrasts with Aurora *provisioned DB clusters*, for which you manually manage capacity. Aurora Serverless v1 provides a relatively simple, cost-effective option for infrequent, intermittent, or unpredictable workloads. It is cost-effective because it automatically starts up, scales compute capacity to match your application's usage, and shuts down when it's not in use.

To learn more about pricing, see Serverless Pricing under **MySQL-Compatible Edition** or **PostgreSQL-Compatible Edition** on the Amazon Aurora pricing page.

Aurora Serverless v1 clusters have the same kind of high-capacity, distributed, and highly available storage volume that is used by provisioned DB clusters.

For an Aurora Serverless v2 cluster, you can choose whether to encrypt the cluster volume.

For an Aurora Serverless v1 cluster, the cluster volume is always encrypted. You can choose the encryption key, but you can't disable encryption. That means that you can perform the same operations on an Aurora Serverless v1 that you can on encrypted snapshots. For more information, see Aurora Serverless v1 and snapshots.

Topics

- Scaling Aurora Serverless v1 DB cluster capacity manually
- Viewing Aurora Serverless v1 DB clusters
- Deleting an Aurora Serverless v1 DB cluster
- Aurora Serverless v1 and Aurora database engine versions

> ⚠ **Important**
>
> Aurora has two generations of serverless technology, Aurora Serverless v2 and Aurora Serverless v1. If your application can run on MySQL 8.0 or PostgreSQL 13, we recommend that you use Aurora Serverless v2. Aurora Serverless v2 scales more quickly and in a more granular way. Aurora Serverless v2 also has more compatibility with other Aurora features such as reader DB instances. Thus, if you're already familiar with Aurora, you don't have to learn as many new procedures or limitations to use Aurora Serverless v2 as with Aurora Serverless v1.
>
> You can learn about Aurora Serverless v2 in Using Aurora Serverless v2.

Region and version availability

Feature availability and support varies across specific versions of each Aurora database engine, and across AWS Regions. For more information on version and Region availability with Aurora and Aurora Serverless v1, see Supported Regions and Aurora DB engines for Aurora Serverless v1.

Advantages of Aurora Serverless v1

Aurora Serverless v1 provides the following advantages:

- **Simpler than provisioned –** Aurora Serverless v1 removes much of the complexity of managing DB instances and capacity.
- **Scalable –** Aurora Serverless v1 seamlessly scales compute and memory capacity as needed, with no disruption to client connections.
- **Cost-effective –** When you use Aurora Serverless v1, you pay only for the database resources that you consume, on a per-second basis.
- **Highly available storage –** Aurora Serverless v1 uses the same fault-tolerant, distributed storage system with six-way replication as Aurora to protect against data loss.

Use cases for Aurora Serverless v1

Aurora Serverless v1 is designed for the following use cases:

- **Infrequently used applications** – You have an application that is only used for a few minutes several times per day or week, such as a low-volume blog site. With Aurora Serverless v1, you pay for only the database resources that you consume on a per-second basis.
- **New applications** – You're deploying a new application and you're unsure about the instance size you need. By using Aurora Serverless v1, you can create a database endpoint and have the database autoscale to the capacity requirements of your application.
- **Variable workloads** – You're running a lightly used application, with peaks of 30 minutes to several hours a few times each day, or several times per year. Examples are applications for human resources, budgeting, and operational reporting applications. With Aurora Serverless v1, you no longer need to provision for peak or average capacity.
- **Unpredictable workloads** – You're running daily workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. With Aurora Serverless v1, your database autoscales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over.
- **Development and test databases** – Your developers use databases during work hours but don't need them on nights or weekends. With Aurora Serverless v1, your database automatically shuts down when it's not in use.
- **Multi-tenant applications** – With Aurora Serverless v1, you don't have to individually manage database capacity for each application in your fleet. Aurora Serverless v1 manages individual database capacity for you.

Limitations of Aurora Serverless v1

The following limitations apply to Aurora Serverless v1:

- Aurora Serverless v1 doesn't support the following features:
  - Aurora global databases
  - Aurora Replicas
  - AWS Identity and Access Management (IAM) database authentication
  - Backtracking in Aurora
  - Database activity streams

  - Kerberos authentication
  - Performance Insights
  - RDS Proxy
  - Viewing logs in the AWS Management Console
- Connections to an Aurora Serverless v1 DB cluster are closed automatically if held open for longer than one day.
- All Aurora Serverless v1 DB clusters have the following limitations:
  - You can't export Aurora Serverless v1 snapshots to Amazon S3 buckets.
  - You can't use AWS Database Migration Service and Change Data Capture (CDC) with Aurora Serverless v1 DB clusters. Only provisioned Aurora DB clusters support CDC with AWS DMS as a source.
  - You can't save data to text files in Amazon S3 or load text file data to Aurora Serverless v1 from S3.
  - You can't attach an IAM role to an Aurora Serverless v1 DB cluster. However, you can load data to Aurora Serverless v1 from Amazon S3 by using the `aws_s3` extension with the `aws_s3.table_import_from_s3` function and the `credentials` parameter. For more information, see Importing data from Amazon S3 into an Aurora PostgreSQL DB cluster.
  - When using the query editor, a Secrets Manager secret is created for the DB credentials to access the database. If you delete the credentials from the query editor, the associated secret is also deleted from Secrets Manager. You can't recover this secret after it's deleted.
- Aurora MySQL–based DB clusters running Aurora Serverless v1 don't support the following:
  - Invoking AWS Lambda functions from within your Aurora MySQL DB cluster. However, AWS Lambda functions can make calls to your Aurora Serverless v1 DB cluster.
  - Restoring a snapshot from a DB instance that isn't Aurora MySQL or RDS for MySQL.
  - Replicating data using replication based on binary logs (binlogs). This limitation is true regardless of whether your Aurora MySQL-based DB cluster Aurora Serverless v1 is the source or the target of the replication. To replicate data into an Aurora Serverless v1 DB cluster from a MySQL DB instance outside Aurora, such as one running on Amazon EC2, consider using AWS Database Migration Service. For more information, see the AWS Database Migration Service User Guide.
  - Creating users with host-based access (`'`*`username`*`'@'`*`IP_address`*`'`). This is because Aurora Serverless v1 uses a router fleet between the client and the database host for seamless scaling.

  The IP address that the Aurora Serverless DB cluster sees is that of the router host and not your client. For more information, see Aurora Serverless v1 architecture.

  Instead, use the wildcard (`'username'@'%'`).

- Aurora PostgreSQL–based DB clusters running Aurora Serverless v1 have the following limitations:
  - Aurora PostgreSQL query plan management (`apg_plan_management` extension) isn't supported.
  - The logical replication feature available in Amazon RDS PostgreSQL and Aurora PostgreSQL isn't supported.
  - Outbound communications such as those enabled by Amazon RDS for PostgreSQL extensions aren't supported. For example, you can't access external data with the `postgres_fdw/dblink` extension. For more information about RDS PostgreSQL extensions, see PostgreSQL on Amazon RDS in the *RDS User Guide*.
  - Currently, certain SQL queries and commands aren't recommended. These include session-level advisory locks, temporary relations, asynchronous notifications (`LISTEN`), and cursors with hold (`DECLARE name ... CURSOR WITH HOLD FOR query`). Also, `NOTIFY` commands prevent scaling and aren't recommended.

    For more information, see Autoscaling for Aurora Serverless v1.
- You can't set the preferred automated backup window for an Aurora Serverless v1 DB cluster.
- You can set the maintenance window for an Aurora Serverless v1 DB cluster. For more information, see Adjusting the preferred DB cluster maintenance window.

Configuration requirements for Aurora Serverless v1

When you create an Aurora Serverless v1 DB cluster, pay attention to the following requirements:

- Use these specific port numbers for each DB engine:
  - Aurora MySQL – `3306`
  - Aurora PostgreSQL – `5432`
- Create your Aurora Serverless v1 DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. When you create an Aurora Serverless v1 DB cluster in your VPC, you consume two (2) of the fifty (50) Interface and Gateway Load Balancer endpoints allotted to your VPC.

These endpoints are created automatically for you. To increase your quota, you can contact AWS Support. For more information, see Amazon VPC quotas.

- You can't give an Aurora Serverless v1 DB cluster a public IP address. You can access an Aurora Serverless v1 DB cluster only from within a VPC.
- Create subnets in different Availability Zones for the DB subnet group that you use for your Aurora Serverless v1 DB cluster. In other words, you can't have more than one subnet in the same Availability Zone.
- Changes to a subnet group used by an Aurora Serverless v1 DB cluster aren't applied to the cluster.
- You can access an Aurora Serverless v1 DB cluster from AWS Lambda. To do so, you must configure your Lambda function to run in the same VPC as your Aurora Serverless v1 DB cluster. For more information about working with AWS Lambda, see Configuring a Lambda function to access resources in an Amazon VPC in the *AWS Lambda Developer Guide*.

Using TLS/SSL with Aurora Serverless v1

By default, Aurora Serverless v1 uses the Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocol to encrypt communications between clients and your Aurora Serverless v1 DB cluster. It supports TLS/SSL versions 1.0, 1.1, and 1.2. You don't need to configure your Aurora Serverless v1 DB cluster to use TLS/SSL.

However, the following limitations apply:

- TLS/SSL support for Aurora Serverless v1 DB clusters isn't currently available in the China (Beijing) AWS Region.
- When you create database users for an Aurora MySQL–based Aurora Serverless v1 DB cluster, don't use the `REQUIRE` clause for SSL permissions. Doing so prevents users from connecting to the Aurora DB instance.
- For both MySQL Client and PostgreSQL Client utilities, session variables that you might use in other environments have no effect when using TLS/SSL between client and Aurora Serverless v1.
- For the MySQL Client, when connecting with TLS/SSL's `VERIFY_IDENTITY` mode, currently you need to use the MySQL 8.0-compatible `mysql` command. For more information, see Connecting to a DB instance running the MySQL database engine.

Depending on the client that you use to connect to Aurora Serverless v1 DB cluster, you might not need to specify TLS/SSL to get an encrypted connection. For example, to use the PostgreSQL Client to connect to an Aurora Serverless v1 DB cluster running Aurora PostgreSQL-Compatible Edition, connect as you normally do.

```
psql -h endpoint -U user
```

After you enter your password, the PostgreSQL Client shows you see the connection details, including the TLS/SSL version and cipher.

```
psql (12.5 (Ubuntu 12.5-0ubuntu0.20.04.1), server 10.12)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256,
 compression: off)
Type "help" for help.
```

> **Important**
>
> Aurora Serverless v1 uses the Transport Layer Security/Secure Sockets Layer (TLS/SSL) protocol to encrypt connections by default unless SSL/TLS is disabled by the client application. The TLS/SSL connection terminates at the router fleet. Communication between the router fleet and your Aurora Serverless v1 DB cluster occurs within the service's internal network boundary.
>
> You can check the status of the client connection to examine whether the connection to Aurora Serverless v1 is TLS/SSL encrypted. The PostgreSQL `pg_stat_ssl` and `pg_stat_activity` tables and its `ssl_is_used` function don't show the TLS/SSL state for the communication between the client application and Aurora Serverless v1. Similarly, the TLS/SSL state can't be derived from the MySQL `status` statement.
>
> The Aurora cluster parameters `force_ssl` for PostgreSQL and `require_secure_transport` for MySQL formerly weren't supported for Aurora Serverless v1. These parameters are available now for Aurora Serverless v1. For a complete list of parameters supported by Aurora Serverless v1, call the DescribeEngineDefaultClusterParameters API operation. For more information on parameter groups and Aurora Serverless v1, see Parameter groups for Aurora Serverless v1.

To use the MySQL Client to connect to an Aurora Serverless v1 DB cluster running Aurora MySQL-Compatible Edition, you specify TLS/SSL in your request. The following example includes the

Amazon root CA 1 trust store downloaded from Amazon Trust Services, which is necessary for this connection to succeed.

```
mysql -h endpoint -P 3306 -u user -p --ssl-ca=amazon-root-CA-1.pem --ssl-mode=REQUIRED
```

When prompted, enter your password. Soon, the MySQL monitor opens. You can confirm that the session is encrypted by using the `status` command.

```
mysql> status
--------------
mysql  Ver 14.14 Distrib 5.5.62, for Linux (x86_64) using readline 5.1
Connection id:          19
Current database:
Current user:           ***@*******
SSL:                    Cipher in use is ECDHE-RSA-AES256-SHA
...
```

To learn more about connecting to Aurora MySQL database with the MySQL Client, see Connecting to a DB instance running the MySQL database engine.

Aurora Serverless v1 supports all TLS/SSL modes available to the MySQL Client (`mysql`) and PostgreSQL Client (`psql`), including those listed in the following table.

Description of TLS/SSL mode	mysql	psql
Connect without using TLS/SSL.	DISABLED	disable
Try the connection using TLS/SSL first, but fall back to non-SSL if necessary.	PREFERRED	prefer (default)
Enforce using TLS/SSL.	REQUIRED	require
Enforce TLS/SSL and verify the CA.	VERIFY_CA	verify-ca

Description of TLS/SSL mode	mysql	psql
Enforce TLS/SSL, verify the CA, and verify the CA hostname.	VERIFY_IDENTITY	verify-full

Aurora Serverless v1 uses wildcard certificates. If you specify the "verify CA" or the "verify CA and CA hostname" option when using TLS/SSL, first download the Amazon root CA 1 trust store from Amazon Trust Services. After doing so, you can identify this PEM-formatted file in your client command. To do so using the PostgreSQL Client:

For Linux, macOS, or Unix:

```
psql 'host=endpoint user=user sslmode=require sslrootcert=amazon-root-CA-1.pem
 dbname=db-name'
```

To learn more about working with the Aurora PostgreSQL database using the Postgres Client, see Connecting to a DB instance running the PostgreSQL database engine.

For more information about connecting to Aurora DB clusters in general, see Connecting to an Amazon Aurora DB cluster.

Supported cipher suites for connections to Aurora Serverless v1 DB clusters

By using configurable cipher suites, you can have more control over the security of your database connections. You can specify a list of cipher suites that you want to allow to secure client TLS/SSL connections to your database. With configurable cipher suites, you can control the connection encryption that your database server accepts. Doing this prevents the use of ciphers that aren't secure or that are no longer used.

Aurora Serverless v1 DB clusters that are based on Aurora MySQL support the same cipher suites as Aurora MySQL provisioned DB clusters. For information about these cipher suites, see Configuring cipher suites for connections to Aurora MySQL DB clusters.

Aurora Serverless v1 DB clusters that are based on Aurora PostgreSQL don't support cipher suites.

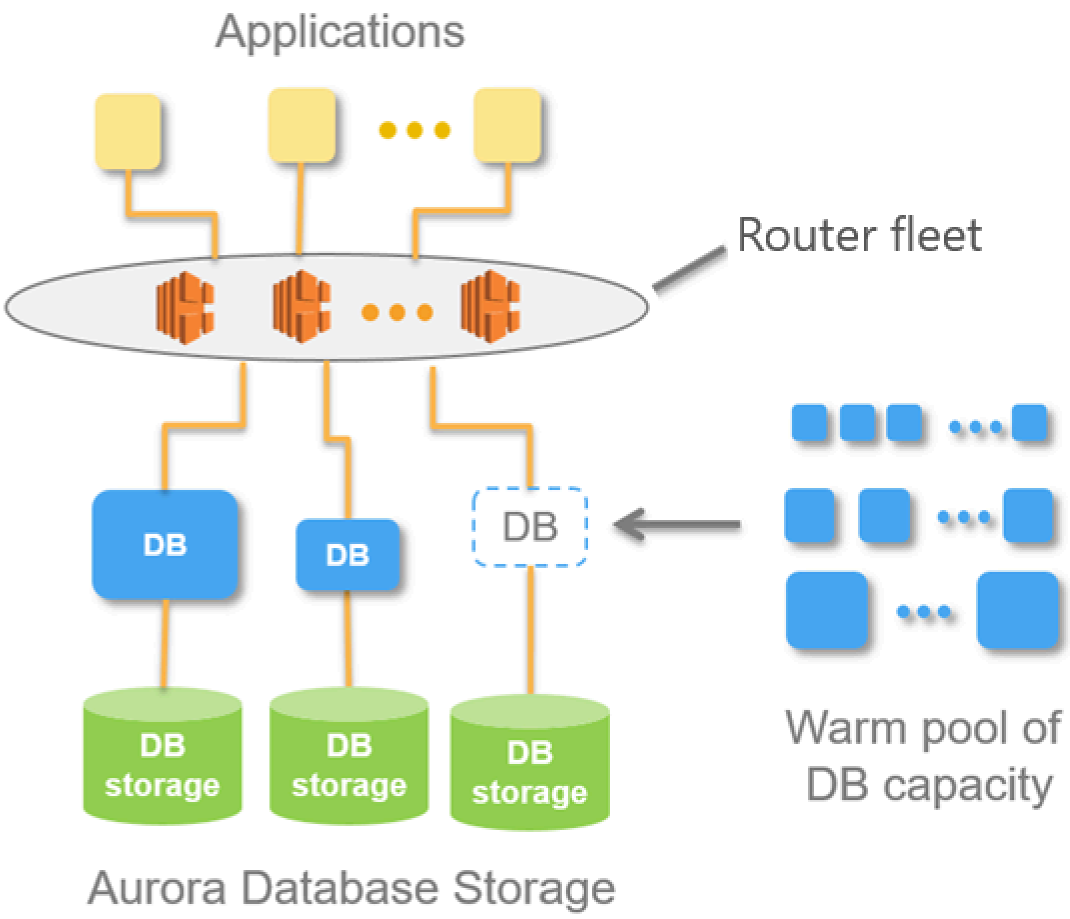
How Aurora Serverless v1 works

Following, you can learn how Aurora Serverless v1 works.

Topics

Aurora Serverless v1 architecture

The following image shows an overview of the Aurora Serverless v1 architecture.



Instead of provisioning and managing database servers, you specify Aurora capacity units (ACUs). Each ACU is a combination of approximately 2 gigabytes (GB) of memory, corresponding CPU, and networking. Database storage automatically scales from 10 gibibytes (GiB) to 128 tebibytes (TiB), the same as storage in a standard Aurora DB cluster.

You can specify the minimum and maximum ACU. The *minimum Aurora capacity unit* is the lowest ACU to which the DB cluster can scale down. The *maximum Aurora capacity unit* is the highest ACU to which the DB cluster can scale up. Based on your settings, Aurora Serverless v1 automatically creates scaling rules for thresholds for CPU utilization, connections, and available memory.

Aurora Serverless v1 manages the warm pool of resources in an AWS Region to minimize scaling time. When Aurora Serverless v1 adds new resources to the Aurora DB cluster, it uses the router fleet to switch active client connections to the new resources. At any specific time, you are charged only for the ACUs that are being actively used in your Aurora DB cluster.

Autoscaling for Aurora Serverless v1

The capacity allocated to your Aurora Serverless v1 DB cluster seamlessly scales up and down based on the load generated by your client application. Here, load is CPU utilization and the number of connections. When capacity is constrained by either of these, Aurora Serverless v1 scales up. Aurora Serverless v1 also scales up when it detects performance issues that can be resolved by doing so.

You can view scaling events for your Aurora Serverless v1 cluster in the AWS Management Console. During autoscaling, Aurora Serverless v1 resets the `EngineUptime` metric. The value of the reset metric value doesn't mean that seamless scaling had problems or that Aurora Serverless v1 dropped connections. It's simply the starting point for uptime at the new capacity. To learn more about metrics, see Monitoring metrics in an Amazon Aurora cluster.

When your Aurora Serverless v1 DB cluster has no active connections, it can scale down to zero capacity (0 ACUs). To learn more, see Pause and resume for Aurora Serverless v1.

When it does need to perform a scaling operation, Aurora Serverless v1 first tries to identify a *scaling point*, a moment when no queries are being processed. Aurora Serverless v1 might not be able to find a scaling point for the following reasons:

- Long-running queries
- In-progress transactions
- Temporary tables or table locks

To increase your Aurora Serverless v1 DB cluster's success rate when finding a scaling point, we recommend that you avoid long-running queries and long-running transactions. To learn more about operations that block scaling and how to avoid them, see Best practices for working with Aurora Serverless v1.

By default, Aurora Serverless v1 tries to find a scaling point for 5 minutes (300 seconds). You can specify a different timeout period when you create or modify the cluster. The timeout period can be between 60 seconds and 10 minutes (600 seconds). If Aurora Serverless v1 can't find a scaling point within the specified period, the autoscaling operation times out.

By default, if autoscaling doesn't find a scaling point before timing out, Aurora Serverless v1 keeps the cluster at the current capacity. You can change this default behavior when you create or modify your Aurora Serverless v1 DB cluster by selecting the **Force the capacity change** option. For more information, see Timeout action for capacity changes.

Timeout action for capacity changes

If autoscaling times out without finding a scaling point, by default Aurora keeps the current capacity. You can choose to have Aurora force the change by selecting the **Force the capacity change** option. This option is available in the **Autoscaling timeout and action** section of the **Create database** page when you create the cluster.

By default, the **Force the capacity change** option isn't selected. Keep this option clear to have your Aurora Serverless v1 DB cluster's capacity remain unchanged if the scaling operation times out without finding a scaling point.

Selecting this option causes your Aurora Serverless v1 DB cluster to enforce the capacity change, even without a scaling point. Before selecting this option, be aware of the consequences of this selection:

- Any in-process transactions are interrupted, and the following error message appears.

  Aurora MySQL version 2 – ERROR 1105 (HY000): The last transaction was aborted due to Seamless Scaling. Please retry.

  You can resubmit the transactions as soon as your Aurora Serverless v1 DB cluster is available.
- Connections to temporary tables and locks are dropped.

  We recommend that you select the **Force the capacity change** option only if your application can recover from dropped connections or incomplete transactions.

The choices that you make in the AWS Management Console when you create an Aurora Serverless v1 DB cluster are stored in the `ScalingConfigurationInfo` object, in the `SecondsBeforeTimeout` and `TimeoutAction` properties. The value of the `TimeoutAction` property is set to one of the following values when you create your cluster:

- `RollbackCapacityChange` – This value is set when you select the **Roll back the capacity change** option. This is the default behavior.
- `ForceApplyCapacityChange` – This value is set when you select the **Force the capacity change** option.

You can get the value of this property on an existing Aurora Serverless v1 DB cluster by using the describe-db-clusters AWS CLI command, as shown following.

For Linux, macOS, or Unix:

```
aws rds describe-db-clusters --region region \
  --db-cluster-identifier your-cluster-name \
  --query '*[].{ScalingConfigurationInfo:ScalingConfigurationInfo}'
```

For Windows:

```
aws rds describe-db-clusters --region region ^
  --db-cluster-identifier your-cluster-name ^
  --query "*[].{ScalingConfigurationInfo:ScalingConfigurationInfo}"
```

As an example, the following shows the query and response for an Aurora Serverless v1 DB cluster named `west-coast-sles` in the US West (N. California) Region.

```
$ aws rds describe-db-clusters --region us-west-1 --db-cluster-identifier west-coast-
sles
--query '*[].{ScalingConfigurationInfo:ScalingConfigurationInfo}'

[
    {
        "ScalingConfigurationInfo": {
            "MinCapacity": 1,
            "MaxCapacity": 64,
            "AutoPause": false,
            "SecondsBeforeTimeout": 300,
            "SecondsUntilAutoPause": 300,
            "TimeoutAction": "RollbackCapacityChange"
        }
    }
]
```

As the response shows, this Aurora Serverless v1 DB cluster uses the default setting.

For more information, see Creating an Aurora Serverless v1 DB cluster. After creating your Aurora Serverless v1, you can modify the timeout action and other capacity settings at any time. To learn how, see Modifying an Aurora Serverless v1 DB cluster.

Pause and resume for Aurora Serverless v1

You can choose to pause your Aurora Serverless v1 DB cluster after a given amount of time with no activity. You specify the amount of time with no activity before the DB cluster is paused. When you select this option, the default inactivity time is five minutes, but you can change this value. This is an optional setting.

When the DB cluster is paused, no compute or memory activity occurs, and you are charged only for storage. If database connections are requested when an Aurora Serverless v1 DB cluster is paused, the DB cluster automatically resumes and services the connection requests.

When the DB cluster resumes activity, it has the same capacity as it had when Aurora paused the cluster. The number of ACUs depends on how much Aurora scaled the cluster up or down before pausing it.

> **Note**
>
> If a DB cluster is paused for more than seven days, the DB cluster might be backed up with a snapshot. In this case, Aurora restores the DB cluster from the snapshot when there is a request to connect to it.

Determining the maximum number of database connections for Aurora Serverless v1

The following examples are for an Aurora Serverless v1 DB cluster that's compatible with MySQL 5.7. You can use a MySQL client or the query editor, if you've configured access to it. For more information, see Running queries in the query editor.

To find the maximum number of database connections

1. Find the capacity range for your Aurora Serverless v1 DB cluster using the AWS CLI.

```
aws rds describe-db-clusters \
    --db-cluster-identifier my-serverless-57-cluster \
    --query 'DBClusters[*].ScalingConfigurationInfo|[0]'
```

   The result shows that its capacity range is 1–4 ACUs.

```
{
    "MinCapacity": 1,
    "AutoPause": true,
    "MaxCapacity": 4,
    "TimeoutAction": "RollbackCapacityChange",
    "SecondsUntilAutoPause": 3600
}
```

2. Run the following SQL query to find the maximum number of connections.

   ```
   select @@max_connections;
   ```

   The result shown is for the minimum capacity of the cluster, 1 ACU.

   ```
   @@max_connections
   90
   ```

3. Scale the cluster to 8–32 ACUs.

   For more information on scaling, see Modifying an Aurora Serverless v1 DB cluster.

4. Confirm the capacity range.

   ```
   {
       "MinCapacity": 8,
       "AutoPause": true,
       "MaxCapacity": 32,
       "TimeoutAction": "RollbackCapacityChange",
       "SecondsUntilAutoPause": 3600
   }
   ```

5. Find the maximum number of connections.

   ```
   select @@max_connections;
   ```

   The result shown is for the minimum capacity of the cluster, 8 ACUs.

   ```
   @@max_connections
   1000
   ```

6. Scale the cluster to the maximum possible, 256–256 ACUs.

7. Confirm the capacity range.

   ```
   {
       "MinCapacity": 256,
       "AutoPause": true,
   ```

```
    "MaxCapacity": 256,
    "TimeoutAction": "RollbackCapacityChange",
    "SecondsUntilAutoPause": 3600
}
```

8. Find the maximum number of connections.

   ```
   select @@max_connections;
   ```

   The result shown is for 256 ACUs.

   ```
   @@max_connections
   6000
   ```

   > **Note**
   >
   > The `max_connections` value doesn't scale linearly with the number of ACUs.

9. Scale the cluster back down to 1–4 ACUs.

   ```
   {
       "MinCapacity": 1,
       "AutoPause": true,
       "MaxCapacity": 4,
       "TimeoutAction": "RollbackCapacityChange",
       "SecondsUntilAutoPause": 3600
   }
   ```

   This time, the `max_connections` value is for 4 ACUs.

   ```
   @@max_connections
   270
   ```

10. Let the cluster scale down to 2 ACUs.

    ```
    @@max_connections
    180
    ```

    If you've configured the cluster to pause after a certain amount of time idle, it scales down to 0 ACUs. However, `max_connections` doesn't drop below the value for 1 ACU.

```
@@max_connections
90
```

Parameter groups for Aurora Serverless v1

When you create your Aurora Serverless v1 DB cluster, you choose a specific Aurora DB engine and an associated DB cluster parameter group. Unlike provisioned Aurora DB clusters, an Aurora Serverless v1 DB cluster has a single read/write DB instance that's configured with a DB cluster parameter group only—it doesn't have a separate DB parameter group. During autoscaling, Aurora Serverless v1 needs to be able to change parameters for the cluster to work best for the increased or decreased capacity. Thus, with an Aurora Serverless v1 DB cluster, some of the changes that you might make to parameters for a particular DB engine type might not apply.

For example, an Aurora PostgreSQL–based Aurora Serverless v1 DB cluster can't use `apg_plan_mgmt.capture_plan_baselines` and other parameters that might be used on provisioned Aurora PostgreSQL DB clusters for query plan management.

You can get a list of default values for the default parameter groups for the various Aurora DB engines by using the describe-engine-default-cluster-parameters CLI command and querying the AWS Region. The following are values that you can use for the `--db-parameter-group-family` option.

Aurora MySQL version 2	`aurora-mysql5.7`
Aurora PostgreSQL version 11	`aurora-postgresql11`
Aurora PostgreSQL version 13	`aurora-postgresql13`

We recommend that you configure your AWS CLI with your AWS access key ID and AWS secret access key, and that you set your AWS Region before using AWS CLI commands. Providing the Region to your CLI configuration saves you from entering the `--region` parameter when running commands. To learn more about configuring AWS CLI, see Configuration basics in the *AWS Command Line Interface User Guide*.

The following example gets a list of parameters from the default DB cluster group for Aurora MySQL version 2.

For Linux, macOS, or Unix:

```
aws rds describe-engine-default-cluster-parameters \
  --db-parameter-group-family aurora-mysql5.7 --query \
  'EngineDefaults.Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} | [?
contains(SupportedEngineModes, `serverless`) == `true`] | [*].{param:ParameterName}' \
  --output text
```

For Windows:

```
aws rds describe-engine-default-cluster-parameters ^
   --db-parameter-group-family aurora-mysql5.7 --query ^
   "EngineDefaults.Parameters[*].
{ParameterName:ParameterName,SupportedEngineModes:SupportedEngineModes} | [?
contains(SupportedEngineModes, 'serverless') == `true`] | [*].{param:ParameterName}" ^
   --output text
```

Modifying parameter values for Aurora Serverless v1

As explained in Working with parameter groups, you can't directly change values in a default parameter group, regardless of its type (DB cluster parameter group, DB parameter group). Instead, you create a custom parameter group based on the default DB cluster parameter group for your Aurora DB engine and change settings as needed on that parameter group. For example, you might want to change some of the settings for your Aurora Serverless v1 DB cluster to log queries or to upload DB engine specific logs to Amazon CloudWatch.

To create a custom DB cluster parameter group

1. Sign in to the AWS Management Console and then open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Parameter groups**.
3. Choose **Create parameter group** to open the Parameter group details pane.
4. Choose the appropriate default DB cluster group for the DB engine you want to use for your Aurora Serverless v1 DB cluster. Be sure that you choose the following options:

   a. For **Parameter group family**, choose the appropriate family for your chosen DB engine. Be sure that your choice has the prefix `aurora-` in its name.

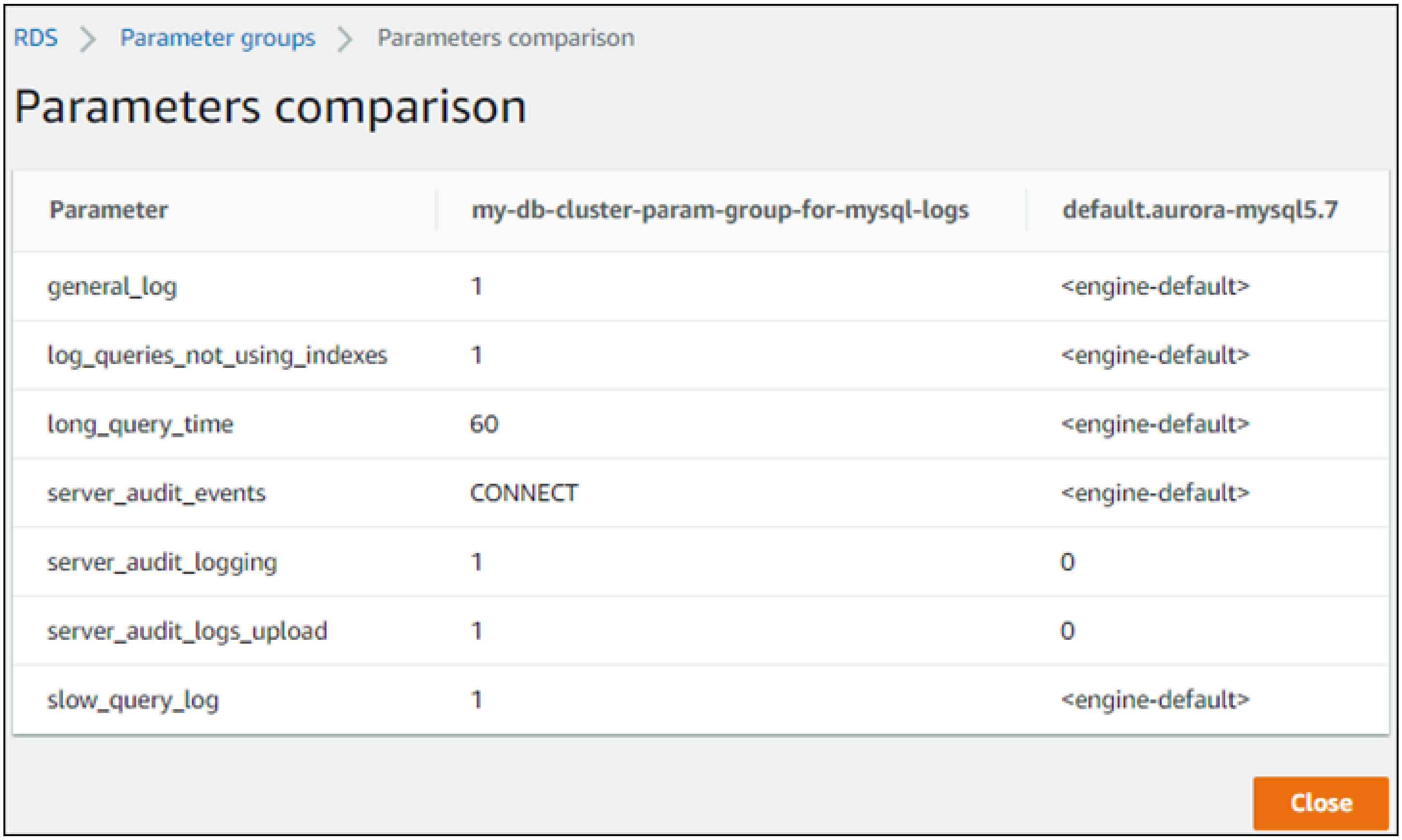
   b. For **Type**, choose **DB Cluster Parameter Group**.

c. For **Group name** and **Description**, enter meaningful names for you or others who might need to work with your Aurora Serverless v1 DB cluster and its parameters.

d. Choose **Create**.

Your custom DB cluster parameter group is added to the list of parameter groups available in your AWS Region. You can use your custom DB cluster parameter group when you create new Aurora Serverless v1 DB clusters. You can also modify an existing Aurora Serverless v1 DB cluster to use your custom DB cluster parameter group. After your Aurora Serverless v1 DB cluster starts using your custom DB cluster parameter group, you can change values for dynamic parameters using either the AWS Management Console or the AWS CLI.

You can also use the console to view a side-by-side comparison of the values in your custom DB cluster parameter group compared to the default DB cluster parameter group, as shown in the following screenshot.



Parameters comparison

Parameter	my-db-cluster-param-group-for-mysql-logs	default.aurora-mysql5.7
general_log	1	<engine-default>
log_queries_not_using_indexes	1	<engine-default>
long_query_time	60	<engine-default>
server_audit_events	CONNECT	<engine-default>
server_audit_logging	1	0
server_audit_logs_upload	1	0
slow_query_log	1	<engine-default>

When you change parameter values on an active DB cluster, Aurora Serverless v1 starts a seamless scale in order to apply the parameter changes. If your Aurora Serverless v1 DB cluster is in a paused state, it resumes and starts scaling so that it can make the change. The scaling operation for a parameter group change always forces a capacity change, so be aware that modifying parameters might result in dropped connections if a scaling point can't be found during the scaling period.

Logging for Aurora Serverless v1

By default, error logs for Aurora Serverless v1 are enabled and automatically uploaded to Amazon CloudWatch. You can also have your Aurora Serverless v1 DB cluster upload Aurora database-engine specific logs to CloudWatch. To do this, enable configuration parameters in your custom DB cluster parameter group. Your Aurora Serverless v1 DB cluster then uploads all available logs to Amazon CloudWatch. At this point, you can use CloudWatch to analyze log data, create alarms, and view metrics.

For Aurora MySQL, the following table shows the logs that you can enable. When enabled, they're automatically uploaded from your Aurora Serverless v1 DB cluster to Amazon CloudWatch.

Aurora MySQL log	Description
`general_log`	Creates the general log. Set to 1 to turn on. Default is off (0).
`log_queries_not_using_indexes`	Logs any queries to the slow query log that don't use an index. Default is off (0). Set to 1 to turn on this log.
`long_query_time`	Prevents fast-running queries from being logged in the slow query log. Can be set to a float between 0 and 3,1536,000. Default is 0 (not active).
`server_audit_events`	The list of events to capture in the logs. Supported values are `CONNECT`, `QUERY`, `QUERY_DCL` , `QUERY_DDL` , `QUERY_DML` , and `TABLE`.
`server_audit_logging`	Set to 1 to turn on server audit logging. If you turn this on, you can specify the audit events to send to CloudWatch by listing them in the `server_audit_events` parameter.

Aurora MySQL log	Description
slow_query_log	Creates a slow query log. Set to 1 to turn on the slow query log. Default is off (0).

For more information, see Using Advanced Auditing with an Amazon Aurora MySQL DB cluster.

For Aurora PostgreSQL, the following table shows the logs that you can enable. When enabled, they're automatically uploaded from your Aurora Serverless v1 DB cluster to Amazon CloudWatch along with the regular error logs.

Aurora PostgreSQL log	Description
log_connections	Turned on by default and can't be changed. It logs details for all new client connections.
log_disconnections	Turned on by default and can't be changed. Logs all client disconnections.
log_hostname	Turned off by default and can't be changed. Hostnames aren't logged.
log_lock_waits	Default is 0 (off). Set to 1 to log lock waits.
log_min_duration_statement	The minimum duration (in milliseconds) for a statement to run before it's logged.
log_min_messages	Sets the message levels that are logged. Supported values are debug5, debug4, debug3, debug2, debug1, info, notice, warning, error, log, fatal, panic. To log performance data to the postgres log, set the value to debug1.
log_temp_files	Logs the use of temporary files that are above the specified kilobytes (kB).

Aurora PostgreSQL log	Description
`log_statement`	Controls the specific SQL statements that get logged. Supported values are none, `ddl`, `mod`, and `all`. Default is none.

After you turn on logs for Aurora MySQL or Aurora PostgreSQL for your Aurora Serverless v1 DB cluster, you can view the logs in CloudWatch.

Viewing Aurora Serverless v1 logs with Amazon CloudWatch

Aurora Serverless v1 automatically uploads ("publishes") to Amazon CloudWatch all logs that are enabled in your custom DB cluster parameter group. You don't need to choose or specify the log types. Uploading logs starts as soon as you enable the log configuration parameter. If you later disable the log parameter, further uploads stop. However, all the logs that have already been published to CloudWatch remain until you delete them.

For more information on using CloudWatch with Aurora MySQL logs, see Monitoring log events in Amazon CloudWatch.

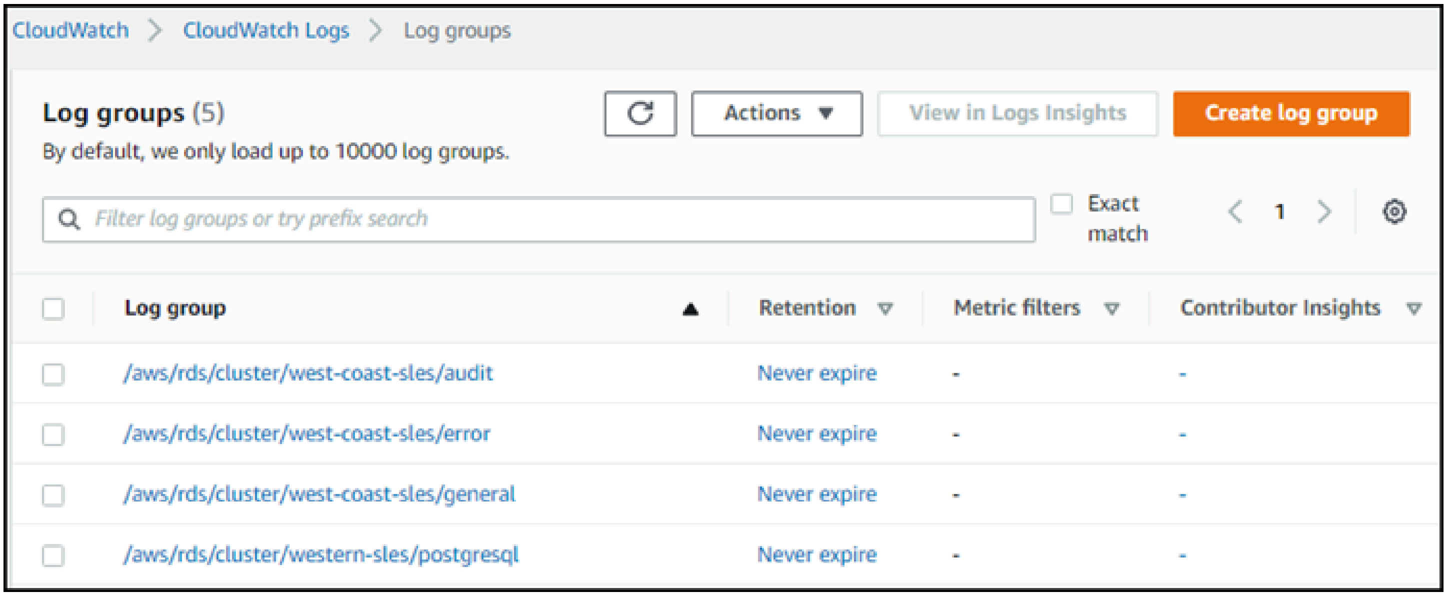
For more information about CloudWatch and Aurora PostgreSQL, see Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs.

To view logs for your Aurora Serverless v1 DB cluster

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose your AWS Region.
3. Choose **Log groups**.
4. Choose your Aurora Serverless v1 DB cluster log from the list. For error logs, the naming pattern is as follows.

```
/aws/rds/cluster/cluster-name/error
```

For example, in the following screenshot you can find listings for logs published for an Aurora PostgreSQL Aurora Serverless v1 DB cluster named `western-sles`. You can also find several listings for Aurora MySQL Aurora Serverless v1 DB cluster, `west-coast-sles`. Choose the log that you're interest in to start exploring its content.


CloudWatch > CloudWatch Logs > Log groups
C
Actions ▼
View in Logs Insights
Create log group
Filter log groups or try prefix search
Exact match
< 1 >
⚙

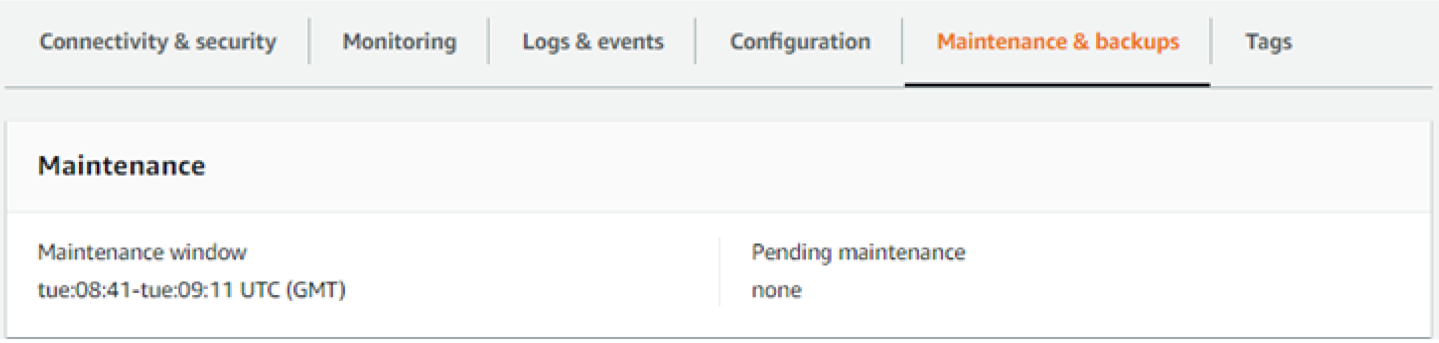
Log groups (5)

By default, we only load up to 10000 log groups.

□	Log group ▲	Retention ▽	Metric filters ▽	Contributor Insights ▽
□	/aws/rds/cluster/west-coast-sles/audit	Never expire	-	-
□	/aws/rds/cluster/west-coast-sles/error	Never expire	-	-
□	/aws/rds/cluster/west-coast-sles/general	Never expire	-	-
□	/aws/rds/cluster/western-sles/postgresql	Never expire	-	-

Aurora Serverless v1 and maintenance

Maintenance for Aurora Serverless v1 DB cluster, such as applying the latest features, fixes, and security updates, is performed automatically for you. Aurora Serverless v1 has a maintenance window that you can view in the AWS Management Console in **Maintenance & backups** for your Aurora Serverless v1 DB cluster. You can find the date and time that maintenance might be performed and if any maintenance is pending for your Aurora Serverless v1 DB cluster, as shown in the following figure.



Maintenance

Maintenance window
tue:08:41-tue:09:11 UTC (GMT)

Pending maintenance
none

You can set the maintenance window when you create the Aurora Serverless v1 DB cluster, and you can modify the window later. For more information, see Adjusting the preferred DB cluster maintenance window.

Maintenance windows are used for scheduled major version upgrades. Minor version upgrades and patches are applied immediately during scaling. Scaling happens according to your setting for `TimeoutAction`:

- `ForceApplyCapacityChange` – The change is applied immediately.
- `RollbackCapacityChange` – Aurora forcibly updates the cluster after 3 days from the first patch attempt.

As with any change that's forced without an appropriate scaling point, this might interrupt your workload.

Whenever possible, Aurora Serverless v1 performs maintenance in a nondisruptive manner. When maintenance is required, your Aurora Serverless v1 DB cluster scales its capacity to handle the necessary operations. Before scaling, Aurora Serverless v1 looks for a scaling point. It does so for up to three days if necessary.

At the end of each day that Aurora Serverless v1 can't find a scaling point, it creates a cluster event. This event notifies you of the pending maintenance and the need to scale to perform maintenance. The notification includes the date when Aurora Serverless v1 can force the DB cluster to scale.

For more information, see Timeout action for capacity changes.

Aurora Serverless v1 and failover

If the DB instance for an Aurora Serverless v1 DB cluster becomes unavailable or the Availability Zone (AZ) it's in fails, Aurora recreates the DB instance in a different AZ. However, the Aurora Serverless v1 cluster isn't a Multi-AZ cluster. That's because it consists of a single DB instance in a single AZ. Thus, this failover mechanism takes longer than for an Aurora cluster with provisioned or Aurora Serverless v2 instances. The Aurora Serverless v1 failover time is undefined because it depends on demand and capacity availability in other AZs within the given AWS Region.

Because Aurora separates computation capacity and storage, the storage volume for the cluster is spread across multiple AZs. Your data remains available even if outages affect the DB instance or the associated AZ.

Aurora Serverless v1 and snapshots

The cluster volume for an Aurora Serverless v1 cluster is always encrypted. You can choose the encryption key, but you can't disable encryption. To copy or share a snapshot of an Aurora Serverless v1 cluster, encrypt the snapshot using your own AWS KMS key. For more information, see Copying a DB cluster snapshot. To learn more about encryption and Amazon Aurora, see Encrypting an Amazon Aurora DB cluster

Creating an Aurora Serverless v1 DB cluster

The following procedure creates an Aurora Serverless v1 cluster without any of your schema objects or data. If you want to create an Aurora Serverless v1 cluster that's a duplicate of an existing provisioned or Aurora Serverless v1 cluster, you can perform a snapshot restore or cloning operation instead. For those details, see Restoring from a DB cluster snapshot and Cloning a volume for an Amazon Aurora DB cluster. You can't convert an existing provisioned cluster to Aurora Serverless v1. You also can't convert an existing Aurora Serverless v1 cluster back to a provisioned cluster.

When you create an Aurora Serverless v1 DB cluster, you can set the minimum and maximum capacity for the cluster. A capacity unit is equivalent to a specific compute and memory configuration. Aurora Serverless v1 creates scaling rules for thresholds for CPU utilization, connections, and available memory and seamlessly scales to a range of capacity units as needed for your applications. For more information see Aurora Serverless v1 architecture.

You can set the following specific values for your Aurora Serverless v1 DB cluster:

- **Minimum Aurora capacity unit** – Aurora Serverless v1 can reduce capacity down to this capacity unit.
- **Maximum Aurora capacity unit** – Aurora Serverless v1 can increase capacity up to this capacity unit.

You can also choose the following optional scaling configuration options:

- **Force scaling the capacity to the specified values when the timeout is reached** – You can choose this setting if you want Aurora Serverless v1 to force Aurora Serverless v1 to scale even if it can't find a scaling point before it times out. If you want Aurora Serverless v1 to cancel capacity changes if it can't find a scaling point, don't choose this setting. For more information, see Timeout action for capacity changes.
- **Pause compute capacity after consecutive minutes of inactivity** – You can choose this setting if you want Aurora Serverless v1 to scale to zero when there's no activity on your DB cluster for an amount of time you specify. With this setting enabled, your Aurora Serverless v1 DB cluster automatically resumes processing and scales to the necessary capacity to handle the workload when database traffic resumes. To learn more, see Pause and resume for Aurora Serverless v1.

Before you can create an Aurora Serverless v1 DB cluster, you need an AWS account. You also need to have completed the setup tasks for working with Amazon Aurora. For more information, see Setting up your environment for Amazon Aurora. You also need to complete other preliminary steps for creating any Aurora DB cluster. To learn more, see Creating an Amazon Aurora DB cluster.

Aurora Serverless v1 is available in certain AWS Regions and for specific Aurora MySQL and Aurora PostgreSQL versions only. For more information, see Supported Regions and Aurora DB engines for Aurora Serverless v1.

> **Note**
>
> The cluster volume for an Aurora Serverless v1 cluster is always encrypted. When you create your Aurora Serverless v1 DB cluster, you can't turn off encryption, but you can choose to use your own encryption key. With Aurora Serverless v2, you can choose whether to encrypt the cluster volume.

You can create an Aurora Serverless v1 DB cluster with the AWS Management Console, the AWS CLI, or the RDS API.

> **Note**
>
> If you receive the following error message when trying to create your cluster, your account needs additional permissions.
>
> `Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later.`
>
> See Using service-linked roles for Amazon Aurora for more information.

You can't directly connect to the DB instance on your Aurora Serverless v1 DB cluster. To connect to your Aurora Serverless v1 DB cluster, you use the database endpoint. You can find the endpoint for your Aurora Serverless v1 DB cluster on the **Connectivity & security** tab for your cluster in the AWS Management Console. For more information, see Connecting to an Amazon Aurora DB cluster.

Console

Use the following general procedure. For more information on creating an Aurora DB cluster using the AWS Management Console, see Creating an Amazon Aurora DB cluster.

To create a new Aurora Serverless v1 DB cluster

1. Sign in to the AWS Management Console.
2. Choose an AWS Region that supports Aurora Serverless v1.
3. Choose Amazon RDS from the AWS Services list.
4. Choose **Create database**.
5. On the **Create database** page:

   a. Choose **Standard create** for the database creation method.

   b. Continue creating the Aurora Serverless v1 DB cluster by using the steps from the following examples.

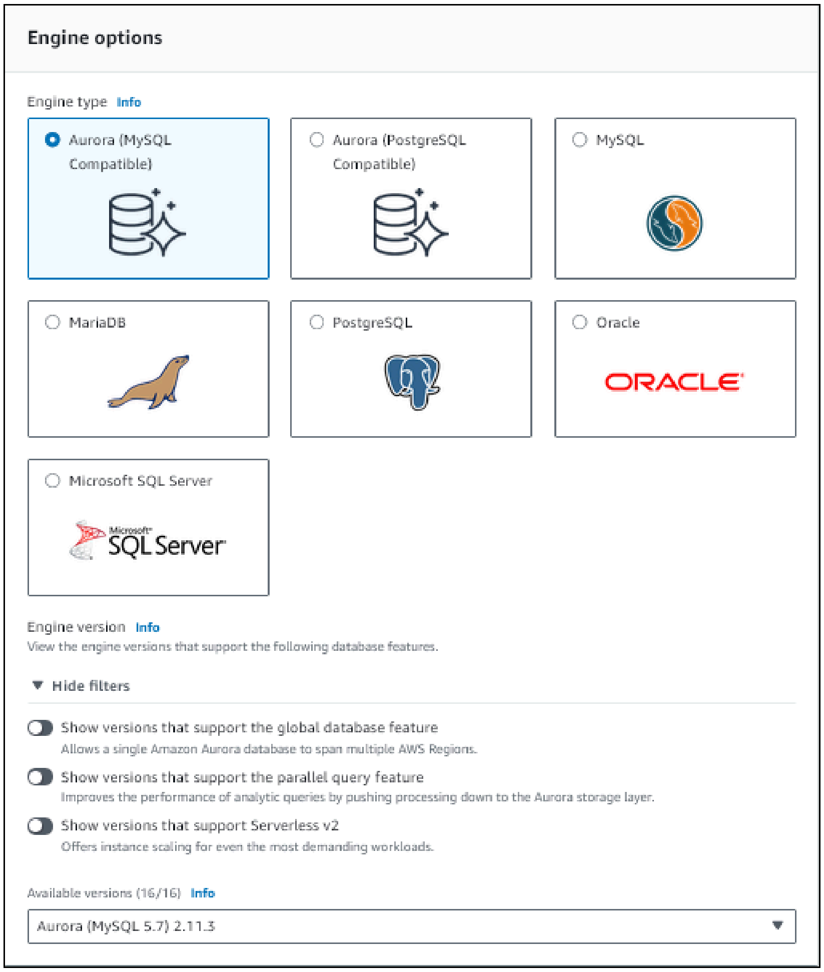
   > **Note**
   >
   > If you choose a version of the DB engine that doesn't support Aurora Serverless v1, the **Serverless** option doesn't display for **DB instance class**.

Example for Aurora MySQL

Use the following procedure.

To create an Aurora Serverless v1 DB cluster for Aurora MySQL

1. For **Engine type**, choose **Aurora (MySQL Compatibile)**.
2. Choose the Aurora MySQL version, compatible with Aurora Serverless v1, that you want for your DB cluster. The supported versions are shown on the right side of the page.



Engine options

Engine type Info

● Aurora (MySQL Compatible)

○ Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

Engine version Info
View the engine versions that support the following database features.

▼ Hide filters

Show versions that support the global database feature
Allows a single Amazon Aurora database to span multiple AWS Regions.

Show versions that support the parallel query feature
Improves the performance of analytic queries by pushing processing down to the Aurora storage layer.

Show versions that support Serverless v2
Offers instance scaling for even the most demanding workloads.

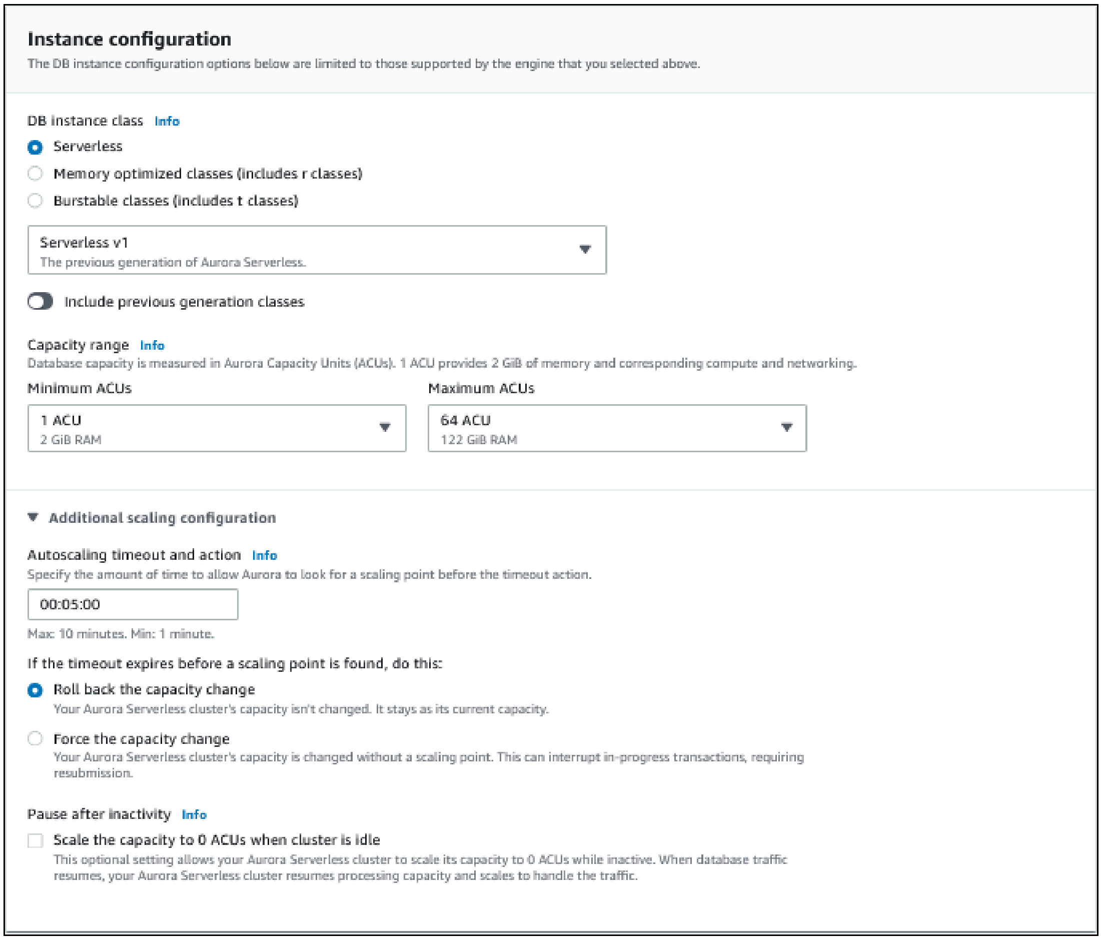
Available versions (16/16) Info

Aurora (MySQL 5.7) 2.11.3 ▼

3. For **DB instance class**, choose **Serverless**.

4. Set the **Capacity range** for the DB cluster.

5. Adjust values as needed in the **Additional scaling configuration** section of the page. To learn more about capacity settings, see Autoscaling for Aurora Serverless v1.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

Serverless v1
The previous generation of Aurora Serverless.

Include previous generation classes

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs	Maximum ACUs
1 ACU 2 GiB RAM	64 ACU 122 GiB RAM

▼ Additional scaling configuration

Autoscaling timeout and action Info

Specify the amount of time to allow Aurora to look for a scaling point before the timeout action.

00:05:00

Max: 10 minutes. Min: 1 minute.

If the timeout expires before a scaling point is found, do this:

● Roll back the capacity change
Your Aurora Serverless cluster's capacity isn't changed. It stays as its current capacity.

○ Force the capacity change
Your Aurora Serverless cluster's capacity is changed without a scaling point. This can interrupt in-progress transactions, requiring resubmission.

Pause after inactivity Info

□ Scale the capacity to 0 ACUs when cluster is idle
This optional setting allows your Aurora Serverless cluster to scale its capacity to 0 ACUs while inactive. When database traffic resumes, your Aurora Serverless cluster resumes processing capacity and scales to handle the traffic.

6. To enable the Data API for your Aurora Serverless v1 DB cluster, select the **Data API** check box under **Additional configuration** in the **Connectivity** section.

   To learn more about the Data API, see Using RDS Data API.

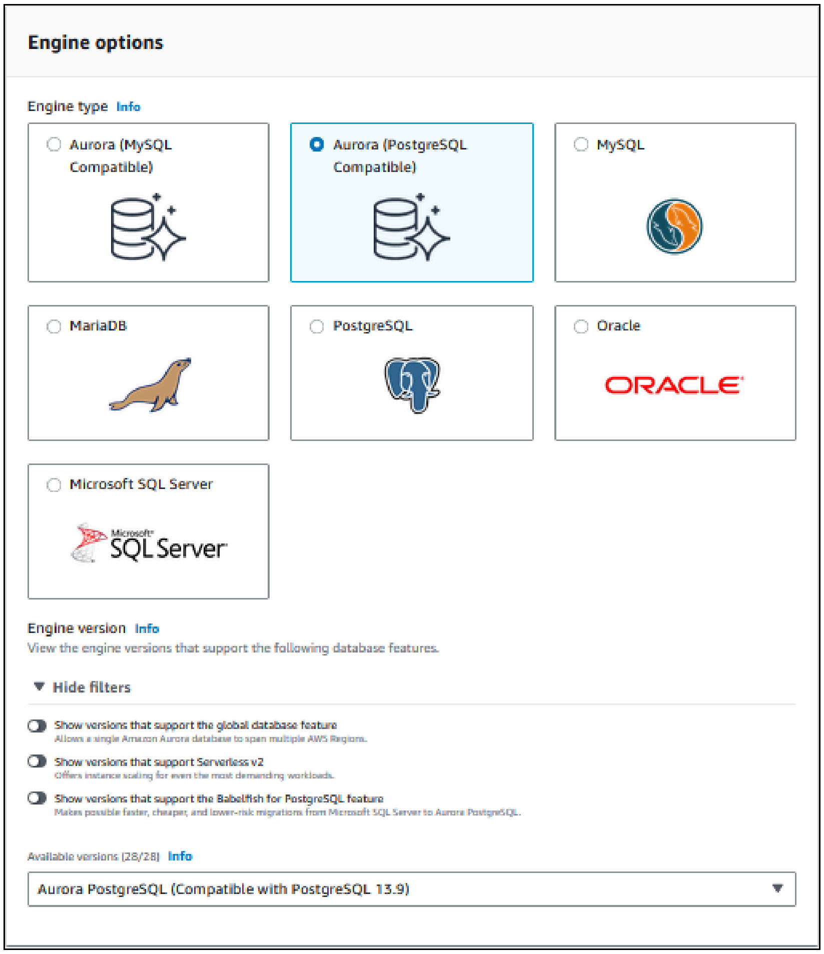
7. Choose other database settings as needed, then choose **Create database.**

Example for Aurora PostgreSQL

Use the following procedure.

To create an Aurora Serverless v1 DB cluster for Aurora PostgreSQL

1. For **Engine type**, choose **Aurora (PostgreSQL Compatibile)**.
2. Choose the Aurora PostgreSQL version, compatible with Aurora Serverless v1, that you want for your DB cluster. The supported versions are shown on the right side of the page.



Engine options

Engine type Info

○ Aurora (MySQL Compatible)

● Aurora (PostgreSQL Compatible)

○ MySQL

○ MariaDB

○ PostgreSQL

○ Oracle

○ Microsoft SQL Server

Engine version Info

View the engine versions that support the following database features.

▼ Hide filters

○ Show versions that support the global database feature
Allows a single Amazon Aurora database to span multiple AWS Regions.

○ Show versions that support Serverless v2
Offers instance scaling for even the most demanding workloads.

○ Show versions that support the Babelfish for PostgreSQL feature
Makes possible faster, cheaper, and lower-risk migrations from Microsoft SQL Server to Aurora PostgreSQL.

Available versions (28/28) Info

Aurora PostgreSQL (Compatible with PostgreSQL 13.9) ▼

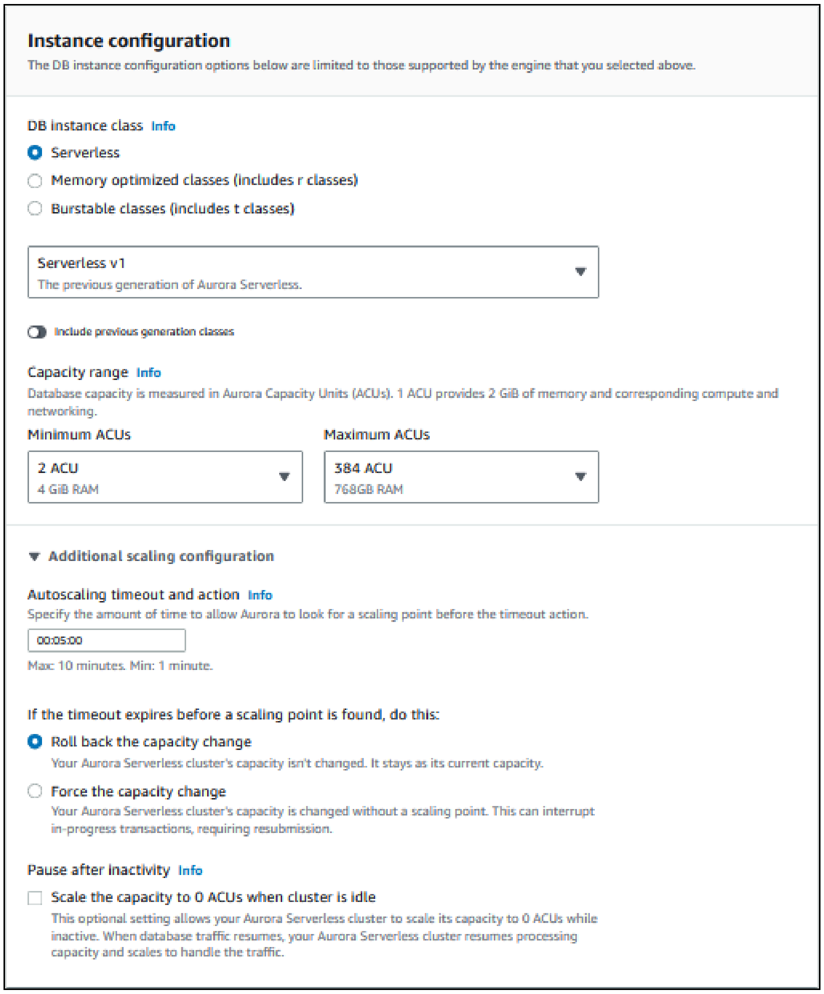
3. For **DB instance class**, choose **Serverless**.

4. If you chose an Aurora PostgreSQL version 13 minor version, choose **Serverless v1** from the menu.

> **Note**
>
> Aurora PostgreSQL version 13 also supports Aurora Serverless v2.

5. Set the **Capacity range** for the DB cluster.

6. Adjust values as needed in the **Additional scaling configuration** section of the page. To learn more about capacity settings, see Autoscaling for Aurora Serverless v1.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

Serverless v1
The previous generation of Aurora Serverless.

Include previous generation classes

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs: 2 ACU, 4 GiB RAM

Maximum ACUs: 384 ACU, 768GB RAM

▼ Additional scaling configuration

Autoscaling timeout and action Info

Specify the amount of time to allow Aurora to look for a scaling point before the timeout action.

00:05:00

Max: 10 minutes. Min: 1 minute.

If the timeout expires before a scaling point is found, do this:

● Roll back the capacity change
Your Aurora Serverless cluster's capacity isn't changed. It stays as its current capacity.

○ Force the capacity change
Your Aurora Serverless cluster's capacity is changed without a scaling point. This can interrupt in-progress transactions, requiring resubmission.

Pause after inactivity Info

□ Scale the capacity to 0 ACUs when cluster is idle
This optional setting allows your Aurora Serverless cluster to scale its capacity to 0 ACUs while inactive. When database traffic resumes, your Aurora Serverless cluster resumes processing capacity and scales to handle the traffic.

7. To use the Data API with your Aurora Serverless v1 DB cluster, select the **Data API** check box under **Additional configuration** in the **Connectivity** section.

   To learn more about the Data API, see Using RDS Data API.

8. Choose other database settings as needed, then choose **Create database.**

AWS CLI

To create a new Aurora Serverless v1 DB cluster with the AWS CLI, run the create-db-cluster command and specify `serverless` for the `--engine-mode` option.

You can optionally specify the `--scaling-configuration` option to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections.

The following command examples create a new Serverless DB cluster by setting the `--engine-mode` option to `serverless`. The examples also specify values for the `--scaling-configuration` option.

Example for Aurora MySQL

The following command creates a new Aurora MySQL–compatible Serverless DB cluster. Valid capacity values for Aurora MySQL are 1, 2, 4, 8, 16, 32, 64, 128, and 256.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster \
    --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.4 \
    --engine-mode serverless \
    --scaling-configuration
 MinCapacity=4,MaxCapacity=32,SecondsUntilAutoPause=1000,AutoPause=true \
    --master-username username --master-user-password password
```

For Windows:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster ^
    --engine aurora-mysql --engine-version 5.7.mysql_aurora.2.11.4 ^
    --engine-mode serverless ^
    --scaling-configuration
 MinCapacity=4,MaxCapacity=32,SecondsUntilAutoPause=1000,AutoPause=true ^
    --master-username username --master-user-password password
```

Example for Aurora PostgreSQL

The following command creates a new PostgreSQL 13.9–compatible Serverless DB cluster. Valid capacity values for Aurora PostgreSQL are 2, 4, 8, 16, 32, 64, 192, and 384.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster \
    --engine aurora-postgresql --engine-version 13.9 \
    --engine-mode serverless \
    --scaling-configuration
 MinCapacity=8,MaxCapacity=64,SecondsUntilAutoPause=1000,AutoPause=true \
    --master-username username --master-user-password password
```

For Windows:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster ^
    --engine aurora-postgresql --engine-version 13.9 ^
    --engine-mode serverless ^
    --scaling-configuration
 MinCapacity=8,MaxCapacity=64,SecondsUntilAutoPause=1000,AutoPause=true ^
    --master-username username --master-user-password password
```

RDS API

To create a new Aurora Serverless v1 DB cluster with the RDS API, run the CreateDBCluster operation and specify `serverless` for the `EngineMode` parameter.

You can optionally specify the `ScalingConfiguration` parameter to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

Restoring an Aurora Serverless v1 DB cluster

You can configure an Aurora Serverless v1 DB cluster when you restore a provisioned DB cluster snapshot with the AWS Management Console, the AWS CLI, or the RDS API.

When you restore a snapshot to an Aurora Serverless v1 DB cluster, you can set the following specific values:

- **Minimum Aurora capacity unit** – Aurora Serverless v1 can reduce capacity down to this capacity unit.
- **Maximum Aurora capacity unit** – Aurora Serverless v1 can increase capacity up to this capacity unit.
- **Timeout action** – The action to take when a capacity modification times out because it can't find a scaling point. Aurora Serverless v1 DB cluster can force your DB cluster to the new capacity settings if set the **Force scaling the capacity to the specified values...** option. Or, it can roll back the capacity change to cancel it if you don't choose the option. For more information, see Timeout action for capacity changes.

- **Pause after inactivity** – The amount of time with no database traffic to scale to zero processing capacity. When database traffic resumes, Aurora automatically resumes processing capacity and scales to handle the traffic.

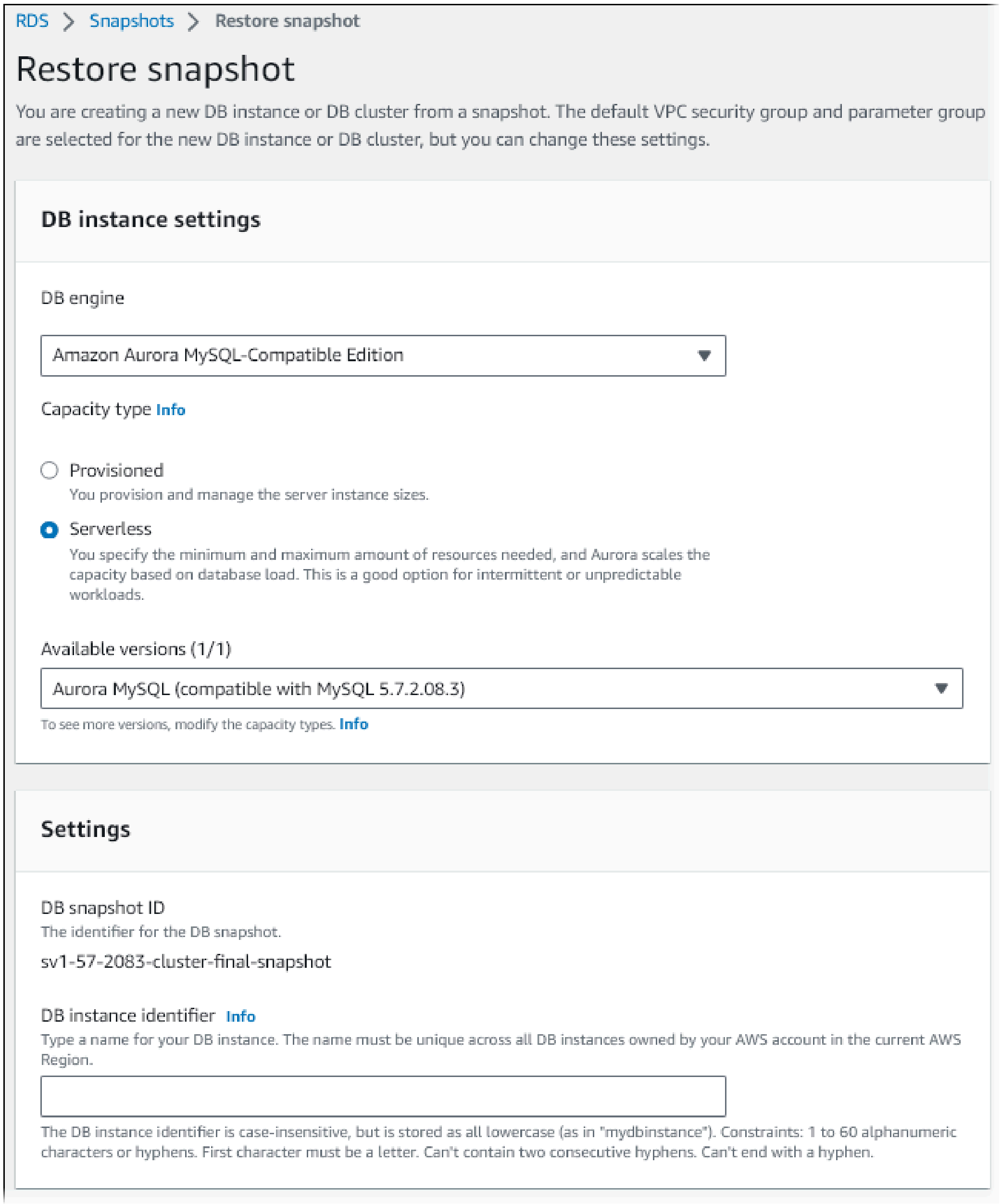
For general information about restoring a DB cluster from a snapshot, see Restoring from a DB cluster snapshot.

Console

You can restore a DB cluster snapshot to an Aurora DB cluster with the AWS Management Console.

To restore a DB cluster snapshot to an Aurora DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region that hosts your source DB cluster.
3. In the navigation pane, choose **Snapshots**, and choose the DB cluster snapshot that you want to restore.
4. For **Actions**, choose **Restore Snapshot**.
5. On the **Restore DB Cluster** page, choose **Serverless** for **Capacity type**.



RDS > Snapshots > Restore snapshot

Restore snapshot

You are creating a new DB instance or DB cluster from a snapshot. The default VPC security group and parameter group are selected for the new DB instance or DB cluster, but you can change these settings.

DB instance settings

DB engine

Amazon Aurora MySQL-Compatible Edition

Capacity type Info

○ Provisioned
You provision and manage the server instance sizes.

● Serverless
You specify the minimum and maximum amount of resources needed, and Aurora scales the capacity based on database load. This is a good option for intermittent or unpredictable workloads.

Available versions (1/1)

Aurora MySQL (compatible with MySQL 5.7.2.08.3)

To see more versions, modify the capacity types. Info

Settings

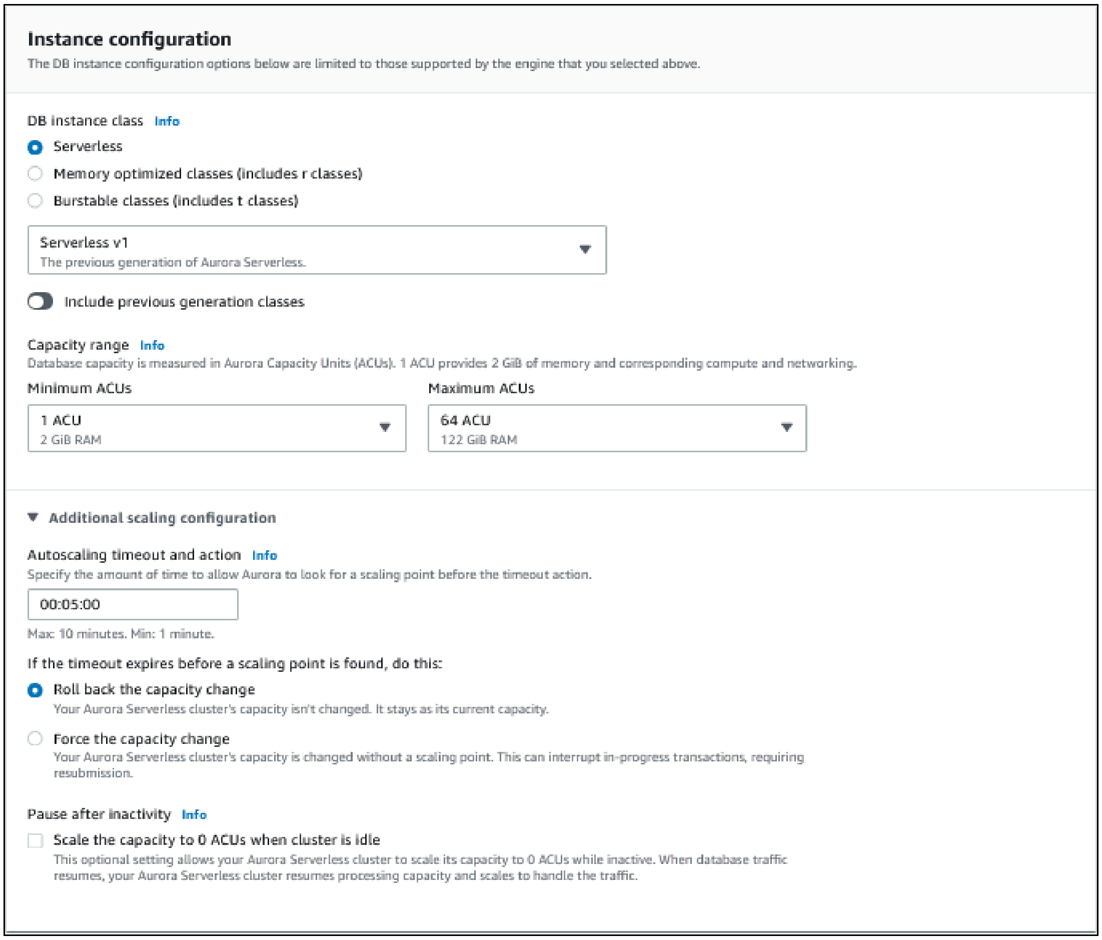
DB snapshot ID
The identifier for the DB snapshot.
sv1-57-2083-cluster-final-snapshot

DB instance identifier Info
Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

6. In the **DB cluster identifier** field, type the name for your restored DB cluster, and complete the other fields.

7. In the **Capacity settings** section, modify the scaling configuration.



Instance configuration

The DB instance configuration options below are limited to those supported by the engine that you selected above.

DB instance class Info

● Serverless

○ Memory optimized classes (includes r classes)

○ Burstable classes (includes t classes)

Serverless v1
The previous generation of Aurora Serverless.

Include previous generation classes

Capacity range Info

Database capacity is measured in Aurora Capacity Units (ACUs). 1 ACU provides 2 GiB of memory and corresponding compute and networking.

Minimum ACUs: 1 ACU, 2 GiB RAM

Maximum ACUs: 64 ACU, 122 GiB RAM

▼ Additional scaling configuration

Autoscaling timeout and action Info

Specify the amount of time to allow Aurora to look for a scaling point before the timeout action.

00:05:00

Max: 10 minutes. Min: 1 minute.

If the timeout expires before a scaling point is found, do this:

● Roll back the capacity change
Your Aurora Serverless cluster's capacity isn't changed. It stays as its current capacity.

○ Force the capacity change
Your Aurora Serverless cluster's capacity is changed without a scaling point. This can interrupt in-progress transactions, requiring resubmission.

Pause after inactivity Info

□ Scale the capacity to 0 ACUs when cluster is idle
This optional setting allows your Aurora Serverless cluster to scale its capacity to 0 ACUs while inactive. When database traffic resumes, your Aurora Serverless cluster resumes processing capacity and scales to handle the traffic.

8. Choose **Restore DB Cluster**.

To connect to an Aurora Serverless v1 DB cluster, use the database endpoint. For details, see the instructions in Connecting to an Amazon Aurora DB cluster.

Note

If you encounter the following error message, your account requires additional permissions: `Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later.` For more information, see Using service-linked roles for Amazon Aurora.

AWS CLI

You can configure an Aurora Serverless DB cluster when you restore a provisioned DB cluster snapshot with the AWS Management Console, the AWS CLI, or the RDS API.

When you restore a snapshot to an Aurora Serverless DB cluster, you can set the following specific values:

- **Minimum Aurora capacity unit** – Aurora Serverless can reduce capacity down to this capacity unit.
- **Maximum Aurora capacity unit** – Aurora Serverless can increase capacity up to this capacity unit.
- **Timeout action** – The action to take when a capacity modification times out because it can't find a scaling point. Aurora Serverless v1 DB cluster can force your DB cluster to the new capacity settings if set the **Force scaling the capacity to the specified values...** option. Or, it can roll back the capacity change to cancel it if you don't choose the option. For more information, see Timeout action for capacity changes.
- **Pause after inactivity** – The amount of time with no database traffic to scale to zero processing capacity. When database traffic resumes, Aurora automatically resumes processing capacity and scales to handle the traffic.

> **Note**
>
> The version of the DB cluster snapshot must be compatible with Aurora Serverless v1. For the list of supported versions, see Supported Regions and Aurora DB engines for Aurora Serverless v1.

To restore a snapshot to an Aurora Serverless v1 cluster with MySQL 5.7 compatibility, include the following additional parameters:

- `--engine` **`aurora-mysql`**
- `--engine-version` **`5.7`**

The `--engine` and `--engine-version` parameters let you create a MySQL 5.7-compatible Aurora Serverless v1 cluster from a MySQL 5.6-compatible Aurora or Aurora Serverless v1 snapshot. The following example restores a snapshot from a MySQL 5.6-compatible cluster named *`mydbclustersnapshot`* to a MySQL 5.7-compatible Aurora Serverless v1 cluster named *`mynewdbcluster`*.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier mynewdbcluster \
    --snapshot-identifier mydbclustersnapshot \
    --engine-mode serverless \
    --engine aurora-mysql \
    --engine-version 5.7
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-instance-identifier mynewdbcluster ^
    --db-snapshot-identifier mydbclustersnapshot ^
    --engine aurora-mysql ^
    --engine-version 5.7
```

You can optionally specify the `--scaling-configuration` option to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

In the following example, you restore from a previously created DB cluster snapshot named *mydbclustersnapshot* to a new DB cluster named *mynewdbcluster*. You set the `--scaling-configuration` so that the new Aurora Serverless v1 DB cluster can scale from 8 ACUs to 64 ACUs (Aurora capacity units) as needed to process the workload. After processing completes and after 1000 seconds with no connections to support, the cluster shuts down until connection requests prompt it to restart.

For Linux, macOS, or Unix:

```
aws rds restore-db-cluster-from-snapshot \
    --db-cluster-identifier mynewdbcluster \
    --snapshot-identifier mydbclustersnapshot \
    --engine-mode serverless --scaling-configuration
 MinCapacity=8,MaxCapacity=64,TimeoutAction='ForceApplyCapacityChange',SecondsUntilAutoPause=10
```

For Windows:

```
aws rds restore-db-cluster-from-snapshot ^
    --db-instance-identifier mynewdbcluster ^
    --db-snapshot-identifier mydbclustersnapshot ^
    --engine-mode serverless --scaling-configuration
 MinCapacity=8,MaxCapacity=64,TimeoutAction='ForceApplyCapacityChange',SecondsUntilAutoPause=10
```

RDS API

To configure an Aurora Serverless v1 DB cluster when you restore from a DB cluster using the RDS API, run the RestoreDBClusterFromSnapshot operation and specify `serverless` for the `EngineMode` parameter.

You can optionally specify the `ScalingConfiguration` parameter to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

Modifying an Aurora Serverless v1 DB cluster

After you configure an Aurora Serverless v1 DB cluster, you can modify certain properties with the AWS Management Console, the AWS CLI, or the RDS API. Most of the properties you can modify are the same as for other kinds of Aurora clusters.

The most relevant changes for Aurora Serverless v1 are the following:

- Modifying the scaling configuration
- Upgrading the major version
- Converting from Aurora Serverless v1 to provisioned

Modifying the scaling configuration of an Aurora Serverless v1 DB cluster

You can set the minimum and maximum capacity for the DB cluster. Each capacity unit is equivalent to a specific compute and memory configuration. Aurora Serverless automatically creates scaling rules for thresholds for CPU utilization, connections, and available memory. You can also set whether Aurora Serverless pauses the database when there's no activity and then resumes when activity begins again.

You can set the following specific values for the scaling configuration:

- **Minimum Aurora capacity unit** – Aurora Serverless can reduce capacity down to this capacity unit.
- **Maximum Aurora capacity unit** – Aurora Serverless can increase capacity up to this capacity unit.
- **Autoscaling timeout and action** – This section specifies how long Aurora Serverless waits to find a scaling point before timing out. It also specifies the action to take when a capacity modification times out because it can't find a scaling point. Aurora can force the capacity change to set the capacity to the specified value as soon as possible. Or, it can roll back the capacity change to cancel it. For more information, see Timeout action for capacity changes.
- **Pause after inactivity** – Use the optional **Scale the capacity to 0 ACUs when cluster is idle** setting to scale the database to zero processing capacity while it's inactive. When database traffic resumes, Aurora automatically resumes processing capacity and scales to handle the traffic.

Console

You can modify the scaling configuration of an Aurora DB cluster with the AWS Management Console.

To modify an Aurora Serverless v1 DB cluster

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora Serverless v1 DB cluster that you want to modify.
4. For **Actions**, choose **Modify cluster**.
5. In the **Capacity settings** section, modify the scaling configuration.
6. Choose **Continue**.
7. On the **Modify DB cluster** page, review your modifications, then choose when to apply them.
8. Choose **Modify cluster**.

AWS CLI

To modify the scaling configuration of an Aurora Serverless v1 DB cluster using the AWS CLI, run the modify-db-cluster AWS CLI command. Specify the `--scaling-configuration` option to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

In this example, you modify the scaling configuration of an Aurora Serverless v1 DB cluster named *`sample-cluster`*.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --scaling-configuration
 MinCapacity=8,MaxCapacity=64,SecondsUntilAutoPause=500,TimeoutAction='ForceApplyCapacityChange
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --scaling-configuration
 MinCapacity=8,MaxCapacity=64,SecondsUntilAutoPause=500,TimeoutAction='ForceApplyCapacityChange
```

RDS API

You can modify the scaling configuration of an Aurora DB cluster with the ModifyDBCluster API operation. Specify the `ScalingConfiguration` parameter to configure the minimum capacity, maximum capacity, and automatic pause when there are no connections. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

Upgrading the major version of an Aurora Serverless v1 DB cluster

You can upgrade the major version for an Aurora Serverless v1 DB cluster compatible with PostgreSQL 11 to a corresponding PostgreSQL 13–compatible version.

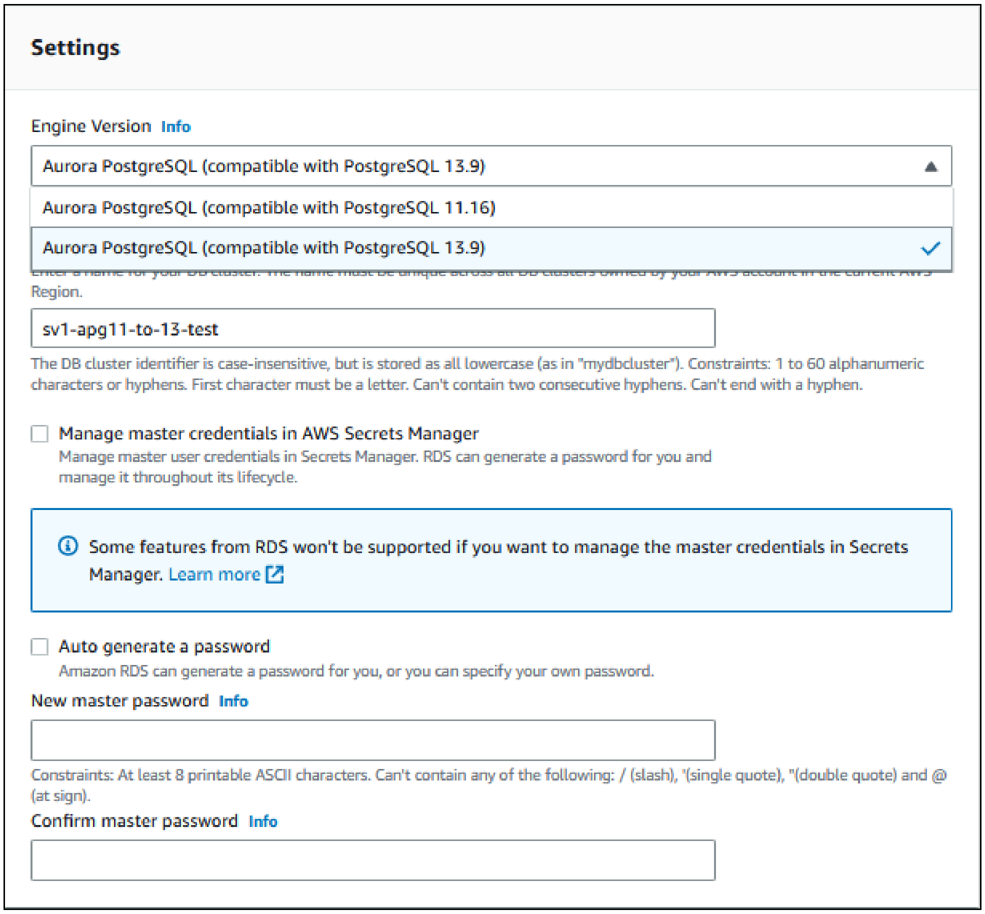
Console

You can perform an in-place upgrade of an Aurora Serverless v1 DB cluster using the AWS Management Console.

To upgrade an Aurora Serverless v1 DB cluster

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora Serverless v1 DB cluster that you want to upgrade.
4. For **Actions**, choose **Modify cluster**.
5. For **Version**, choose an Aurora PostgreSQL version 13 version number.

   The following example shows an in-place upgrade from Aurora PostgreSQL 11.16 to 13.9.



Settings

Engine Version Info

Aurora PostgreSQL (compatible with PostgreSQL 13.9) ▲

Aurora PostgreSQL (compatible with PostgreSQL 11.16)

Aurora PostgreSQL (compatible with PostgreSQL 13.9) ✓

Region.

sv1-apg11-to-13-test

The DB cluster identifier is case-insensitive, but is stored as all lowercase (as in "mydbcluster"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

□ Manage master credentials in AWS Secrets Manager
Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

ⓘ Some features from RDS won't be supported if you want to manage the master credentials in Secrets Manager. Learn more

□ Auto generate a password
Amazon RDS can generate a password for you, or you can specify your own password.

New master password Info

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password Info

If you perform a major version upgrade, leave all of the other properties the same. To change any other properties, do another **Modify** operation after the upgrade finishes.

6. Choose **Continue**.
7. On the **Modify DB cluster** page, review your modifications, then choose when to apply them.
8. Choose **Modify cluster**.

AWS CLI

To perform an in-place upgrade from a PostgreSQL 11–compatible Aurora Serverless v1 DB cluster to a PostgreSQL 13–compatible one, specify the `--engine-version` parameter with an Aurora PostgreSQL version 13 version number that's compatible with Aurora Serverless v1. Also include the `--allow-major-version-upgrade` parameter.

In this example, you modify the major version of a PostgreSQL 11–compatible Aurora Serverless v1 DB cluster named `sample-cluster`. Doing so performs an in-place upgrade to a PostgreSQL 13–compatible Aurora Serverless v1 DB cluster.

```
aws rds modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --engine-version 13.9 \
    --allow-major-version-upgrade
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --engine-version 13.9 ^
    --allow-major-version-upgrade
```

RDS API

To perform an in-place upgrade from a PostgreSQL 11–compatible Aurora Serverless v1 DB cluster to a PostgreSQL 13–compatible one, specify the `EngineVersion` parameter with an Aurora PostgreSQL version 13 version number that's compatible with Aurora Serverless v1. Also include the `AllowMajorVersionUpgrade` parameter.

Converting an Aurora Serverless v1 DB cluster to provisioned

You can convert an Aurora Serverless v1 DB cluster to a provisioned DB cluster. To perform the conversion, you change the DB instance class to **Provisioned**. You can use this conversion as part of upgrading your DB cluster from Aurora Serverless v1 to Aurora Serverless v2. For more information, see Upgrading from an Aurora Serverless v1 cluster to Aurora Serverless v2.

The conversion process creates a reader DB instance in the DB cluster, promotes the reader instance to a writer instance, and then deletes the original Aurora Serverless v1 instance. When you convert the DB cluster, you can't perform any other modifications at the same time, such as changing the DB engine version or DB cluster parameter group. The conversion operation is applied immediately, and can't be undone.

During the conversion, a backup DB cluster snapshot is taken of the DB cluster in case an error occurs. The identifier for the DB cluster snapshot has the form pre-modify-engine-mode-*DB_cluster_identifier*-*timestamp*.

Aurora uses the current default DB minor engine version for the provisioned DB cluster.

If you don't provide a DB instance class for your converted DB cluster, Aurora recommends one based on the maximum capacity of the original Aurora Serverless v1 DB cluster. The recommended capacity to instance class mappings are shown in the following table.

Serverless maximum capacity (ACUs)	Provisioned DB instance class
1	db.t3.small
2	db.t3.medium
4	db.t3.large
8	db.r5.large
16	db.r5.xlarge
32	db.r5.2xlarge
64	db.r5.4xlarge
128	db.r5.8xlarge
192	db.r5.12xlarge
256	db.r5.16xlarge
384	db.r5.24xlarge

Note

Depending on the DB instance class you choose, and your database usage, you might see different costs for a provisioned DB cluster compared to Aurora Serverless v1.

If you convert your Aurora Serverless v1 DB cluster to a burstable (db.t*) DB instance class, you might incur additional costs for using the DB cluster. For more information, see DB instance class types.

AWS CLI

To convert an Aurora Serverless v1 DB cluster to a provisioned cluster, run the modify-db-cluster AWS CLI command.

The following parameters are required:

- `--db-cluster-identifier` – The Aurora Serverless v1 DB cluster that you're converting to provisioned.
- `--engine-mode` – Use the value `provisioned`.
- `--allow-engine-mode-change`
- `--db-cluster-instance-class` – Choose the DB instance class for the provisioned DB cluster based on the capacity of the Aurora Serverless v1 DB cluster.

In this example, you convert an Aurora Serverless v1 DB cluster named `sample-cluster` and use the `db.r5.xlarge` DB instance class.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --engine-mode provisioned \
    --allow-engine-mode-change \
    --db-cluster-instance-class db.r5.xlarge
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --engine-mode provisioned ^
    --allow-engine-mode-change ^
    --db-cluster-instance-class db.r5.xlarge
```

RDS API

To convert an Aurora Serverless v1 DB cluster to a provisioned cluster, use the ModifyDBCluster API operation.

The following parameters are required:

- `DBClusterIdentifier` – The Aurora Serverless v1 DB cluster that you're converting to provisioned.
- `EngineMode` – Use the value `provisioned`.
- `AllowEngineModeChange`
- `DBClusterInstanceClass` – Choose the DB instance class for the provisioned DB cluster based on the capacity of the Aurora Serverless v1 DB cluster.

Scaling Aurora Serverless v1 DB cluster capacity manually

Typically, Aurora Serverless v1 DB clusters scale seamlessly based on the workload. However, capacity might not always scale fast enough to meet sudden extremes, such as an exponential increase in transactions. In such cases you can initiate the scaling operation manually by setting a new capacity value. After you set the capacity explicitly, Aurora Serverless v1 automatically scales the DB cluster. It does so based on the cooldown period for scaling down.

You can explicitly set the capacity of an Aurora Serverless v1 DB cluster to a specific value with the AWS Management Console, the AWS CLI, or the RDS API.

Console

You can set the capacity of an Aurora DB cluster with the AWS Management Console.

To modify an Aurora Serverless v1 DB cluster

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the Aurora Serverless v1 DB cluster that you want to modify.
4. For **Actions**, choose **Set capacity**.
5. In the **Scale database capacity** window, choose the following:

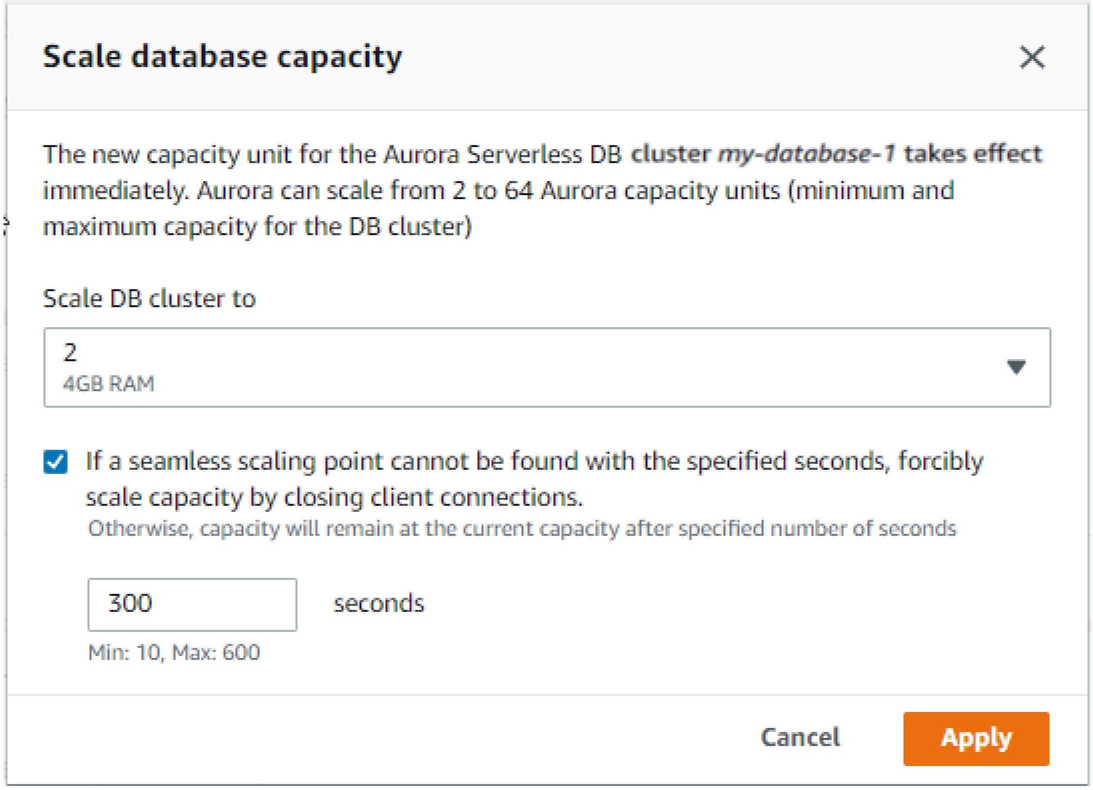
    a. For the **Scale DB cluster to** drop-down selector, choose the new capacity that you want for your DB cluster.

    b. For the **If a seamless scaling point cannot be found** check box, choose the behavior that you want for your Aurora Serverless v1 DB cluster's `TimeoutAction` setting, as follows:

    - Clear this option if you want your capacity to remain unchanged if Aurora Serverless v1 can't find a scaling point before timing out.

  - Select this option if you want to force your Aurora Serverless v1 DB cluster change its capacity even if it can't find a scaling point before timing out. This option can result Aurora Serverless v1 dropping connections that prevent it from finding a scaling point.

c. For **seconds**, enter the amount of time you want to allow your Aurora Serverless v1 DB cluster to look for a scaling point before timing out. You can specify anywhere from 10 seconds to 600 seconds (10 minutes). The default is five minutes (300 seconds). This following example forces the Aurora Serverless v1 DB cluster to scale down to 2 ACUs even if it can't find a scaling point within five minutes.



Scale database capacity

The new capacity unit for the Aurora Serverless DB cluster *my-database-1* takes effect immediately. Aurora can scale from 2 to 64 Aurora capacity units (minimum and maximum capacity for the DB cluster)

Scale DB cluster to

2
4GB RAM

☑ If a seamless scaling point cannot be found with the specified seconds, forcibly scale capacity by closing client connections.
Otherwise, capacity will remain at the current capacity after specified number of seconds

300 seconds
Min: 10, Max: 600

6. Choose **Apply**.

To learn more about scaling points, `TimeoutAction`, and cooldown periods, see Autoscaling for Aurora Serverless v1.

AWS CLI

To set the capacity of an Aurora Serverless v1 DB cluster using the AWS CLI, run the modify-current-db-cluster-capacity AWS CLI command, and specify the `--capacity` option. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.

- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

In this example, you set the capacity of an Aurora Serverless v1 DB cluster named *sample-cluster* to *64*.

```
aws rds modify-current-db-cluster-capacity --db-cluster-identifier sample-cluster --
capacity 64
```

RDS API

You can set the capacity of an Aurora DB cluster with the ModifyCurrentDBClusterCapacity API operation. Specify the `Capacity` parameter. Valid capacity values include the following:

- Aurora MySQL: 1, 2, 4, 8, 16, 32, 64, 128, and 256.
- Aurora PostgreSQL: 2, 4, 8, 16, 32, 64, 192, and 384.

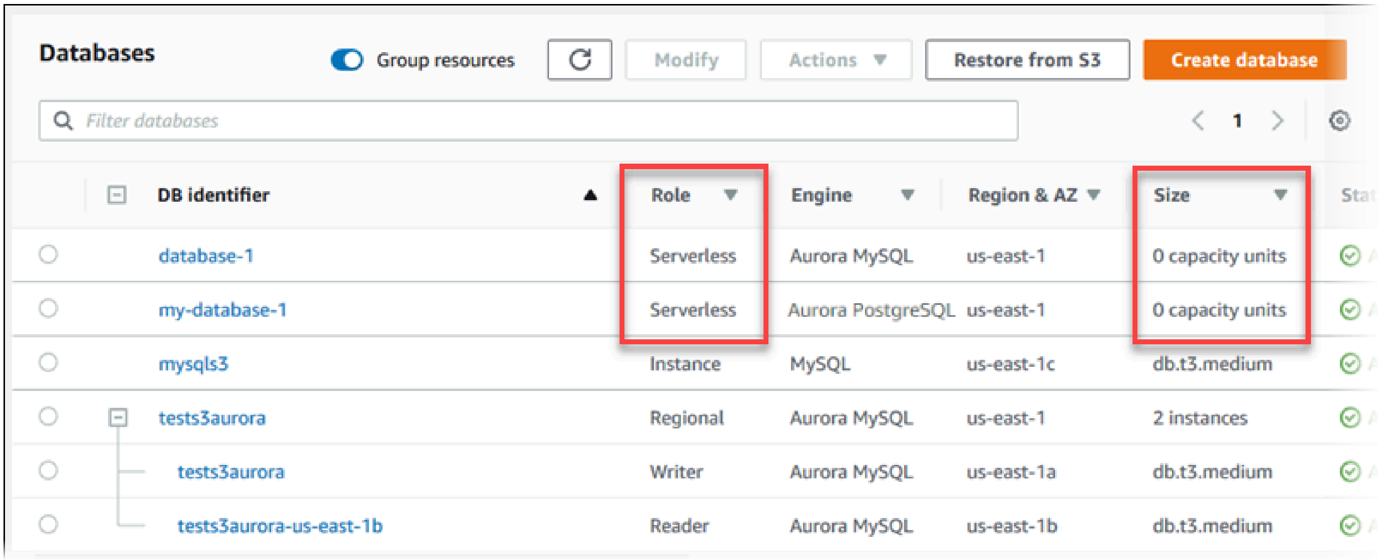
Viewing Aurora Serverless v1 DB clusters

After you create one or more Aurora Serverless v1 DB clusters, you can view which DB clusters are type **Serverless** and which are type **Instance**. You can also view the current number of Aurora capacity units (ACUs) each Aurora Serverless v1 DB cluster is using. Each ACU is a combination of processing (CPU) and memory (RAM) capacity.

To view your Aurora Serverless v1 DB clusters

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you created the Aurora Serverless v1 DB clusters.
3. In the navigation pane, choose **Databases**.

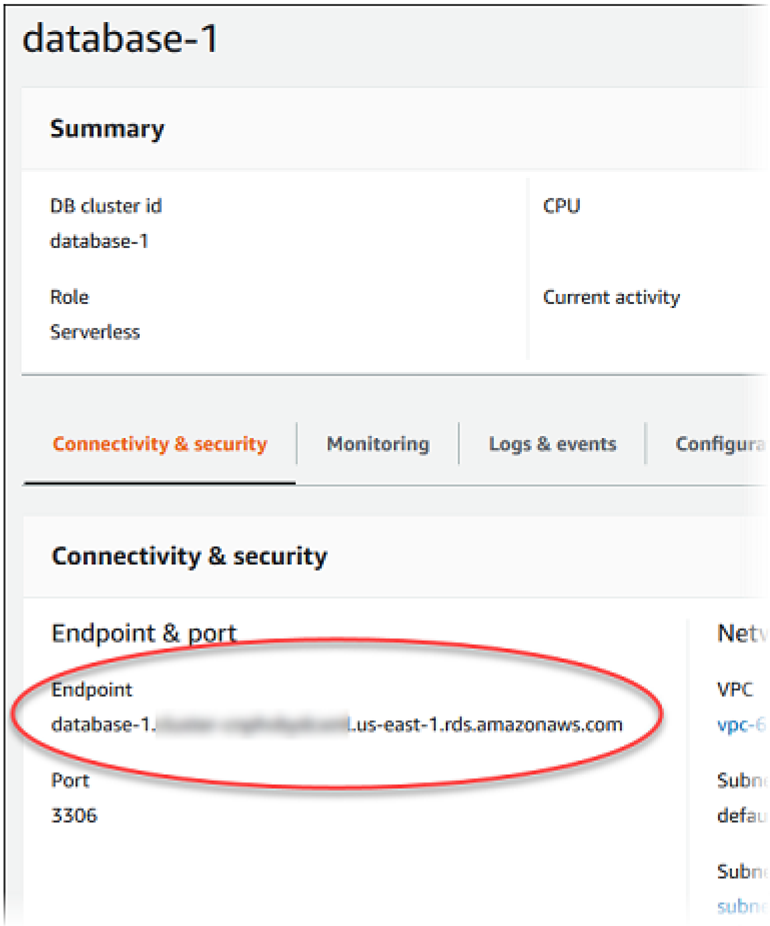
   For each DB cluster, the DB cluster type is shown under **Role**. The Aurora Serverless v1 DB clusters show **Serverless** for the type. You can view an Aurora Serverless v1 DB cluster's current capacity under **Size**.



	DB identifier ▲	Role ▼	Engine ▼	Region & AZ ▼	Size ▼	Sta
○	database-1	Serverless	Aurora MySQL	us-east-1	0 capacity units	✓
○	my-database-1	Serverless	Aurora PostgreSQL	us-east-1	0 capacity units	✓
○	mysqls3	Instance	MySQL	us-east-1c	db.t3.medium	✓
○	⊟ tests3aurora	Regional	Aurora MySQL	us-east-1	2 instances	✓
○	tests3aurora	Writer	Aurora MySQL	us-east-1a	db.t3.medium	✓
○	tests3aurora-us-east-1b	Reader	Aurora MySQL	us-east-1b	db.t3.medium	✓

4. Choose the name of an Aurora Serverless v1 DB cluster to display its details.

   On the **Connectivity & security** tab, note the database endpoint. Use this endpoint to connect to your Aurora Serverless v1 DB cluster.



database-1

Summary

DB cluster id
database-1

Role
Serverless

CPU

Current activity

Connectivity & security

Endpoint & port

Endpoint
database-1.[redacted].us-east-1.rds.amazonaws.com

Port
3306

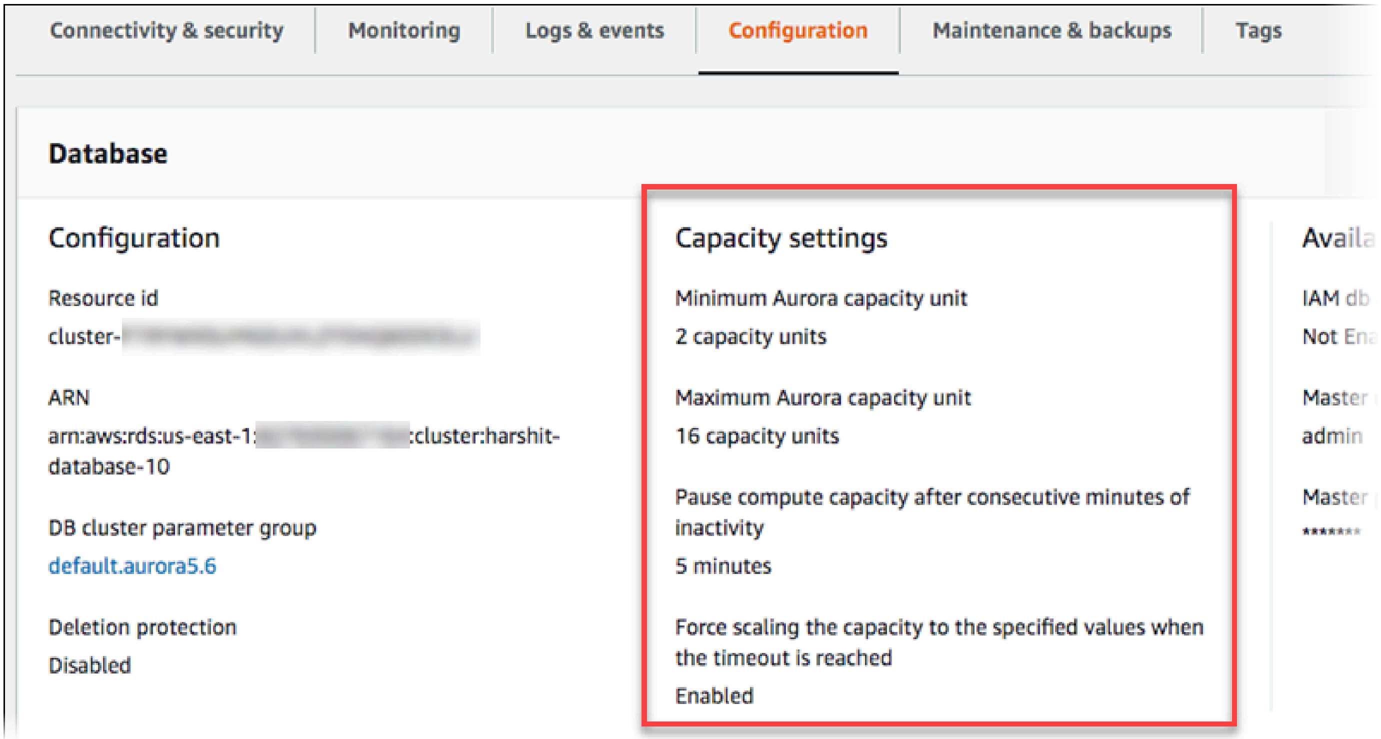
Netv

VPC
vpc-6

Subn
defau

Subn
subn

Choose the **Configuration** tab to view the capacity settings.



Database

Configuration

Resource id

cluster-[redacted]

ARN

arn:aws:rds:us-east-1:[redacted]:cluster:harshit-database-10

DB cluster parameter group

default.aurora5.6

Deletion protection

Disabled

Capacity settings

Minimum Aurora capacity unit

2 capacity units

Maximum Aurora capacity unit

16 capacity units

Pause compute capacity after consecutive minutes of inactivity

5 minutes

Force scaling the capacity to the specified values when the timeout is reached

Enabled

Availa

IAM db

Not Ena

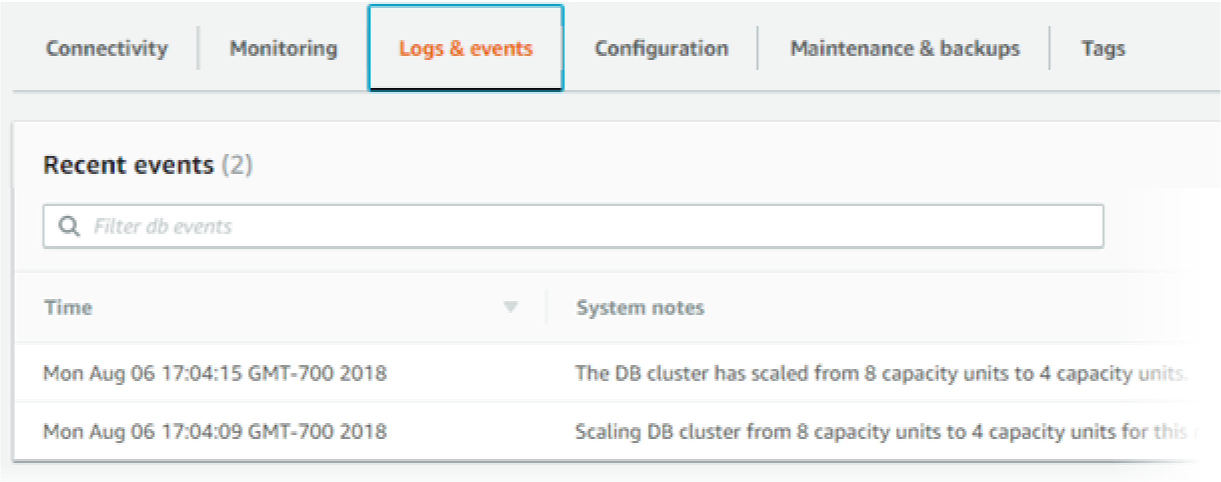
Master

admin

Master

****

A *scaling event* is generated when the DB cluster scales up, scales down, pauses, or resumes. Choose the **Logs & events** tab to see recent events. The following image shows examples of these events.



Recent events (2)

Filter db events

Time	System notes
Mon Aug 06 17:04:15 GMT-700 2018	The DB cluster has scaled from 8 capacity units to 4 capacity units.
Mon Aug 06 17:04:09 GMT-700 2018	Scaling DB cluster from 8 capacity units to 4 capacity units for this

Monitoring capacity and scaling events for your Aurora Serverless v1 DB cluster

You can view your Aurora Serverless v1 DB cluster in CloudWatch to monitor the capacity allocated to the DB cluster with the `ServerlessDatabaseCapacity` metric. You can also monitor all of the standard Aurora CloudWatch metrics, such as `CPUUtilization`, `DatabaseConnections`, `Queries`, and so on.

You can have Aurora publish some or all database logs to CloudWatch. You select the logs to publish by enabling the configuration parameters such as `general_log` and `slow_query_log` in the DB cluster parameter group associated with theAurora Serverless v1 cluster. Unlike provisioned clusters, Aurora Serverless v1 clusters don't require you to specify in the DB cluster settings which log types to upload to CloudWatch. Aurora Serverless v1 clusters automatically upload all the available logs. When you disable a log configuration parameter, publishing of the log to CloudWatch stops. You can also delete the logs in CloudWatch if they are no longer needed.

To get started with Amazon CloudWatch for your Aurora Serverless v1 DB cluster, see Viewing Aurora Serverless v1 logs with Amazon CloudWatch. To learn more about how to monitor Aurora DB clusters through CloudWatch, see Monitoring log events in Amazon CloudWatch.

To connect to an Aurora Serverless v1 DB cluster, use the database endpoint. For more information, see Connecting to an Amazon Aurora DB cluster.

Note

You can't connect directly to specific DB instances in your Aurora Serverless v1 DB clusters.

Deleting an Aurora Serverless v1 DB cluster

When you create an Aurora Serverless v1 DB cluster using the AWS Management Console, the **Enable default protection** option is enabled by default unless you deselect it. That means that you can't immediately delete an Aurora Serverless v1 DB cluster that has **Deletion protection** enabled. To delete Aurora Serverless v1 DB clusters that have deletion protection by using the AWS Management Console, you first modify the cluster to remove this protection. For information about using the AWS CLI for this task, see AWS CLI.

To disable deletion protection using the AWS Management Console

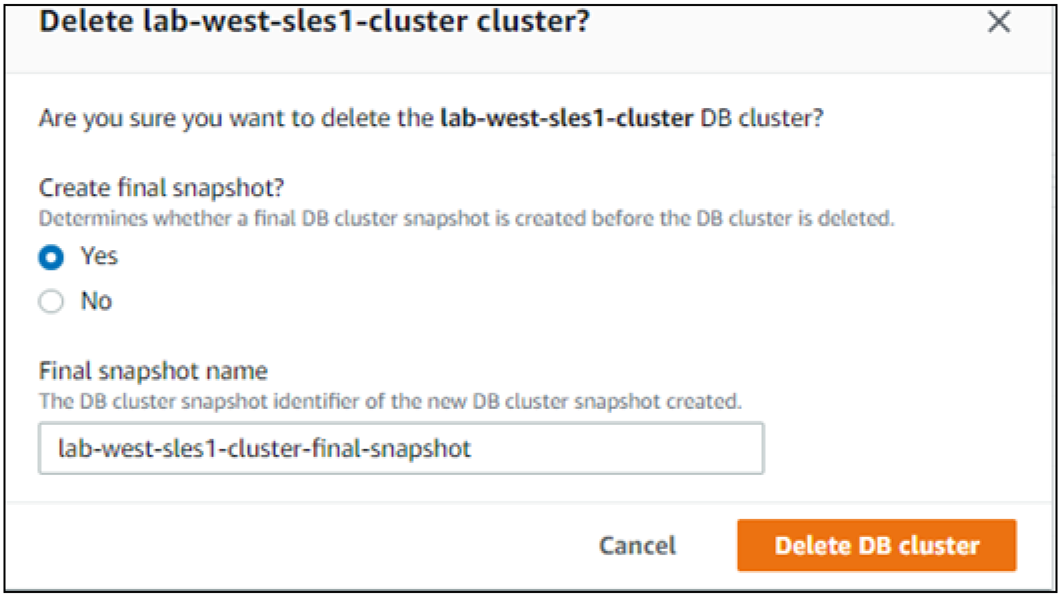
1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **DB clusters**.
3. Choose your Aurora Serverless v1 DB cluster from the list.
4. Choose **Modify** to open your DB cluster's configuration. The Modify DB cluster page opens the Settings, Capacity settings, and other configuration details for your Aurora Serverless v1 DB cluster. Deletion protection is in the **Additional configuration** section.
5. Clear the **Enable deletion protection** check box in the **Additional configuration** properties card.
6. Choose **Continue**. The **Summary of modifications** appears.
7. Choose **Modify cluster** to accept the summary of modifications. You can also choose **Back** to modify your changes or **Cancel** to discard your changes.

After deletion protection is no longer active, you can delete your Aurora Serverless v1 DB cluster by using the AWS Management Console.

Console

To delete an Aurora Serverless v1 DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the **Resources** section, choose **DB Clusters**.
3. Choose the Aurora Serverless v1 DB cluster that you want to delete.
4. For **Actions**, choose **Delete**. You're prompted to confirm that you want to delete your Aurora Serverless v1 DB cluster.
5. We recommend that you keep the preselected options:
   - **Yes** for **Create final snapshot?**
   - Your Aurora Serverless v1 DB cluster name plus `-final-snapshot` for **Final snapshot name**. However, you can change the name for your final snapshot in this field.



Delete lab-west-sles1-cluster cluster?

Are you sure you want to delete the **lab-west-sles1-cluster** DB cluster?

Create final snapshot?
Determines whether a final DB cluster snapshot is created before the DB cluster is deleted.

● Yes

○ No

Final snapshot name
The DB cluster snapshot identifier of the new DB cluster snapshot created.

lab-west-sles1-cluster-final-snapshot

If you choose **No** for **Create final snapshot?** you can't restore your DB cluster using snapshots or point-in-time recovery.

6. Choose **Delete DB cluster**.

Aurora Serverless v1 deletes your DB cluster. If you chose to have a final snapshot, you see your Aurora Serverless v1 DB cluster's status change to "Backing-up" before it's deleted and no longer appears in the list.

AWS CLI

Before you begin, configure your AWS CLI with your AWS Access Key ID, AWS Secret Access Key, and the AWS Region where your Aurora Serverless v1 DB cluster is. For more information, see Configuration basics in the AWS Command Line Interface User Guide.

You can't delete an Aurora Serverless v1 DB cluster until after you first disable deletion protection for clusters configured with this option. If you try to delete a cluster that has this protection option enabled, you see the following error message.

```
An error occurred (InvalidParameterCombination) when calling the DeleteDBCluster
  operation: Cannot delete protected Cluster, please disable deletion protection and
 try again.
```

You can change your Aurora Serverless v1 DB cluster's deletion-protection setting by using the modify-db-cluster AWS CLI command as shown in the following:

```
aws rds modify-db-cluster --db-cluster-identifier your-cluster-name --no-deletion-
protection
```

This command returns the revised properties for the specified DB cluster. You can now delete your Aurora Serverless v1 DB cluster.

We recommend that you always create a final snapshot whenever you delete an Aurora Serverless v1 DB cluster. The following example of using the AWS CLI delete-db-cluster shows you how. You provide the name of your DB cluster and a name for the snapshot.

For Linux, macOS, or Unix:

```
aws rds delete-db-cluster --db-cluster-identifier \
  your-cluster-name --no-skip-final-snapshot \
  --final-db-snapshot-identifier name-your-snapshot
```

For Windows:

```
aws rds delete-db-cluster --db-cluster-identifier ^
  your-cluster-name --no-skip-final-snapshot ^
  --final-db-snapshot-identifier name-your-snapshot
```

Aurora Serverless v1 and Aurora database engine versions

Aurora Serverless v1 is available in certain AWS Regions and for specific Aurora MySQL and Aurora PostgreSQL versions only. For the current list of AWS Regions that support Aurora Serverless v1 and the specific Aurora MySQL and Aurora PostgreSQL versions available in each Region, see Supported Regions and Aurora DB engines for Aurora Serverless v1.

Aurora Serverless v1 uses its associated Aurora database engine to identify specific supported releases for each database engine supported, as follows:

- Aurora MySQL Serverless
- Aurora PostgreSQL Serverless

When minor releases of the database engines become available for Aurora Serverless v1, they are applied automatically in the various AWS Regions where Aurora Serverless v1 is available. In other words, you don't need to upgrade your Aurora Serverless v1 DB cluster to get a new minor release for your cluster's DB engine when it's available for Aurora Serverless v1.

Aurora MySQL Serverless

If you want to use Aurora MySQL-Compatible Edition for your Aurora Serverless v1 DB cluster, you can choose an Aurora MySQL version 2 that's compatible with MySQL 5.7. To learn about enhancements and bug fixes for Aurora MySQL version 2, see Database engine updates for Amazon Aurora MySQL version 2 in the *Release Notes for Aurora MySQL*.

Aurora PostgreSQL Serverless

If you want to use Aurora PostgreSQL for your Aurora Serverless v1 DB cluster, you can choose among Aurora PostgreSQL 11-compatible and 13-compatible versions. Minor releases for Aurora PostgreSQL-Compatible Edition include only changes that are backward-compatible. Your Aurora Serverless v1 DB cluster is transparently upgraded when an Aurora PostgreSQL minor release becomes available for Aurora Serverless v1 in your AWS Region.

For example, the minor version Aurora PostgreSQL 11.16 release was transparently applied to all Aurora Serverless v1 DB clusters running the previous Aurora PostgreSQL version. For more information about the Aurora PostgreSQL version 11.16 update, see PostgreSQL 11.16 in the *Release Notes for Aurora PostgreSQL*.

Using RDS Data API

By using RDS Data API (Data API), you can work with a web-services interface to your Aurora DB cluster. Data API doesn't require a persistent connection to the DB cluster. Instead, it provides a secure HTTP endpoint and integration with AWS SDKs. You can use the endpoint to run SQL statements without managing connections.

Users don't need to pass credentials with calls to Data API, because Data API uses database credentials stored in AWS Secrets Manager. To store credentials in Secrets Manager, users must be granted the appropriate permissions to use Secrets Manager, and also Data API. For more information about authorizing users, see Authorizing access to RDS Data API.

You can also use Data API to integrate Amazon Aurora with other AWS applications such as AWS Lambda, AWS AppSync, and AWS Cloud9. Data API provides a more secure way to use AWS Lambda. It enables you to access your DB cluster without your needing to configure a Lambda function to access resources in a virtual private cloud (VPC). For more information, see AWS Lambda, AWS AppSync, and AWS Cloud9.

You can enable Data API when you create the Aurora DB cluster. You can also modify the configuration later. For more information, see Enabling RDS Data API.

After you enable Data API, you can also use the query editor to run ad hoc queries without configuring a query tool to access Aurora in a VPC. For more information, see Using the Aurora query editor.

Topics

- Logging RDS Data API calls with AWS CloudTrail

Region and version availability

For information about the Regions and engine versions available for Data API, see the following sections.

Cluster type	Region and version availability
Aurora PostgreSQL provisioned and Serverless v2	Data API with Aurora PostgreSQL Serverless v2 and provisioned
Aurora PostgreSQL Serverless v1	Data API with Aurora PostgreSQL Serverless v1
Aurora MySQL Serverless v1	Data API with Aurora MySQL Serverless v1

> **Note**
>
> Currently, Data API isn't available for provisioned or Aurora Serverless v2 DB clusters that use the MySQL engine.

If you require cryptographic modules validated by FIPS 140-2 when accessing Data API through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

Limitations with RDS Data API

RDS Data API (Data API) has the following limitations:

- You can only execute Data API queries on writer instances in a DB cluster. However, writer instances can accept both write and read queries.
- With Aurora global databases, you can enable Data API on both primary and secondary DB clusters. However, until a secondary cluster is promoted to be the primary, it has no writer

instance. Thus, Data API queries that you send to the secondary fail. After a promoted secondary has an available writer instance, Data API queries on that DB instance should succeed.

- Performance Insights doesn't support monitoring database queries that you make using Data API.
- Data API isn't supported on T DB instance classes.
- For Aurora Serverless v2 and provisioned DB clusters that use the PostgreSQL engine, RDS Data API doesn't support some data types. For the list of supported types, see the section called “Comparison with Serverless v2 and provisioned, and Aurora Serverless v1”.
- For Aurora PostgreSQL version 14 and higher databases, Data API only supports scram-sha-256 for password encryption.

Comparison of RDS Data API with Serverless v2 and provisioned, and Aurora Serverless v1

The following table describes differences between RDS Data API (Data API) with Aurora PostgreSQL Serverless v2 and provisioned DB clusters, and Aurora Serverless v1 DB clusters. Aurora Serverless v1 DB clusters use the `serverless` engine mode. Provisioned DB clusters use the `provisioned` engine mode. An Aurora Serverless v2 DB cluster also uses the `provisioned` engine mode, and contains one or more Aurora Serverless v2 DB instances with the `db.serverless` instance class.

Difference	Aurora PostgreSQL Serverless v2 and provisioned	Aurora Serverless v1
Maximum number of requests per second	Unlimited	1,000
Enabling or disabling Data API on an existing database by using the RDS API or AWS CLI	• RDS API – Use the `EnableHttpEndpoint` and `DisableHttpEndpoint` operations. • AWS CLI – Use the `enable-http-endpoint` and `disable-http-endpoint` operations.	• RDS API – Use the `ModifyDBCluster` operation, and specify `true` or `false`, as applicable, for the `EnableHttpEndpoint` parameter. • AWS CLI – Use the `modify-db-cluster` operation

Difference	Aurora PostgreSQL Serverless v2 and provisioned	Aurora Serverless v1
		with the `--enable-http-endpoint` or `--no-enable-http-endpoint` option, as applicable.
CloudTrail events	Events from Data API calls are data events. These events are automatically excluded in a trail by default. For more information, see the section called "Including Data API events in an CloudTrail trail".	Events from Data API calls are management events. These events are automatically included in a trail by default. For more information, see the section called "Excluding Data API events from a CloudTrail trail (Aurora Serverless v1 only)".
Multistatement support	Multistatements aren't supported. In this case, Data API throws `ValidationException: Multistatements aren't supported.`	For Aurora PostgreSQL, multistatements return only the first query response. For Aurora MySQL, multistatements aren't supported.
BatchExecuteStatement	The generated fields object in the update result is empty.	The generated fields object in the update result includes inserted values.
ExecuteSQL	Not supported	Deprecated

Difference	Aurora PostgreSQL Serverless v2 and provisioned	Aurora Serverless v1
ExecuteStatement	ExecuteStatement doesn't support retrieving multidimentional array columns. In this case, Data API throws UnsupportedResultException . Data API doesn't support some data types, such as geometric and monetary types. In this case, Data API throws UnsupportedResultException: The result contains the unsupported data type *data_type* . Only the following types are supported: - BOOL - BYTEA - DATE - CIDR - DECIMAL, NUMERIC - ENUM - FLOAT8, DOUBLE PRECISION - INET - INT, INT4, SERIAL - INT2, SMALLINT, SMALLSERIAL	ExecuteStatement supports retrieving multidimentional array columns and all advanced data types.

Difference	Aurora PostgreSQL Serverless v2 and provisioned	Aurora Serverless v1
	• INT8, BIGINT, BIGSERIAL • JSONB, JSON • REAL, FLOAT • TEXT, CHAR(N), VARCHAR, NAME • TIME • TIMESTAMP • UUID • VECTOR Only the following array types are supported: • BOOL[], BIT[] • DATE[] • DECIMAL[] , NUMERIC[] • FLOAT8[], DOUBLE PRECISION[] • INT[], INT4[] • INT2[] • INT8[], BIGINT[] • JSON[] • REAL[], FLOAT[] • TEXT[], CHAR(N)[] , VARCHAR[] , NAME[] • TIME[] • TIMESTAMP[] • UUID[]	

Authorizing access to RDS Data API

Users can invoke RDS Data API (Data API) operations only if they are authorized to do so. You can give a user permission to use Data API by attaching an AWS Identity and Access Management (IAM) policy that defines their privileges. You can also attach the policy to a role if you're using IAM roles. An AWS managed policy, `AmazonRDSDataFullAccess`, includes permissions for Data API.

The `AmazonRDSDataFullAccess` policy also includes permissions for the user to get the value of a secret from AWS Secrets Manager. Users need to use Secrets Manager to store secrets that they can use in their calls to Data API. Using secrets means that users don't need to include database credentials for the resources that they target in their calls to Data API. Data API transparently calls Secrets Manager, which allows (or denies) the user's request for the secret. For information about setting up secrets to use with Data API, see Storing database credentials in AWS Secrets Manager.

The `AmazonRDSDataFullAccess` policy provides complete access (through Data API) to resources. You can narrow the scope by defining your own policies that specify the Amazon Resource Name (ARN) of a resource.

For example, the following policy shows an example of the minimum required permissions for a user to access Data API for the DB cluster identified by its ARN. The policy includes the needed permissions to access Secrets Manager and get authorization to the DB instance for the user.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SecretsManagerDbCredentialsAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:*:*:secret:rds-db-credentials/*"
        },
        {
            "Sid": "RDSDataServiceAccess",
            "Effect": "Allow",
            "Action": [
                "rds-data:BatchExecuteStatement",
                "rds-data:BeginTransaction",
                "rds-data:CommitTransaction",
                "rds-data:ExecuteStatement",
                "rds-data:RollbackTransaction"
            ],
            "Resource": "arn:aws:rds:us-east-2:111122223333:cluster:prod"
        }
    ]
}
```

We recommend that you use a specific ARN for the "Resources" element in your policy statements (as shown in the example) rather than a wildcard (*).

Working with tag-based authorization

RDS Data API (Data API) and Secrets Manager both support tag-based authorization. *Tags* are key-value pairs that label a resource, such as an RDS cluster, with an additional string value, for example:

- `environment:production`
- `environment:development`

You can apply tags to your resources for cost allocation, operations support, access control, and many other reasons. (If you don't already have tags on your resources and you want to apply them, you can learn more at Tagging Amazon RDS resources.) You can use the tags in your policy statements to limit access to the RDS clusters that are labeled with these tags. As an example, an Aurora DB cluster might have tags that identify its environment as either production or development.

The following example shows how you can use tags in your policy statements. This statement requires that both the cluster and the secret passed in the Data API request have an `environment:production` tag.

Here's how the policy is applied: When a user makes a call using Data API, the request is sent to the service. Data API first verifies that the cluster ARN passed in the request is tagged with `environment:production`. It then calls Secrets Manager to retrieve the value of the user's secret in the request. Secrets Manager also verifies that the user's secret is tagged with `environment:production`. If so, Data API then uses the retrieved value for the user's DB password. Finally, if that's also correct, the Data API request is invoked successfully for the user.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "SecretsManagerDbCredentialsAccess",
            "Effect": "Allow",
            "Action": [
                 "secretsmanager:GetSecretValue"
               ],
            "Resource": "arn:aws:secretsmanager:*:*:secret:rds-db-credentials/*",
            "Condition": {
                    "StringEquals": {
                        "aws:ResourceTag/environment": [
                                         "production"
                                        ]
                     }
             }
        },
        {
            "Sid": "RDSDataServiceAccess",
            "Effect": "Allow",
            "Action": [
                  "rds-data:*"
               ],
            "Resource": "arn:aws:rds:us-east-2:111122223333:cluster:*",
            "Condition": {
                    "StringEquals": {
                        "aws:ResourceTag/environment": [
                                         "production"
                                        ]
                     }
             }
         }
     ]
}
```

The example shows separate actions for `rds-data` and `secretsmanager` for Data API and Secrets Manager. However, you can combine actions and define tag conditions in many different ways to support your specific use cases. For more information, see Using identity-based policies (IAM policies) for Secrets Manager.

In the "Condition" element of the policy, you can choose tag keys from among the following:

- `aws:TagKeys`
- `aws:ResourceTag/${TagKey}`

To learn more about resource tags and how to use `aws:TagKeys`, see Controlling access to AWS resources using resource tags.

> **Note**
>
> Both Data API and AWS Secrets Manager authorize users. If you don't have permissions for all actions defined in a policy, you get an `AccessDeniedException` error.

Storing database credentials in AWS Secrets Manager

When you call RDS Data API (Data API), you pass credentials for the Aurora DB cluster by using a secret in Secrets Manager. To pass credentials in this way, you specify the name of the secret or the Amazon Resource Name (ARN) of the secret.

To store DB cluster credentials in a secret

1. Use Secrets Manager to create a secret that contains credentials for the Aurora DB cluster.

   For instructions, see Create a database secret in the *AWS Secrets Manager User Guide*.
2. Use the Secrets Manager console to view the details for the secret you created, or run the `aws secretsmanager describe-secret` AWS CLI command.

   Note the name and ARN of the secret. You can use them in calls to Data API.

For more information about using Secrets Manager, see the AWS Secrets Manager User Guide.

To understand how Amazon Aurora manages identity and access management, see How Amazon Aurora works with IAM.

For more information about creating an IAM policy, see Creating IAM Policies in the *IAM User Guide*. For information about adding an IAM policy to a user, see Adding and Removing IAM Identity Permissions in the *IAM User Guide*.

Enabling RDS Data API

To use RDS Data API (Data API), enable it for your Aurora DB cluster. You can enable Data API when you create or modify the DB cluster.

Note

For Aurora PostgreSQL, Data API is supported with Aurora Serverless v2, Aurora Serverless v1, and provisioned databases. For Aurora MySQL, Data API is only supported with Aurora Serverless v1 databases.

Topics

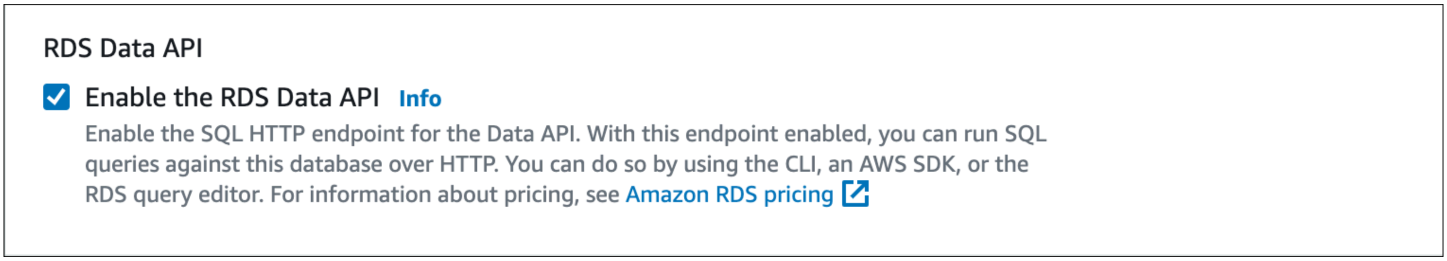
- Enabling RDS Data API when you create a database
- Enabling RDS Data API on an existing database

Enabling RDS Data API when you create a database

While you are creating a database that supports RDS Data API (Data API), you can enable this feature. The following procedures describe how to do so when you use the AWS Management Console, the AWS CLI, or the RDS API.

Console

To enable Data API when you create a DB cluster, select the **Enable the RDS Data API** checkbox in the **Connectivity** section of the **Create database** page, as in the following screenshot.



RDS Data API

☑ Enable the RDS Data API Info

Enable the SQL HTTP endpoint for the Data API. With this endpoint enabled, you can run SQL queries against this database over HTTP. You can do so by using the CLI, an AWS SDK, or the RDS query editor. For information about pricing, see Amazon RDS pricing

For instructions on how to create an Aurora DB cluster that can use the RDS Data API, see the following:

- For Aurora PostgreSQL Serverless v2 and provisioned clusters – Creating an Amazon Aurora DB cluster
- For Aurora Serverless v1 – Creating an Aurora Serverless v1 DB cluster

AWS CLI

To enable Data API while you're creating an Aurora DB cluster, run the create-db-cluster AWS CLI command with the `--enable-http-endpoint` option.

The following example creates an Aurora PostgreSQL DB cluster with Data API enabled.

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
    --db-cluster-identifier my_pg_cluster \
    --engine aurora-postgresql \
    --enable-http-endpoint
```

For Windows:

```
aws rds create-db-cluster ^
    --db-cluster-identifier my_pg_cluster ^
    --engine aurora-postgresql ^
    --enable-http-endpoint
```

RDS API

To enable Data API while you're creating an Aurora DB cluster, use the CreateDBCluster operation with the value of the `EnableHttpEndpoint` parameter set to `true`.

Enabling RDS Data API on an existing database

You can modify a DB cluster that supports RDS Data API (Data API) to enable or disable this feature.

Topics

- Enabling or disabling Data API (Aurora PostgreSQL Serverless v2 and provisioned)
- Enabling or disabling Data API (Aurora Serverless v1 only)

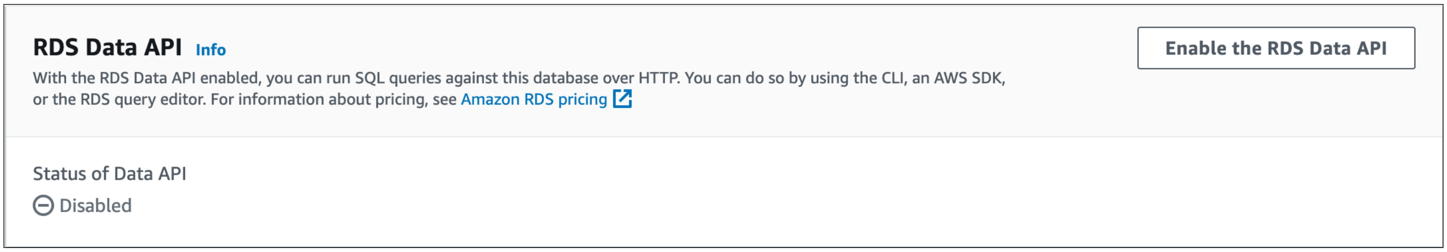
Enabling or disabling Data API (Aurora PostgreSQL Serverless v2 and provisioned)

Use the following procedures to enable or disable Data API on Aurora PostgreSQL Serverless v2 and provisioned databases. To enable or disable Data API on Aurora Serverless v1 databases, use the procedures in the section called "Enabling or disabling Data API (Aurora Serverless v1 only)".

Console

You can enable or disable Data API by using the RDS console for a DB cluster that supports this feature. To do so, open the cluster details page of the database on which you want to enable or disable Data API, and on the **Connectivity & security** tab, go to the **RDS Data API** section. This section displays the status of Data API, and allows you to enable or disable it.

The following screenshot shows that the **RDS Data API** isn't enabled.



With the RDS Data API enabled, you can run SQL queries against this database over HTTP. You can do so by using the CLI, an AWS SDK, or the RDS query editor. For information about pricing, see Amazon RDS pricing

Status of Data API

⊖ Disabled

AWS CLI

To enable or disable Data API on an existing database, run the enable-http-endpoint or disable-http-endpoint AWS CLI command, and specify the ARN of your DB cluster.

The following example enables Data API.

For Linux, macOS, or Unix:

```
aws rds enable-http-endpoint \
    --resource-arn cluster_arn
```

For Windows:

```
aws rds enable-http-endpoint ^
    --resource-arn cluster_arn
```

RDS API

To enable or disable Data API on an existing database, use the EnableHttpEndpoint and DisableHttpEndpoint operations.

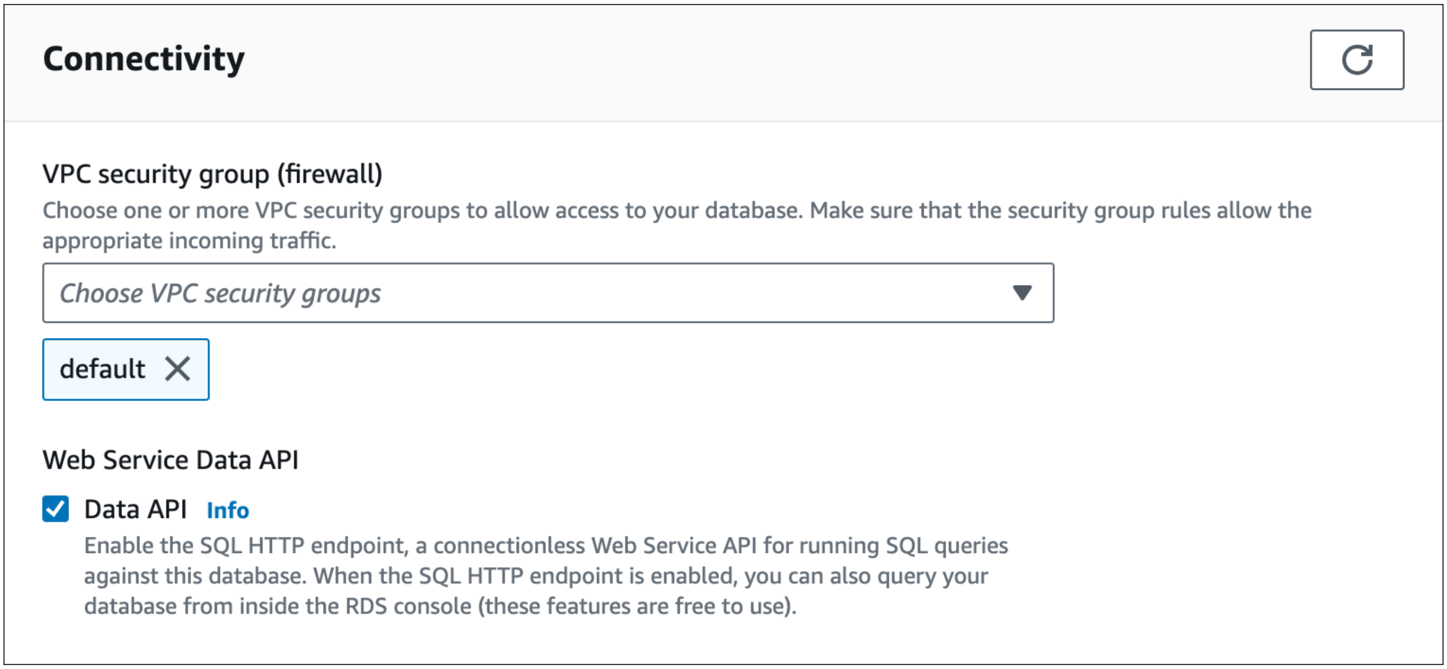
Enabling or disabling Data API (Aurora Serverless v1 only)

Use the following procedures to enable or disable Data API on existing Aurora Serverless v1 databases. To enable or disable Data API on Aurora PostgreSQL Serverless v2 and provisioned databases, use the procedures in the section called “Enabling or disabling Data API”.

Console

When you modify an Aurora Serverless v1 DB cluster, you enable Data API in the RDS console's **Connectivity** section.

The following screenshot shows the enabled **Data API** when modifying an Aurora DB cluster.



Connectivity

VPC security group (firewall)

Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

Choose VPC security groups ▼

default ✕

Web Service Data API

☑ Data API Info

Enable the SQL HTTP endpoint, a connectionless Web Service API for running SQL queries against this database. When the SQL HTTP endpoint is enabled, you can also query your database from inside the RDS console (these features are free to use).

For instructions on how to modify an Aurora Serverless v1 DB cluster, see [Modifying an Aurora Serverless v1 DB cluster](#).

AWS CLI

To enable or disable Data API, run the [modify-db-cluster](#) AWS CLI command, with the `--enable-http-endpoint` or `--no-enable-http-endpoint`, as applicable.

The following example enables Data API on `sample-cluster`.

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --enable-http-endpoint
```

For Windows:

```
aws rds modify-db-cluster ^
```

```
    --db-cluster-identifier sample-cluster ^
    --enable-http-endpoint
```

RDS API

To enable Data API, use the ModifyDBCluster operation, and set the value of `EnableHttpEndpoint` to `true` or `false`, as applicable.

Creating an Amazon VPC endpoint for RDS Data API (AWS PrivateLink)

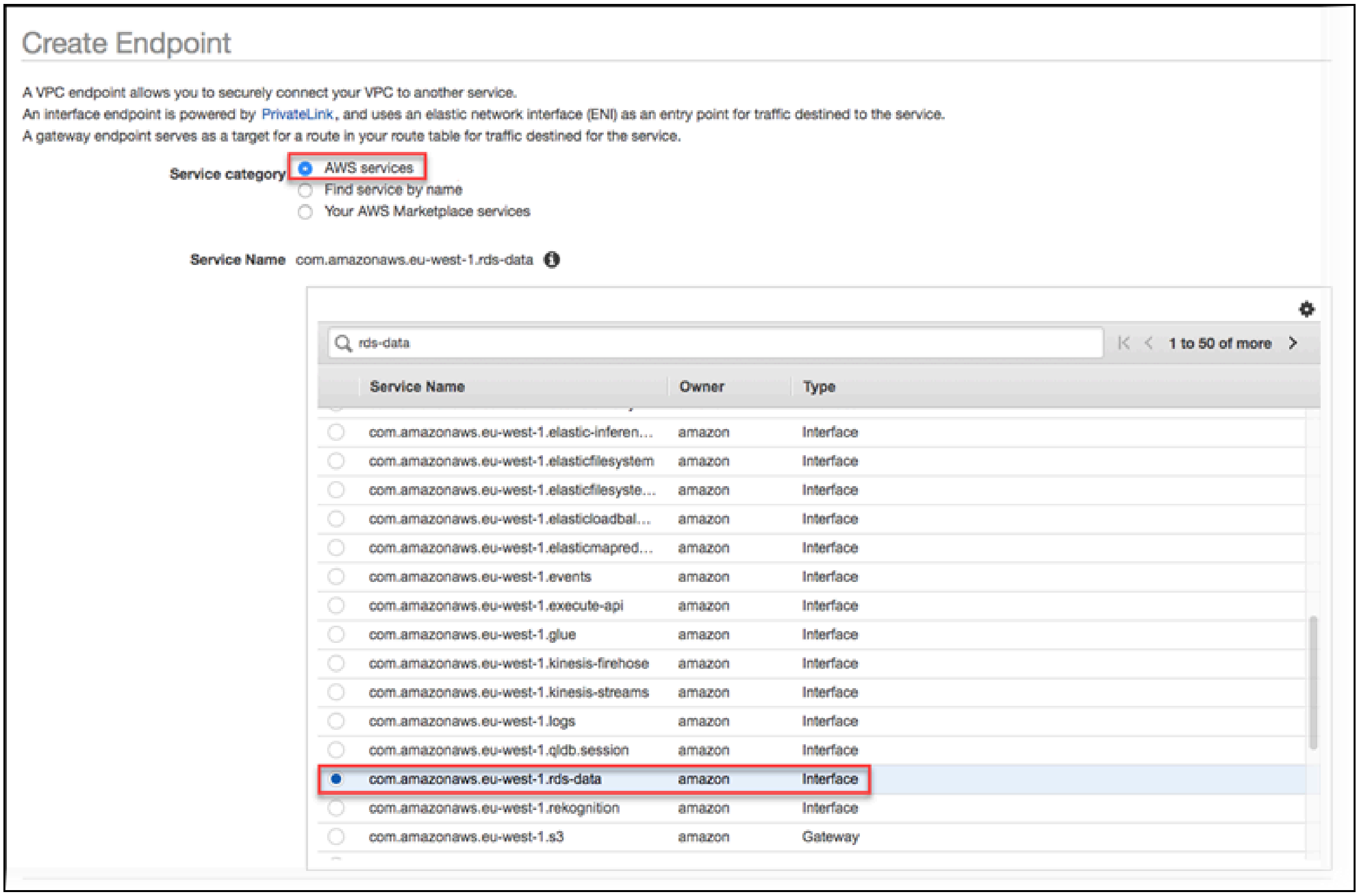
Amazon VPC enables you to launch AWS resources, such as Aurora DB clusters and applications, into a virtual private cloud (VPC). AWS PrivateLink provides private connectivity between VPCs and AWS services with high security on the Amazon network. Using AWS PrivateLink, you can create Amazon VPC endpoints, which enable you to connect to services across different accounts and VPCs based on Amazon VPC. For more information about AWS PrivateLink, see VPC Endpoint Services (AWS PrivateLink) in the *Amazon Virtual Private Cloud User Guide*.

You can call RDS Data API (Data API) with Amazon VPC endpoints. Using an Amazon VPC endpoint keeps traffic between applications in your Amazon VPC and Data API in the AWS network, without using public IP addresses. Amazon VPC endpoints can help you meet compliance and regulatory requirements related to limiting public internet connectivity. For example, if you use an Amazon VPC endpoint, you can keep traffic between an application running on an Amazon EC2 instance and Data API in the VPCs that contain them.

After you create the Amazon VPC endpoint, you can start using it without making any code or configuration changes in your application.

To create an Amazon VPC endpoint for Data API

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **Endpoints**, and then choose **Create Endpoint**.
3. On the **Create Endpoint** page, for **Service category**, choose **AWS services**. For **Service Name**, choose **rds-data**.



Create Endpoint

A VPC endpoint allows you to securely connect your VPC to another service.
An interface endpoint is powered by PrivateLink, and uses an elastic network interface (ENI) as an entry point for traffic destined to the service.
A gateway endpoint serves as a target for a route in your route table for traffic destined for the service.

Service category
● AWS services
○ Find service by name
○ Your AWS Marketplace services

Service Name com.amazonaws.eu-west-1.rds-data

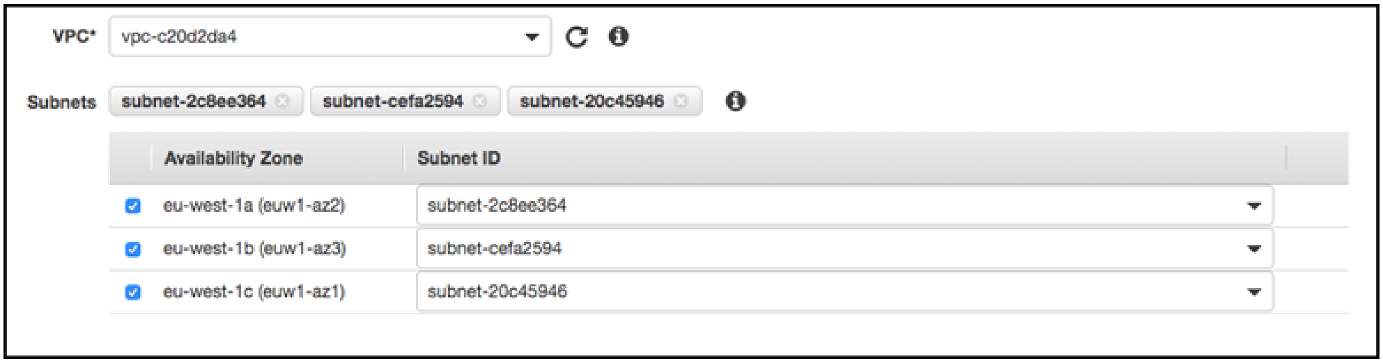
rds-data

	Service Name	Owner	Type
○	com.amazonaws.eu-west-1.elastic-inferen...	amazon	Interface
○	com.amazonaws.eu-west-1.elasticfilesystem	amazon	Interface
○	com.amazonaws.eu-west-1.elasticfilesyste...	amazon	Interface
○	com.amazonaws.eu-west-1.elasticloadbal...	amazon	Interface
○	com.amazonaws.eu-west-1.elasticmapred...	amazon	Interface
○	com.amazonaws.eu-west-1.events	amazon	Interface
○	com.amazonaws.eu-west-1.execute-api	amazon	Interface
○	com.amazonaws.eu-west-1.glue	amazon	Interface
○	com.amazonaws.eu-west-1.kinesis-firehose	amazon	Interface
○	com.amazonaws.eu-west-1.kinesis-streams	amazon	Interface
○	com.amazonaws.eu-west-1.logs	amazon	Interface
○	com.amazonaws.eu-west-1.qldb.session	amazon	Interface
●	com.amazonaws.eu-west-1.rds-data	amazon	Interface
○	com.amazonaws.eu-west-1.rekognition	amazon	Interface
○	com.amazonaws.eu-west-1.s3	amazon	Gateway

4. For **VPC**, choose the VPC to create the endpoint in.

   Choose the VPC that contains the application that makes Data API calls.

5. For **Subnets**, choose the subnet for each Availability Zone (AZ) used by the AWS service that is running your application.



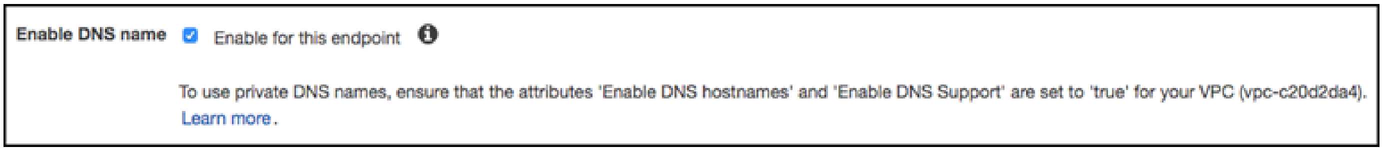
VPC* vpc-c20d2da4

Subnets subnet-2c8ee364 subnet-cefa2594 subnet-20c45946

	Availability Zone	Subnet ID
☑	eu-west-1a (euw1-az2)	subnet-2c8ee364
☑	eu-west-1b (euw1-az3)	subnet-cefa2594
☑	eu-west-1c (euw1-az1)	subnet-20c45946

To create an Amazon VPC endpoint, specify the private IP address range in which the endpoint will be accessible. To do this, choose the subnet for each Availability Zone. Doing so restricts the VPC endpoint to the private IP address range specific to each Availability Zone and also creates an Amazon VPC endpoint in each Availability Zone.

6. For **Enable DNS name**, select **Enable for this endpoint**.



Enable DNS name ☑ Enable for this endpoint

To use private DNS names, ensure that the attributes 'Enable DNS hostnames' and 'Enable DNS Support' are set to 'true' for your VPC (vpc-c20d2da4). Learn more.

Private DNS resolves the standard Data API DNS hostname (`https://rds-data.`*`region`*`.amazonaws.com`) to the private IP addresses associated with the DNS hostname specific to your Amazon VPC endpoint. As a result, you can access the Data API VPC endpoint using the AWS CLI or AWS SDKs without making any code or configuration changes to update Data API's endpoint URL.

7. For **Security group**, choose a security group to associate with the Amazon VPC endpoint.

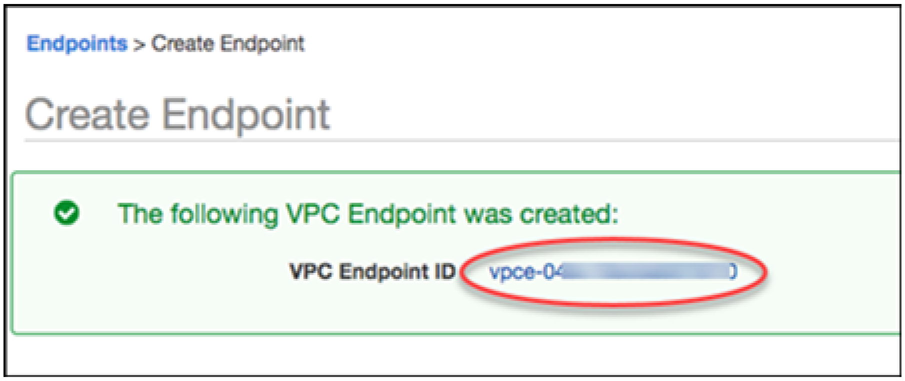
   Choose the security group that allows access to the AWS service that is running your application. For example, if an Amazon EC2 instance is running your application, choose the security group that allows access to the Amazon EC2 instance. The security group enables you to control the traffic to the Amazon VPC endpoint from resources in your VPC.

8. For **Policy**, choose **Full Access** to allow anyone inside the Amazon VPC to access the Data API through this endpoint. Or choose **Custom** to specify a policy that limits access.

   If you choose **Custom**, enter the policy in the policy creation tool.

9. Choose **Create endpoint**.

After the endpoint is created, choose the link in the AWS Management Console to view the endpoint details.



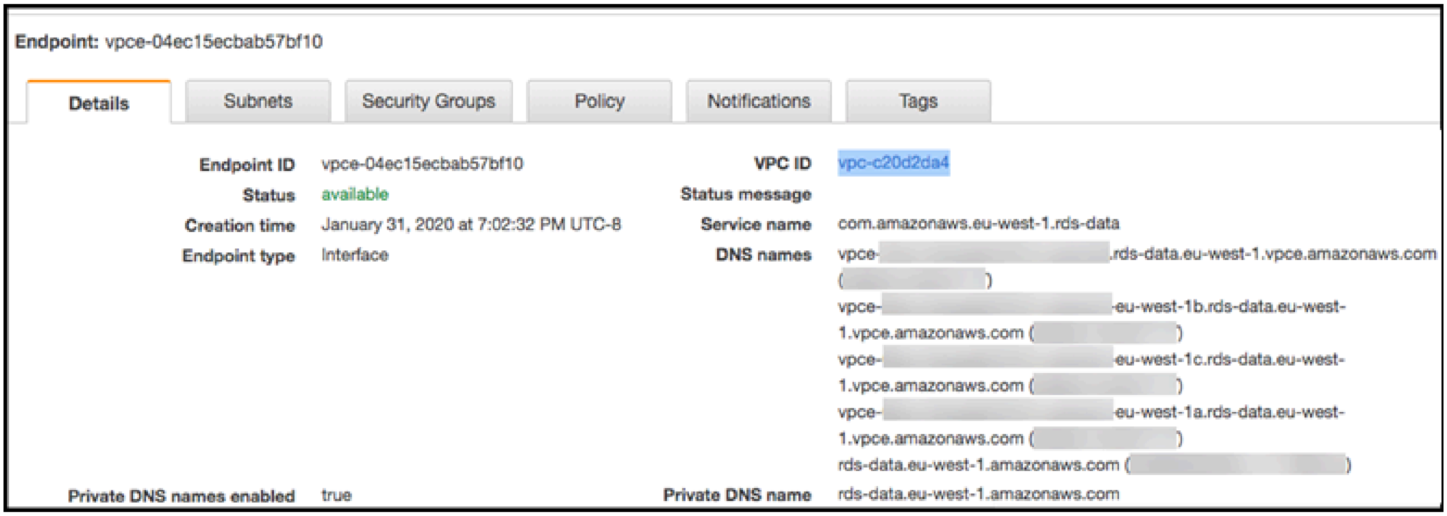
Endpoints > Create Endpoint

Create Endpoint

The following VPC Endpoint was created:

VPC Endpoint ID vpce-04[redacted]

The endpoint **Details** tab shows the DNS hostnames that were generated while creating the Amazon VPC endpoint.



Endpoint: vpce-04ec15ecbab57bf10

Endpoint ID vpce-04ec15ecbab57bf10

Status available

Creation time January 31, 2020 at 7:02:32 PM UTC-8

Endpoint type Interface

Private DNS names enabled true

VPC ID vpc-c20d2da4

Status message

Service name com.amazonaws.eu-west-1.rds-data

DNS names vpce-[redacted].rds-data.eu-west-1.vpce.amazonaws.com ([redacted])
vpce-[redacted]-eu-west-1b.rds-data.eu-west-1.vpce.amazonaws.com ([redacted])
vpce-[redacted]-eu-west-1c.rds-data.eu-west-1.vpce.amazonaws.com ([redacted])
vpce-[redacted]-eu-west-1a.rds-data.eu-west-1.vpce.amazonaws.com ([redacted])
rds-data.eu-west-1.amazonaws.com ([redacted])

Private DNS name rds-data.eu-west-1.amazonaws.com

You can use the standard endpoint (`rds-data.`*`region`*`.amazonaws.com`) or one of the VPC-specific endpoints to call the Data API within the Amazon VPC. The standard Data API endpoint automatically routes to the Amazon VPC endpoint. This routing occurs because the Private DNS hostname was enabled when the Amazon VPC endpoint was created.

When you use an Amazon VPC endpoint in a Data API call, all traffic between your application and Data API remains in the Amazon VPCs that contain them. You can use an Amazon VPC endpoint for any type of Data API call. For information about calling Data API, see Calling RDS Data API.

Calling RDS Data API

With RDS Data API (Data API) enabled on your Aurora DB cluster, you can run SQL statements on the Aurora DB cluster by using Data API or the AWS CLI. Data API supports the programming languages supported by the AWS SDKs. For more information, see Tools to build on AWS.

Topics

- Data API operations reference
- Calling RDS Data API with the AWS CLI
- Calling RDS Data API from a Python application
- Calling RDS Data API from a Java application
- Controlling Data API timeout behavior

Data API operations reference

Data API provides the following operations to perform SQL statements.

Data API operation	AWS CLI command	Description
ExecuteStatement	aws rds-data execute-statement	Runs a SQL statement on a database.
BatchExecuteStatement	aws rds-data batch-execute-statement	Runs a batch SQL statement over an array of data for bulk update and insert operations. You can run a data manipulation language (DML) statement with an array of parameter sets. A batch SQL statement can provide a significant performance improvement over individual insert and update statements.

You can use either operation to run individual SQL statements or to run transactions. For transactions, Data API provides the following operations.

Data API operation	AWS CLI command	Description
BeginTransaction	aws rds-data begin-transaction	Starts a SQL transaction.
CommitTransaction	aws rds-data commit-transaction	Ends a SQL transaction and commits the changes.
RollbackTransaction	aws rds-data rollback-transaction	Performs a rollback of a transaction.

The operations for performing SQL statements and supporting transactions have the following common Data API parameters and AWS CLI options. Some operations support other parameters or options.

Data API operation parameter	AWS CLI command option	Required	Description
`resourceArn`	`--resource-arn`	Yes	The Amazon Resource Name (ARN) of the Aurora DB cluster.
`secretArn`	`--secret-arn`	Yes	The name or ARN of the secret that enables access to the DB cluster.

You can use parameters in Data API calls to `ExecuteStatement` and `BatchExecuteStatement`, and when you run the AWS CLI commands `execute-statement` and `batch-execute-statement`. To use a parameter, you specify a name-value pair in the `SqlParameter` data type. You specify the value with the `Field` data type. The following table maps Java Database Connectivity (JDBC) data types to the data types that you specify in Data API calls.

JDBC data type	Data API data type
`INTEGER, TINYINT, SMALLINT, BIGINT`	`LONG` (or `STRING`)
`FLOAT, REAL, DOUBLE`	`DOUBLE`
`DECIMAL`	`STRING`
`BOOLEAN, BIT`	`BOOLEAN`
`BLOB, BINARY, LONGVARBINARY, VARBINARY`	`BLOB`
`CLOB`	`STRING`
Other types (including types related to date and time)	`STRING`

Note

You can specify the LONG or STRING data type in your Data API call for LONG values returned by the database. We recommend that you do so to avoid losing precision for extremely large numbers, which can happen when you work with JavaScript.

Certain types, such as DECIMAL and TIME, require a hint so that Data API passes String values to the database as the correct type. To use a hint, include values for typeHint in the SqlParameter data type. The possible values for typeHint are the following:

- DATE – The corresponding String parameter value is sent as an object of DATE type to the database. The accepted format is YYYY-MM-DD.
- DECIMAL – The corresponding String parameter value is sent as an object of DECIMAL type to the database.
- JSON – The corresponding String parameter value is sent as an object of JSON type to the database.
- TIME – The corresponding String parameter value is sent as an object of TIME type to the database. The accepted format is HH:MM:SS[.FFF].
- TIMESTAMP – The corresponding String parameter value is sent as an object of TIMESTAMP type to the database. The accepted format is YYYY-MM-DD HH:MM:SS[.FFF].
- UUID – The corresponding String parameter value is sent as an object of UUID type to the database.

  Note

  Currently, Data API doesn't support arrays of Universal Unique Identifiers (UUIDs).

Note

For Amazon Aurora PostgreSQL, Data API always returns the Aurora PostgreSQL data type TIMESTAMPTZ in UTC time zone.

Calling RDS Data API with the AWS CLI

You can call RDS Data API (Data API) using the AWS CLI.

The following examples use the AWS CLI for Data API. For more information, see AWS CLI reference for the Data API.

In each example, replace the Amazon Resource Name (ARN) for the DB cluster with the ARN for your Aurora DB cluster. Also, replace the secret ARN with the ARN of the secret in Secrets Manager that allows access to the DB cluster.

> **Note**
>
> The AWS CLI can format responses in JSON.

Topics

Starting a SQL transaction

You can start a SQL transaction using the `aws rds-data begin-transaction` CLI command. The call returns a transaction identifier.

> **Important**
>
> Within Data API, a transaction times out if there are no calls that use its transaction ID in three minutes. If a transaction times out before it's committed, Data API rolls it back automatically.
> MySQL data definition language (DDL) statements inside a transaction cause an implicit commit. We recommend that you run each MySQL DDL statement in a separate `execute-statement` command with the `--continue-after-timeout` option.

In addition to the common options, specify the `--database` option, which provides the name of the database.

For example, the following CLI command starts a SQL transaction.

For Linux, macOS, or Unix:

```
aws rds-data begin-transaction --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
```

For Windows:

```
aws rds-data begin-transaction --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
```

The following is an example of the response.

```
{
    "transactionId": "ABC1234567890xyz"
}
```

Running a SQL statement

You can run a SQL statement using the `aws rds-data execute-statement` CLI command.

You can run the SQL statement in a transaction by specifying the transaction identifier with the `--transaction-id` option. You can start a transaction using the `aws rds-data begin-transaction` CLI command. You can end and commit a transaction using the `aws rds-data commit-transaction` CLI command.

> **Important**
>
> If you don't specify the `--transaction-id` option, changes that result from the call are committed automatically.

In addition to the common options, specify the following options:

- `--sql` (required) – A SQL statement to run on the DB cluster.
- `--transaction-id` (optional) – The identifier of a transaction that was started using the `begin-transaction` CLI command. Specify the transaction ID of the transaction that you want to include the SQL statement in.
- `--parameters` (optional) – The parameters for the SQL statement.
- `--include-result-metadata | --no-include-result-metadata` (optional) – A value that indicates whether to include metadata in the results. The default is `--no-include-result-metadata`.
- `--database` (optional) – The name of the database.

  The `--database` option might not work when you run a SQL statement after running `--sql "use *database_name*;"` in the previous request. We recommend that you use the `--database` option instead of running `--sql "use *database_name*;"` statements.
- `--continue-after-timeout | --no-continue-after-timeout` (optional) – A value that indicates whether to continue running the statement after the call exceeds the Data API timeout interval of 45 seconds. The default is `--no-continue-after-timeout`.

  For data definition language (DDL) statements, we recommend continuing to run the statement after the call times out to avoid errors and the possibility of corrupted data structures.
- `--format-records-as "JSON"|"NONE"` – An optional value that specifies whether to format the result set as a JSON string. The default is `"NONE"`. For usage information about processing JSON result sets, see Processing RDS Data API query results in JSON format.

The DB cluster returns a response for the call.

> **Note**
>
> The response size limit is 1 MiB. If the call returns more than 1 MiB of response data, the call is terminated.
> For Aurora Serverless v1, the maximum number of requests per second is 1,000. For all other supported databases, there is no limit.

For example, the following CLI command runs a single SQL statement and omits the metadata in the results (the default).

For Linux, macOS, or Unix:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" \
--sql "select * from mytable"
```

For Windows:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" ^
--sql "select * from mytable"
```

The following is an example of the response.

```
{
    "numberOfRecordsUpdated": 0,
    "records": [
        [
            {
                "longValue": 1
            },
            {
                "stringValue": "ValueOne"
            }
        ],
        [
            {
                "longValue": 2
            },
            {
                "stringValue": "ValueTwo"
            }
        ],
        [
            {
                "longValue": 3
            },
            {
                "stringValue": "ValueThree"
            }
        ]
    ]
}
```

The following CLI command runs a single SQL statement in a transaction by specifying the `--transaction-id` option.

For Linux, macOS, or Unix:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" \
--sql "update mytable set quantity=5 where id=201" --transaction-id "ABC1234567890xyz"
```

For Windows:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" ^
--sql "update mytable set quantity=5 where id=201" --transaction-id "ABC1234567890xyz"
```

The following is an example of the response.

```
{
    "numberOfRecordsUpdated": 1
}
```

The following CLI command runs a single SQL statement with parameters.

For Linux, macOS, or Unix:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" \
--sql "insert into mytable values (:id, :val)" --parameters "[{\"name\": \"id\",
 \"value\": {\"longValue\": 1}},{\"name\": \"val\", \"value\": {\"stringValue\":
 \"value1\"}}]"
```

For Windows:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" ^
--sql "insert into mytable values (:id, :val)" --parameters "[{\"name\": \"id\",
 \"value\": {\"longValue\": 1}},{\"name\": \"val\", \"value\": {\"stringValue\":
 \"value1\"}}]"
```

The following is an example of the response.

```
{
    "numberOfRecordsUpdated": 1
}
```

The following CLI command runs a data definition language (DDL) SQL statement. The DDL statement renames column `job` to column `role`.

> **Important**
>
> For DDL statements, we recommend continuing to run the statement after the call times out. When a DDL statement terminates before it is finished running, it can result in errors and possibly corrupted data structures. To continue running a statement after a call exceeds the RDS Data API timeout interval of 45 seconds, specify the `--continue-after-timeout` option.

For Linux, macOS, or Unix:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" \
--sql "alter table mytable change column job role varchar(100)" --continue-after-
timeout
```

For Windows:

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" ^
--sql "alter table mytable change column job role varchar(100)" --continue-after-
timeout
```

The following is an example of the response.

```
{
    "generatedFields": [],
    "numberOfRecordsUpdated": 0
}
```

Note

The `generatedFields` data isn't supported by Aurora PostgreSQL. To get the values of generated fields, use the `RETURNING` clause. For more information, see [Returning data from modified rows]() in the PostgreSQL documentation.

Running a batch SQL statement over an array of data

You can run a batch SQL statement over an array of data by using the `aws rds-data batch-execute-statement` CLI command. You can use this command to perform a bulk import or update operation.

You can run the SQL statement in a transaction by specifying the transaction identifier with the `--transaction-id` option. You can start a transaction by using the `aws rds-data begin-transaction` CLI command. You can end and commit a transaction by using the `aws rds-data commit-transaction` CLI command.

Important

If you don't specify the `--transaction-id` option, changes that result from the call are committed automatically.

In addition to the common options, specify the following options:

- `--sql` (required) – A SQL statement to run on the DB cluster.

> **Tip**
>
> For MySQL-compatible statements, don't include a semicolon at the end of the `--sql` parameter. A trailing semicolon might cause a syntax error.

- `--transaction-id` (optional) – The identifier of a transaction that was started using the `begin-transaction` CLI command. Specify the transaction ID of the transaction that you want to include the SQL statement in.
- `--parameter-set` (optional) – The parameter sets for the batch operation.
- `--database` (optional) – The name of the database.

The DB cluster returns a response to the call.

> **Note**
>
> There isn't a fixed upper limit on the number of parameter sets. However, the maximum size of the HTTP request submitted through Data API is 4 MiB. If the request exceeds this limit, Data API returns an error and doesn't process the request. This 4 MiB limit includes the size of the HTTP headers and the JSON notation in the request. Thus, the number of parameter sets that you can include depends on a combination of factors, such as the size of the SQL statement and the size of each parameter set.
> The response size limit is 1 MiB. If the call returns more than 1 MiB of response data, the call is terminated.
> For Aurora Serverless v1, the maximum number of requests per second is 1,000. For all other supported databases, there is no limit.

For example, the following CLI command runs a batch SQL statement over an array of data with a parameter set.

For Linux, macOS, or Unix:

```
aws rds-data batch-execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" \
--sql "insert into mytable values (:id, :val)" \
--parameter-sets "[[{\"name\": \"id\", \"value\": {\"longValue\": 1}},{\"name\":
 \"val\", \"value\": {\"stringValue\": \"ValueOne\"}}],
[{\"name\": \"id\", \"value\": {\"longValue\": 2}},{\"name\": \"val\", \"value\":
 {\"stringValue\": \"ValueTwo\"}}],
[{\"name\": \"id\", \"value\": {\"longValue\": 3}},{\"name\": \"val\", \"value\":
 {\"stringValue\": \"ValueThree\"}}]]"
```

For Windows:

```
aws rds-data batch-execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--database "mydb" --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret" ^
--sql "insert into mytable values (:id, :val)" ^
--parameter-sets "[[{\"name\": \"id\", \"value\": {\"longValue\": 1}},{\"name\":
 \"val\", \"value\": {\"stringValue\": \"ValueOne\"}}],
[{\"name\": \"id\", \"value\": {\"longValue\": 2}},{\"name\": \"val\", \"value\":
 {\"stringValue\": \"ValueTwo\"}}],
[{\"name\": \"id\", \"value\": {\"longValue\": 3}},{\"name\": \"val\", \"value\":
 {\"stringValue\": \"ValueThree\"}}]]"
```

Note

Don't include line breaks in the `--parameter-sets` option.

Committing a SQL transaction

Using the `aws rds-data commit-transaction` CLI command, you can end a SQL transaction that you started with `aws rds-data begin-transaction` and commit the changes.

In addition to the common options, specify the following option:

- `--transaction-id` (required) – The identifier of a transaction that was started using the `begin-transaction` CLI command. Specify the transaction ID of the transaction that you want to end and commit.

For example, the following CLI command ends a SQL transaction and commits the changes.

For Linux, macOS, or Unix:

```
aws rds-data commit-transaction --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster" \
--secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret" \
--transaction-id "ABC1234567890xyz"
```

For Windows:

```
aws rds-data commit-transaction --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster" ^
--secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret" ^
--transaction-id "ABC1234567890xyz"
```

The following is an example of the response.

```
{
    "transactionStatus": "Transaction Committed"
}
```

Rolling back a SQL transaction

Using the `aws rds-data rollback-transaction` CLI command, you can roll back a SQL transaction that you started with `aws rds-data begin-transaction`. Rolling back a transaction cancels its changes.

> **Important**
>
> If the transaction ID has expired, the transaction was rolled back automatically. In this case, an `aws rds-data rollback-transaction` command that specifies the expired transaction ID returns an error.

In addition to the common options, specify the following option:

- `--transaction-id` (required) – The identifier of a transaction that was started using the `begin-transaction` CLI command. Specify the transaction ID of the transaction that you want to roll back.

For example, the following AWS CLI command rolls back a SQL transaction.

For Linux, macOS, or Unix:

```
aws rds-data rollback-transaction --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" \
--secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret" \
--transaction-id "ABC1234567890xyz"
```

For Windows:

```
aws rds-data rollback-transaction --resource-arn "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster" ^
--secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret" ^
--transaction-id "ABC1234567890xyz"
```

The following is an example of the response.

```
{
    "transactionStatus": "Rollback Complete"
    }
```

Calling RDS Data API from a Python application

You can call RDS Data API (Data API) from a Python application.

The following examples use the AWS SDK for Python (Boto). For more information about Boto, see the AWS SDK for Python (Boto 3) documentation.

In each example, replace the DB cluster's Amazon Resource Name (ARN) with the ARN for your Aurora DB cluster. Also, replace the secret ARN with the ARN of the secret in Secrets Manager that allows access to the DB cluster.

Topics

Running a SQL query

You can run a SELECT statement and fetch the results with a Python application.

The following example runs a SQL query.

```
import boto3

rdsData = boto3.client('rds-data')

cluster_arn = 'arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster'
secret_arn = 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'

response1 = rdsData.execute_statement(
            resourceArn = cluster_arn,
            secretArn = secret_arn,
            database = 'mydb',
            sql = 'select * from employees limit 3')

print (response1['records'])
[
    [
        {
            'longValue': 1
        },
        {
            'stringValue': 'ROSALEZ'
        },
        {
            'stringValue': 'ALEJANDRO'
        },
        {
            'stringValue': '2016-02-15 04:34:33.0'
        }
    ],
    [
        {
            'longValue': 1
        },
        {
            'stringValue': 'DOE'
        },
        {
            'stringValue': 'JANE'
        },
        {
            'stringValue': '2014-05-09 04:34:33.0'
        }
    ],
    [
        {
            'longValue': 1
        },
        {
            'stringValue': 'STILES'
        },
        {
            'stringValue': 'JOHN'
        },
        {
            'stringValue': '2017-09-20 04:34:33.0'
        }
    ]
]
```

Running a DML SQL statement

You can run a data manipulation language (DML) statement to insert, update, or delete data in your database. You can also use parameters in DML statements.

Important

If a call isn't part of a transaction because it doesn't include the `transactionID` parameter, changes that result from the call are committed automatically.

The following example runs an insert SQL statement and uses parameters.

```
import boto3

cluster_arn = 'arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster'
secret_arn = 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'

rdsData = boto3.client('rds-data')


param1 = {'name':'firstname', 'value':{'stringValue': 'JACKSON'}}
param2 = {'name':'lastname', 'value':{'stringValue': 'MATEO'}}
paramSet = [param1, param2]
```

```
response2 = rdsData.execute_statement(resourceArn=cluster_arn,
                                      secretArn=secret_arn,
                                      database='mydb',
                                      sql='insert into employees(first_name, last_name)
 VALUES(:firstname, :lastname)',
                                      parameters = paramSet)

print (response2["numberOfRecordsUpdated"])
```

Running a SQL transaction

You can start a SQL transaction, run one or more SQL statements, and then commit the changes with a Python application.

> **Important**
>
> A transaction times out if there are no calls that use its transaction ID in three minutes. If a transaction times out before it's committed, it's rolled back automatically.
> If you don't specify a transaction ID, changes that result from the call are committed automatically.

The following example runs a SQL transaction that inserts a row in a table.

```
import boto3

rdsData = boto3.client('rds-data')

cluster_arn = 'arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster'
secret_arn = 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'

tr = rdsData.begin_transaction(
     resourceArn = cluster_arn,
     secretArn = secret_arn,
     database = 'mydb')

response3 = rdsData.execute_statement(
     resourceArn = cluster_arn,
     secretArn = secret_arn,
     database = 'mydb',
     sql = 'insert into employees(first_name, last_name) values('XIULAN', 'WANG')',
    transactionId = tr['transactionId'])

cr = rdsData.commit_transaction(
    resourceArn = cluster_arn,
    secretArn = secret_arn,
    transactionId = tr['transactionId'])

cr['transactionStatus']
'Transaction Committed'

response3['numberOfRecordsUpdated']
1
```

Note

If you run a data definition language (DDL) statement, we recommend continuing to run the statement after the call times out. When a DDL statement terminates before it is finished running, it can result in errors and possibly corrupted data structures. To continue running a statement after a call exceeds the RDS Data API timeout interval of 45 seconds, set the `continueAfterTimeout` parameter to `true`.

Calling RDS Data API from a Java application

You can call RDS Data API (Data API) from a Java application.

The following examples use the AWS SDK for Java. For more information, see the AWS SDK for Java Developer Guide.

In each example, replace the DB cluster's Amazon Resource Name (ARN) with the ARN for your Aurora DB cluster. Also, replace the secret ARN with the ARN of the secret in Secrets Manager that allows access to the DB cluster.

Topics

- Running a SQL query
- Running a SQL transaction
- Running a batch SQL operation

Running a SQL query

You can run a SELECT statement and fetch the results with a Java application.

The following example runs a SQL query.

```
package com.amazonaws.rdsdata.examples;

import com.amazonaws.services.rdsdata.AWSRDSData;
import com.amazonaws.services.rdsdata.AWSRDSDataClient;
import com.amazonaws.services.rdsdata.model.ExecuteStatementRequest;
import com.amazonaws.services.rdsdata.model.ExecuteStatementResult;
import com.amazonaws.services.rdsdata.model.Field;

import java.util.List;

public class FetchResultsExample {
  public static final String RESOURCE_ARN = "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster";
  public static final String SECRET_ARN = "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret";

  public static void main(String[] args) {
    AWSRDSData rdsData = AWSRDSDataClient.builder().build();

    ExecuteStatementRequest request = new ExecuteStatementRequest()
            .withResourceArn(RESOURCE_ARN)
            .withSecretArn(SECRET_ARN)
            .withDatabase("mydb")
            .withSql("select * from mytable");

    ExecuteStatementResult result = rdsData.executeStatement(request);

    for (List<Field> fields: result.getRecords()) {
      String stringValue = fields.get(0).getStringValue();
      long numberValue = fields.get(1).getLongValue();

      System.out.println(String.format("Fetched row: string = %s, number = %d",
 stringValue, numberValue));
    }
  }
}
```

Running a SQL transaction

You can start a SQL transaction, run one or more SQL statements, and then commit the changes with a Java application.

> ⚠ **Important**
>
> A transaction times out if there are no calls that use its transaction ID in three minutes. If a transaction times out before it's committed, it's rolled back automatically.
>
> If you don't specify a transaction ID, changes that result from the call are committed automatically.

The following example runs a SQL transaction.

```
package com.amazonaws.rdsdata.examples;

import com.amazonaws.services.rdsdata.AWSRDSData;
import com.amazonaws.services.rdsdata.AWSRDSDataClient;
import com.amazonaws.services.rdsdata.model.BeginTransactionRequest;
import com.amazonaws.services.rdsdata.model.BeginTransactionResult;
import com.amazonaws.services.rdsdata.model.CommitTransactionRequest;
import com.amazonaws.services.rdsdata.model.ExecuteStatementRequest;

public class TransactionExample {
  public static final String RESOURCE_ARN = "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster";
  public static final String SECRET_ARN = "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret";

  public static void main(String[] args) {
    AWSRDSData rdsData = AWSRDSDataClient.builder().build();

    BeginTransactionRequest beginTransactionRequest = new BeginTransactionRequest()
            .withResourceArn(RESOURCE_ARN)
            .withSecretArn(SECRET_ARN)
            .withDatabase("mydb");
    BeginTransactionResult beginTransactionResult =
 rdsData.beginTransaction(beginTransactionRequest);
    String transactionId = beginTransactionResult.getTransactionId();

    ExecuteStatementRequest executeStatementRequest = new ExecuteStatementRequest()
```

```
                .withTransactionId(transactionId)
                .withResourceArn(RESOURCE_ARN)
                .withSecretArn(SECRET_ARN)
                .withSql("INSERT INTO test_table VALUES ('hello world!')");
        rdsData.executeStatement(executeStatementRequest);

        CommitTransactionRequest commitTransactionRequest = new CommitTransactionRequest()
                .withTransactionId(transactionId)
                .withResourceArn(RESOURCE_ARN)
                .withSecretArn(SECRET_ARN);
        rdsData.commitTransaction(commitTransactionRequest);
    }
}
```

Note

If you run a data definition language (DDL) statement, we recommend continuing to run the statement after the call times out. When a DDL statement terminates before it is finished running, it can result in errors and possibly corrupted data structures. To continue running a statement after a call exceeds the RDS Data API timeout interval of 45 seconds, set the `continueAfterTimeout` parameter to `true`.

Running a batch SQL operation

You can run bulk insert and update operations over an array of data with a Java application. You can run a DML statement with array of parameter sets.

Important

If you don't specify a transaction ID, changes that result from the call are committed automatically.

The following example runs a batch insert operation.

```
package com.amazonaws.rdsdata.examples;

import com.amazonaws.services.rdsdata.AWSRDSData;
import com.amazonaws.services.rdsdata.AWSRDSDataClient;
import com.amazonaws.services.rdsdata.model.BatchExecuteStatementRequest;
import com.amazonaws.services.rdsdata.model.Field;
import com.amazonaws.services.rdsdata.model.SqlParameter;

import java.util.Arrays;

public class BatchExecuteExample {
  public static final String RESOURCE_ARN = "arn:aws:rds:us-
east-1:123456789012:cluster:mydbcluster";
  public static final String SECRET_ARN = "arn:aws:secretsmanager:us-
east-1:123456789012:secret:mysecret";

  public static void main(String[] args) {
      AWSRDSData rdsData = AWSRDSDataClient.builder().build();

    BatchExecuteStatementRequest request = new BatchExecuteStatementRequest()
            .withDatabase("test")
            .withResourceArn(RESOURCE_ARN)
            .withSecretArn(SECRET_ARN)
            .withSql("INSERT INTO test_table2 VALUES (:string, :number)")
            .withParameterSets(Arrays.asList(
                    Arrays.asList(
                            new SqlParameter().withName("string").withValue(new
 Field().withStringValue("Hello")),
                            new SqlParameter().withName("number").withValue(new
 Field().withLongValue(1L))
                    ),
                    Arrays.asList(
                            new SqlParameter().withName("string").withValue(new
 Field().withStringValue("World")),
                            new SqlParameter().withName("number").withValue(new
 Field().withLongValue(2L))
                    )
            ));

    rdsData.batchExecuteStatement(request);
  }
}
```

Controlling Data API timeout behavior

All calls to Data API are synchronous. Suppose that you perform a Data API operation that runs a SQL statement such as `INSERT` or `CREATE TABLE`. If the Data API call returns successfully, the SQL processing is finished when the call returns.

By default, Data API cancels an operation and returns a timeout error if the operation doesn't finish processing within 45 seconds. In that case, the data isn't inserted, the table isn't created, and so on.

You can use Data API to perform long-running operations that can't complete within 45 seconds. If you expect that an operation such as a bulk `INSERT` or a DDL operation on a large table takes longer than 45 seconds, you can specify the `continueAfterTimeout` parameter for the `ExecuteStatement` operation. Your application still receives the timeout error. However, the operation continues running and isn't cancelled. For an example, see Running a SQL transaction.

If the AWS SDK for your programming language has its own timeout period for API calls or HTTP socket connections, make sure that all such timeout periods are more than 45 seconds. For some SDKs, the timeout period is less than 45 seconds by default. We recommend setting any SDK-specific or client-specific timeout periods to at least one minute. Doing so avoids the possibility that your application receives a timeout error while the Data API operation still completes successfully. That way, you can be sure whether to retry the operation or not.

For example, suppose that the SDK returns a timeout error to your application, but the Data API operation still completes within the Data API timeout interval. In that case, retrying the operation might insert duplicate data or otherwise produce incorrect results. The SDK might retry the operation automatically, causing incorrect data without any action from your application.

The timeout interval is especially important for the Java 2 SDK. In that SDK, the API call timeout and the HTTP socket timeout are both 30 seconds by default. Here is an example of setting those timeouts to a higher value:

```
public RdsDataClient createRdsDataClient() {
    return RdsDataClient.builder()
        .region(Region.US_EAST_1) // Change this to your desired Region
        .overrideConfiguration(createOverrideConfiguration())
        .httpClientBuilder(createHttpClientBuilder())
        .credentialsProvider(defaultCredentialsProvider()) // Change this to your
 desired credentials provider
        .build();
}

private static ClientOverrideConfiguration createOverrideConfiguration() {
    return ClientOverrideConfiguration.builder()
        .apiCallTimeout(Duration.ofSeconds(60))
        .build();
}
```

```
private HttpClientBuilder createHttpClientBuilder() {
    return ApacheHttpClient.builder() // Change this to your desired HttpClient
        .socketTimeout(Duration.ofSeconds(60));
}
```

Here is an equivalent example using the asynchronous data client:

```
public static RdsDataAsyncClient createRdsDataAsyncClient() {
    return RdsDataAsyncClient.builder()
        .region(Region.US_EAST_1) // Change this to your desired Region
        .overrideConfiguration(createOverrideConfiguration())
        .credentialsProvider(defaultCredentialsProvider())  // Change this to your
 desired credentials provider
        .build();
}

private static ClientOverrideConfiguration createOverrideConfiguration() {
    return ClientOverrideConfiguration.builder()
        .apiCallAttemptTimeout(Duration.ofSeconds(60))
        .build();
}

private HttpClientBuilder createHttpClientBuilder() {
    return NettyNioAsyncHttpClient.builder() // Change this to your desired
 AsyncHttpClient
        .readTimeout(Duration.ofSeconds(60));
}
```

Using the Java client library for RDS Data API

You can download and use a Java client library for RDS Data API (Data API). This Java client library provides an alternative way to use Data API. Using this library, you can map your client-side classes to Data API requests and responses. This mapping support can ease integration with some specific Java types, such as `Date`, `Time`, and `BigDecimal`.

Downloading the Java client library for Data API

The Data API Java client library is open source in GitHub at the following location:

https://github.com/awslabs/rds-data-api-client-library-java

You can build the library manually from the source files, but the best practice is to consume the library using Apache Maven dependency management. Add the following dependency to your Maven POM file.

For version 2.x, which is compatible with AWS SDK 2.x, use the following:

```
<dependency>
   <groupId>software.amazon.rdsdata</groupId>
   <artifactId>rds-data-api-client-library-java</artifactId>
   <version>2.0.0</version>
</dependency>
```

For version 1.x, which is compatible with AWS SDK 1.x, use the following:

```
<dependency>
    <groupId>software.amazon.rdsdata</groupId>
    <artifactId>rds-data-api-client-library-java</artifactId>
    <version>1.0.8</version>
</dependency>
```

Java client library examples

Following, you can find some common examples of using the Data API Java client library. These examples assume that you have a table `accounts` with two columns: `accountId` and `name`. You also have the following data transfer object (DTO).

```
public class Account {
    int accountId;
    String name;
    // getters and setters omitted
}
```

The client library enables you to pass DTOs as input parameters. The following example shows how customer DTOs are mapped to input parameters sets.

```
var account1 = new Account(1, "John");
var account2 = new Account(2, "Mary");
client.forSql("INSERT INTO accounts(accountId, name) VALUES(:accountId, :name)")
         .withParamSets(account1, account2)
         .execute();
```

In some cases, it's easier to work with simple values as input parameters. You can do so with the following syntax.

```
client.forSql("INSERT INTO accounts(accountId, name) VALUES(:accountId, :name)")
         .withParameter("accountId", 3)
         .withParameter("name", "Zhang")
         .execute();
```

The following is another example that works with simple values as input parameters.

```

client.forSql("INSERT INTO accounts(accountId, name) VALUES(?, ?)", 4, "Carlos")
         .execute();

```

The client library provides automatic mapping to DTOs when a result is returned. The following examples show how the result is mapped to your DTOs.

```
List<Account> result = client.forSql("SELECT * FROM accounts")
          .execute()
          .mapToList(Account.class);

Account result = client.forSql("SELECT * FROM accounts WHERE account_id = 1")
          .execute()
          .mapToSingle(Account.class);
```

In many cases, the database result set contains only a single value. In order to simplify retrieving such results, the client library offers the following API:

```
int numberOfAccounts = client.forSql("SELECT COUNT(*) FROM accounts")
          .execute()
          .singleValue(Integer.class);
```

Note

The `mapToList` function converts a SQL result set into a user-defined object list. We don't support using the `.withFormatRecordsAs(RecordsFormatType.JSON)` statement in an `ExecuteStatement` call for the Java client library, because it serves the same purpose. For more information, see Processing RDS Data API query results in JSON format.

Processing RDS Data API query results in JSON format

When you call the ExecuteStatement operation, you can choose to have the query results returned as a string in JSON format. That way, you can use your programming language's JSON parsing capabilities to interpret and reformat the result set. Doing so can help to avoid writing extra code to loop through the result set and interpret each column value.

To request the result set in JSON format, you pass the optional formatRecordsAs parameter with a value of JSON. The JSON-formatted result set is returned in the formattedRecords field of the ExecuteStatementResponse structure.

The BatchExecuteStatement action doesn't return a result set. Thus, the JSON option doesn't apply to that action.

To customize the keys in the JSON hash structure, define column aliases in the result set. You can do so by using the AS clause in the column list of your SQL query.

You might use the JSON capability to make the result set easier to read and map its contents to language-specific frameworks. Because the volume of the ASCII-encoded result set is larger than the default representation, you might choose the default representation for queries that return large numbers of rows or large column values that consume more memory than is available to your application.

Topics

Retrieving query results in JSON format

To receive the result set as a JSON string, include .withFormatRecordsAs(RecordsFormatType.JSON) in the ExecuteStatement call. The return value comes back as a JSON string in the formattedRecords field. In this case, the columnMetadata is null. The column labels are the keys of the object that represents each row. These column names are repeated for each row in the result set. The column values are quoted strings, numeric values, or special values representing true, false, or null. Column metadata such as length constraints and the precise type for numbers and strings isn't preserved in the JSON response.

If you omit the .withFormatRecordsAs() call or specify a parameter of NONE, the result set is returned in binary format using the Records and columnMetadata fields.

Data Type Mapping

The SQL values in the result set are mapped to a smaller set of JSON types. The values are represented in JSON as strings, numbers, and some special constants such as true, false, and null. You can convert these values into variables in your application, using strong or weak typing as appropriate for your programming language.

JDBC data type	JSON data type
INTEGER, TINYINT, SMALLINT, BIGINT	Number by default. String if the LongReturnType option is set to STRING.
FLOAT, REAL, DOUBLE	Number
DECIMAL	String by default. Number if the DecimalReturnType option is set to DOUBLE_OR_LONG .
STRING	String
BOOLEAN, BIT	Boolean
BLOB, BINARY, VARBINARY , LONGVARBINARY	String in base64 encoding.
CLOB	String
ARRAY	Array
NULL	null
Other types (including types related to date and time)	String

Troubleshooting

The JSON response is limited to 10 megabytes. If the response is larger than this limit, your program receives a BadRequestException error. In this case, you can resolve the error using one of the following techniques:

- Reduce the number of rows in the result set. To do so, add a LIMIT clause. You might split a large result set into multiple smaller ones by submitting several queries with LIMIT and OFFSET clauses.

  If the result set includes rows that are filtered out by application logic, you can remove those rows from the result set by adding more conditions in the WHERE clause.
- Reduce the number of columns in the result set. To do so, remove items from the select list of the query.
- Shorten the column labels by using column aliases in the query. Each column name is repeated in the JSON string for each row in the result set. Thus, a query result with long column names and many rows could exceed the size limit. In particular, use column aliases for complicated expressions to avoid having the entire expression repeated in the JSON string.
- Although with SQL you can use column aliases to produce a result set having more than one column with the same name, duplicate key names aren't allowed in JSON. The RDS Data API returns an error if you request the result set in JSON format and more than one column has the same name. Thus, make sure that all the column labels have unique names.

Examples

The following Java examples show how to call ExecuteStatement with the response as a JSON-formatted string, then interpret the result set. Substitute the appropriate values for the *databaseName*, *secretStoreArn*, and *clusterArn* parameters.

The following Java example demonstrates a query that returns a decimal numeric value in the result set. The assertThat calls test that the fields of the response have the expected properties based on the rules for JSON result sets.

This example works with the following schema and sample data:

```
create table test_simplified_json (a float);
insert into test_simplified_json values(10.0);
```

```
public void JSON_result_set_demo() {
    var sql = "select * from test_simplified_json";
    var request = new ExecuteStatementRequest()
      .withDatabase(databaseName)
      .withSecretArn(secretStoreArn)
      .withResourceArn(clusterArn)
      .withSql(sql)
      .withFormatRecordsAs(RecordsFormatType.JSON);
    var result = rdsdataClient.executeStatement(request);
}
```

The value of the `formattedRecords` field from the preceding program is:

```
[{"a":10.0}]
```

The `Records` and `ColumnMetadata` fields in the response are both null, due to the presence of the JSON result set.

The following Java example demonstrates a query that returns an integer numeric value in the result set. The example calls `getFormattedRecords` to return only the JSON-formatted string and ignore the other response fields that are blank or null. The example deserializes the result into a structure representing a list of records. Each record has fields whose names correspond to the column aliases from the result set. This technique simplifies the code that parses the result set. Your application doesn't have to loop through the rows and columns of the result set and convert each value to the appropriate type.

This example works with the following schema and sample data:

```
create table test_simplified_json (a int);
insert into test_simplified_json values(17);
```

```
public void JSON_deserialization_demo() {
    var sql = "select * from test_simplified_json";
    var request = new ExecuteStatementRequest()
      .withDatabase(databaseName)
      .withSecretArn(secretStoreArn)
      .withResourceArn(clusterArn)
      .withSql(sql)
      .withFormatRecordsAs(RecordsFormatType.JSON);
    var result = rdsdataClient.executeStatement(request)
      .getFormattedRecords();
/* Turn the result set into a Java object, a list of records.
   Each record has a field 'a' corresponding to the column
   labelled 'a' in the result set. */
    private static class Record { public int a; }
    var recordsList = new ObjectMapper().readValue(
        response, new TypeReference<List<Record>>() {
        });
}
```

The value of the `formattedRecords` field from the preceding program is:

```
[{"a":17}]
```

To retrieve the a column of result row 0, the application would refer to `recordsList.get(0).a`.

In contrast, the following Java example shows the kind of code that's required to construct a data structure holding the result set when you don't use the JSON format. In this case, each row of the result set contains fields with information about a single user. Building a data structure to represent the result set requires looping through the rows. For each row, the code retrieves the value of each field, performs an appropriate type conversion, and assigns the result to the corresponding field in the object representing the row. Then the code adds the object representing each user to the data structure representing the entire result set. If the query was changed to reorder, add, or remove fields in the result set, the application code would have to change also.

```
/* Verbose result-parsing code that doesn't use the JSON result set format */
for (var row: response.getRecords()) {
    var user = User.builder()
      .userId(row.get(0).getLongValue())
      .firstName(row.get(1).getStringValue())
      .lastName(row.get(2).getStringValue())
      .dob(Instant.parse(row.get(3).getStringValue()))
      .build();
    result.add(user);
  }
```

The following sample values show the values of the `formattedRecords` field for result sets with different numbers of columns, column aliases, and column data types.

If the result set includes multiple rows, each row is represented as an object that is an array element. Each column in the result set becomes a key in the object. The keys are repeated for each row in the result set. Thus, for result sets consisting of many rows and columns, you might need to define short column aliases to avoid exceeding the length limit for the entire response.

This example works with the following schema and sample data:

```
create table sample_names (id int, name varchar(128));
insert into sample_names values (0, "Jane"), (1, "Mohan"), (2, "Maria"), (3, "Bruce"),
 (4, "Jasmine");
```

```
[{"id":0,"name":"Jane"},{"id":1,"name":"Mohan"},
{"id":2,"name":"Maria"},{"id":3,"name":"Bruce"},{"id":4,"name":"Jasmine"}]
```

If a column in the result set is defined as an expression, the text of the expression becomes the JSON key. Thus, it's typically convenient to define a descriptive column alias for each expression in the select list of the query. For example, the following query includes expressions such as function calls and arithmetic operations in its select list.

```
select count(*), max(id), 4+7 from sample_names;
```

Those expressions are passed through to the JSON result set as keys.

```
[{"count(*)":5,"max(id)":4,"4+7":11}]
```

Adding AS columns with descriptive labels makes the keys simpler to interpret in the JSON result set.

```
select count(*) as rows, max(id) as largest_id, 4+7 as addition_result from
 sample_names;
```

With the revised SQL query, the column labels defined by the AS clauses are used as the key names.

```
[{"rows":5,"largest_id":4,"addition_result":11}]
```

The value for each key-value pair in the JSON string can be a quoted string. The string might contain unicode characters. If the string contains escape sequences or the " or \ characters, those characters are preceded by backslash escape characters. The following examples of JSON strings demonstrate these possibilities. For example, the `string_with_escape_sequences` result contains the special characters backspace, newline, carriage return, tab, form feed, and \.

```
[{"quoted_string":"hello"}]
[{"unicode_string":"####"}]
[{"string_with_escape_sequences":"\b \n \r \t \f \\ '"}]
```

The value for each key-value pair in the JSON string can also represent a number. The number might be an integer, a floating-point value, a negative value, or a value represented as exponential notation. The following examples of JSON strings demonstrate these possibilities.

```
[{"integer_value":17}]
[{"float_value":10.0}]
[{"negative_value":-9223372036854775808,"positive_value":9223372036854775807}]
[{"very_small_floating_point_value":4.9E-324,"very_large_floating_point_value":1.79769313486231
```

Boolean and null values are represented with the unquoted special keywords `true`, `false`, and `null`. The following examples of JSON strings demonstrate these possibilities.

```
[{"boolean_value_1":true,"boolean_value_2":false}]
[{"unknown_value":null}]
```

If you select a value of a BLOB type, the result is represented in the JSON string as a base64-encoded value. To convert the value back to its original representation, you can use the appropriate decoding function in your application's language. For example, in Java you call the function `Base64.getDecoder().decode()`. The following sample output shows the result of selecting a BLOB value of `hello world` and returning the result set as a JSON string.

```
[{"blob_column":"aGVsbG8gd29ybGQ="}]
```

The following Python example shows how to access the values from the result of a call to the Python `execute_statement` function. The result set is a string value in the field `response['formattedRecords']`. The code turns the JSON string into a data structure by calling the `json.loads` function. Then each row of the result set is a list element within the data structure, and within each row you can refer to each field of the result set by name.

```
import json
```

```
result = json.loads(response['formattedRecords'])
print (result[0]["id"])
```

The following JavaScript example shows how to access the values from the result of a call to the JavaScript `executeStatement` function. The result set is a string value in the field `response.formattedRecords`. The code turns the JSON string into a data structure by calling the `JSON.parse` function. Then each row of the result set is an array element within the data structure, and within each row you can refer to each field of the result set by name.

```
<script>
    const result = JSON.parse(response.formattedRecords);
    document.getElementById("display").innerHTML = result[0].id;
</script>
```

Troubleshooting RDS Data API issues

Use the following sections, titled with common error messages, to help troubleshoot problems that you have with RDS Data API (Data API).

Topics

Transaction <transaction_ID> is not found

In this case, the transaction ID specified in a Data API call wasn't found. The cause for this issue is appended to the error message, and is one of the following:

- Transaction may be expired.

  Make sure that each transactional call runs within three minutes of the previous one.

  It's also possible that the specified transaction ID wasn't created by a BeginTransaction call. Make sure that your call has a valid transaction ID.

- One previous call resulted in a termination of your transaction.

The transaction was already ended by your CommitTransaction or RollbackTransaction call.

- Transaction has been aborted due to an error from a previous call.

  Check whether your previous calls have thrown any exceptions.

For information about running transactions, see Calling RDS Data API.

Packet for query is too large

In this case, the result set returned for a row was too large. The Data API size limit is 64 KB per row in the result set returned by the database.

To solve this issue, make sure that each row in a result set is 64 KB or less.

Database response exceeded size limit

In this case, the size of the result set returned by the database was too large. The Data API limit is 1 MiB in the result set returned by the database.

To solve this issue, make sure that calls to Data API return 1 MiB of data or less. If you need to return more than 1 MiB, you can use multiple ExecuteStatement calls with the LIMIT clause in your query.

For more information about the LIMIT clause, see SELECT syntax in the MySQL documentation.

HttpEndpoint is not enabled for cluster <cluster_ID>

Check the following potential causes for this issue:

- The Aurora DB cluster doesn't support Data API. For example, for Aurora MySQL, you can only use Data API with Aurora Serverless v1. For information about the types of DB clusters RDS Data API supports, see the section called “Region and version availability”.
- Data API isn't enabled for the Aurora DB cluster. To use Data API with an Aurora DB cluster, Data API must be enabled for the DB cluster. For information about enabling Data API, see Enabling RDS Data API.
- The DB cluster was renamed after Data API was enabled for it. In that case, turn off Data API for that cluster and then enable it again.

- The ARN you specified doesn't precisely match the ARN of the cluster. Check that the ARN returned from another source or constructed by program logic matches the ARN of the cluster exactly. For example, make sure that the ARN you use has the correct letter case for all alphabetic characters.

Logging RDS Data API calls with AWS CloudTrail

RDS Data API (Data API) is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Data API. CloudTrail captures all API calls for Data API as events, including calls from the Amazon RDS console and from code calls to Data API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Data API. Using the data collected by CloudTrail, you can determine a lot of information. This information includes the request that was made to Data API, the IP address the request was made from, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Working with Data API information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When supported activity (management events) occurs in Data API, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent management events in your AWS account. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide.*

For an ongoing record of events in your AWS account, including events for Data API, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all AWS Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following topics in the *AWS CloudTrail User Guide*:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail

- Receiving CloudTrail log files from multiple Regions and Receiving CloudTrail log files from multiple accounts

All Data API operations are logged by CloudTrail and documented in the *Amazon RDS data service API reference*. For example, calls to the `BatchExecuteStatement`, `BeginTransaction`, `CommitTransaction`, and `ExecuteStatement` operations generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Including and excluding Data API events from an AWS CloudTrail trail

Most Data API users rely on the events in an AWS CloudTrail trail to provide a record of Data API operations. Event data doesn't reveal the database name, schema name, or SQL statements in requests to the Data API. However, knowing which user made a type of call against a specific DB cluster at a given time can help to detect anomalous access patterns.

Including Data API events in an AWS CloudTrail trail

For Aurora PostgreSQL Serverless v2 and provisioned databases, the following Data API operations are logged to AWS CloudTrail as *data events*. Data events are high-volume data-plane API operations that CloudTrail doesn't log by default. Additional charges apply for data events. For information about CloudTrail pricing, see AWS CloudTrail Pricing.

- BatchExecuteStatement
- BeginTransaction
- CommitTransaction
- ExecuteStatement
- RollbackTransaction

You can use the CloudTrail console,AWS CLI, or CloudTrail API operations to log these Data API operations. In the CloudTrail console, choose **RDS Data API - DB Cluster** for the Data event type. For more information, see Logging data events with the AWS Management Console in the *AWS CloudTrail User Guide*.

Using the AWS CLI, run the `aws cloudtrail put-event-selectors` command to log these Data API operations for your trail. To log all Data API events on DB clusters, specify `AWS::RDS::DBCluster` for the resource type. The following example logs all Data API events on DB clusters. For more information, see Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

```
aws cloudtrail put-event-selectors --trail-name trail_name --advanced-event-selectors \
'{
   "Name": "RDS Data API Selector",
   "FieldSelectors": [
      {
         "Field": "eventCategory",
         "Equals": [
            "Data"
         ]
      },
      {
         "Field": "resources.type",
         "Equals": [
            "AWS::RDS::DBCluster"
         ]
      }
   ]
}'
```

You can configure advanced event selectors to additionally filter on the `readOnly`, `eventName`, and `resources.ARN` fields. For more information on these fields, see AdvancedFieldSelector.

Excluding Data API events from an AWS CloudTrail trail (Aurora Serverless v1 only)

For Aurora Serverless v1, Data API events are management events. By default, all Data API events are included in an AWS CloudTrail trail. However, because Data API can generate a large number of events, you might want to exclude these events from your CloudTrail trail. The **Exclude Amazon RDS Data API events** setting excludes all Data API events from the trail. You can't exclude specific Data API events.

To exclude Data API events from a trail, do the following:

- In the CloudTrail console, choose the **Exclude Amazon RDS Data API events** setting when you create a trail or update a trail.
- In the CloudTrail API, use the PutEventSelectors operation. If you're using advanced event selectors, you can exclude Data API events by setting the `eventSource` field not equal to `rdsdata.amazonaws.com`. If you're using basic event selectors, you can exclude Data API events by setting the value of the `ExcludeManagementEventSources` attribute to `rdsdata.amazonaws.com`. For more information, see Logging events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

⚠ **Warning**

Excluding Data API events from a CloudTrail log can obscure Data API actions. Be cautious when giving principals the `cloudtrail:PutEventSelectors` permission that is required to perform this operation.

You can turn off this exclusion at any time by changing the console setting or the event selectors for a trail. The trail will then start recording Data API events. However, it can't recover Data API events that occurred while the exclusion was effective.

When you exclude Data API events by using the console or API, the resulting CloudTrail `PutEventSelectors` API operation is also logged in your CloudTrail logs. If Data API events don't appear in your CloudTrail logs, look for a `PutEventSelectors` event with the `ExcludeManagementEventSources` attribute set to `rdsdata.amazonaws.com`.

For more information, see Logging management events for trails in the *AWS CloudTrail User Guide*.

Understanding Data API log file entries

A *trail* is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An *event* represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

Aurora PostgreSQL Serverless v2 and provisioned

The following example shows a CloudTrail log entry that demonstrates the `ExecuteStatement` operation for Aurora PostgreSQL Serverless v2 and provisioned databases. For these databases, all Data API events are data events where the event source is **rdsdataapi.amazonaws.com** and the event type is **Rds Data Service**.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AKIAIOSFODNN7EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/johndoe",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
        "userName": "johndoe"
    },
    "eventTime": "2019-12-18T00:49:34Z",
    "eventSource": "rdsdataapi.amazonaws.com",
    "eventName": "ExecuteStatement",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "aws-cli/1.16.102 Python/3.7.2 Windows/10 botocore/1.12.92",
    "requestParameters": {
        "continueAfterTimeout": false,
        "database": "**********",
        "includeResultMetadata": false,
        "parameters": [],
        "resourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:my-database-1",
        "schema": "**********",
        "secretArn": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:dataapisecret-ABC123",
        "sql": "**********"
    },
    "responseElements": null,
    "requestID": "6ba9a36e-b3aa-4ca8-9a2e-15a9eada988e",
    "eventID": "a2c7a357-ee8e-4755-a0d0-aed11ed4253a",
    "eventType": "Rds Data Service",
    "recipientAccountId": "123456789012"
}
```

Aurora Serverless v1

The following example shows how the preceding example CloudTrail log entry appears for Aurora Serverless v1. For Aurora Serverless v1, all events are management events where the event source is **rdsdata.amazonaws.com** and the event type is **AwsApiCall**.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AKIAIOSFODNN7EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/johndoe",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
        "userName": "johndoe"
    },
    "eventTime": "2019-12-18T00:49:34Z",
    "eventSource": "rdsdata.amazonaws.com",
    "eventName": "ExecuteStatement",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "aws-cli/1.16.102 Python/3.7.2 Windows/10 botocore/1.12.92",
    "requestParameters": {
        "continueAfterTimeout": false,
        "database": "**********",
        "includeResultMetadata": false,
        "parameters": [],
        "resourceArn": "arn:aws:rds:us-east-1:123456789012:cluster:my-database-1",
        "schema": "**********",
        "secretArn": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:dataapisecret-ABC123",
        "sql": "**********"
    },
    "responseElements": null,
    "requestID": "6ba9a36e-b3aa-4ca8-9a2e-15a9eada988e",
    "eventID": "a2c7a357-ee8e-4755-a0d0-aed11ed4253a",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

Using the Aurora query editor

The Aurora query editor lets you run SQL statements on your Aurora DB cluster through the AWS Management Console. You can run SQL queries, data manipulation (DML) statements, and data definition (DDL) statements. Using the console interface lets you perform database maintenance, produce reports, and conduct SQL experiments. You can avoid setting up the network configuration to connect to your DB cluster from a separate client system such as an EC2 instance or a laptop computer.

The query editor requires an Aurora DB cluster with RDS Data API (Data API) enabled. For information about DB clusters that support Data API and how to enable it, see Using RDS Data API. The SQL that you can run is subject to the Data API limitations. For more information, see the section called “Limitations”.

Availability of the query editor

The query editor is available for Aurora DB clusters using Aurora MySQL and Aurora PostgreSQL engine versions that support Data API, and in the AWS Regions where Data API is available. For more information, see Supported Regions and Aurora DB engines for RDS Data API.

Authorizing access to the query editor

A user must be authorized to run queries in the query editor. You can authorize a user to run queries in the query editor by adding the `AmazonRDSDataFullAccess` policy, a predefined AWS Identity and Access Management (IAM) policy, to that user.

Note

Make sure to use the same user name and password when you create the IAM user as you did for the database user, such as the administrative user name and password. For more information, see Creating an IAM user in your AWS account in the *AWS Identity and Access Management User Guide*.

You can also create an IAM policy that grants access to the query editor. After you create the policy, add it to each user that requires access to the query editor.

The following policy provides the minimum required permissions for a user to access the query editor.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "QueryEditor0",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:PutResourcePolicy",
                "secretsmanager:PutSecretValue",
                "secretsmanager:DeleteSecret",
                "secretsmanager:DescribeSecret",
                "secretsmanager:TagResource"
            ],
            "Resource": "arn:aws:secretsmanager:*:*:secret:rds-db-credentials/*"
        },
        {
            "Sid": "QueryEditor1",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetRandomPassword",
                "tag:GetResources",
                "secretsmanager:CreateSecret",
                "secretsmanager:ListSecrets",
                "dbqms:CreateFavoriteQuery",
                "dbqms:DescribeFavoriteQueries",
                "dbqms:UpdateFavoriteQuery",
                "dbqms:DeleteFavoriteQueries",
                "dbqms:GetQueryString",
                "dbqms:CreateQueryHistory",
                "dbqms:UpdateQueryHistory",
                "dbqms:DeleteQueryHistory",
                "dbqms:DescribeQueryHistory",
                "rds-data:BatchExecuteStatement",
                "rds-data:BeginTransaction",
                "rds-data:CommitTransaction",
                "rds-data:ExecuteStatement",
                "rds-data:RollbackTransaction"
            ],
            "Resource": "*"
        }
    ]
}
```

For information about creating an IAM policy, see Creating IAM policies in the *AWS Identity and Access Management User Guide*.

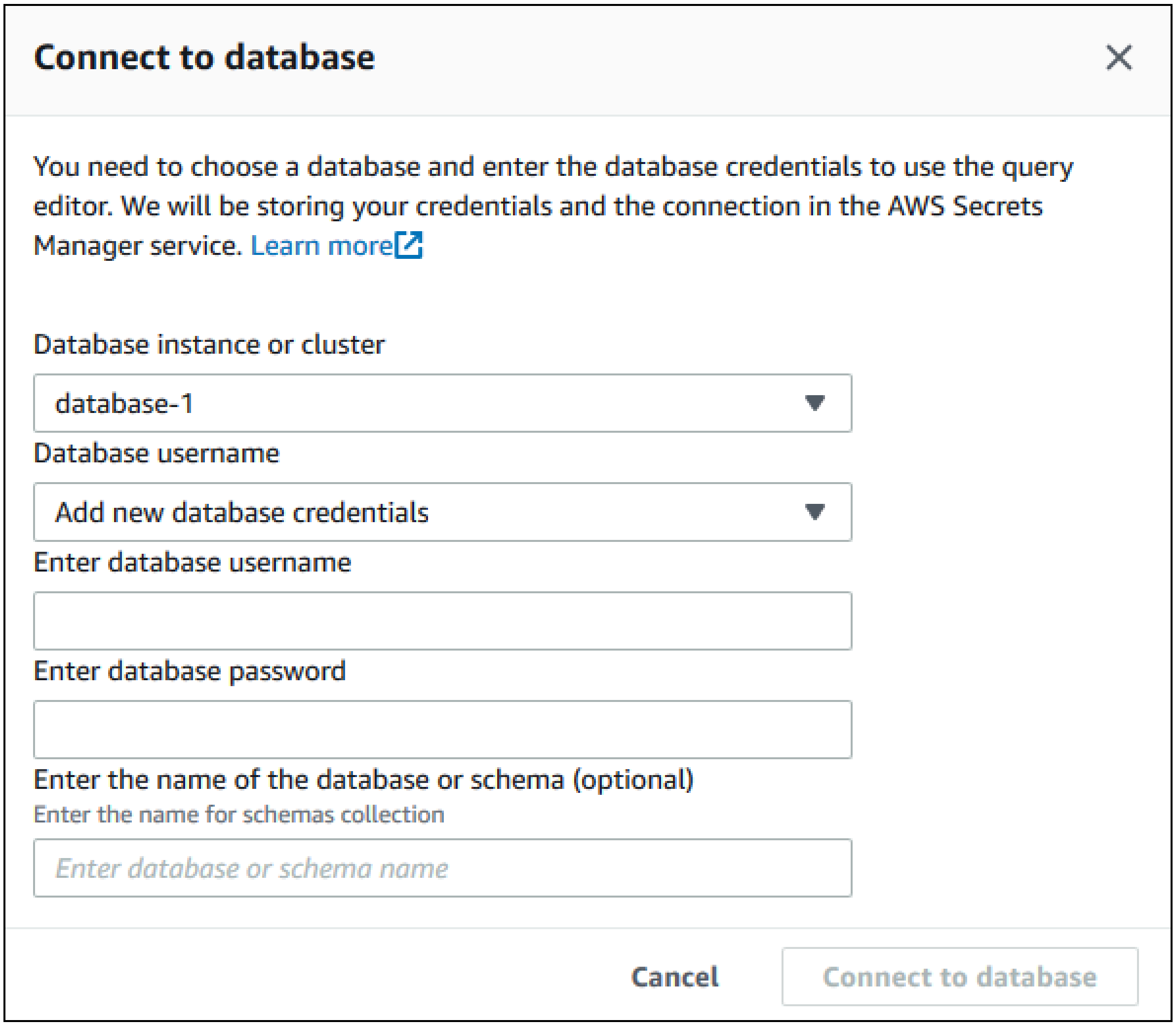
For information about adding an IAM policy to a user, see Adding and removing IAM identity permissions in the *AWS Identity and Access Management User Guide*.

Running queries in the query editor

You can run SQL statements on an Aurora DB cluster in the query editor. The SQL that you can run is subject to the Data API limitations. For more information, see the section called “Limitations”.

To run a query in the query editor

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the AWS Management Console, choose the AWS Region in which you created the Aurora DB clusters that you want to query.
3. In the navigation pane, choose **Databases**.
4. Choose the Aurora DB cluster that you want to run SQL queries on.
5. For **Actions**, choose **Query**. If you haven't connected to the database before, the **Connect to database** page opens.



Connect to database

You need to choose a database and enter the database credentials to use the query editor. We will be storing your credentials and the connection in the AWS Secrets Manager service. Learn more

Database instance or cluster: database-1

Database username: Add new database credentials

Enter database username

Enter database password

Enter the name of the database or schema (optional)

Enter the name for schemas collection

Enter database or schema name

6. Enter the following information:

   a. For **Database instance or cluster**, choose the Aurora DB cluster that you want to run SQL queries on.

   b. For **Database username**, choose the user name of the database user to connect with, or choose **Add new database credentials**. If you choose **Add new database credentials**, enter the user name for the new database credentials in **Enter database username**.

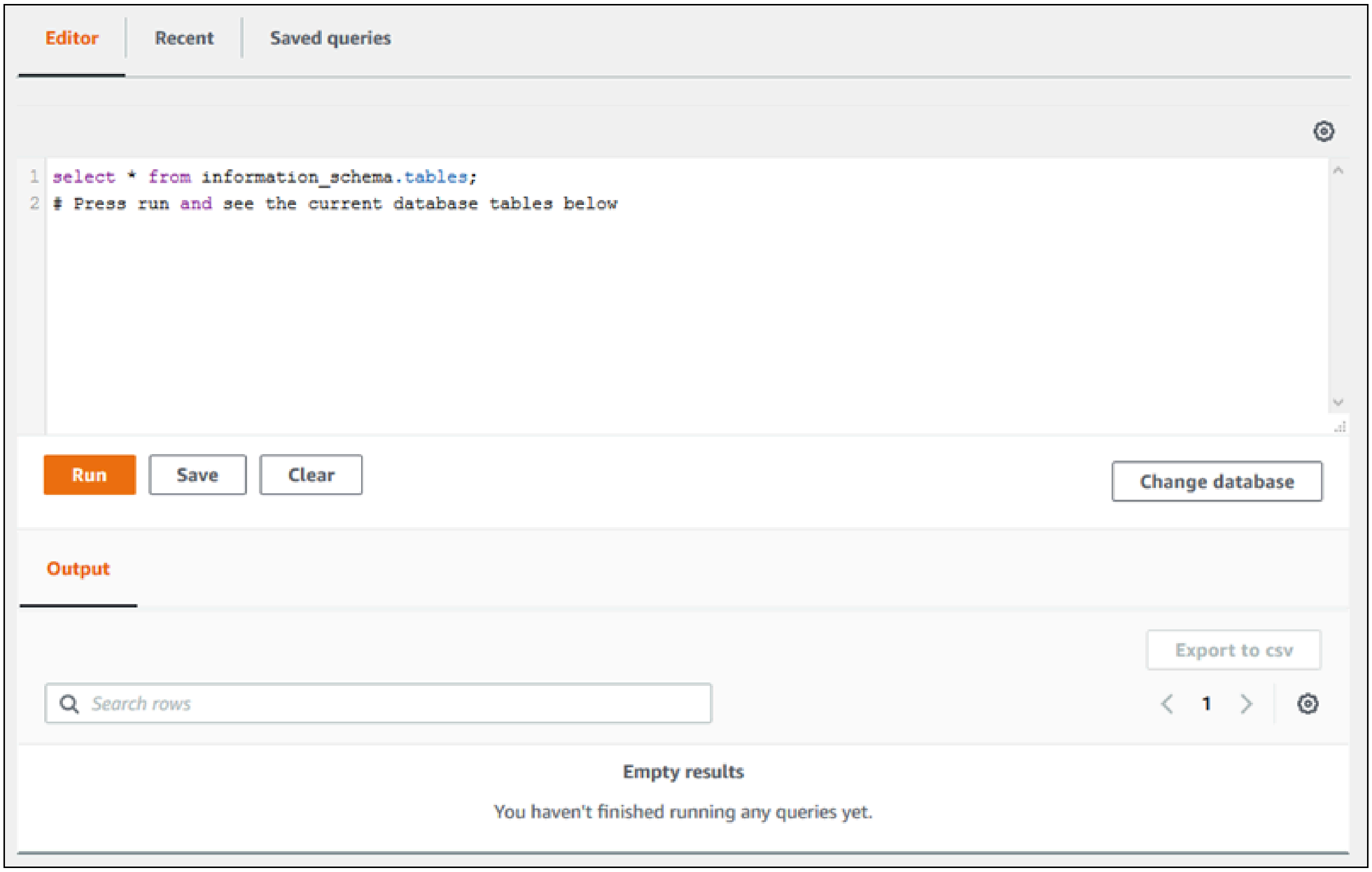
   c. For **Enter database password**, enter the password for the database user that you chose.

   d. In the last box, enter the name of the database or schema that you want to use for the Aurora DB cluster.

   e. Choose **Connect to database**.

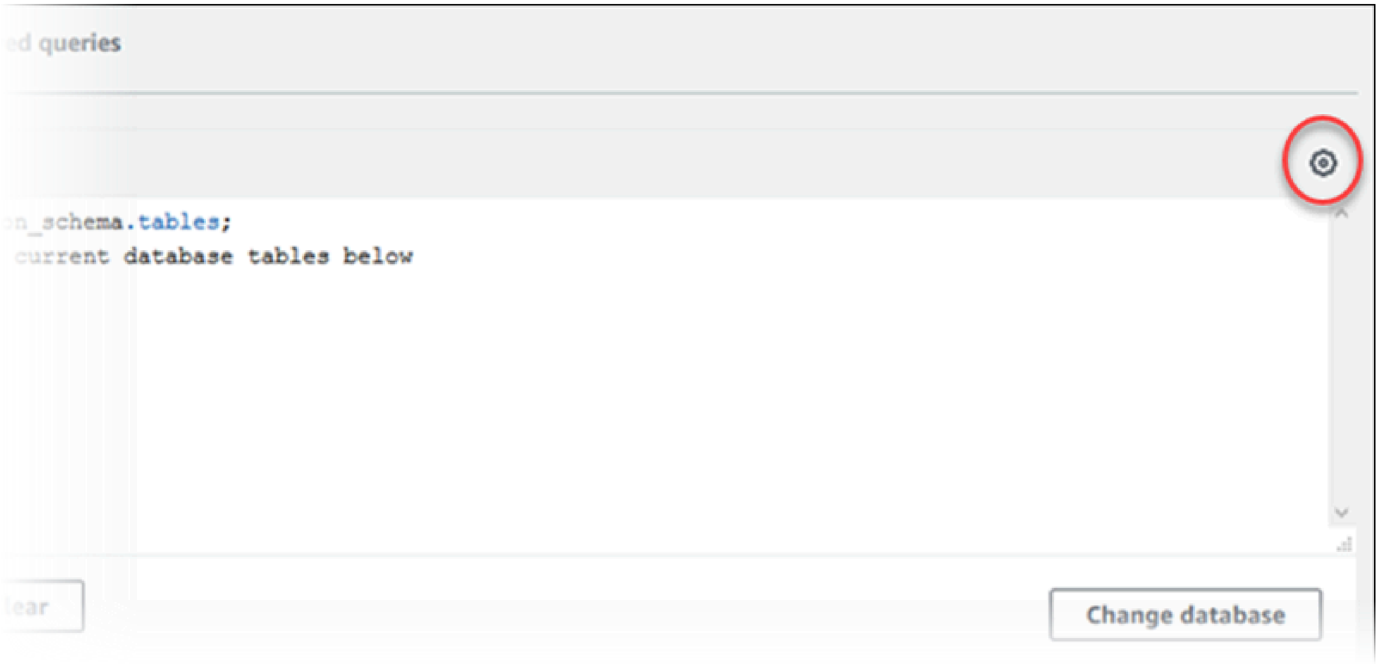
> **Note**
>
> If your connection is successful, your connection and authentication information are stored in AWS Secrets Manager. You don't need to enter the connection information again.

7. In the query editor, enter the SQL query that you want to run on the database.



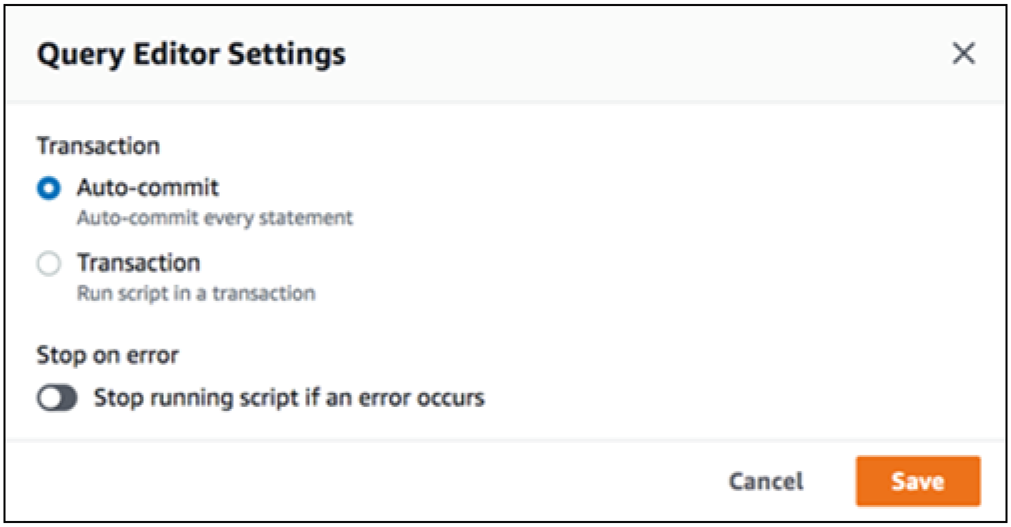
```
select * from information_schema.tables;
# Press run and see the current database tables below
```

Each SQL statement can commit automatically, or you can run SQL statements in a script as part of a transaction. To control this behavior, choose the gear icon above the query window.



```
n_schema.tables;
current database tables below
```

The **Query Editor Settings** window appears.



Query Editor Settings ✕

Transaction

● Auto-commit
Auto-commit every statement

○ Transaction
Run script in a transaction

Stop on error

Stop running script if an error occurs

If you choose **Auto-commit**, every SQL statement commits automatically. If you choose **Transaction**, you can run a group of statements in a script. Statements are automatically committed at the end of the script unless explicitly committed or rolled back before then. Also, you can choose to stop a running script if an error occurs by enabling **Stop on error**.

> **Note**
>
> In a group of statements, data definition language (DDL) statements can cause previous data manipulation language (DML) statements to commit. You can also include `COMMIT` and `ROLLBACK` statements in a group of statements in a script.

After you make your choices in the **Query Editor Settings** window, choose **Save**.

8. Choose **Run** or press Ctrl+Enter, and the query editor displays the results of your query.

   After running the query, save it to **Saved queries** by choosing **Save**.

   Export the query results to spreadsheet format by choosing **Export to csv**.

You can find, edit, and rerun previous queries. To do so, choose the **Recent** tab or the **Saved queries** tab, choose the query text, and then choose **Run**.

To change the database, choose **Change database**.

Database Query Metadata Service (DBQMS) API reference

The Database Query Metadata Service (`dbqms`) is an internal-only service. It provides your recent and saved queries for the query editor on the AWS Management Console for multiple AWS services, including Amazon RDS.

The following DBQMS actions are supported:

Topics

- CreateFavoriteQuery
- CreateQueryHistory
- CreateTab
- DeleteFavoriteQueries
- DeleteQueryHistory
- DeleteTab
- DescribeFavoriteQueries

- DescribeQueryHistory
- DescribeTabs
- GetQueryString
- UpdateFavoriteQuery
- UpdateQueryHistory
- UpdateTab

CreateFavoriteQuery

Save a new favorite query. Each user can create up to 1000 saved queries. This limit is subject to change in the future.

CreateQueryHistory

Save a new query history entry.

CreateTab

Save a new query tab. Each user can create up to 10 query tabs.

DeleteFavoriteQueries

Delete one or more saved queries.

DeleteQueryHistory

Delete query history entries.

DeleteTab

Delete query tab entries.

DescribeFavoriteQueries

List saved queries created by a user in a given account.

DescribeQueryHistory

List query history entries.

DescribeTabs

List query tabs created by a user in a given account.

GetQueryString

Retrieve full query text from a query ID.

UpdateFavoriteQuery

Update the query string, description, name, or expiration date.

UpdateQueryHistory

Update the status of query history.

UpdateTab

Update the query tab name and query string.

Using Amazon Aurora machine learning

By using Amazon Aurora machine learning, you can integrate your Aurora DB cluster with one of the following AWS machine learning services, depending on your needs. They each support specific machine learning use cases.

Amazon Bedrock

Amazon Bedrock is a fully managed service that makes leading foundation models from AI companies available through an API, along with developer tooling to help build and scale generative AI applications. With Amazon Bedrock, you pay to run inference on any of the third-party foundation models. Pricing is based on the volume of input tokens and output tokens, and on whether you have purchased provisioned throughput for the model. For more information, see What is Amazon Bedrock? in the *Amazon Bedrock User Guide*.

Amazon Comprehend

Amazon Comprehend is a managed natural language processing (NLP) service that's used to extract insights from documents. With Amazon Comprehend, you can deduce sentiment based on the content of documents, by analyzing entities, key phrases, language, and other features. To learn more, see What is Amazon Comprehend? in the *Amazon Comprehend Developer Guide*.

SageMaker

Amazon SageMaker is a fully managed machine learning service. Data scientists and developers use Amazon SageMaker to build, train, and test machine learning models for a variety of inference tasks, such as fraud detection and product recommendation. When a machine learning model is ready for use in production, it can be deployed to the Amazon SageMaker hosted environment. For more information, see What Is Amazon SageMaker? in the *Amazon SageMaker Developer Guide*.

Using Amazon Comprehend with your Aurora DB cluster has less preliminary setup than using SageMaker. If you're new to AWS machine learning, we recommend that you start by exploring Amazon Comprehend.

Topics

Using Amazon Aurora machine learning with Aurora MySQL

By using Amazon Aurora machine learning with your Aurora MySQL DB cluster, you can use Amazon Bedrock, Amazon Comprehend, or Amazon SageMaker, depending on your needs. They each support different machine learning use cases.

Contents

Requirements for using Aurora machine learning with Aurora MySQL

AWS machine learning services are managed services that are set up and run in their own production environments. Aurora machine learning supports integration with Amazon Bedrock, Amazon Comprehend, and SageMaker. Before trying to set up your Aurora MySQL DB cluster to use Aurora machine learning, be sure you understand the following requirements and prerequisites.

- The machine learning services must be running in the same AWS Region as your Aurora MySQL DB cluster. You can't use machine learning services from an Aurora MySQL DB cluster in a different Region.
- If your Aurora MySQL DB cluster is in a different virtual public cloud (VPC) from your Amazon Bedrock, Amazon Comprehend, or SageMaker service, the VPC's Security group needs to allow outbound connections to the target Aurora machine learning service. For more information, see Control traffic to your AWS resources using security groups in the *Amazon VPC User Guide*.
- You can upgrade an Aurora cluster that's running a lower version of Aurora MySQL to a supported higher version if you want to use Aurora machine learning with that cluster. For more information, see Database engine updates for Amazon Aurora MySQL.
- Your Aurora MySQL DB cluster must use a custom DB cluster parameter group. At the end of the setup process for each Aurora machine learning service that you want to use, you add the Amazon Resource Name (ARN) of the associated IAM role that was created for the service. We recommend that you create a custom DB cluster parameter group for your Aurora MySQL in advance and configure your Aurora MySQL DB cluster to use it so that it's ready for you to modify at the end of the setup process.
- For SageMaker:
  - The machine learning components that you want to use for inferences must be set up and ready to use. During the configuration process for your Aurora MySQL DB cluster, make sure to have the ARN of the SageMaker endpoint available. The data scientists on your team are likely best able to handle working with SageMaker to prepare the models and handle the other such tasks. To get started with Amazon SageMaker, see Get Started with Amazon SageMaker. For more information about inferences and endpoints, see Real-time inference.
  - To use SageMaker with your own training data, you must set up an Amazon S3 bucket as part of your Aurora MySQL configuration for Aurora machine learning. To do so, you follow the same general process as for setting up the SageMaker integration. For a summary of this optional setup process, see Setting up your Aurora MySQL DB cluster to use Amazon S3 for SageMaker (Optional).

- For Aurora global databases, you set up the Aurora machine learning services that you want to use in all AWS Regions that make up your Aurora global database. For example, if you want to use Aurora machine learning with SageMaker for your Aurora global database, you do the following for every Aurora MySQL DB cluster in every AWS Region:
  - Set up the Amazon SageMaker services with the same SageMaker training models and endpoints. These must also use the same names.
  - Create the IAM roles as detailed in Setting up your Aurora MySQL DB cluster to use Aurora machine learning.
  - Add the ARN of the IAM role to the custom DB cluster parameter group for each Aurora MySQL DB cluster in every AWS Region.

  These tasks require that Aurora machine learning is available for your version of Aurora MySQL in all AWS Regions that make up your Aurora global database.

Region and version availability

Feature availability and support varies across specific versions of each Aurora database engine, and across AWS Regions.

- For information on version and Region availability for Amazon Comprehend and Amazon SageMaker with Aurora MySQL, see Aurora machine learning with Aurora MySQL.
- Amazon Bedrock is supported only on Aurora MySQL version 3.06 and higher.

  For information on Region availability for Amazon Bedrock, see Supported models in Amazon Bedrock in the *Amazon Bedrock User Guide*.

Supported features and limitations of Aurora machine learning with Aurora MySQL

When using Aurora MySQL with Aurora machine learning, the following limitations apply:

- The Aurora machine learning extension doesn't support vector interfaces.
- Aurora machine learning integrations aren't supported when used in a trigger.
- Aurora machine learning functions aren't compatible with binary logging (binlog) replication.
  - The setting `--binlog-format=STATEMENT` throws an exception for calls to Aurora machine learning functions.

  - Aurora machine learning functions are nondeterministic, and nondeterministic stored functions aren't compatible with the binlog format.

  For more information, see Binary Logging Formats in the MySQL documentation.

- Stored functions that call tables with generated-always columns aren't supported. This applies to any Aurora MySQL stored function. To learn more about this column type, see CREATE TABLE and Generated Columns in the MySQL documentation.
- Amazon Bedrock functions don't support `RETURNS JSON`. You can use `CONVERT` or `CAST` to convert from `TEXT` to `JSON` if needed.
- Amazon Bedrock doesn't support batch requests.
- Aurora MySQL supports any SageMaker endpoint that reads and writes the comma-separated value (CSV) format, through a `ContentType` of `text/csv`. This format is accepted by the following built-in SageMaker algorithms:
  - Linear Learner
  - Random Cut Forest
  - XGBoost

  To learn more about these algorithms, see Choose an Algorithm in the *Amazon SageMaker Developer Guide*.

Setting up your Aurora MySQL DB cluster to use Aurora machine learning

In the following topics, you can find separate setup procedures for each of these Aurora machine learning services.

Topics

- Granting access to SageMaker functions

Setting up your Aurora MySQL DB cluster to use Amazon Bedrock

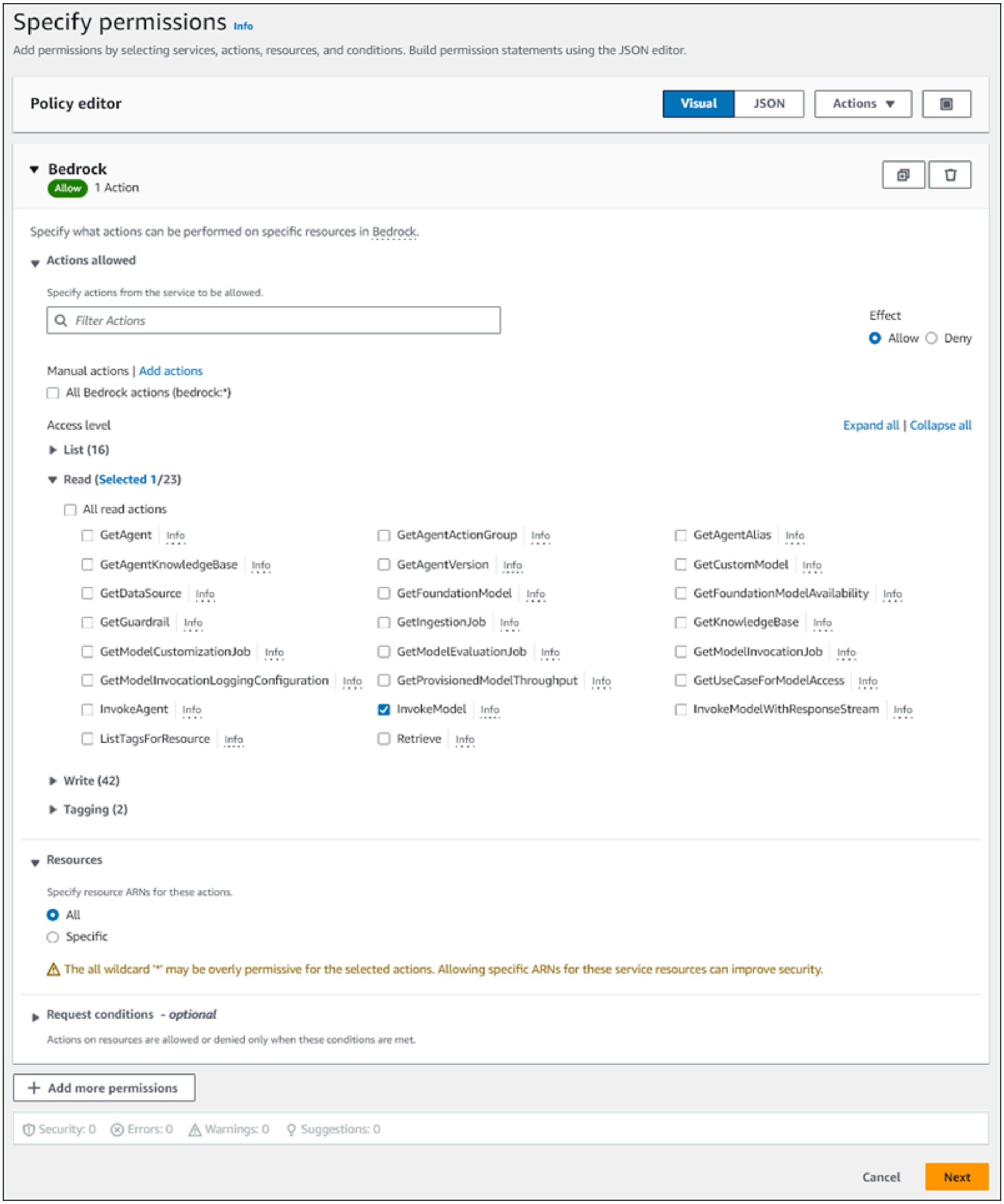
Aurora machine learning relies on AWS Identity and Access Management (IAM) roles and policies to allow your Aurora MySQL DB cluster to access and use the Amazon Bedrock services. The following procedures create an IAM permission policy and role so that your DB cluster can integrate with Amazon Bedrock.

To create the IAM policy

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Policies** in the navigation pane.
3. Choose **Create a policy**.
4. On the **Specify permissions** page, for **Select a service**, choose **Bedrock**.

   The Amazon Bedrock permissions display.
5. Expand **Read**, then select **InvokeModel**.
6. For **Resources**, select **All**.

   The **Specify permissions** page should resemble the following figure.



Specify permissions Info

Add permissions by selecting services, actions, resources, and conditions. Build permission statements using the JSON editor.

Policy editor

▼ Bedrock

Allow 1 Action

Specify what actions can be performed on specific resources in Bedrock.

▼ Actions allowed

Specify actions from the service to be allowed.

Effect

● Allow ○ Deny

Manual actions | Add actions

□ All Bedrock actions (bedrock:*)

Access level

Expand all | Collapse all

▶ List (16)

▼ Read (Selected 1/23)

□ All read actions

□ GetAgent Info
□ GetAgentActionGroup Info
□ GetAgentAlias Info
□ GetAgentKnowledgeBase Info
□ GetAgentVersion Info
□ GetCustomModel Info
□ GetDataSource Info
□ GetFoundationModel Info
□ GetFoundationModelAvailability Info
□ GetGuardrail Info
□ GetIngestionJob Info
□ GetKnowledgeBase Info
□ GetModelCustomizationJob Info
□ GetModelEvaluationJob Info
□ GetModelInvocationJob Info
□ GetModelInvocationLoggingConfiguration Info
□ GetProvisionedModelThroughput Info
□ GetUseCaseForModelAccess Info
□ InvokeAgent Info
☑ InvokeModel Info
□ InvokeModelWithResponseStream Info
□ ListTagsForResource Info
□ Retrieve Info

▶ Write (42)

▶ Tagging (2)

▼ Resources

Specify resource ARNs for these actions.

● All

○ Specific

⚠ The all wildcard '*' may be overly permissive for the selected actions. Allowing specific ARNs for these service resources can improve security.

▶ Request conditions - *optional*

Actions on resources are allowed or denied only when these conditions are met.

7. Choose **Next**.

8. On the **Review and create** page, enter a name for your policy, for example **`BedrockInvokeModel`**.

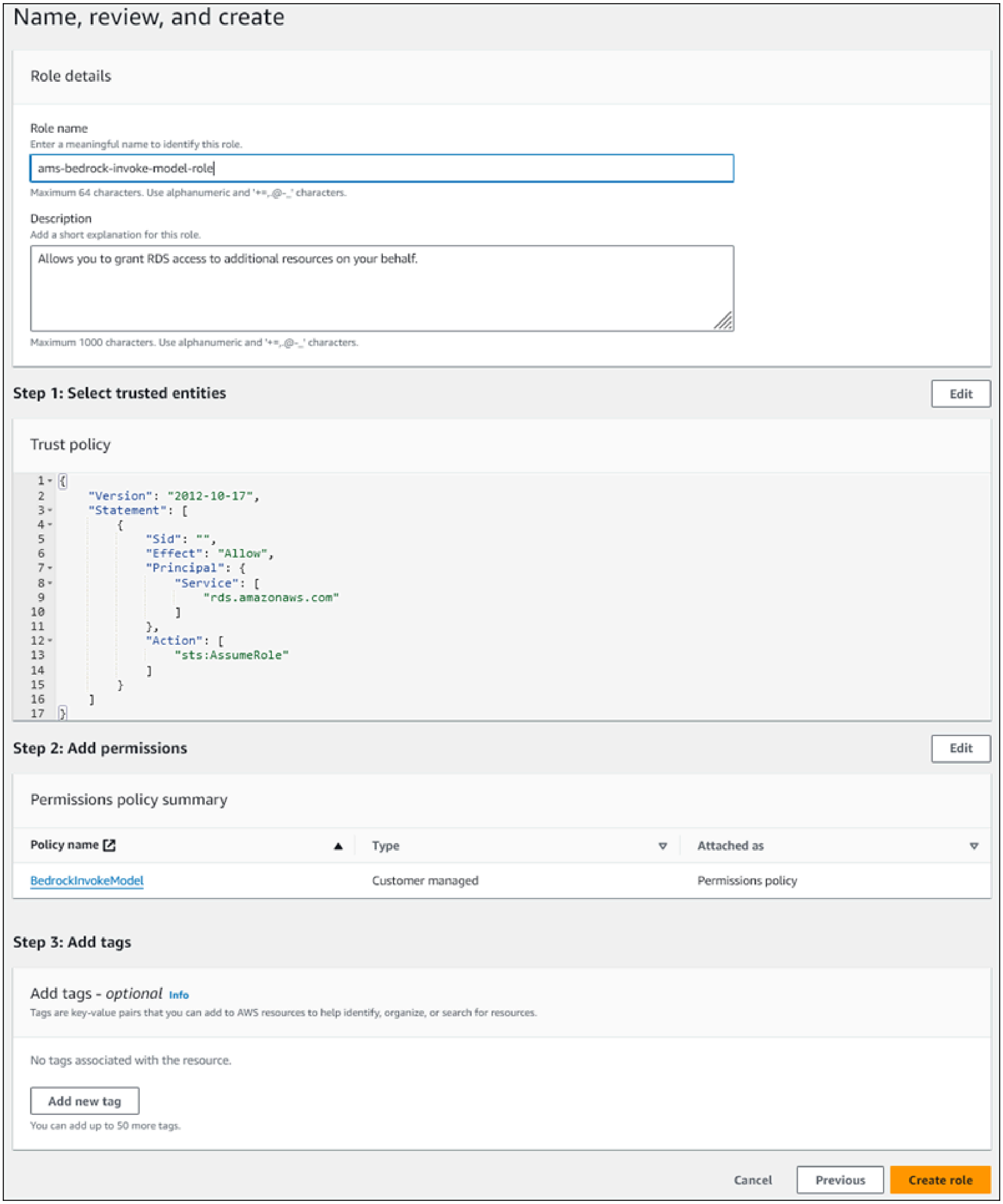
9. Review your policy, then choose **Create policy**.

Next you create the IAM role that uses the Amazon Bedrock permission policy.

To create the IAM role

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Roles** in the navigation pane.
3. Choose **Create role**.
4. On the **Select trusted entity** page, for **Use case**, choose **RDS**.
5. Select **RDS - Add Role to Database**, then choose **Next**.
6. On the **Add permissions** page, for **Permissions policies**, select the IAM policy that you created, then choose **Next**.
7. On the **Name, review, and create** page, enter a name for your role, for example **`ams-bedrock-invoke-model-role`**.

   The role should resemble the following figure.



Name, review, and create

Role details

Role name
Enter a meaningful name to identify this role.
ams-bedrock-invoke-model-role
Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.

Description
Add a short explanation for this role.
Allows you to grant RDS access to additional resources on your behalf.
Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.

Step 1: Select trusted entities

Trust policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "rds.amazonaws.com"
                ]
            },
            "Action": [
                "sts:AssumeRole"
            ]
        }
    ]
}
```

Step 2: Add permissions

Permissions policy summary

Policy name	Type	Attached as
BedrockInvokeModel	Customer managed	Permissions policy

Step 3: Add tags

Add tags - *optional* Info
Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.

No tags associated with the resource.

You can add up to 50 more tags.

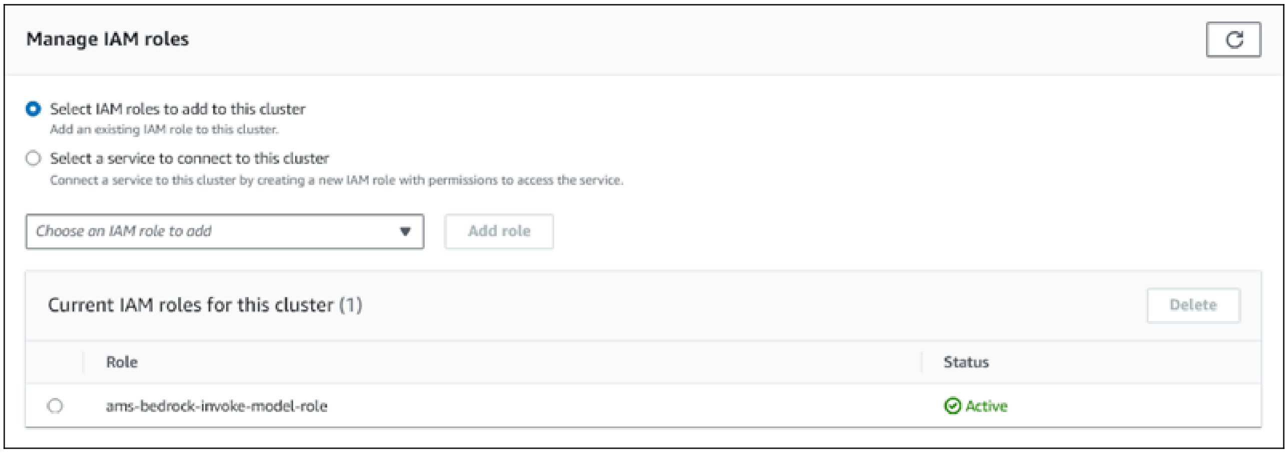
8. Review your role, then choose **Create role**.

Next you associate the Amazon Bedrock IAM role with your DB cluster.

To associate the IAM role with your DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** from the navigation pane.
3. Choose the Aurora MySQL DB cluster that you want to connect to Amazon Bedrock services.
4. Choose the **Connectivity & security** tab.
5. For **Manage IAM roles** section, choose **Select IAM to add to this cluster**.
6. Choose the IAM that you created, and then choose **Add role**.

   The IAM role is associated with your DB cluster, first with the status **Pending**, then **Active**. When the process completes, you can find the role in the **Current IAM roles for this cluster** list.



Manage IAM roles

● Select IAM roles to add to this cluster
Add an existing IAM role to this cluster.

○ Select a service to connect to this cluster
Connect a service to this cluster by creating a new IAM role with permissions to access the service.

Choose an IAM role to add ▼ | Add role

Current IAM roles for this cluster (1) | Delete

	Role	Status
○	ams-bedrock-invoke-model-role	✓ Active

You must add the ARN of this IAM role to the `aws_default_bedrock_role` parameter of the custom DB cluster parameter group associated with your Aurora MySQL DB cluster. If your Aurora MySQL DB cluster doesn't use a custom DB cluster parameter group, you need to create one to use with your Aurora MySQL DB cluster to complete the integration. For more information, see [Working with DB cluster parameter groups](#).

To configure the DB cluster parameter

1. In the Amazon RDS Console, open the **Configuration** tab of your Aurora MySQL DB cluster.
2. Locate the DB cluster parameter group configured for your cluster. Choose the link to open your custom DB cluster parameter group, then choose **Edit**.

3. Find the `aws_default_bedrock_role` parameter in your custom DB cluster parameter group.
4. In the **Value** field, enter the ARN of the IAM role.
5. Choose **Save changes** to save the setting.
6. Reboot the primary instance of your Aurora MySQL DB cluster so that this parameter setting takes effect.

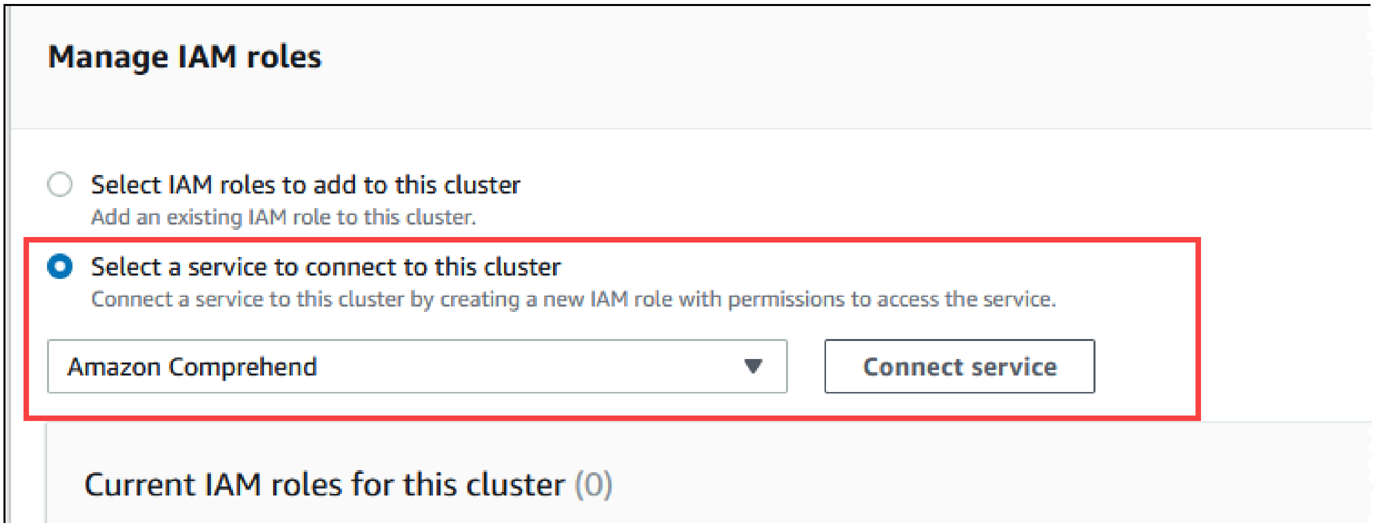
The IAM integration for Amazon Bedrock is complete. Continue setting up your Aurora MySQL DB cluster to work with Amazon Bedrock by Granting database users access to Aurora machine learning.

Setting up your Aurora MySQL DB cluster to use Amazon Comprehend

Aurora machine learning relies on AWS Identity and Access Management roles and policies to allow your Aurora MySQL DB cluster to access and use the Amazon Comprehend services. The following procedure automatically creates an IAM role and policy for your cluster so that it can use Amazon Comprehend.

To set up your Aurora MySQL DB cluster to use Amazon Comprehend

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** from the navigation pane.
3. Choose the Aurora MySQL DB cluster that you want to connect to Amazon Comprehend services.
4. Choose the **Connectivity & security** tab.
5. For **Manage IAM roles** section, choose **Select a service to connect to this cluster**.
6. Choose **Amazon Comprehend** from the menu, and then choose **Connect service**.



Manage IAM roles

○ Select IAM roles to add to this cluster
Add an existing IAM role to this cluster.

● Select a service to connect to this cluster
Connect a service to this cluster by creating a new IAM role with permissions to access the service.

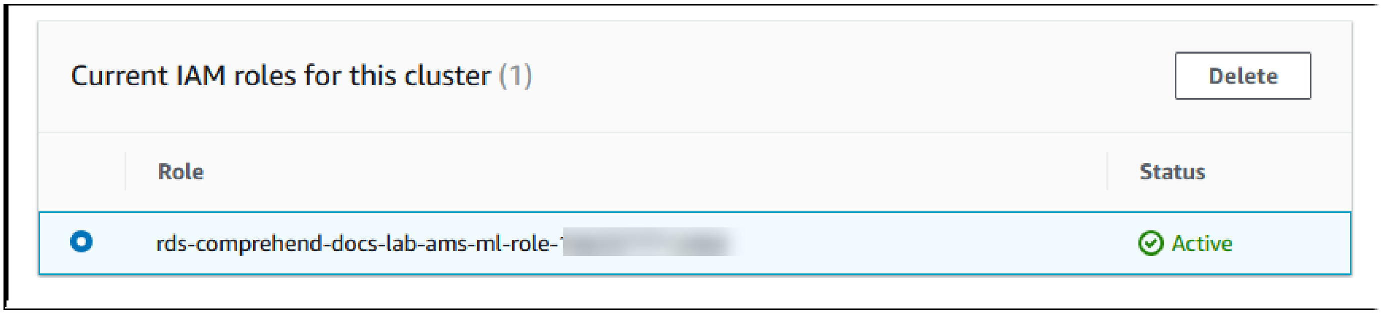
Amazon Comprehend ▼

Current IAM roles for this cluster (0)

7. The **Connect cluster to Amazon Comprehend** dialog doesn't require any additional information. However, you might see a message notifying you that the integration between Aurora and Amazon Comprehend is currently in preview. Be sure to read the message before you continue. You can choose **Cancel** if you prefer not to proceed.

8. Choose **Connect service** to complete the integration process.

   Aurora creates the IAM role. It also creates the policy that allows the Aurora MySQL DB cluster to use Amazon Comprehend services and attaches the policy to the role. When the process completes, you can find the role in the **Current IAM roles for this cluster** list as shown in the following image.



Current IAM roles for this cluster (1)

	Role	Status
●	rds-comprehend-docs-lab-ams-ml-role-[redacted]	✓ Active

   You need to add the ARN of this IAM role to the `aws_default_comprehend_role` parameter of the custom DB cluster parameter group associated with your Aurora MySQL DB cluster. If your Aurora MySQL DB cluster doesn't use a custom DB cluster parameter group, you need to create one to use with your Aurora MySQL DB cluster to complete the integration. For more information, see Working with DB cluster parameter groups.

   After creating your custom DB cluster parameter group and associating it with your Aurora MySQL DB cluster, you can continue following these steps.

If your cluster uses a custom DB cluster parameter group, do as follows.

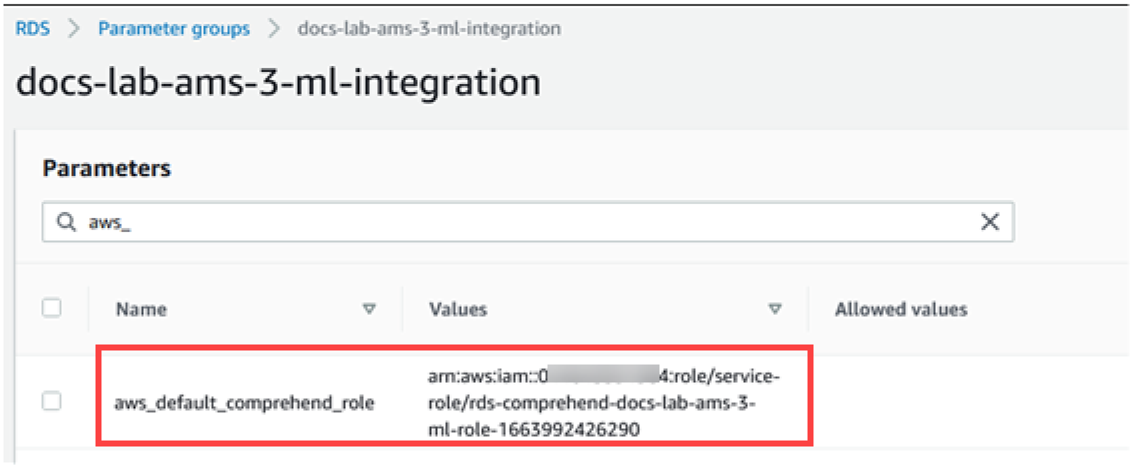
a. In the Amazon RDS Console, open the **Configuration** tab of your Aurora MySQL DB cluster.

b. Locate the DB cluster parameter group configured for your cluster. Choose the link to open your custom DB cluster parameter group, then choose **Edit**.

c. Find the `aws_default_comprehend_role` parameter in your custom DB cluster parameter group.

d. In the **Value** field, enter the ARN of the IAM role.

e. Choose **Save changes** to save the setting. In the following image, you can find an example.



RDS > Parameter groups > docs-lab-ams-3-ml-integration

docs-lab-ams-3-ml-integration

Parameters

aws_

□	Name	Values	Allowed values
□	aws_default_comprehend_role	arn:aws:iam::0[redacted]4:role/service-role/rds-comprehend-docs-lab-ams-3-ml-role-1663992426290	

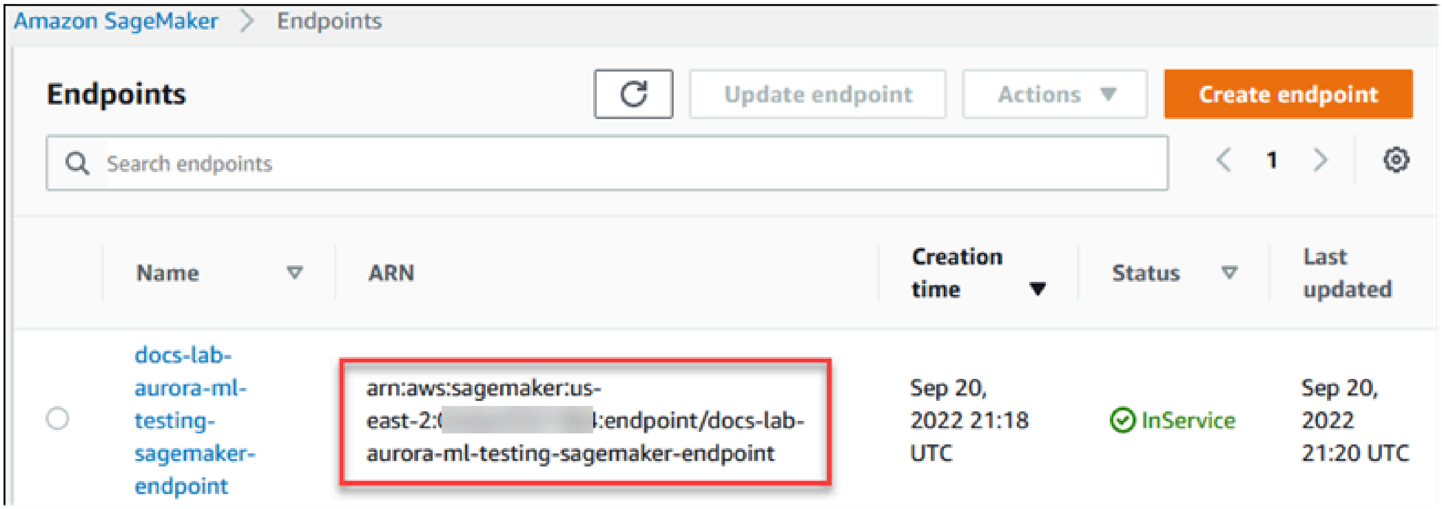
Reboot the primary instance of your Aurora MySQL DB cluster so that this parameter setting takes effect.

The IAM integration for Amazon Comprehend is complete. Continue setting up your Aurora MySQL DB cluster to work with Amazon Comprehend by granting access to the appropriate database users.

Setting up your Aurora MySQL DB cluster to use SageMaker

The following procedure automatically creates the IAM role and policy for your Aurora MySQL DB cluster so that it can use SageMaker. Before trying to follow this procedure, be sure that you have the SageMaker endpoint available so that you can enter it when needed. Typically, data scientists on your team would do the work to produce an endpoint that you can use from your Aurora MySQL

DB cluster. You can find such endpoints in the [SageMaker console](). In the navigation pane, open the **Inference** menu and choose **Endpoints**. In the following image, you can find an example.



Amazon SageMaker > Endpoints

Endpoints

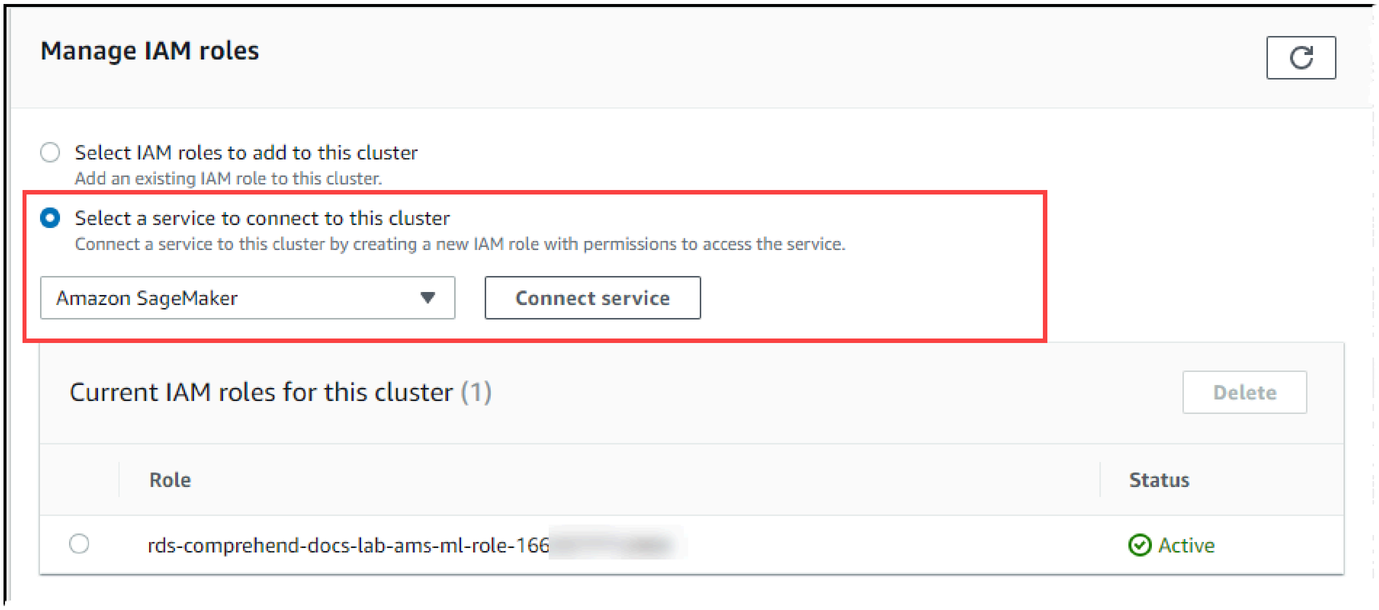
Update endpoint | Actions ▼ | Create endpoint

Search endpoints

	Name	ARN	Creation time	Status	Last updated
○	docs-lab-aurora-ml-testing-sagemaker-endpoint	arn:aws:sagemaker:us-east-2:[redacted]:endpoint/docs-lab-aurora-ml-testing-sagemaker-endpoint	Sep 20, 2022 21:18 UTC	✓ InService	Sep 20, 2022 21:20 UTC

To set up your Aurora MySQL DB cluster to use SageMaker

1. Sign in to the AWS Management Console and open the Amazon RDS console at [https://console.aws.amazon.com/rds/]().
2. Choose **Databases** from the Amazon RDS navigation menu and then choose the Aurora MySQL DB cluster that you want to connect to SageMaker services.
3. Choose the **Connectivity & security** tab.
4. Scroll to the **Manage IAM roles** section, and then choose **Select a service to connect to this cluster**. Choose **SageMaker** from the selector.



Manage IAM roles

○ Select IAM roles to add to this cluster
Add an existing IAM role to this cluster.

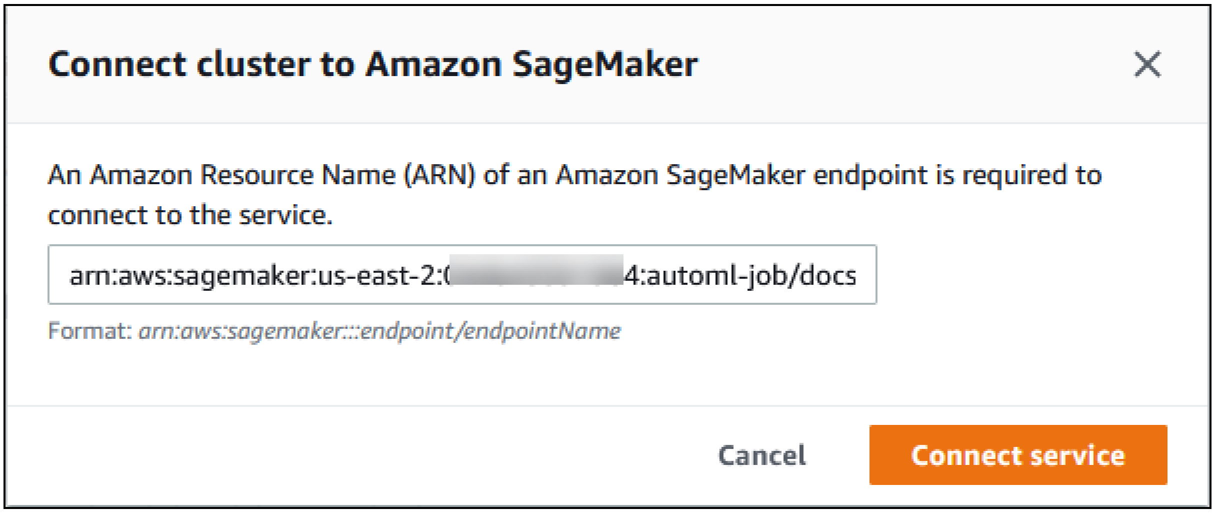
● Select a service to connect to this cluster
Connect a service to this cluster by creating a new IAM role with permissions to access the service.

Amazon SageMaker ▼ | Connect service

Current IAM roles for this cluster (1) | Delete

	Role	Status
○	rds-comprehend-docs-lab-ams-ml-role-166[redacted]	✓ Active

5. Choose **Connect service**.
6. In the **Connect cluster to SageMaker** dialog, enter the ARN of the SageMaker endpoint.



Connect cluster to Amazon SageMaker

An Amazon Resource Name (ARN) of an Amazon SageMaker endpoint is required to connect to the service.

arn:aws:sagemaker:us-east-2:[redacted]4:automl-job/docs

Format: *arn:aws:sagemaker:::endpoint/endpointName*

7. Aurora creates the IAM role. It also creates the policy that allows the Aurora MySQL DB cluster to use SageMaker services and attaches the policy to the role. When the process completes, you can find the role in the **Current IAM roles for this cluster** list.
8. Open the IAM console at https://console.aws.amazon.com/iam/.
9. Choose **Roles** from the Access management section of the AWS Identity and Access Management navigation menu.
10. Find the role from among those listed. Its name uses the following pattern.

```
rds-sagemaker-your-cluster-name-role-auto-generated-digits
```

11. Open the role's Summary page and locate the ARN. Note the ARN or copy it using the copy widget.
12. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
13. Choose your Aurora MySQL DB cluster, and then choose its **Configuration** tab.
14. Locate the DB cluster parameter group, and choose the link to open your custom DB cluster parameter group. Find the `aws_default_sagemaker_role` parameter and enter the ARN of the IAM role in the Value field and Save the setting.
15. Reboot the primary instance of your Aurora MySQL DB cluster so that this parameter setting takes effect.

The IAM setup is now complete. Continue setting up your Aurora MySQL DB cluster to work with SageMaker by granting access to the appropriate database users.

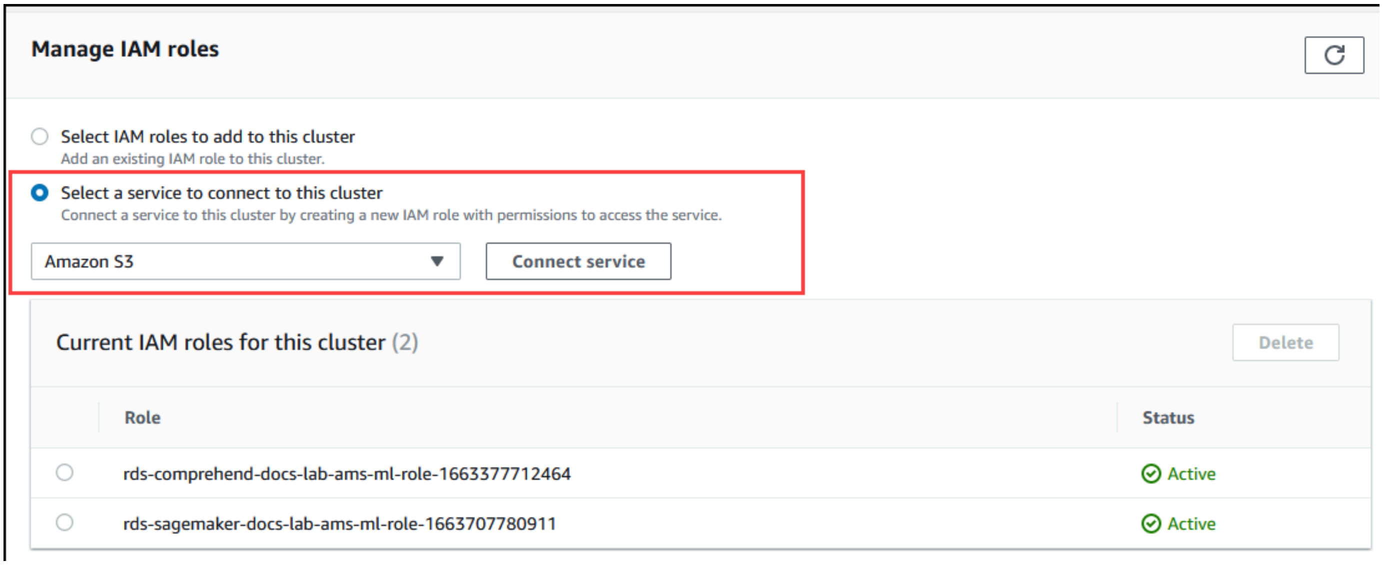
If you want to use your SageMaker models for training rather than using pre-built SageMaker components, you also need to add the Amazon S3 bucket to your Aurora MySQL DB cluster, as outlined in the Setting up your Aurora MySQL DB cluster to use Amazon S3 for SageMaker (Optional) that follows.

Setting up your Aurora MySQL DB cluster to use Amazon S3 for SageMaker (Optional)

To use SageMaker with your own models rather than using the pre-built components provided by SageMaker, you need to set up an Amazon S3 bucket for the Aurora MySQL DB cluster to use. For more information about creating an Amazon S3 bucket, see Creating a bucket in the *Amazon Simple Storage Service User Guide*.

To set up your Aurora MySQL DB cluster to use an Amazon S3 bucket for SageMaker

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. Choose **Databases** from the Amazon RDS navigation menu and then choose the Aurora MySQL DB cluster that you want to connect to SageMaker services.
3. Choose the **Connectivity & security** tab.
4. Scroll to the **Manage IAM roles** section, and then choose **Select a service to connect to this cluster**. Choose **Amazon S3** from the selector.



Manage IAM roles

○ Select IAM roles to add to this cluster
Add an existing IAM role to this cluster.

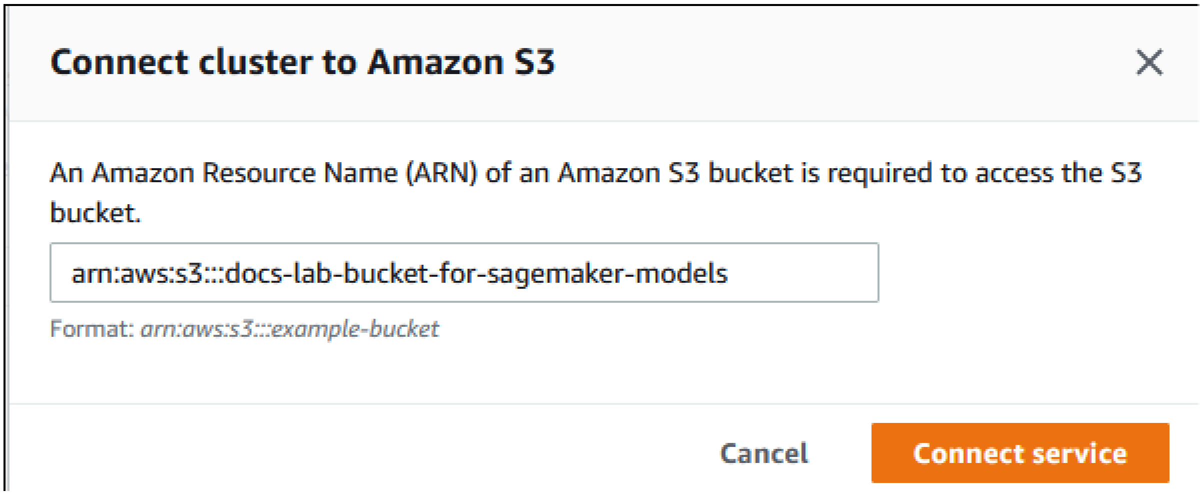
● Select a service to connect to this cluster
Connect a service to this cluster by creating a new IAM role with permissions to access the service.

Amazon S3 ▼ | Connect service

Current IAM roles for this cluster (2) | Delete

	Role	Status
○	rds-comprehend-docs-lab-ams-ml-role-1663377712464	✓ Active
○	rds-sagemaker-docs-lab-ams-ml-role-1663707780911	✓ Active

5. Choose **Connect service**.
6. In the **Connect cluster to Amazon S3** dialog, enter the ARN of the Amazon S3 bucket, as shown in the following image.



Connect cluster to Amazon S3

An Amazon Resource Name (ARN) of an Amazon S3 bucket is required to access the S3 bucket.

arn:aws:s3:::docs-lab-bucket-for-sagemaker-models

Format: *arn:aws:s3:::example-bucket*

7. Choose **Connect service** to complete this process.

For more information about using Amazon S3 buckets with SageMaker, see Specify an Amazon S3 Bucket to Upload Training Datasets and Store Output Data in the *Amazon SageMaker Developer Guide*. To learn more about working with SageMaker, see Get Started with Amazon SageMaker Notebook Instancesin the in the *Amazon SageMaker Developer Guide*.

Granting database users access to Aurora machine learning

Database users must be granted permission to invoke Aurora machine learning functions. How you grant permission depends on the version of MySQL that you use for your Aurora MySQL DB cluster, as outlined in the following. How you do so depends on the version of MySQL that your Aurora MySQL DB cluster uses.

- For Aurora MySQL version 3 (MySQL 8.0 compatible), database users must be granted the appropriate *database role*. For more informations, see Using Roles in the *MySQL 8.0 Reference Manual*.
- For Aurora MySQL version 2 (MySQL 5.7 compatible), database users are granted *privileges*. For more information, see Access Control and Account Management in the *MySQL 5.7 Reference Manual*.

The following table shows the roles and privileges that database users need to work with machine learning functions.

Aurora MySQL version 3 (role)	Aurora MySQL version 2 (privilege)
AWS_BEDROCK_ACCESS	–
AWS_COMPREHEND_ACCESS	INVOKE COMPREHEND
AWS_SAGEMAKER_ACCESS	INVOKE SAGEMAKER

Granting access to Amazon Bedrock functions

To give database users access to Amazon Bedrock functions, use the following SQL statement:

```
GRANT AWS_BEDROCK_ACCESS TO user@domain-or-ip-address;
```

Database users also need to be granted EXECUTE permissions for the functions that you create for working with Amazon Bedrock:

```
GRANT EXECUTE ON FUNCTION database_name.function_name TO user@domain-or-ip-address;
```

Finally, database users must have their roles set to AWS_BEDROCK_ACCESS:

```
SET ROLE AWS_BEDROCK_ACCESS;
```

The Amazon Bedrock functions are now available for use.

Granting access to Amazon Comprehend functions

To give database users access to Amazon Comprehend functions, use the appropriate statement for your Aurora MySQL version.

- Aurora MySQL version 3 (MySQL 8.0 compatible)

  ```
  GRANT AWS_COMPREHEND_ACCESS TO user@domain-or-ip-address;
  ```

- Aurora MySQL version 2 (MySQL 5.7 compatible)

  ```
  GRANT INVOKE COMPREHEND ON *.* TO user@domain-or-ip-address;
  ```

The Amazon Comprehend functions are now available for use. For usage examples, see Using Amazon Comprehend with your Aurora MySQL DB cluster.

Granting access to SageMaker functions

To give database users access to SageMaker functions, use the appropriate statement for your Aurora MySQL version.

- Aurora MySQL version 3 (MySQL 8.0 compatible)

  ```
  GRANT AWS_SAGEMAKER_ACCESS TO user@domain-or-ip-address;
  ```

- Aurora MySQL version 2 (MySQL 5.7 compatible)

  ```
  GRANT INVOKE SAGEMAKER ON *.* TO user@domain-or-ip-address;
  ```

Database users also need to be granted EXECUTE permissions for the functions that you create for working with SageMaker. Suppose that you created two functions, db1.anomoly_score and db2.company_forecasts, to invoke the services of your SageMaker endpoint. You grant execute privileges as shown in the following example.

```
GRANT EXECUTE ON FUNCTION db1.anomaly_score TO user1@domain-or-ip-address1;
GRANT EXECUTE ON FUNCTION db2.company_forecasts TO user2@domain-or-ip-address2;
```

The SageMaker functions are now available for use. For usage examples, see Using SageMaker with your Aurora MySQL DB cluster.

Using Amazon Bedrock with your Aurora MySQL DB cluster

To use Amazon Bedrock, you create a user-defined function (UDF) in your Aurora MySQL database that invokes a model. For more information, see Supported models in Amazon Bedrock in the *Amazon Bedrock User Guide*.

A UDF uses the following syntax:

```
CREATE FUNCTION function_name (argument type)
        [DEFINER = user]
    RETURNS mysql_data_type
    [SQL SECURITY {DEFINER | INVOKER}]
    ALIAS AWS_BEDROCK_INVOKE_MODEL
```

```
    MODEL ID 'model_id'
    [CONTENT_TYPE 'content_type']
    [ACCEPT 'content_type']
    [TIMEOUT_MS timeout_in_milliseconds];
```

- Amazon Bedrock functions don't support `RETURNS JSON`. You can use `CONVERT` or `CAST` to convert from `TEXT` to `JSON` if needed.
- If you don't specify `CONTENT_TYPE` or `ACCEPT`, the default is `application/json`.
- If you don't specify `TIMEOUT_MS`, the value for `aurora_ml_inference_timeout` is used.

For example, the following UDF invokes the Amazon Titan Text Express model:

```
CREATE FUNCTION invoke_titan (request_body TEXT)
    RETURNS TEXT
    ALIAS AWS_BEDROCK_INVOKE_MODEL
    MODEL ID 'amazon.titan-text-express-v1'
    CONTENT_TYPE 'application/json'
    ACCEPT 'application/json';
```

To allow a DB user to use this function, use the following SQL command:

```
GRANT EXECUTE ON FUNCTION database_name.invoke_titan TO user@domain-or-ip-address;
```

Then the user can call `invoke_titan` like any other function, as shown in the following example. Make sure to format the request body according to the Amazon Titan text models.

```
CREATE TABLE prompts (request varchar(1024));
INSERT INTO prompts VALUES (
'{
    "inputText": "Generate synthetic data for daily product sales in various categories
 - include row number, product name, category, date of sale and price. Produce output
 in JSON format. Count records and ensure there are no more than 5.",
    "textGenerationConfig": {
        "maxTokenCount": 1024,
        "stopSequences": [],
        "temperature":0,
        "topP":1
    }
}');
```

~~~
SELECT invoke_titan(request) FROM prompts;

{"inputTextTokenCount":44,"results":[{"tokenCount":296,"outputText":"
```tabular-data-json
{
    "rows": [
        {
            "Row Number": "1",
            "Product Name": "T-Shirt",
            "Category": "Clothing",
            "Date of Sale": "2024-01-01",
            "Price": "$20"
        },
        {
            "Row Number": "2",
            "Product Name": "Jeans",
            "Category": "Clothing",
            "Date of Sale": "2024-01-02",
            "Price": "$30"
        },
        {
            "Row Number": "3",
            "Product Name": "Hat",
            "Category": "Accessories",
            "Date of Sale": "2024-01-03",
            "Price": "$15"
        },
        {
            "Row Number": "4",
            "Product Name": "Watch",
            "Category": "Accessories",
            "Date of Sale": "2024-01-04",
            "Price": "$40"
        },
        {
            "Row Number": "5",
            "Product Name": "Phone Case",
            "Category": "Accessories",
            "Date of Sale": "2024-01-05",
            "Price": "$25"
        }
    ]
}
~~~

~~~
```","completionReason":"FINISH"}]}
~~~

For other models that you use, make sure to format the request body appropriately for them. For more information, see Inference parameters for foundation models in the *Amazon Bedrock User Guide*.

Using Amazon Comprehend with your Aurora MySQL DB cluster

For Aurora MySQL, Aurora machine learning provides the following two built-in functions for working with Amazon Comprehend and your text data. You provide the text to analyze (`input_data`) and specify the language (`language_code`).

aws_comprehend_detect_sentiment

This function identifies the text as having a positive, negative, neutral, or mixed emotional posture. This function's reference documentation is as follows.

```
aws_comprehend_detect_sentiment(
  input_text,
  language_code
  [,max_batch_size]
)
```

To learn more, see Sentiment in the *Amazon Comprehend Developer Guide*.

aws_comprehend_detect_sentiment_confidence

This function measures the confidence level of the sentiment detected for a given text. It returns a value (type, `double`) that indicates the confidence of the sentiment assigned by the aws_comprehend_detect_sentiment function to the text. Confidence is a statistical metric between 0 and 1. The higher the confidence level, the more weight you can give the result. A summary of the function's documentation is as follows.

```
aws_comprehend_detect_sentiment_confidence(
  input_text,
  language_code
  [,max_batch_size]
)
```

In both functions (aws_comprehend_detect_sentiment_confidence, aws_comprehend_detect_sentiment) the `max_batch_size` uses a default value of 25 if none is specified. Batch size should always be greater than 0. You can use `max_batch_size` to tune the performance of the Amazon Comprehend function calls. A large batch size trades off faster performance for greater memory usage on the Aurora MySQL DB cluster. For more information, see Performance considerations for using Aurora machine learning with Aurora MySQL.

For more information about parameters and return types for the sentiment detection functions in Amazon Comprehend, see DetectSentiment

Example Example: A simple query using Amazon Comprehend functions

Here's an example of a simple query that invokes these two functions to see how happy your customers are with your support team. Suppose you have a database table (`support`) that stores customer feedback after each request for help. This example query applies both built-in functions to the text in the `feedback` column of the table and outputs the results. The confidence values returned by the function are doubles between 0.0 and 1.0. For more readable output, this query rounds the results to 6 decimal points. For easier comparisons, this query also sorts the results in descending order, from the result having the highest degree of confidence, first.

```
SELECT feedback AS 'Customer feedback',
    aws_comprehend_detect_sentiment(feedback, 'en') AS Sentiment,
    ROUND(aws_comprehend_detect_sentiment_confidence(feedback, 'en'), 6)
    AS Confidence FROM support
    ORDER BY Confidence DESC;
+---------------------------------------------------------+-----------+------------+
| Customer feedback                                       | Sentiment | Confidence |
+---------------------------------------------------------+-----------+------------+
| Thank you for the excellent customer support!           | POSITIVE  |   0.999771 |
| The latest version of this product stinks!              | NEGATIVE  |   0.999184 |
| Your support team is just awesome! I am blown away.     | POSITIVE  |   0.997774 |
| Your product is too complex, but your support is great. | MIXED     |   0.957958 |
| Your support tech helped me in fifteen minutes.         | POSITIVE  |   0.949491 |
| My problem was never resolved!                          | NEGATIVE  |   0.920644 |
| When will the new version of this product be released?  | NEUTRAL   |   0.902706 |
| I cannot stand that chatbot.                            | NEGATIVE  |   0.895219 |
| Your support tech talked down to me.                    | NEGATIVE  |   0.868598 |
| It took me way too long to get a real person.           | NEGATIVE  |   0.481805 |
+---------------------------------------------------------+-----------+------------+
 10 rows in set (0.1898 sec)
```

Example Example: Determining the average sentiment for text above a specific confidence level

A typical Amazon Comprehend query looks for rows where the sentiment is a certain value, with a confidence level greater than a certain number. For example, the following query shows how you can determine the average sentiment of documents in your database. The query considers only documents where the confidence of the assessment is at least 80%.

```
SELECT AVG(CASE aws_comprehend_detect_sentiment(productTable.document, 'en')
  WHEN 'POSITIVE' THEN 1.0
  WHEN 'NEGATIVE' THEN -1.0
  ELSE 0.0 END) AS avg_sentiment, COUNT(*) AS total
FROM productTable
WHERE productTable.productCode = 1302 AND
    aws_comprehend_detect_sentiment_confidence(productTable.document, 'en') >= 0.80;
```

Using SageMaker with your Aurora MySQL DB cluster

To use SageMaker functionality from your Aurora MySQL DB cluster, you need to create stored functions that embed your calls to the SageMaker endpoint and its inference features. You do so by using MySQL's `CREATE FUNCTION` in generally the same way that you do for other processing tasks on your Aurora MySQL DB cluster.

To use models deployed in SageMaker for inference, you create user-defined functions using MySQL data definition language (DDL) statements for stored functions. Each stored function represents the SageMaker endpoint hosting the model. When you define such a function, you specify the input parameters to the model, the specific SageMaker endpoint to invoke, and the return type. The function returns the inference computed by the SageMaker endpoint after applying the model to the input parameters.

All Aurora machine learning stored functions return numeric types or `VARCHAR`. You can use any numeric type except `BIT`. Other types, such as `JSON`, `BLOB`, `TEXT`, and `DATE` aren't allowed.

The following example shows the `CREATE FUNCTION` syntax for working with SageMaker.

```
CREATE FUNCTION function_name (
     arg1 type1,
     arg2 type2, ...)
     [DEFINER = user]
  RETURNS mysql_type
  [SQL SECURITY { DEFINER | INVOKER } ]
  ALIAS AWS_SAGEMAKER_INVOKE_ENDPOINT
  ENDPOINT NAME 'endpoint_name'
  [MAX_BATCH_SIZE max_batch_size];
```

This is an extension of the regular CREATE FUNCTION DDL statement. In the CREATE FUNCTION statement that defines the SageMaker function, you don't specify a function body. Instead, you specify the keyword ALIAS where the function body usually goes. Currently, Aurora machine learning only supports aws_sagemaker_invoke_endpoint for this extended syntax. You must specify the endpoint_name parameter. An SageMaker endpoint can have different characteristics for each model.

> **Note**
>
> For more information about CREATE FUNCTION, see CREATE PROCEDURE and CREATE FUNCTION Statements in the MySQL 8.0 Reference Manual.

The max_batch_size parameter is optional. By default, maximum batch size is 10,000. You can use this parameter in your function to restrict the maximum number of inputs processed in a batched request to SageMaker. The max_batch_size parameter can help to avoid an error caused by inputs that are too large, or to make SageMaker return a response more quickly. This parameter affects the size of an internal buffer used for SageMaker request processing. Specifying too large a value for max_batch_size might cause substantial memory overhead on your DB instance.

We recommend that you leave the MANIFEST setting at its default value of OFF. Although you can use the MANIFEST ON option, some SageMaker features can't directly use the CSV exported with this option. The manifest format is not compatible with the expected manifest format from SageMaker.

You create a separate stored function for each of your SageMaker models. This mapping of functions to models is required because an endpoint is associated with a specific model, and each model accepts different parameters. Using SQL types for the model inputs and the model output type helps to avoid type conversion errors passing data back and forth between the AWS services. You can control who can apply the model. You can also control the runtime characteristics by specifying a parameter representing the maximum batch size.

Currently, all Aurora machine learning functions have the NOT DETERMINISTIC property. If you don't specify that property explicitly, Aurora sets NOT DETERMINISTIC automatically. This requirement is because the SageMaker model can be changed without any notification to the database. If that happens, calls to an Aurora machine learning function might return different results for the same input within a single transaction.

You can't use the characteristics CONTAINS SQL, NO SQL, READS SQL DATA, or MODIFIES SQL DATA in your CREATE FUNCTION statement.

Following is an example usage of invoking an SageMaker endpoint to detect anomalies. There is an SageMaker endpoint random-cut-forest-model. The corresponding model is already trained by the random-cut-forest algorithm. For each input, the model returns an anomaly score. This example shows the data points whose score is greater than 3 standard deviations (approximately the 99.9th percentile) from the mean score.

```
CREATE FUNCTION anomaly_score(value real) returns real
  alias aws_sagemaker_invoke_endpoint endpoint name 'random-cut-forest-model-demo';

set @score_cutoff = (select avg(anomaly_score(value)) + 3 * std(anomaly_score(value))
 from nyc_taxi);

select *, anomaly_detection(value) score from nyc_taxi
  where anomaly_detection(value) > @score_cutoff;
```

Character set requirement for SageMaker functions that return strings

We recommend specifying a character set of utf8mb4 as the return type for your SageMaker functions that return string values. If that isn't practical, use a large enough string length for the return type to hold a value represented in the utf8mb4 character set. The following example shows how to declare the utf8mb4 character set for your function.

```
CREATE FUNCTION my_ml_func(...) RETURNS VARCHAR(5) CHARSET utf8mb4 ALIAS ...
```

Currently, each SageMaker function that returns a string uses the character set utf8mb4 for the return value. The return value uses this character set even if your SageMaker function declares a different character set for its return type implicitly or explicitly. If your SageMaker function declares a different character set for the return value, the returned data might be silently truncated if you store it in a table column that isn't long enough. For example, a query with a DISTINCT clause creates a temporary table. Thus, the SageMaker function result might be truncated due to the way strings are handled internally during a query.

Exporting data to Amazon S3 for SageMaker model training (Advanced)

We recommend that you get started with Aurora machine learning and SageMaker by using some of the provided algorithms, and that the data scientists on your team provide you with the SageMaker endpoints that you can use with your SQL code. In the following, you can find minimal information about using your own Amazon S3 bucket with your your own SageMaker models and your Aurora MySQL DB cluster.

Machine learning consists of two major steps: training, and inference. To train SageMaker models, you export data to an Amazon S3 bucket. The Amazon S3 bucket is used by a Jupyter SageMaker notebook instance to train your model before it is deployed. You can use the `SELECT INTO OUTFILE S3` statement to query data from an Aurora MySQL DB cluster and save it directly into text files stored in an Amazon S3 bucket. Then the notebook instance consumes the data from the Amazon S3 bucket for training.

Aurora machine learning extends the existing `SELECT INTO OUTFILE` syntax in Aurora MySQL to export data to CSV format. The generated CSV file can be directly consumed by models that need this format for training purposes.

```
SELECT * INTO OUTFILE S3 's3_uri' [FORMAT {CSV|TEXT} [HEADER]] FROM table_name;
```

The extension supports the standard CSV format.

- Format `TEXT` is the same as the existing MySQL export format. This is the default format.
- Format `CSV` is a newly introduced format that follows the specification in RFC-4180.
- If you specify the optional keyword `HEADER`, the output file contains one header line. The labels in the header line correspond to the column names from the `SELECT` statement.
- You can still use the keywords `CSV` and `HEADER` as identifiers.

The extended syntax and grammar of `SELECT INTO` is now as follows:

```
INTO OUTFILE S3 's3_uri'
[CHARACTER SET charset_name]
[FORMAT {CSV|TEXT} [HEADER]]
[{FIELDS | COLUMNS}
  [TERMINATED BY 'string']
  [[OPTIONALLY] ENCLOSED BY 'char']
  [ESCAPED BY 'char']
]
[LINES
  [STARTING BY 'string']
  [TERMINATED BY 'string']
]
```

Performance considerations for using Aurora machine learning with Aurora MySQL

The Amazon Bedrock, Amazon Comprehend, and SageMaker services do most of the work when invoked by an Aurora machine learning function. That means that you can scale those resources as needed, independently. For your Aurora MySQL DB cluster, you can make your function calls as efficient as possible. Following, you can find some performance considerations to note when working with Aurora machine learning.

Model and prompt

Performance when using Amazon Bedrock is highly dependent on the model and prompt that you use. Choose a model and prompt that are optimal for your use case.

Query cache

The Aurora MySQL query cache doesn't work for Aurora machine learning functions. Aurora MySQL doesn't store query results in the query cache for any SQL statements that call Aurora machine learning functions.

Batch optimization for Aurora machine learning function calls

The main Aurora machine learning performance aspect that you can influence from your Aurora cluster is the batch mode setting for calls to the Aurora machine learning stored functions. Machine learning functions typically require substantial overhead, making it impractical to call an external service separately for each row. Aurora machine learning can minimize this overhead by combining the calls to the external Aurora machine learning service for many rows into a single batch. Aurora machine learning receives the responses for all the input rows, and delivers the responses, one row at a time, to the query as it runs. This optimization improves the throughput and latency of your Aurora queries without changing the results.

When you create an Aurora stored function that's connected to an SageMaker endpoint, you define the batch size parameter. This parameter influences how many rows are transferred for every underlying call to SageMaker. For queries that process large numbers of rows, the overhead to make a separate SageMaker call for each row can be substantial. The larger the data set processed by the stored procedure, the larger you can make the batch size.

If the batch mode optimization can be applied to an SageMaker function, you can tell by checking the query plan produced by the `EXPLAIN PLAN` statement. In this case, the `extra` column in the execution plan includes `Batched machine learning`. The following example shows a call to an SageMaker function that uses batch mode.

```
mysql> CREATE FUNCTION anomaly_score(val real) returns real alias
 aws_sagemaker_invoke_endpoint endpoint name 'my-rcf-model-20191126';
Query OK, 0 rows affected (0.01 sec)

mysql> explain select timestamp, value, anomaly_score(value) from nyc_taxi;
+----+-------------+----------+------------+------+---------------+------+---------
+------+------+----------+--------------------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len |
 ref  | rows | filtered | Extra                    |
+----+-------------+----------+------------+------+---------------+------+---------
+------+------+----------+--------------------------+
|  1 | SIMPLE      | nyc_taxi | NULL       | ALL  | NULL          | NULL | NULL    |
 NULL |   48 |   100.00 | Batched machine learning |
+----+-------------+----------+------------+------+---------------+------+---------
+------+------+----------+--------------------------+
1 row in set, 1 warning (0.01 sec)
```

When you call one of the built-in Amazon Comprehend functions, you can control the batch size by specifying the optional `max_batch_size` parameter. his parameter restricts the maximum number of `input_text` values processed in each batch. By sending multiple items at once, it reduces the number of round trips between Aurora and Amazon Comprehend. Limiting the batch size is useful in situations such as a query with a `LIMIT` clause. By using a small value for `max_batch_size`, you can avoid invoking Amazon Comprehend more times than you have input texts.

The batch optimization for evaluating Aurora machine learning functions applies in the following cases:

- Function calls within the select list or the `WHERE` clause of `SELECT` statements
- Function calls in the `VALUES` list of `INSERT` and `REPLACE` statements
- SageMaker functions in `SET` values in `UPDATE` statements:

```
INSERT INTO MY_TABLE (col1, col2, col3) VALUES
  (ML_FUNC(1), ML_FUNC(2), ML_FUNC(3)),
  (ML_FUNC(4), ML_FUNC(5), ML_FUNC(6));
UPDATE MY_TABLE SET col1 = ML_FUNC(col2), SET col3 = ML_FUNC(col4) WHERE ...;
```

Monitoring Aurora machine learning

You can monitor Aurora machine learning batch operations by querying several global variables, as shown in the following example.

```
show status like 'Aurora_ml%';
```

You can reset the status variables by using a `FLUSH STATUS` statement. Thus, all of the figures represent totals, averages, and so on, since the last time the variable was reset.

`Aurora_ml_logical_request_cnt`

The number of logical requests that the DB instance has evaluated to be sent to the Aurora machine learning services since the last status reset. Depending on whether batching has been used, this value can be higher than `Aurora_ml_actual_request_cnt`.

`Aurora_ml_logical_response_cnt`

The aggregate response count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance.

`Aurora_ml_actual_request_cnt`

The aggregate request count that Aurora MySQL makes to the Aurora machine learning services across all queries run by users of the DB instance.

`Aurora_ml_actual_response_cnt`

The aggregate response count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance.

`Aurora_ml_cache_hit_cnt`

The aggregate internal cache hit count that Aurora MySQL receives from the Aurora machine learning services across all queries run by users of the DB instance.

`Aurora_ml_retry_request_cnt`

The number of retried requests that the DB instance has sent to the Aurora machine learning services since the last status reset.

`Aurora_ml_single_request_cnt`

The aggregate count of Aurora machine learning functions that are evaluated by non-batch mode across all queries run by users of the DB instance.

For information about monitoring the performance of the SageMaker operations called from Aurora machine learning functions, see Monitor Amazon SageMaker.

Using Amazon Aurora machine learning with Aurora PostgreSQL

By using Amazon Aurora machine learning with your Aurora PostgreSQL DB cluster, you can use Amazon Comprehend or Amazon SageMaker or Amazon Bedrock, depending on your needs. These services each support specific machine learning use cases.

Aurora machine learning is supported in certain AWS Regions and for specific versions of Aurora PostgreSQL only. Before trying to set up Aurora machine learning, check availability for your Aurora PostgreSQL version and your Region. For details, see Aurora machine learning with Aurora PostgreSQL.

Topics

Requirements for using Aurora machine learning with Aurora PostgreSQL

AWS machine learning services are managed services that are set up and run in their own production environments. Aurora machine learning supports integration with Amazon Comprehend, SageMaker, and Amazon Bedrock. Before trying to set up your Aurora PostgreSQL DB cluster to use Aurora machine learning, be sure you understand the following requirements and prerequisites.

- The Amazon Comprehend, SageMaker, and Amazon Bedrock services must be running in the same AWS Region as your Aurora PostgreSQL DB cluster. You can't use Amazon Comprehend or SageMaker or Amazon Bedrock services from an Aurora PostgreSQL DB cluster in a different Region.
- If your Aurora PostgreSQL DB cluster is in a different virtual public cloud (VPC) based on the Amazon VPC service than your Amazon Comprehend and SageMaker services, the VPC's Security group needs to allow outbound connections to the target Aurora machine learning service. For more information, see Enabling network communication from Amazon Aurora MySQL to other AWS services.
- For SageMaker, the machine learning components that you want to use for inferences must be set up and ready to use. During the configuration process for your Aurora PostgreSQL DB cluster, you need to have the Amazon Resource Name (ARN) of the SageMaker endpoint available. The data scientists on your team are likely best able to handle working with SageMaker to prepare the models and handle the other such tasks. To get started with Amazon SageMaker, see Get Started with Amazon SageMaker. For more information about inferences and endpoints, see Real-time inference.
- For Amazon Bedrock, you need to have the model ID of the Bedrock models that you want to use for inferences available during the configuration process of your Aurora PostgreSQL DB cluster. The data scientists on your team are likely best able to work with Bedrock to decide which models to use, fine tune them if needed and handle other such tasks. To get started with Amazon Bedrock, see How to setup Bedrock.
- Amazon Bedrock users need to request access to models before they are available for use. If you want to add additional models for text, chat, and image generation, you need to request access to models in Amazon Bedrock. For more information, see Model access.

Supported features and limitations of Aurora machine learning with Aurora PostgreSQL

Aurora machine learning supports any SageMaker endpoint that can read and write the comma-separated value (CSV) format through a `ContentType` value of `text/csv`. The built-in SageMaker algorithms that currently accept this format are the following.

- Linear Learner
- Random Cut Forest
- XGBoost

  To learn more about these algorithms, see Choose an Algorithm in the *Amazon SageMaker Developer Guide*.

When using Amazon Bedrock with Aurora machine learning, the following limitations apply:

- The user-defined functions (UDFs) provide a native way to interact with Amazon Bedrock. The UDFs don't have specific request or response requirements, so they can use any model.
- You can use UDFs to build any work flow desired. For example, you can combine base primitives such as `pg_cron` to run a query, fetch data, generate inferences, and write to tables to serve queries directly.
- UDFs don't support batched or parallel calls.
- The Aurora Machine Learning extension doesn't support vector interfaces. As part of the extension, a function is available to output the embeddings of model’s response in the `float8[]` format to store those embeddings in Aurora. For more information on the usage of `float8[]`, see Using Amazon Bedrock with your Aurora PostgreSQL DB cluster.

Setting up your Aurora PostgreSQL DB cluster to use Aurora machine learning

For Aurora machine learning to work with your Aurora PostgreSQL DB cluster, you need to create an AWS Identity and Access Management (IAM) role for each of the services that you want to use. The IAM role allows your Aurora PostgreSQL DB cluster to use the Aurora machine learning service on the cluster's behalf. You also need to install the Aurora machine learning extension. In the following topics, you can find setup procedures for each of these Aurora machine learning services.

Topics

Setting up Aurora PostgreSQL to use Amazon Bedrock

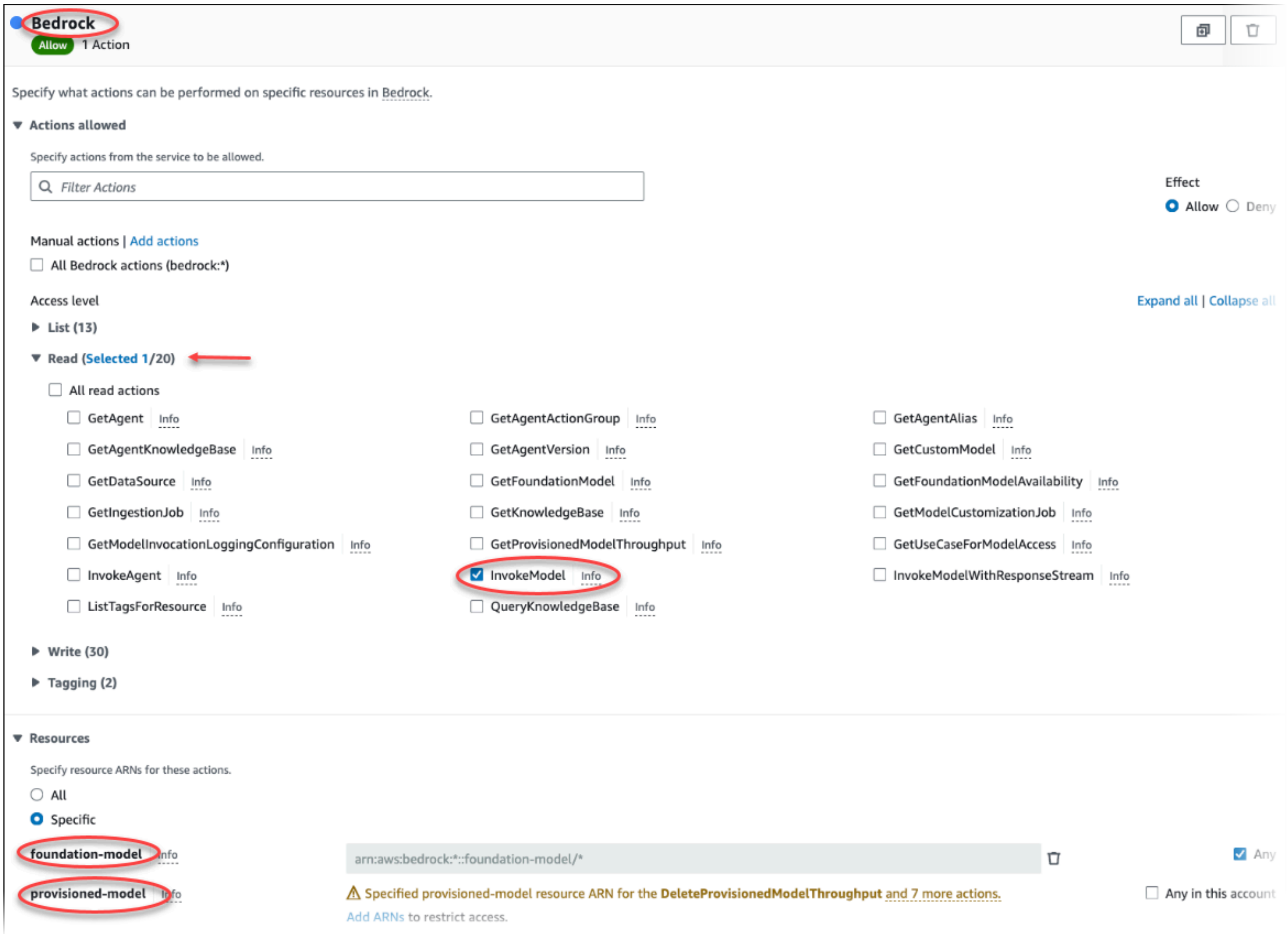
In the procedure following, you first create the IAM role and policy that gives your Aurora PostgreSQL permission to use Amazon Bedrock on the cluster's behalf. You then attach the policy to an IAM role that your Aurora PostgreSQL DB cluster uses to work with Amazon Bedrock. For simplicity's sake, this procedure uses the AWS Management Console to complete all tasks.

To set up your Aurora PostgreSQL DB cluster to use Amazon Bedrock

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Open the IAM console at https://console.aws.amazon.com/iam/.
3. Choose **Policies** (under Access management) on the AWS Identity and Access Management (IAM) Console menu.

   a. Choose **Create policy**. In the Visual editor page, choose **Service** and then enter **Bedrock** in the Select a service field. Expand the Read access level. Choose **InvokeModel** from the Amazon Bedrock read settings.

   b. Choose the Foundation/Provisioned model you want to grant read access via the policy.



Bedrock

Allow 1 Action

Specify what actions can be performed on specific resources in Bedrock.

▼ Actions allowed

Specify actions from the service to be allowed.

Filter Actions

Effect

● Allow ○ Deny

Manual actions | Add actions

☐ All Bedrock actions (bedrock:*)

Access level

Expand all | Collapse all

▶ List (13)

▼ Read (Selected 1/20)

☐ All read actions

☐ GetAgent Info ☐ GetAgentActionGroup Info ☐ GetAgentAlias Info

☐ GetAgentKnowledgeBase Info ☐ GetAgentVersion Info ☐ GetCustomModel Info

☐ GetDataSource Info ☐ GetFoundationModel Info ☐ GetFoundationModelAvailability Info

☐ GetIngestionJob Info ☐ GetKnowledgeBase Info ☐ GetModelCustomizationJob Info

☐ GetModelInvocationLoggingConfiguration Info ☐ GetProvisionedModelThroughput Info ☐ GetUseCaseForModelAccess Info

☐ InvokeAgent Info ☑ InvokeModel Info ☐ InvokeModelWithResponseStream Info

☐ ListTagsForResource Info ☐ QueryKnowledgeBase Info

▶ Write (30)

▶ Tagging (2)

▼ Resources

Specify resource ARNs for these actions.

○ All

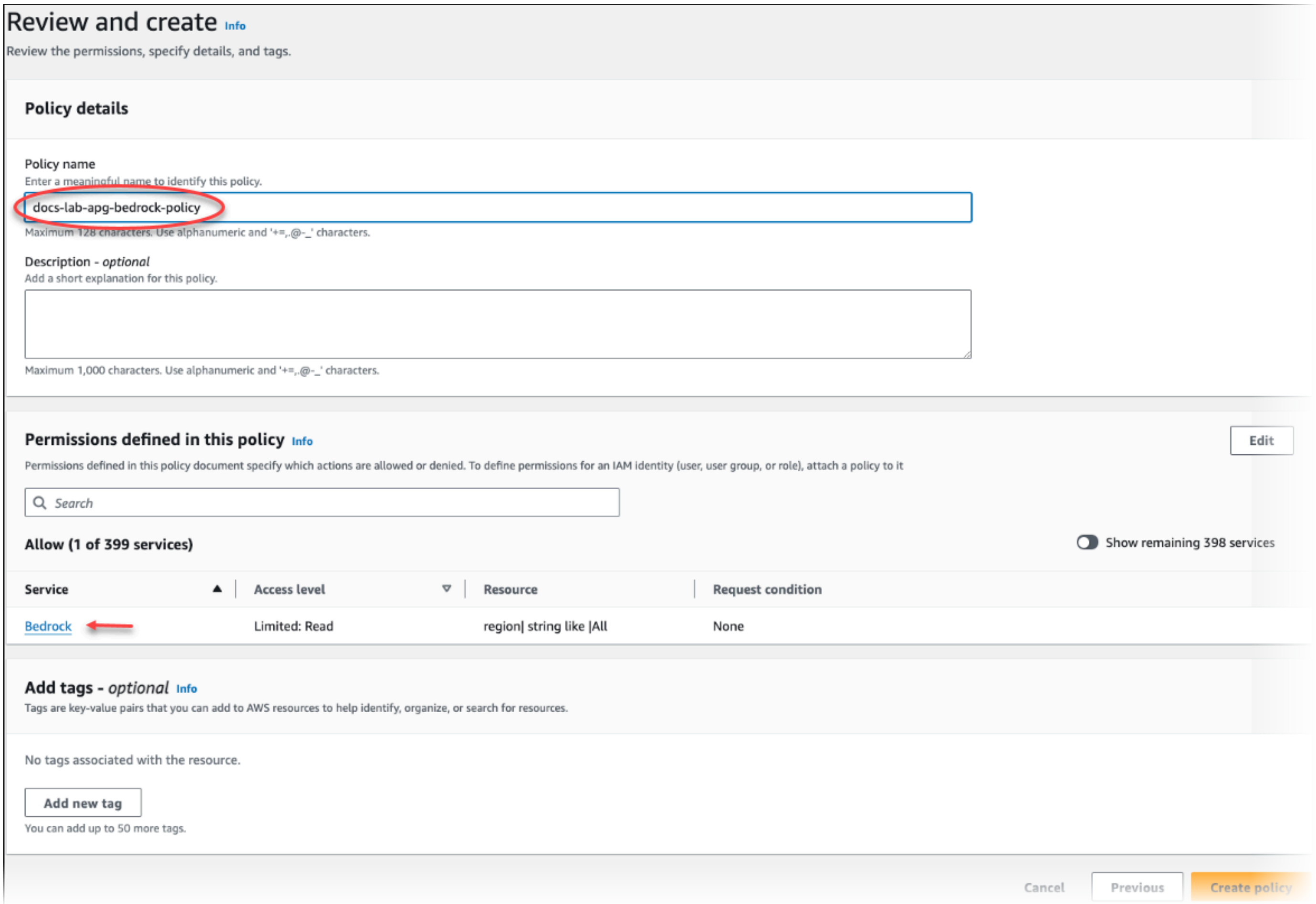
● Specific

foundation-model Info arn:aws:bedrock:*::foundation-model/* ☑ Any

provisioned-model Info ⚠ Specified provisioned-model resource ARN for the DeleteProvisionedModelThroughput and 7 more actions. ☐ Any in this account

Add ARNs to restrict access.

4. Choose **Next: Tags** and define any tags (this is optional). Choose **Next: Review**. Enter a Name for the policy and description, as shown in the image.



Review and create

Review the permissions, specify details, and tags.

Policy details

Policy name

Enter a meaningful name to identify this policy.

docs-lab-apg-bedrock-policy

Maximum 128 characters. Use alphanumeric and '+=,.@-_' characters.

Description - *optional*

Add a short explanation for this policy.

Maximum 1,000 characters. Use alphanumeric and '+=,.@-_' characters.

Permissions defined in this policy

Permissions defined in this policy document specify which actions are allowed or denied. To define permissions for an IAM identity (user, user group, or role), attach a policy to it

Allow (1 of 399 services)

Show remaining 398 services

Service	Access level	Resource	Request condition
Bedrock	Limited: Read	region| string like |All	None

Add tags - *optional*

Tags are key-value pairs that you can add to AWS resources to help identify, organize, or search for resources.

No tags associated with the resource.

You can add up to 50 more tags.

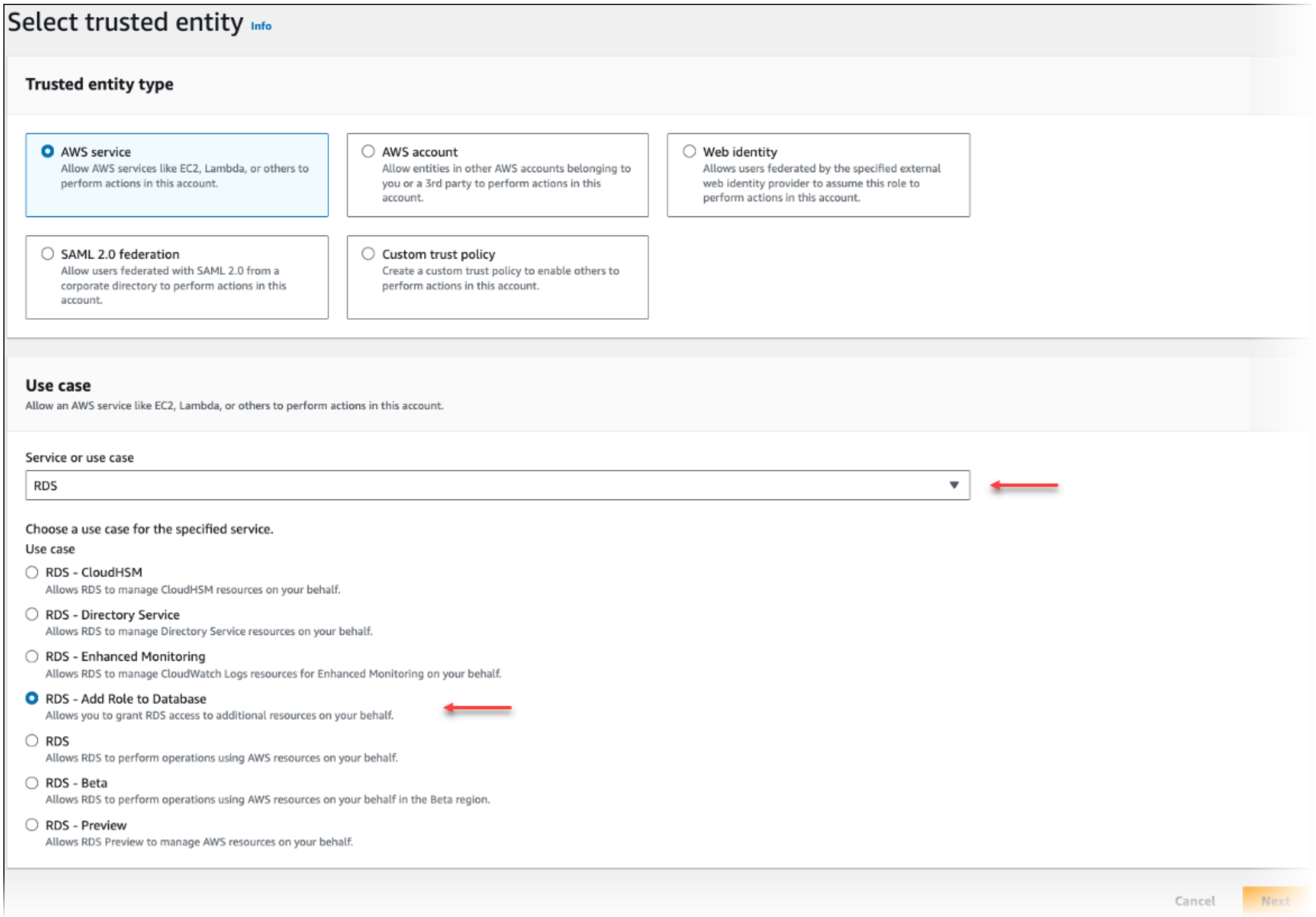
5. Choose **Create policy**. The Console displays an alert when the policy has been saved. You can find it in the list of Policies.

6. Choose **Roles** (under Access management) on the IAM Console.

7. Choose **Create role**.

8. On the Select trusted entity page, choose the **AWS service** tile, and then choose **RDS** to open the selector.

9. Choose **RDS – Add Role to Database**.



Select trusted entity Info

Trusted entity type

● AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

○ AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

○ Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

○ SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

○ Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case
RDS

Choose a use case for the specified service.
Use case

○ RDS - CloudHSM
Allows RDS to manage CloudHSM resources on your behalf.

○ RDS - Directory Service
Allows RDS to manage Directory Service resources on your behalf.

○ RDS - Enhanced Monitoring
Allows RDS to manage CloudWatch Logs resources for Enhanced Monitoring on your behalf.

● RDS - Add Role to Database
Allows you to grant RDS access to additional resources on your behalf.

○ RDS
Allows RDS to perform operations using AWS resources on your behalf.

○ RDS - Beta
Allows RDS to perform operations using AWS resources on your behalf in the Beta region.

○ RDS - Preview
Allows RDS Preview to manage AWS resources on your behalf.

10. Choose **Next**. On the Add permissions page, find the policy that you created in the previous step and choose it from among those listed. Choose **Next**.

11. **Next: Review**. Enter a name for the IAM role and a description.

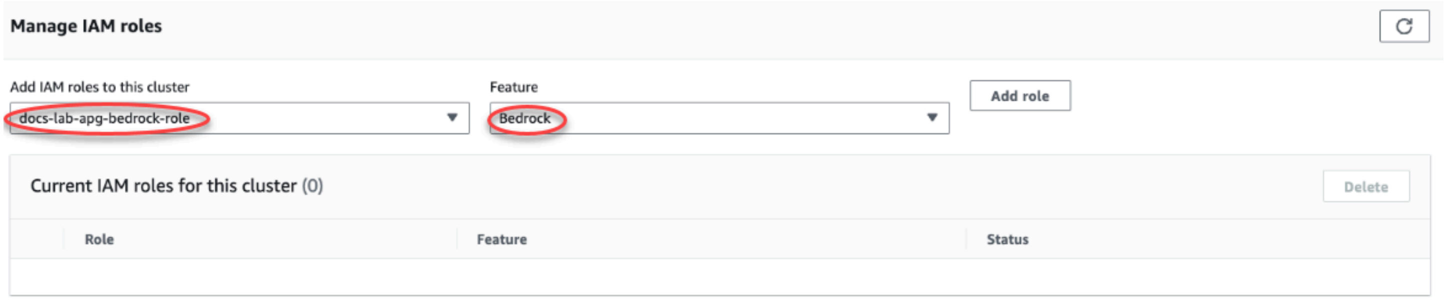
12. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

13. Navigate to the AWS Region where your Aurora PostgreSQL DB cluster is located.

14. In the navigation pane, choose **Databases**, and then choose the Aurora PostgreSQL DB cluster that you want to use with Bedrock.

15. Choose the **Connectivity & security** tab and scroll to find the **Manage IAM roles** section of the page. From the **Add IAM roles to this cluster** selector, choose the role that you created in the previous steps. In the **Feature** selector, choose Bedrock, and then choose **Add role**.

The role (with its policy) are associated with the Aurora PostgreSQL DB cluster. When the process completes, the role is listed in the Current IAM roles for this cluster listing, as shown following.



Manage IAM roles

Add IAM roles to this cluster: docs-lab-apg-bedrock-role

Feature: Bedrock

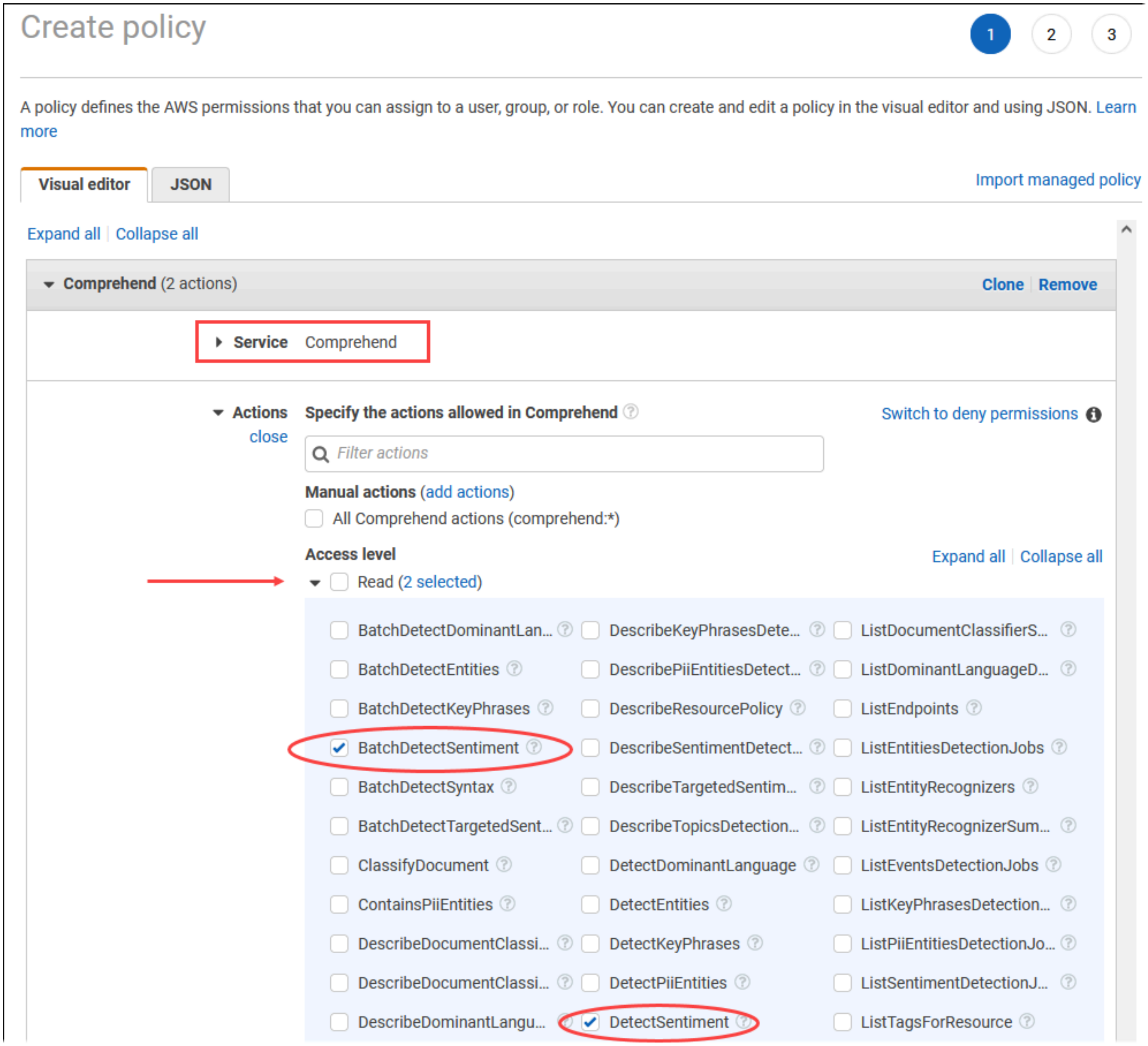
The IAM setup for Amazon Bedrock is complete. Continue setting up your Aurora PostgreSQL to work with Aurora machine learning by installing the extension as detailed in [Installing the Aurora machine learning extension](#)

Setting up Aurora PostgreSQL to use Amazon Comprehend

In the procedure following, you first create the IAM role and policy that gives your Aurora PostgreSQL permission to use Amazon Comprehend on the cluster's behalf. You then attach the policy to an IAM role that your Aurora PostgreSQL DB cluster uses to work with Amazon Comprehend For simplicity's sake, this procedure uses the AWS Management Console to complete all tasks.

To set up your Aurora PostgreSQL DB cluster to use Amazon Comprehend

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Open the IAM console at https://console.aws.amazon.com/iam/.
3. Choose **Policies** (under Access management) on the AWS Identity and Access Management (IAM) Console menu.



Create policy

1 2 3

A policy defines the AWS permissions that you can assign to a user, group, or role. You can create and edit a policy in the visual editor and using JSON. Learn more

Visual editor | JSON | Import managed policy

Expand all | Collapse all

Comprehend (2 actions) Clone | Remove

Service Comprehend

Actions close

Specify the actions allowed in Comprehend

Switch to deny permissions

Filter actions

Manual actions (add actions)

□ All Comprehend actions (comprehend:*)

Access level

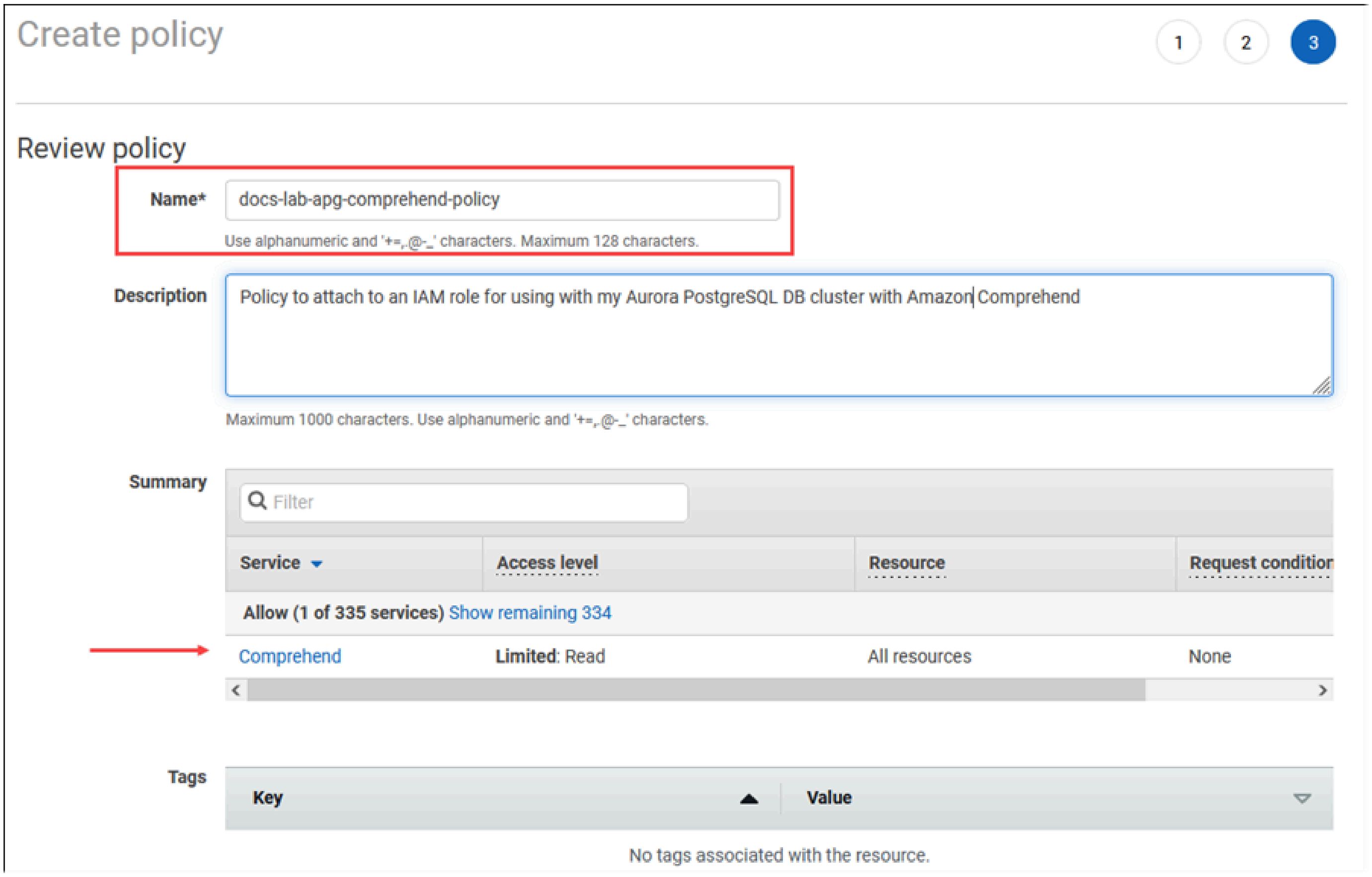
Expand all | Collapse all

□ Read (2 selected)

- □ BatchDetectDominantLan...
- □ BatchDetectEntities
- □ BatchDetectKeyPhrases
- ☑ BatchDetectSentiment
- □ BatchDetectSyntax
- □ BatchDetectTargetedSent...
- □ ClassifyDocument
- □ ContainsPiiEntities
- □ DescribeDocumentClassi...
- □ DescribeDocumentClassi...
- □ DescribeDominantLangu...
- □ DescribeKeyPhrasesDete...
- □ DescribePiiEntitiesDetect...
- □ DescribeResourcePolicy
- □ DescribeSentimentDetect...
- □ DescribeTargetedSentim...
- □ DescribeTopicsDetection...
- □ DetectDominantLanguage
- □ DetectEntities
- □ DetectKeyPhrases
- □ DetectPiiEntities
- ☑ DetectSentiment
- □ ListDocumentClassifierS...
- □ ListDominantLanguageD...
- □ ListEndpoints
- □ ListEntitiesDetectionJobs
- □ ListEntityRecognizers
- □ ListEntityRecognizerSum...
- □ ListEventsDetectionJobs
- □ ListKeyPhrasesDetection...
- □ ListPiiEntitiesDetectionJo...
- □ ListSentimentDetectionJ...
- □ ListTagsForResource

4. Choose **Create policy**. In the Visual editor page, choose **Service** and then enter **Comprehend** in the Select a service field. Expand the Read access level. Choose **BatchDetectSentiment** and **DetectSentiment** from the Amazon Comprehend read settings.

5. Choose **Next: Tags** and define any tags (this is optional). Choose **Next: Review**. Enter a Name for the policy and description, as shown in the image.



Create policy

Review policy

Name* docs-lab-apg-comprehend-policy

Use alphanumeric and '+=,.@-_' characters. Maximum 128 characters.

Description Policy to attach to an IAM role for using with my Aurora PostgreSQL DB cluster with Amazon Comprehend

Maximum 1000 characters. Use alphanumeric and '+=,.@-_' characters.

Summary

Service	Access level	Resource	Request conditior
Allow (1 of 335 services) Show remaining 334			
Comprehend	**Limited**: Read	All resources	None

Tags

Key	Value

No tags associated with the resource.

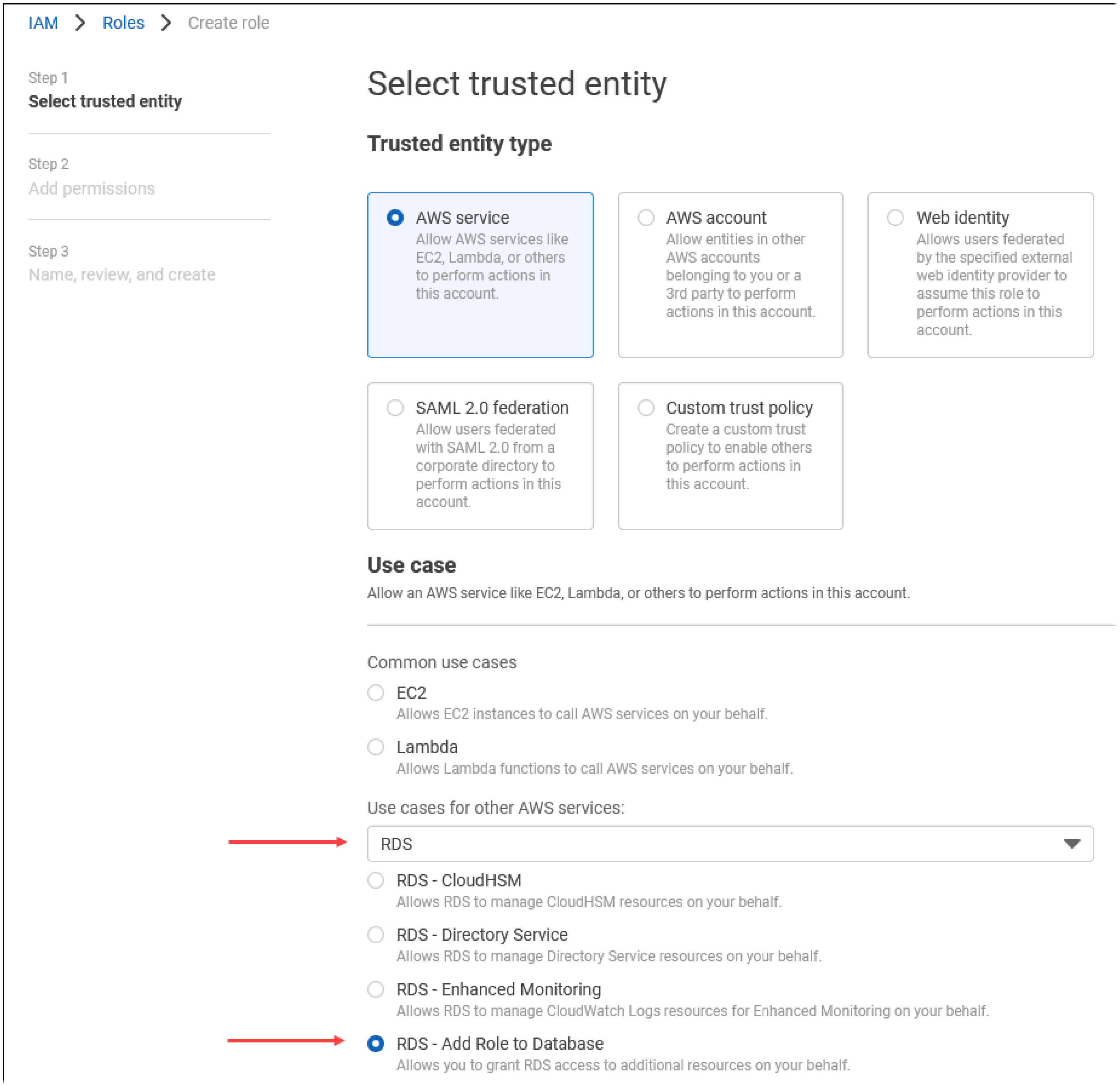
6. Choose **Create policy**. The Console displays an alert when the policy has been saved. You can find it in the list of Policies.

7. Choose **Roles** (under Access management) on the IAM Console.

8. Choose **Create role**.

9. On the Select trusted entity page, choose the **AWS service** tile, and then choose **RDS** to open the selector.

10. Choose **RDS – Add Role to Database**.



IAM > Roles > Create role

Step 1
Select trusted entity

Step 2
Add permissions

Step 3
Name, review, and create

Select trusted entity

Trusted entity type

● AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

○ AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

○ Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

○ SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

○ Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Common use cases

○ EC2
Allows EC2 instances to call AWS services on your behalf.

○ Lambda
Allows Lambda functions to call AWS services on your behalf.

Use cases for other AWS services:

RDS

○ RDS - CloudHSM
Allows RDS to manage CloudHSM resources on your behalf.

○ RDS - Directory Service
Allows RDS to manage Directory Service resources on your behalf.

○ RDS - Enhanced Monitoring
Allows RDS to manage CloudWatch Logs resources for Enhanced Monitoring on your behalf.

● RDS - Add Role to Database
Allows you to grant RDS access to additional resources on your behalf.

11. Choose **Next**. On the Add permissions page, find the policy that you created in the previous step and choose it from among those listed. Choose **Next**

12. **Next: Review**. Enter a name for the IAM role and a description.

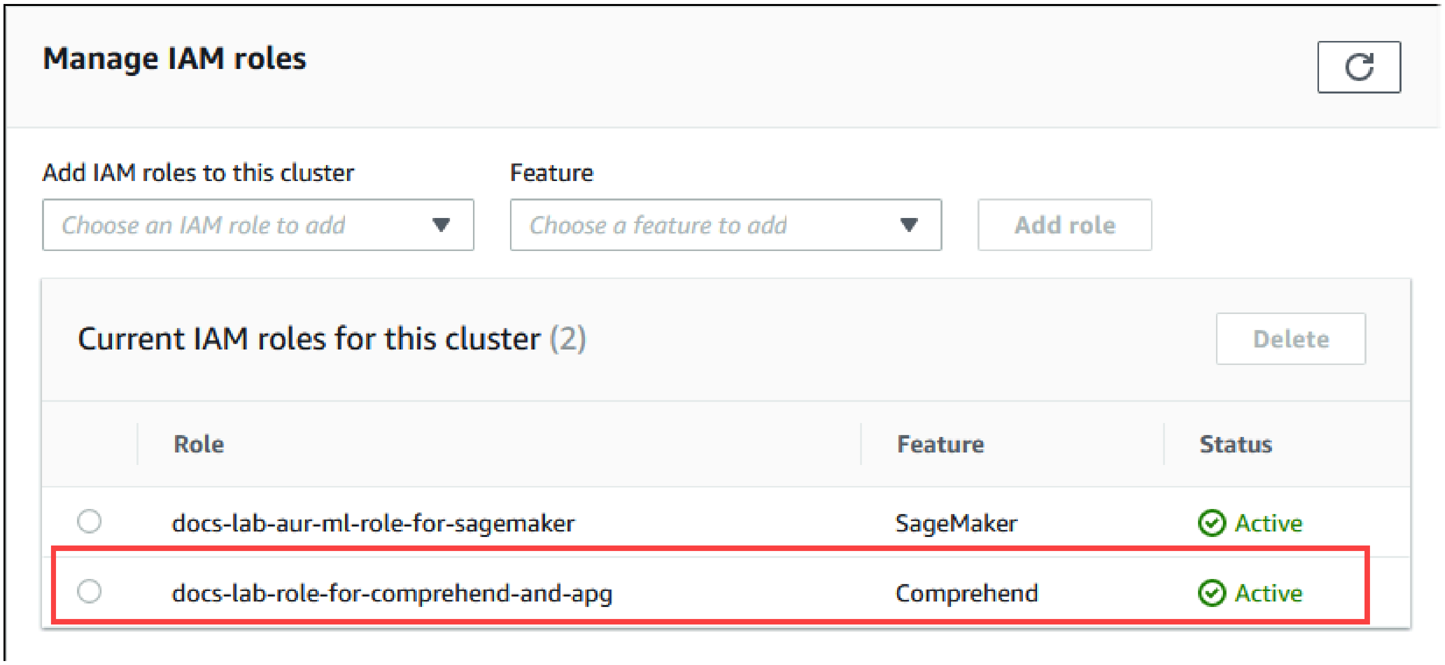
13. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

14. Navigate to the AWS Region where your Aurora PostgreSQL DB cluster is located.

15. In the navigation pane, choose **Databases**, and then choose the Aurora PostgreSQL DB cluster that you want to use with Amazon Comprehend.

16. Choose the **Connectivity & security** tab and scroll to find the **Manage IAM roles** section of the page. From the **Add IAM roles to this cluster** selector, choose the role that you created in the previous steps. In the **Feature** selector, choose Comprehend, and then choose **Add role**.

The role (with its policy) are associated with the Aurora PostgreSQL DB cluster. When the process completes, the role is listed in the Current IAM roles for this cluster listing, as shown following.


Manage IAM roles
Add IAM roles to this cluster
Choose an IAM role to add
Feature
Choose a feature to add
Add role
Current IAM roles for this cluster (2)
Delete

	Role	Feature	Status
○	docs-lab-aur-ml-role-for-sagemaker	SageMaker	✓ Active
○	docs-lab-role-for-comprehend-and-apg	Comprehend	✓ Active

The IAM setup for Amazon Comprehend is complete. Continue setting up your Aurora PostgreSQL to work with Aurora machine learning by installing the extension as detailed in Installing the Aurora machine learning extension

Setting up Aurora PostgreSQL to use Amazon SageMaker

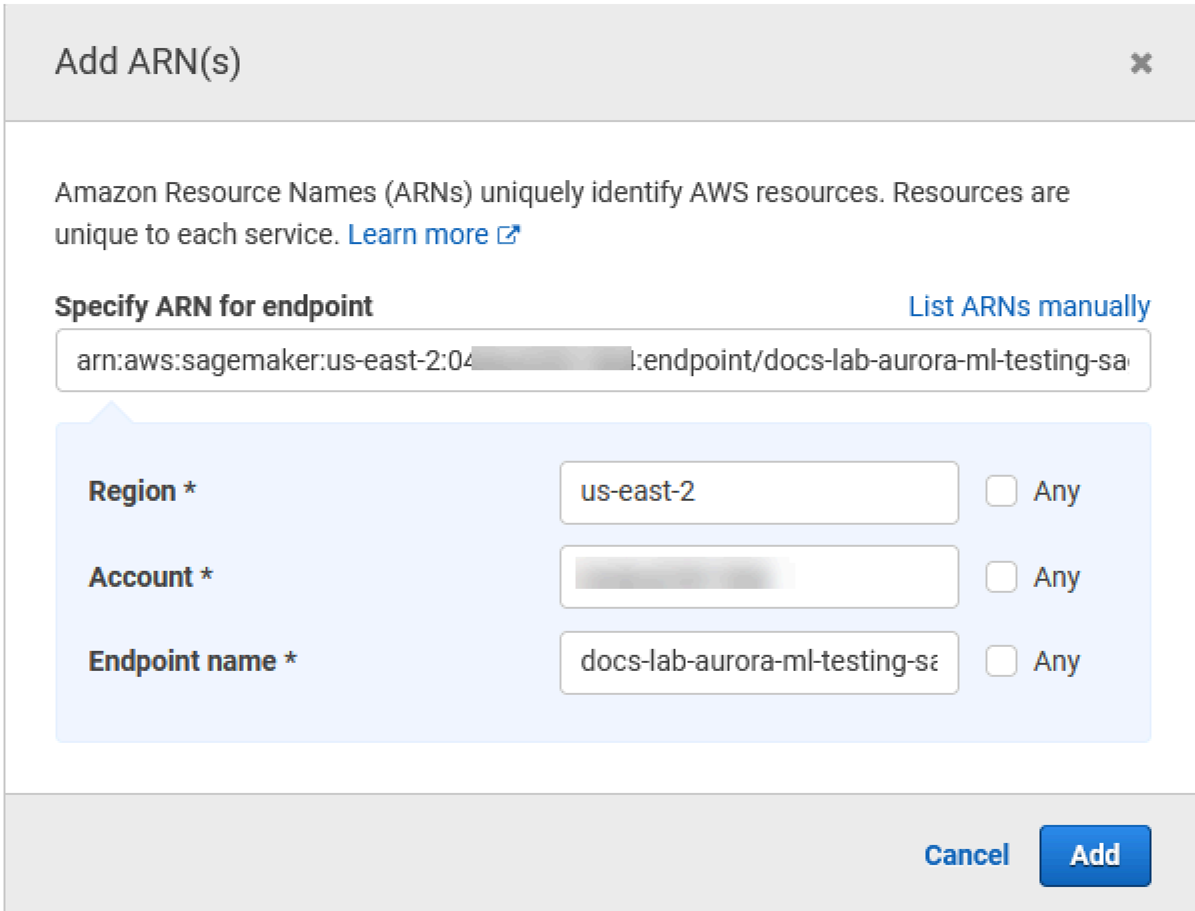
Before you can create the IAM policy and role for your Aurora PostgreSQL DB cluster, you need to have your SageMaker model setup and your endpoint available.

To set up your Aurora PostgreSQL DB cluster to use SageMaker

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Policies** (under Access management) on the AWS Identity and Access Management (IAM) Console menu, and then choose **Create policy**. In the Visual editor, choose **SageMaker** for the Service. For Actions, open the Read selector (under Access level) and choose **InvokeEndpoint**. When you this, a warning icon displays.

3. Open the Resources selector and choose the **Add ARN to restrict access** link under the Specify endpoint resource ARN for the InvokeEndpoint action.

4. Enter the AWS Region of your SageMaker resources and the name of your endpoint. Your AWS account is prefilled.



Amazon Resource Names (ARNs) uniquely identify AWS resources. Resources are unique to each service. Learn more

Specify ARN for endpoint — List ARNs manually

arn:aws:sagemaker:us-east-2:04[redacted]:endpoint/docs-lab-aurora-ml-testing-sa

Region *	us-east-2	□ Any
Account *	[redacted]	□ Any
Endpoint name *	docs-lab-aurora-ml-testing-sa	□ Any

5. Choose **Add** to save. Choose **Next: Tags** and **Next: Review** to get to the last page of the policy creation process.

6. Enter a Name and Description for this policy, and then choose **Create policy**. The policy is created and is added to the Policies list. You see an alert in the Console as this occurs.

7. On the IAM Console, choose **Roles**.

8. Choose **Create role**.

9. On the Select trusted entity page, choose the **AWS service** tile, and then choose **RDS** to open the selector.

10. Choose **RDS – Add Role to Database**.

11. Choose **Next**. On the Add permissions page, find the policy that you created in the previous step and choose it from among those listed. Choose **Next**

12. **Next: Review**. Enter a name for the IAM role and a description.
13. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
14. Navigate to the AWS Region where your Aurora PostgreSQL DB cluster is located.
15. In the navigation pane, choose **Databases**, and then choose the Aurora PostgreSQL DB cluster that you want to use with SageMaker.
16. Choose the **Connectivity & security** tab and scroll to find the **Manage IAM roles** section of the page. From the **Add IAM roles to this cluster** selector, choose the role that you created in the previous steps. In the **Feature** selector, choose SageMaker, and then choose **Add role**.

The role (with its policy) are associated with the Aurora PostgreSQL DB cluster. When the process completes, the role is listed in the Current IAM roles for this cluster listing.

The IAM setup for SageMaker is complete. Continue setting up your Aurora PostgreSQL to work with Aurora machine learning by installing the extension as detailed in Installing the Aurora machine learning extension.

Setting up Aurora PostgreSQL to use Amazon S3 for SageMaker (Advanced)

To use SageMaker with your own models rather than using the pre-built components provided by SageMaker, you need to set up an Amazon Simple Storage Service (Amazon S3) bucket for Aurora PostgreSQL DB cluster to use. This is an advanced topic, and isn't fully documented in this *Amazon Aurora User Guide*. The general process is the same as for integrating support for SageMaker, as follows.

1. Create the IAM policy and role for Amazon S3.
2. Add the IAM role and the Amazon S3 import or export as a feature on the Connectivity & security tab of your Aurora PostgreSQL DB cluster.
3. Add the ARN of the role to your custom DB cluster parameter group for your Aurora DB cluster.

For basic usage information, see Exporting data to Amazon S3 for SageMaker model training (Advanced).

Installing the Aurora machine learning extension

The Aurora machine learning extensions `aws_ml 1.0` provides two functions that you can use to invoke Amazon Comprehend, SageMaker services and `aws_ml 2.0` provides two additional functions that you can use to invoke Amazon Bedrock services. Installing these extensions on your Aurora PostgreSQL DB cluster also creates an administrative role for the feature.

> **Note**
>
> Using these functions depends on having the IAM setup for the Aurora machine learning service (Amazon Comprehend, SageMaker, Amazon Bedrock) complete, as detailed in Setting up your Aurora PostgreSQL DB cluster to use Aurora machine learning.

- **aws_comprehend.detect_sentiment** – You use this function to apply sentiment analysis to text stored in in the database on your Aurora PostgreSQL DB cluster.
- **aws_sagemaker.invoke_endpoint** – You use this function in your SQL code to communicate with the SageMaker endpoint from your cluster.
- **aws_bedrock.invoke_model** – You use this function in your SQL code to communicate with the Bedrock Models from your cluster. The response of this function will be in the format of a TEXT, so if a model responds in the format of a JSON body then the output of this function will be relayed in the format of a string to the end user.
- **aws_bedrock.invoke_model_get_embeddings** – You use this function in your SQL code to invoke Bedrock Models that return output embeddings within a JSON response. This can be leveraged when you want to extract the embeddings directly associated with the json-key to streamline the response with any self-managed workflows.

To install the Aurora machine learning extension in your Aurora PostgreSQL DB cluster

- Use `psql` to connect to the writer instance of your Aurora PostgreSQL DB cluster. Connect to the specific database in which to install the `aws_ml` extension.

  ```
  psql --host=cluster-instance-1.111122223333.aws-region.rds.amazonaws.com --
  port=5432 --username=postgres --password --dbname=labdb
  ```

```
labdb=> CREATE EXTENSION IF NOT EXISTS aws_ml CASCADE;
NOTICE:  installing required extension "aws_commons"
CREATE EXTENSION
labdb=>
```

Installing the aws_ml extensions also creates the aws_ml administrative role and two new schemas, as follows.

- aws_comprehend – Schema for the Amazon Comprehend service and source of the detect_sentiment function (aws_comprehend.detect_sentiment).
- aws_sagemaker – Schema for the SageMaker service and source of the invoke_endpoint function (aws_sagemaker.invoke_endpoint).
- aws_bedrock – Schema for the Amazon Bedrock service and source of the invoke_model(aws_bedrock.invoke_model) and invoke_model_get_embeddings(aws_bedrock.invoke_model_get_embeddings) functions.

The rds_superuser role is granted the aws_ml administrative role and is made the OWNER of these two Aurora machine learning schemas. To allow other database users to access the Aurora machine learning functions, the rds_superuser needs to grant EXECUTE privileges on the Aurora machine learning functions. By default, EXECUTE privileges are revoked from PUBLIC on the functions in the two Aurora machine learning schemas.

In a multi-tenant database configuration, you can prevent tenants from accessing Aurora machine learning functions by using REVOKE USAGE on the specific Aurora machine learning schema that you want to protect.

Using Amazon Bedrock with your Aurora PostgreSQL DB cluster

For Aurora PostgreSQL, Aurora machine learning provides the following Amazon Bedrock function for working with your text data. This function is available only after you install the aws_ml 2.0 extension and complete all setup procedures. For more information, see Setting up your Aurora PostgreSQL DB cluster to use Aurora machine learning.

aws_bedrock.invoke_model

This function takes text formatted in JSON as input and processes it for variety of models hosted on Amazon Bedrock and gets back the JSON text response from the model. This response could contain text, image, or embeddings. A summary of the function's documentation is as follows.

```
aws_bedrock.invoke_model(
   IN model_id      varchar,
   IN content_type  text,
   IN accept_type   text,
   IN model_input   text,
   OUT model_output varchar)
```

The inputs and outputs of this function are as follows.

- `model_id` – Identifier of the model.
- `content_type` – The type of the request to Bedrock's model.
- `accept_type` – The type of the response to expect from Bedrock's model. Usually application/JSON for most of the models.
- `model_input` – Prompts; a specific set of inputs to the model in the format as specified by content_type. For more information on the request format/structure the model accepts, see Inference parameters for foundation models.
- `model_output` – The Bedrock model's output as text.

The following example shows how to invoke a Anthropic Claude 2 model for Bedrock using invoke_model.

Example Example: A simple query using Amazon Bedrock functions

```
SELECT aws_bedrock.invoke_model (
    model_id    := 'anthropic.claude-v2',
    content_type:= 'application/json',
    accept_type := 'application/json',
    model_input := '{"prompt": "\n\nHuman: You are a helpful assistant that answers
 questions directly and only using the information provided in the context below.
\nDescribe the answer
    in detail.\n\nContext: %s \n\nQuestion: %s \n
\nAssistant:","max_tokens_to_sample":4096,"temperature":0.5,"top_k":250,"top_p":0.5,"stop_seque
[]}'
);
```

aws_bedrock.invoke_model_get_embeddings

The model output can point to vector embeddings for some cases. Given the response varies per model, another function invoke_model_get_embeddings can be leveraged which works exactly like invoke_model but outputs the embeddings by specifying the appropriate json-key.

```
aws_bedrock.invoke_model_get_embeddings(
   IN model_id      varchar,
   IN content_type  text,
   IN json_key      text,
   IN model_input   text,
   OUT model_output float8[])
```

The inputs and outputs of this function are as follows.

- `model_id` – Identifier of the model.
- `content_type` – The type of the request to Bedrock's model. Here, the accept_type is set to default value `application/json`.
- `model_input` – Prompts; a specific set of inputs to the Model in the format as specified by content_type. For more information on the request format/structure the Model accepts, see Inference parameters for foundation models.
- `json_key` – Reference to the field to extract the embedding from. This may vary if the embedding model changes.
- `model_output` – The Bedrock model's output as an array of embeddings having 16 bit decimals.

The following example shows how to generate an embedding using the Titan Embeddings G1 – Text embedding model for the phrase PostgreSQL I/O monitoring views.

Example Example: A simple query using Amazon Bedrock functions

```
SELECT aws_bedrock.invoke_model_get_embeddings(
   model_id      := 'amazon.titan-embed-text-v1',
   content_type  := 'application/json',
   json_key      := 'embedding',
   model_input   := '{ "inputText": "PostgreSQL I/O monitoring views"}') AS embedding;
```

Using Amazon Comprehend with your Aurora PostgreSQL DB cluster

For Aurora PostgreSQL, Aurora machine learning provides the following Amazon Comprehend function for working with your text data. This function is available only after you install the `aws_ml` extension and complete all setup procedures. For more information, see Setting up your Aurora PostgreSQL DB cluster to use Aurora machine learning.

aws_comprehend.detect_sentiment

This function takes text as input and evaluates whether the text has a a positive, negative, neutral, or mixed emotional posture. It outputs this sentiment along with a confidence level for its evaluation. A summary of the function's documentation is as follows.

```
aws_comprehend.detect_sentiment(
   IN input_text varchar,
   IN language_code varchar,
   IN max_rows_per_batch int,
   OUT sentiment varchar,
   OUT confidence real)
```

The inputs and outputs of this function are as follows.

- `input_text` – The text to evaluate and to assign sentiment (negative, positive, neutral, mixed).
- `language_code` – The language of the `input_text` identified using the 2-letter ISO 639-1 identifier with regional subtag (as needed) or the ISO 639-2 three-letter code, as appropriate. For example, `en` is the code for English, `zh` is the code for simplified Chinese. For more information, see Supported languages in the *Amazon Comprehend Developer Guide*.
- `max_rows_per_batch` – The maximum number of rows per batch for batch-mode processing. For more information, see Understanding batch mode and Aurora machine learning functions.
- `sentiment` – The sentiment of the input text, identified as POSITIVE, NEGATIVE, NEUTRAL, or MIXED.
- `confidence` – The degree of confidence in the accuracy of the specified `sentiment`. Values range from 0.0 to 1.0.

In the following, you can find examples of how to use this function.

Example Example: A simple query using Amazon Comprehend functions

Here's an example of a simple query that invokes this function to assess customer satisfaction with your support team. Suppose you have a database table (`support`) that stores customer feedback after each request for help. This example query applies the `aws_comprehend.detect_sentiment` function to the text in the `feedback` column of the table and outputs the sentiment and the confidence level for that sentiment. This query also outputs results in descending order.

```
SELECT feedback, s.sentiment,s.confidence
    FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
    ORDER BY s.confidence DESC;
 feedback                         | sentiment | confidence
 ---------------------------------------------------------+-----------+------------
 Thank you for the excellent customer support!            | POSITIVE  |   0.999771
 The latest version of this product stinks!               | NEGATIVE  |   0.999184
 Your support team is just awesome! I am blown away.      | POSITIVE  |   0.997774
 Your product is too complex, but your support is great.  | MIXED     |   0.957958
 Your support tech helped me in fifteen minutes.          | POSITIVE  |   0.949491
 My problem was never resolved!                           | NEGATIVE  |   0.920644
 When will the new version of this product be released?   | NEUTRAL   |   0.902706
 I cannot stand that chatbot.                             | NEGATIVE  |   0.895219
 Your support tech talked down to me.                     | NEGATIVE  |   0.868598
 It took me way too long to get a real person.            | NEGATIVE  |   0.481805

 (10 rows)
```

To avoid being charged for sentiment detection more than once per table row, you can materialize the results. Do this on the rows of interest. For example, the clinician's notes are being updated so that only those in French (`fr`) use the sentiment detection function.

```

UPDATE clinician_notes
SET sentiment = (aws_comprehend.detect_sentiment (french_notes, 'fr')).sentiment,
    confidence = (aws_comprehend.detect_sentiment (french_notes, 'fr')).confidence
WHERE
    clinician_notes.french_notes IS NOT NULL AND
    LENGTH(TRIM(clinician_notes.french_notes)) > 0 AND
    clinician_notes.sentiment IS NULL;
```

For more information on optimizing your function calls, see Performance considerations for using Aurora machine learning with Aurora PostgreSQL.

Using SageMaker with your Aurora PostgreSQL DB cluster

After setting up your SageMaker environment and integrating with Aurora PostgreSQL as outlined in Setting up Aurora PostgreSQL to use Amazon SageMaker , you can invoke operations by using the `aws_sagemaker.invoke_endpoint` function. The `aws_sagemaker.invoke_endpoint` function connects only to a model endpoint in the same AWS Region. If your database instance has replicas in multiple AWS Regions be sure that you setup and deploy each SageMaker model to every AWS Region.

Calls to `aws_sagemaker.invoke_endpoint` are authenticated using the IAM role that you set up to associated your Aurora PostgreSQL DB cluster with the SageMaker service and the endpoint that you provided during the setup process. SageMaker model endpoints are scoped to an individual account and are not public. The `endpoint_name` URL doesn't contain the account ID. SageMaker determines the account ID from the authentication token that is supplied by the SageMaker IAM role of the database instance.

aws_sagemaker.invoke_endpoint

This function takes the SageMaker endpoint as input and the number of rows that should be processed as a batch. It also takes as input the various parameters expected by the SageMaker model endpoint. This function's reference documentation is as follows.

```
aws_sagemaker.invoke_endpoint(
  IN endpoint_name varchar,
  IN max_rows_per_batch int,
  VARIADIC model_input "any",
  OUT model_output varchar
  )
```

The inputs and outputs of this function are as follows.

- `endpoint_name` – An endpoint URL that is AWS Region–independent.
- `max_rows_per_batch` – The maximum number of rows per batch for batch-mode processing. For more information, see Understanding batch mode and Aurora machine learning functions.
- `model_input` – One or more input parameters for the model. These can be any data type needed by the SageMaker model. PostgreSQL allows you to specify up to 100 input parameters for a function. Array data types must be one-dimensional, but can contain as many elements as are expected by the SageMaker model. The number of inputs to a SageMaker model is limited only by the SageMaker 6 MB message size limit.
- `model_output` – The SageMaker model's output as text.

Creating a user-defined function to invoke a SageMaker model

Create a separate user-defined function to call `aws_sagemaker.invoke_endpoint` for each of your SageMaker models. Your user-defined function represents the SageMaker endpoint hosting the model. The `aws_sagemaker.invoke_endpoint` function runs within the user-defined function. User-defined functions provide many advantages:

- You can give your SageMaker model its own name instead of only calling `aws_sagemaker.invoke_endpoint` for all of your SageMaker models.
- You can specify the model endpoint URL in just one place in your SQL application code.
- You can control `EXECUTE` privileges to each Aurora machine learning function independently.
- You can declare the model input and output types using SQL types. SQL enforces the number and type of arguments passed to your SageMaker model and performs type conversion if necessary. Using SQL types will also translate `SQL NULL` to the appropriate default value expected by your SageMaker model.
- You can reduce the maximum batch size if you want to return the first few rows a little faster.

To specify a user-defined function, use the SQL data definition language (DDL) statement `CREATE FUNCTION`. When you define the function, you specify the following:

- The input parameters to the model.
- The specific SageMaker endpoint to invoke.
- The return type.

The user-defined function returns the inference computed by the SageMaker endpoint after running the model on the input parameters. The following example creates a user-defined function for an SageMaker model with two input parameters.

```
CREATE FUNCTION classify_event (IN arg1 INT, IN arg2 DATE, OUT category INT)
AS $$
    SELECT aws_sagemaker.invoke_endpoint (
        'sagemaker_model_endpoint_name', NULL,
        arg1, arg2                        -- model inputs are separate arguments
        )::INT                            -- cast the output to INT
$$ LANGUAGE SQL PARALLEL SAFE COST 5000;
```

Note the following:

- The aws_sagemaker.invoke_endpoint function input can be one or more parameters of any data type.
- This example uses an INT output type. If you cast the output from a varchar type to a different type, then it must be cast to a PostgreSQL builtin scalar type such as INTEGER, REAL, FLOAT, or NUMERIC. For more information about these types, see Data types in the PostgreSQL documentation.
- Specify PARALLEL SAFE to enable parallel query processing. For more information, see Improving response times with parallel query processing.
- Specify COST 5000 to estimate the cost of running the function. Use a positive number giving the estimated run cost for the function, in units of cpu_operator_cost.

Passing an array as input to a SageMaker model

The aws_sagemaker.invoke_endpoint function can have up to 100 input parameters, which is the limit for PostgreSQL functions. If the SageMaker model requires more than 100 parameters of the same type, pass the model parameters as an array.

The following example defines a function that passes an array as input to the SageMaker regression model. The output is cast to a REAL value.

```
CREATE FUNCTION regression_model (params REAL[], OUT estimate REAL)
AS $$
    SELECT aws_sagemaker.invoke_endpoint (
      'sagemaker_model_endpoint_name',
      NULL,
      params
      )::REAL
$$ LANGUAGE SQL PARALLEL SAFE COST 5000;
```

Specifying batch size when invoking a SageMaker model

The following example creates a user-defined function for a SageMaker model that sets the batch size default to NULL. The function also allows you to provide a different batch size when you invoke it.

```
CREATE FUNCTION classify_event (
    IN event_type INT, IN event_day DATE, IN amount REAL, -- model inputs
    max_rows_per_batch INT DEFAULT NULL,  -- optional batch size limit
    OUT category INT)                     -- model output
AS $$
    SELECT aws_sagemaker.invoke_endpoint (
        'sagemaker_model_endpoint_name', max_rows_per_batch,
        event_type, event_day, COALESCE(amount, 0.0)
        )::INT              -- casts output to type INT
$$ LANGUAGE SQL PARALLEL SAFE COST 5000;
```

Note the following:

- Use the optional `max_rows_per_batch` parameter to provide control of the number of rows for a batch-mode function invocation. If you use a value of NULL, then the query optimizer automatically chooses the maximum batch size. For more information, see Understanding batch mode and Aurora machine learning functions.
- By default, passing NULL as a parameter's value is translated to an empty string before passing to SageMaker. For this example the inputs have different types.
- If you have a non-text input, or text input that needs to default to some value other than an empty string, use the `COALESCE` statement. Use `COALESCE` to translate NULL to the desired null replacement value in the call to `aws_sagemaker.invoke_endpoint`. For the `amount` parameter in this example, a NULL value is converted to 0.0.

Invoking a SageMaker model that has multiple outputs

The following example creates a user-defined function for a SageMaker model that returns multiple outputs. Your function needs to cast the output of the `aws_sagemaker.invoke_endpoint` function to a corresponding data type. For example, you could use the built-in PostgreSQL point type for (x,y) pairs or a user-defined composite type.

This user-defined function returns values from a model that returns multiple outputs by using a composite type for the outputs.

```
CREATE TYPE company_forecasts AS (
    six_month_estimated_return real,
    one_year_bankruptcy_probability float);
CREATE FUNCTION analyze_company (
    IN free_cash_flow NUMERIC(18, 6),
    IN debt NUMERIC(18,6),
    IN max_rows_per_batch INT DEFAULT NULL,
    OUT prediction company_forecasts)
```

```
AS $$
    SELECT (aws_sagemaker.invoke_endpoint('endpt_name',
       max_rows_per_batch,free_cash_flow, debt))::company_forecasts;


$$ LANGUAGE SQL PARALLEL SAFE COST 5000;
```

For the composite type, use fields in the same order as they appear in the model output and cast the output of `aws_sagemaker.invoke_endpoint` to your composite type. The caller can extract the individual fields either by name or with PostgreSQL ".*" notation.

Exporting data to Amazon S3 for SageMaker model training (Advanced)

We recommend that you become familiar with Aurora machine learning and SageMaker by using the provided algorithms and examples rather than trying to train your own models. For more information, see Get Started with Amazon SageMaker

To train SageMaker models, you export data to an Amazon S3 bucket. The Amazon S3 bucket is used by SageMaker to train your model before it is deployed. You can query data from an Aurora PostgreSQL DB cluster and save it directly into text files stored in an Amazon S3 bucket. Then SageMaker consumes the data from the Amazon S3 bucket for training. For more about SageMaker model training, see Train a model with Amazon SageMaker.

Note

When you create an Amazon S3 bucket for SageMaker model training or batch scoring, use `sagemaker` in the Amazon S3 bucket name. For more information, see Specify a Amazon S3 Bucket to Upload Training Datasets and Store Output Data in the *Amazon SageMaker Developer Guide*.

For more information about exporting your data, see Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3.

Performance considerations for using Aurora machine learning with Aurora PostgreSQL

The Amazon Comprehend and SageMaker services do most of the work when invoked by an Aurora machine learning function. That means that you can scale those resources as needed, independently. For your Aurora PostgreSQL DB cluster, you can make your function calls as efficient as possible. Following, you can find some performance considerations to note when working with Aurora machine learning from Aurora PostgreSQL.

Topics

Understanding batch mode and Aurora machine learning functions

Typically, PostgreSQL runs functions one row at a time. Aurora machine learning can reduce this overhead by combining the calls to the external Aurora machine learning service for many rows into batches with an approach called *batch-mode execution*. In batch mode, Aurora machine learning receives the responses for a batch of input rows, and then delivers the responses back to the running query one row at a time. This optimization improves the throughput of your Aurora queries without limiting the PostgreSQL query optimizer.

Aurora automatically uses batch mode if the function is referenced from the `SELECT` list, a `WHERE` clause, or a `HAVING` clause. Note that top-level simple `CASE` expressions are eligible for batch-mode execution. Top-level searched `CASE` expressions are also eligible for batch-mode execution provided that the first `WHEN` clause is a simple predicate with a batch-mode function call.

Your user-defined function must be a `LANGUAGE SQL` function and should specify `PARALLEL SAFE` and `COST 5000`.

Function migration from the SELECT statement to the FROM clause

Usually, an `aws_ml` function that is eligible for batch-mode execution is automatically migrated by Aurora to the `FROM` clause.

The migration of eligible batch-mode functions to the FROM clause can be examined manually on a per-query level. To do this, you use EXPLAIN statements (and ANALYZE and VERBOSE) and find the "Batch Processing" information below each batch-mode `Function Scan`. You can also use EXPLAIN (with VERBOSE) without running the query. You then observe whether the calls to the function appear as a `Function Scan` under a nested loop join that was not specified in the original statement.

In the following example, the nested loop join operator in the plan shows that Aurora migrated the anomaly_score function. It migrated this function from the SELECT list to the FROM clause, where it's eligible for batch-mode execution.

```
EXPLAIN (VERBOSE, COSTS false)
SELECT anomaly_score(ts.R.description) from ts.R;
                         QUERY PLAN
-------------------------------------------------------------
 Nested Loop
   Output: anomaly_score((r.description)::text)
   ->  Seq Scan on ts.r
         Output: r.id, r.description, r.score
   ->  Function Scan on public.anomaly_score
         Output: anomaly_score.anomaly_score
         Function Call: anomaly_score((r.description)::text)
```

To disable batch-mode execution, set the apg_enable_function_migration parameter to false. This prevents the migration of aws_ml functions from the SELECT to the FROM clause. The following shows how.

```
SET apg_enable_function_migration = false;
```

The apg_enable_function_migration parameter is a Grand Unified Configuration (GUC) parameter that is recognized by the Aurora PostgreSQL apg_plan_mgmt extension for query plan management. To disable function migration in a session, use query plan management to save the resulting plan as an approved plan. At runtime, query plan management enforces the approved plan with its apg_enable_function_migration setting. This enforcement occurs regardless of the apg_enable_function_migration GUC parameter setting. For more information, see Managing query execution plans for Aurora PostgreSQL.

Using the max_rows_per_batch parameter

Both the aws_comprehend.detect_sentiment and the aws_sagemaker.invoke_endpoint functions have a max_rows_per_batch parameter. This parameter specifies the number of rows that can be sent to the Aurora machine learning service. The larger the dataset processed by your function, the larger you can make the batch size.

Batch-mode functions improve efficiency by building batches of rows that spread the cost of the Aurora machine learning function calls over a large number of rows. However, if a SELECT statement finishes early due to a LIMIT clause, then the batch can be constructed over more rows than the query uses. This approach can result in additional charges to your AWS account. To gain the benefits of batch-mode execution but avoid building batches that are too large, use a smaller value for the max_rows_per_batch parameter in your function calls.

If you do an EXPLAIN (VERBOSE, ANALYZE) of a query that uses batch-mode execution, you see a FunctionScan operator that is below a nested loop join. The number of loops reported by EXPLAIN equals the number of times a row was fetched from the FunctionScan operator. If a statement uses a LIMIT clause, the number of fetches is consistent. To optimize the size of the batch, set the max_rows_per_batch parameter to this value. However, if the batch-mode function is referenced in a predicate in the WHERE clause or HAVING clause, then you probably can't know the number of fetches in advance. In this case, use the loops as a guideline and experiment with max_rows_per_batch to find a setting that optimizes performance.

Verifying batch-mode execution

To see if a function ran in batch mode, use EXPLAIN ANALYZE. If batch-mode execution was used, then the query plan will include the information in a "Batch Processing" section.

```
EXPLAIN ANALYZE SELECT user-defined-function();
 Batch Processing: num batches=1 avg/min/max batch size=3333.000/3333.000/3333.000
                                avg/min/max batch call time=146.273/146.273/146.273
```

In this example, there was 1 batch that contained 3,333 rows, which took 146.273 ms to process. The "Batch Processing" section shows the following:

- How many batches there were for this function scan operation
- The batch size average, minimum, and maximum
- The batch execution time average, minimum, and maximum

Typically the final batch is smaller than the rest, which often results in a minimum batch size that is much smaller than the average.

To return the first few rows more quickly, set the max_rows_per_batch parameter to a smaller value.

To reduce the number of batch mode calls to the ML service when you use a LIMIT in your user-defined function, set the max_rows_per_batch parameter to a smaller value.

Improving response times with parallel query processing

To get results as fast as possible from a large number of rows, you can combine parallel query processing with batch mode processing. You can use parallel query processing for `SELECT`, `CREATE TABLE AS SELECT`, and `CREATE MATERIALIZED VIEW` statements.

> **Note**
>
> PostgreSQL doesn't yet support parallel query for data manipulation language (DML) statements.

Parallel query processing occurs both within the database and within the ML service. The number of cores in the instance class of the database limits the degree of parallelism that can be used when running a query. The database server can construct a parallel query execution plan that partitions the task among a set of parallel workers. Then each of these workers can build batched requests containing tens of thousands of rows (or as many as are allowed by each service).

The batched requests from all of the parallel workers are sent to the SageMaker endpoint. The degree of parallelism that the endpoint can support is constrained by the number and type of instances that support it. For *K* degrees of parallelism, you need a database instance class that has at least *K* cores. You also need to configure the SageMaker endpoint for your model to have *K* initial instances of a sufficiently high-performing instance class.

To use parallel query processing, you can set the `parallel_workers` storage parameter of the table that contains the data that you plan to pass. You set `parallel_workers` to a batch-mode function such as `aws_comprehend.detect_sentiment`. If the optimizer chooses a parallel query plan, the AWS ML services can be called both in batch and in parallel.

You can use the following parameters with the `aws_comprehend.detect_sentiment` function to get a plan with four-way parallelism. If you change either of the following two parameters, you must restart the database instance for the changes to take effect

```
-- SET max_worker_processes to 8;  -- default value is 8
-- SET max_parallel_workers to 8;  -- not greater than max_worker_processes
SET max_parallel_workers_per_gather to 4;  -- not greater than max_parallel_workers

-- You can set the parallel_workers storage parameter on the table that the data
-- for the Aurora machine learning function is coming from in order to manually
 override the degree of
-- parallelism that would otherwise be chosen by the query optimizer
--
ALTER TABLE yourTable SET (parallel_workers = 4);

-- Example query to exploit both batch-mode execution and parallel query
EXPLAIN (verbose, analyze, buffers, hashes)
SELECT aws_comprehend.detect_sentiment(description, 'en')).*
FROM yourTable
WHERE id < 100;
```

For more information about controlling parallel query, see Parallel plans in the PostgreSQL documentation.

Using materialized views and materialized columns

When you invoke an AWS service such as SageMaker or Amazon Comprehend from your database, your account is charged according to the pricing policy of that service. To minimize charges to your account, you can materialize the result of calling the AWS service into a materialized column so that the AWS service is not called more than once per input row. If desired, you can add a `materializedAt` timestamp column to record the time at which the columns were materialized.

The latency of an ordinary single-row `INSERT` statement is typically much less than the latency of calling a batch-mode function. Thus, you might not be able to meet the latency requirements of your application if you invoke the batch-mode function for every single-row `INSERT` that your application performs. To materialize the result of calling an AWS service into a materialized column, high-performance applications generally need to populate the materialized columns. To do this, they periodically issue an `UPDATE` statement that operates on a large batch of rows at the same time.

`UPDATE` takes a row-level lock that can impact a running application. So you might need to use `SELECT ... FOR UPDATE SKIP LOCKED`, or use `MATERIALIZED VIEW`.

Analytic queries that operate on a large number of rows in real time can combine batch-mode materialization with real-time processing. To do this, these queries assemble a `UNION ALL` of the pre-materialized results with a query over the rows that don't yet have materialized results. In some cases, such a `UNION ALL` is needed in multiple places, or the query is generated by a third-party application. If so, you can create a `VIEW` to encapsulate the `UNION ALL` operation so this detail isn't exposed to the rest of the SQL application.

You can use a materialized view to materialize the results of an arbitrary `SELECT` statement at a snapshot in time. You can also use it to refresh the materialized view at any time in the future. Currently PostgreSQL doesn't support incremental refresh, so each time the materialized view is refreshed the materialized view is fully recomputed.

You can refresh materialized views with the `CONCURRENTLY` option, which updates the contents of the materialized view without taking an exclusive lock. Doing this allows a SQL application to read from the materialized view while it's being refreshed.

Monitoring Aurora machine learning

You can monitor the `aws_ml` functions by setting the `track_functions` parameter in your custom DB cluster parameter group to `all`. By default, this parameter is set to `pl` which means that only procedure-language functions are tracked. By changing this to `all`, the `aws_ml` functions are also tracked. For more information, see Run-time Statistics in the PostgreSQL documentation.

For information about monitoring the performance of the SageMaker operations called from Aurora machine learning functions, see Monitor Amazon SageMaker in the *Amazon SageMaker Developer Guide*.

With `track_functions` set to `all`, you can query the `pg_stat_user_functions` view to get statistics about the functions that you define and use to invoke Aurora machine learning services. For each function, the view provides the number of `calls`, `total_time`, and `self_time`.

To view the statistics for the `aws_sagemaker.invoke_endpoint` and the `aws_comprehend.detect_sentiment` functions, you can filter results by schema name using the following query.

```
SELECT * FROM pg_stat_user_functions
    WHERE schemaname
    LIKE 'aws_%';
```

To clear the statistics, do as follows.

```
SELECT pg_stat_reset();
```

You can get the names of your SQL functions that call the `aws_sagemaker.invoke_endpoint` function by querying the PostgreSQL `pg_proc` system catalog. This catalog stores information about functions, procedures, and more. For more information, see pg_proc in the PostgreSQL documentation. Following is an example of querying the table to get the names of functions (`proname`) whose source (`prosrc`) includes the text *invoke_endpoint*.

```
SELECT proname FROM pg_proc WHERE prosrc LIKE '%invoke_endpoint%';
```

Code examples for Aurora using AWS SDKs

The following code examples show how to use Aurora with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

Cross-service examples are sample applications that work across multiple AWS services.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Aurora

The following code examples show how to get started using Aurora.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
using Amazon.RDS;
using Amazon.RDS.Model;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
namespace AuroraActions;

public static class HelloAurora
{
    static async Task Main(string[] args)
    {
        // Use the AWS .NET Core Setup package to set up dependency injection for
 the
        // Amazon Relational Database Service (Amazon RDS).
        // Use your AWS profile name, or leave it blank to use the default
 profile.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureServices((_, services) =>
                services.AddAWSService<IAmazonRDS>()
            ).Build();

        // Now the client is available for injection. Fetching it directly here
 for example purposes only.
        var rdsClient = host.Services.GetRequiredService<IAmazonRDS>();

        // You can use await and any of the async methods to get a response.
        var response = await rdsClient.DescribeDBClustersAsync(new
 DescribeDBClustersRequest { IncludeShared = true });
        Console.WriteLine($"Hello Amazon RDS Aurora! Let's list some clusters in
 this account:");
        foreach (var cluster in response.DBClusters)
        {
            Console.WriteLine($"\tCluster: database: {cluster.DatabaseName}
 identifier: {cluster.DBClusterIdentifier}.");
        }
    }
}
```

- For API details, see DescribeDBClusters in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Code for the CMakeLists.txt CMake file.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS rds)

# Set this project's name.
project("hello_aurora")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.
```

```
     # set(BIN_SUB_DIR "/Debug") # If you are building from the command line, you
 may need to uncomment this
                                    # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_aurora.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code for the hello_aurora.cpp source file.

```
#include <aws/core/Aws.h>
#include <aws/rds/RDSClient.h>
#include <aws/rds/model/DescribeDBClustersRequest.h>
#include <iostream>

/*
 *  A "Hello Aurora" starter application which initializes an Amazon Relational
 Database Service (Amazon RDS) client
 *  and describes the Amazon Aurora (Aurora) clusters.
 *
 *  main function
 *
 *  Usage: 'hello_aurora'
 *
 */
int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";
        Aws::RDS::RDSClient rdsClient(clientConfig);

        Aws::String marker; // Used for pagination.
        std::vector<Aws::String> clusterIds;
        do {
            Aws::RDS::Model::DescribeDBClustersRequest request;

            Aws::RDS::Model::DescribeDBClustersOutcome outcome =
                    rdsClient.DescribeDBClusters(request);

            if (outcome.IsSuccess()) {
                for (auto &cluster: outcome.GetResult().GetDBClusters()) {
                    clusterIds.push_back(cluster.GetDBClusterIdentifier());
                }
                marker = outcome.GetResult().GetMarker();
            } else {
                result = 1;
                std::cerr << "Error with Aurora::GDescribeDBClusters. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                break;
            }
        } while (!marker.empty());

        std::cout << clusterIds.size() << " Aurora clusters found." << std::endl;
        for (auto &clusterId: clusterIds) {
            std::cout << "  clusterId " << clusterId << std::endl;
        }
    }

    Aws::ShutdownAPI(options); // Should only be called once.
    return 0;
}
```

- For API details, see DescribeDBClusters in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/rds"
)

// main uses the AWS SDK for Go V2 to create an Amazon Aurora client and list up
 to 20
// DB clusters in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 sdkConfig, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 auroraClient := rds.NewFromConfig(sdkConfig)
 const maxClusters = 20
 fmt.Printf("Let's list up to %v DB clusters.\n", maxClusters)
 output, err := auroraClient.DescribeDBClusters(context.TODO(),
  &rds.DescribeDBClustersInput{MaxRecords: aws.Int32(maxClusters)})
 if err != nil {
  fmt.Printf("Couldn't list DB clusters: %v\n", err)
  return
 }
 if len(output.DBClusters) == 0 {
  fmt.Println("No DB clusters found.")
 } else {
  for _, cluster := range output.DBClusters {
   fmt.Printf("DB cluster %v has database %v.\n", *cluster.DBClusterIdentifier,
    *cluster.DatabaseName)
  }
 }
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rds.RdsClient;
import software.amazon.awssdk.services.rds.paginators.DescribeDBClustersIterable;

public class DescribeDbClusters {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;
        RdsClient rdsClient = RdsClient.builder()
                .region(region)
                .build();

        describeClusters(rdsClient);
        rdsClient.close();
    }

    public static void describeClusters(RdsClient rdsClient) {
        DescribeDBClustersIterable clustersIterable =
 rdsClient.describeDBClustersPaginator();
        clustersIterable.stream()
                .flatMap(r -> r.dbClusters().stream())
                .forEach(cluster -> System.out
                        .println("Database name: " + cluster.databaseName() + "
 Arn = " + cluster.dbClusterArn()));
    }
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Java 2.x API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_sdk_rds::Client;

#[derive(Debug)]
struct Error(String);
impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}", self.0)
    }
}
impl std::error::Error for Error {}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt::init();
    let sdk_config = aws_config::from_env().load().await;
    let client = Client::new(&sdk_config);
    let describe_db_clusters_output = client
        .describe_db_clusters()
        .send()
        .await
        .map_err(|e| Error(e.to_string()))?;
    println!(
        "Found {} clusters:",
        describe_db_clusters_output.db_clusters().len()
    );
    for cluster in describe_db_clusters_output.db_clusters() {
        let name = cluster.database_name().unwrap_or("Unknown");
        let engine = cluster.engine().unwrap_or("Unknown");
        let id = cluster.db_cluster_identifier().unwrap_or("Unknown");
        let class = cluster.db_cluster_instance_class().unwrap_or("Unknown");
        println!("\tDatabase: {name}",);
        println!("\t  Engine: {engine}",);
        println!("\t      ID: {id}",);
        println!("\tInstance: {class}",);
    }

    Ok(())
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Rust API reference*.

Code examples

- Actions for Aurora using AWS SDKs
  - Use CreateDBCluster with an AWS SDK or CLI
  - Use CreateDBClusterParameterGroup with an AWS SDK or CLI
  - Use CreateDBClusterSnapshot with an AWS SDK or CLI
  - Use CreateDBInstance with an AWS SDK or CLI
  - Use DeleteDBCluster with an AWS SDK or CLI
  - Use DeleteDBClusterParameterGroup with an AWS SDK or CLI
  - Use DeleteDBInstance with an AWS SDK or CLI
  - Use DescribeDBClusterParameterGroups with an AWS SDK or CLI
  - Use DescribeDBClusterParameters with an AWS SDK or CLI
  - Use DescribeDBClusterSnapshots with an AWS SDK or CLI

Actions for Aurora using AWS SDKs

The following code examples demonstrate how to perform individual Aurora actions with AWS SDKs. These excerpts call the Aurora API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Amazon Aurora API Reference.

Examples

- Use DescribeDBClusters with an AWS SDK or CLI
- Use DescribeDBEngineVersions with an AWS SDK or CLI
- Use DescribeDBInstances with an AWS SDK or CLI
- Use DescribeOrderableDBInstanceOptions with an AWS SDK or CLI
- Use ModifyDBClusterParameterGroup with an AWS SDK or CLI

Use `CreateDBCluster` with an AWS SDK or CLI

The following code examples show how to use `CreateDBCluster`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Create a new cluster and database.
    /// </summary>
    /// <param name="dbName">The name of the new database.</param>
    /// <param name="clusterIdentifier">The identifier of the cluster.</param>
    /// <param name="parameterGroupName">The name of the parameter group.</param>
    /// <param name="dbEngine">The engine to use for the new cluster.</param>
    /// <param name="dbEngineVersion">The version of the engine to use.</param>
    /// <param name="adminName">The admin username.</param>
    /// <param name="adminPassword">The primary admin password.</param>
    /// <returns>The cluster object.</returns>
    public async Task<DBCluster> CreateDBClusterWithAdminAsync(
        string dbName,
        string clusterIdentifier,
        string parameterGroupName,
        string dbEngine,
        string dbEngineVersion,
        string adminName,
        string adminPassword)
    {
        var request = new CreateDBClusterRequest
        {
            DatabaseName = dbName,
            DBClusterIdentifier = clusterIdentifier,
            DBClusterParameterGroupName = parameterGroupName,
            Engine = dbEngine,
            EngineVersion = dbEngineVersion,
            MasterUsername = adminName,
            MasterUserPassword = adminPassword,
        };

        var response = await _amazonRDS.CreateDBClusterAsync(request);
        return response.DBCluster;
    }
```

- For API details, see CreateDBCluster in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);

        Aws::RDS::Model::CreateDBClusterRequest request;
        request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        request.SetEngine(engineName);
        request.SetEngineVersion(engineVersionName);
        request.SetMasterUsername(administratorName);
        request.SetMasterUserPassword(administratorPassword);

        Aws::RDS::Model::CreateDBClusterOutcome outcome =
                client.CreateDBCluster(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB cluster creation has started."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::CreateDBCluster. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
            return false;
        }
```

- For API details, see CreateDBCluster in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}
```

```
// CreateDbCluster creates a DB cluster that is configured to use the specified
 parameter group.
// The newly created DB cluster contains a database that uses the specified
 engine and
// engine version.
func (clusters *DbClusters) CreateDbCluster(clusterName string,
 parameterGroupName string,
 dbName string, dbEngine string, dbEngineVersion string, adminName string,
 adminPassword string) (
 *types.DBCluster, error) {

 output, err := clusters.AuroraClient.CreateDBCluster(context.TODO(),
 &rds.CreateDBClusterInput{
  DBClusterIdentifier:         aws.String(clusterName),
  Engine:                      aws.String(dbEngine),
  DBClusterParameterGroupName: aws.String(parameterGroupName),
  DatabaseName:                aws.String(dbName),
  EngineVersion:               aws.String(dbEngineVersion),
  MasterUserPassword:          aws.String(adminPassword),
  MasterUsername:              aws.String(adminName),
 })
 if err != nil {
  log.Printf("Couldn't create DB cluster %v: %v\n", clusterName, err)
  return nil, err
 } else {
  return output.DBCluster, err
 }
}
```

- For API details, see CreateDBCluster in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static String createDBCluster(RdsClient rdsClient, String
dbParameterGroupFamily, String dbName,
            String dbClusterIdentifier, String userName, String password) {
        try {
            CreateDbClusterRequest clusterRequest =
CreateDbClusterRequest.builder()
                    .databaseName(dbName)
                    .dbClusterIdentifier(dbClusterIdentifier)
                    .dbClusterParameterGroupName(dbParameterGroupFamily)
                    .engine("aurora-mysql")
                    .masterUsername(userName)
                    .masterUserPassword(password)
                    .build();

            CreateDbClusterResponse response =
rdsClient.createDBCluster(clusterRequest);
            return response.dbCluster().dbClusterArn();

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
        return "";
    }
```

- For API details, see CreateDBCluster in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createDBCluster(
    dbParameterGroupFamilyVal: String?,
    dbName: String?,
    dbClusterIdentifierVal: String?,
    userName: String?,
    password: String?,
): String? {
    val clusterRequest =
        CreateDbClusterRequest {
            databaseName = dbName
            dbClusterIdentifier = dbClusterIdentifierVal
            dbClusterParameterGroupName = dbParameterGroupFamilyVal
            engine = "aurora-mysql"
            masterUsername = userName
            masterUserPassword = password
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbCluster(clusterRequest)
        return response.dbCluster?.dbClusterArn
    }
}
```

- For API details, see CreateDBCluster in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client
```

```
   @classmethod
   def from_client(cls):
       """
       Instantiates this class from a Boto3 client.
       """
       rds_client = boto3.client("rds")
       return cls(rds_client)


   def create_db_cluster(
       self,
       cluster_name,
       parameter_group_name,
       db_name,
       db_engine,
       db_engine_version,
       admin_name,
       admin_password,
   ):
       """
       Creates a DB cluster that is configured to use the specified parameter
group.
       The newly created DB cluster contains a database that uses the specified
engine and
       engine version.

       :param cluster_name: The name of the DB cluster to create.
       :param parameter_group_name: The name of the parameter group to associate
with
                                    the DB cluster.
       :param db_name: The name of the database to create.
       :param db_engine: The database engine of the database that is created,
such as MySql.
       :param db_engine_version: The version of the database engine.
       :param admin_name: The user name of the database administrator.
       :param admin_password: The password of the database administrator.
       :return: The newly created DB cluster.
       """
       try:
           response = self.rds_client.create_db_cluster(
               DatabaseName=db_name,
               DBClusterIdentifier=cluster_name,
               DBClusterParameterGroupName=parameter_group_name,
               Engine=db_engine,
               EngineVersion=db_engine_version,
               MasterUsername=admin_name,
               MasterUserPassword=admin_password,
           )
           cluster = response["DBCluster"]
       except ClientError as err:
           logger.error(
               "Couldn't create database %s. Here's why: %s: %s",
               db_name,
               err.response["Error"]["Code"],
               err.response["Error"]["Message"],
           )
           raise
       else:
           return cluster
```

- For API details, see CreateDBCluster in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   // Get a list of allowed engine versions.
rds.DescribeDbEngineVersions(Engine='aurora-mysql', DBParameterGroupFamily=<the
family used to create your parameter group in step 2>)
   // Create an Aurora DB cluster database cluster that contains a MySql
database and uses the parameter group you created.
   // Wait for DB cluster to be ready. Call rds.DescribeDBClusters and check for
Status == 'available'.
   // Get a list of instance classes available for the selected engine
and engine version. rds.DescribeOrderableDbInstanceOptions(Engine='mysql',
EngineVersion=).
```

```
   // Create a database instance in the cluster.
   // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check
for DBInstanceStatus == 'available'.
   pub async fn start_cluster_and_instance(&mut self) -> Result<(),
ScenarioError> {
       if self.password.is_none() {
           return Err(ScenarioError::with(
               "Must set Secret Password before starting a cluster",
           ));
       }
       let create_db_cluster = self
           .rds
           .create_db_cluster(
               DB_CLUSTER_IDENTIFIER,
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_ENGINE,
               self.engine_version.as_deref().expect("engine version"),
               self.username.as_deref().expect("username"),
               self.password
                   .replace(SecretString::new("".to_string()))
                   .expect("password"),
           )
           .await;
       if let Err(err) = create_db_cluster {
           return Err(ScenarioError::new(
               "Failed to create DB Cluster with cluster group",
               &err,
           ));
       }

       self.db_cluster_identifier = create_db_cluster
           .unwrap()
           .db_cluster
           .and_then(|c| c.db_cluster_identifier);

       if self.db_cluster_identifier.is_none() {
           return Err(ScenarioError::with("Created DB Cluster missing
Identifier"));
       }

       info!(
           "Started a db cluster: {}",
           self.db_cluster_identifier
               .as_deref()
```

```
                .unwrap_or("Missing ARN")
        );

        let create_db_instance = self
            .rds
            .create_db_instance(
                self.db_cluster_identifier.as_deref().expect("cluster name"),
                DB_INSTANCE_IDENTIFIER,
                self.instance_class.as_deref().expect("instance class"),
                DB_ENGINE,
            )
            .await;
        if let Err(err) = create_db_instance {
            return Err(ScenarioError::new(
                "Failed to create Instance in DB Cluster",
                &err,
            ));
        }

        self.db_instance_identifier = create_db_instance
            .unwrap()
            .db_instance
            .and_then(|i| i.db_instance_identifier);

        // Cluster creation can take up to 20 minutes to become available
        let cluster_max_wait = Duration::from_secs(20 * 60);
        let waiter = Waiter::builder().max(cluster_max_wait).build();
        while waiter.sleep().await.is_ok() {
            let cluster = self
                .rds
                .describe_db_clusters(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = cluster {
                warn!(?err, "Failed to describe cluster while waiting for
ready");
                continue;
            }

            let instance = self
```

```
                .rds
                .describe_db_instance(
                    self.db_instance_identifier
                        .as_deref()
                        .expect("instance identifier"),
                )
                .await;
            if let Err(err) = instance {
                return Err(ScenarioError::new(
                    "Failed to find instance for cluster",
                    &err,
                ));
            }

            let instances_available = instance
                .unwrap()
                .db_instances()
                .iter()
                .all(|instance| instance.db_instance_status() ==
Some("Available"));

            let endpoints = self
                .rds
                .describe_db_cluster_endpoints(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = endpoints {
                return Err(ScenarioError::new(
                    "Failed to find endpoint for cluster",
                    &err,
                ));
            }

            let endpoints_available = endpoints
                .unwrap()
                .db_cluster_endpoints()
                .iter()
                .all(|endpoint| endpoint.status() == Some("available"));

            if instances_available && endpoints_available {
                return Ok(());
            }
        }

        Err(ScenarioError::with("timed out waiting for cluster"))
    }

    pub async fn create_db_cluster(
        &self,
        name: &str,
        parameter_group: &str,
        engine: &str,
        version: &str,
        username: &str,
        password: SecretString,
    ) -> Result<CreateDbClusterOutput, SdkError<CreateDBClusterError>> {
        self.inner
            .create_db_cluster()
            .db_cluster_identifier(name)
            .db_cluster_parameter_group_name(parameter_group)
            .engine(engine)
            .engine_version(version)
            .master_username(username)
            .master_user_password(password.expose_secret())
            .send()
            .await
    }

#[tokio::test]
async fn test_start_cluster_and_instance() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
             Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                 .build())
         });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|id| {
            Ok(DescribeDbClustersOutput::builder()

 .db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_describe_db_instance()
        .with(eq("RustSDKCodeExamplesDBInstance"))
        .return_once(|name| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
```

```
                        .db_instance_identifier(name)
                        .db_instance_status("Available")
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
        assert!(scenario
            .password
            .replace(SecretString::new("BAD SECRET".into()))
            .unwrap()
            .expose_secret()
            .is_empty());
        assert_eq!(
            scenario.db_cluster_identifier,
            Some("RustSDKCodeExamplesDBCluster".into())
        );
    });
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
```

```
async fn test_start_cluster_and_instance_cluster_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Err(SdkError::service_error(
                CreateDBClusterError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _}) if message
 == "Failed to create DB Cluster with cluster group")
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_missing_id() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
                .db_cluster(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));
    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context:_ }) if message
 == "Created DB Cluster missing Identifier");
}

#[tokio::test]
async fn test_start_cluster_and_instance_instance_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .return_once(|_, _, _, _| {
            Err(SdkError::service_error(
                CreateDBInstanceError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db instance error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _ }) if message
 == "Failed to create Instance in DB Cluster")
}

#[tokio::test]
async fn test_start_cluster_and_instance_wait_hiccup() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
        .returning(|_| {
            Err(SdkError::service_error(
                DescribeDBClustersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
            ))
        })
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds.expect_describe_db_instance().return_once(|name| {
        Ok(DescribeDbInstancesOutput::builder()
            .db_instances(
                DbInstance::builder()
                    .db_instance_identifier(name)
                    .db_instance_status("Available")
                    .build(),
            )
            .build())
    });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see CreateDBCluster in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateDBClusterParameterGroup` with an AWS SDK or CLI

The following code examples show how to use `CreateDBClusterParameterGroup`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Create a custom cluster parameter group.
    /// </summary>
    /// <param name="parameterGroupFamily">The family of the parameter group.</
param>
    /// <param name="groupName">The name for the new parameter group.</param>
    /// <param name="description">A description for the new parameter group.</
param>
    /// <returns>The new parameter group object.</returns>
    public async Task<DBClusterParameterGroup>
 CreateCustomClusterParameterGroupAsync(
        string parameterGroupFamily,
        string groupName,
        string description)
    {
        var request = new CreateDBClusterParameterGroupRequest
        {
            DBParameterGroupFamily = parameterGroupFamily,
            DBClusterParameterGroupName = groupName,
            Description = description,
        };

        var response = await
 _amazonRDS.CreateDBClusterParameterGroupAsync(request);
        return response.DBClusterParameterGroup;
    }
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);

        Aws::RDS::Model::CreateDBClusterParameterGroupRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        request.SetDBParameterGroupFamily(dbParameterGroupFamily);
        request.SetDescription("Example cluster parameter group.");

        Aws::RDS::Model::CreateDBClusterParameterGroupOutcome outcome =
                client.CreateDBClusterParameterGroup(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB cluster parameter group was successfully
created."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::CreateDBClusterParameterGroup. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// CreateParameterGroup creates a DB cluster parameter group that is based on the
 specified
// parameter group family.
func (clusters *DbClusters) CreateParameterGroup(
 parameterGroupName string, parameterGroupFamily string, description string) (
 *types.DBClusterParameterGroup, error) {

 output, err :=
 clusters.AuroraClient.CreateDBClusterParameterGroup(context.TODO(),
  &rds.CreateDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   DBParameterGroupFamily:      aws.String(parameterGroupFamily),
   Description:                 aws.String(description),
  })
 if err != nil {
  log.Printf("Couldn't create parameter group %v: %v\n", parameterGroupName, err)
  return nil, err
 } else {
  return output.DBClusterParameterGroup, err
 }
}
```

- For API details, see [CreateDBClusterParameterGroup](https://pkg.go.dev/github.com/aws/aws-sdk-go-v2/service/rds#Client.CreateDBClusterParameterGroup) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void createDBClusterParameterGroup(RdsClient rdsClient, String
dbClusterGroupName,
           String dbParameterGroupFamily) {
       try {
           CreateDbClusterParameterGroupRequest groupRequest =
CreateDbClusterParameterGroupRequest.builder()
                   .dbClusterParameterGroupName(dbClusterGroupName)
                   .dbParameterGroupFamily(dbParameterGroupFamily)
                   .description("Created by using the AWS SDK for Java")
                   .build();

           CreateDbClusterParameterGroupResponse response =
rdsClient.createDBClusterParameterGroup(groupRequest);
           System.out.println("The group name is " +
response.dbClusterParameterGroup().dbClusterParameterGroupName());

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createDBClusterParameterGroup(
    dbClusterGroupNameVal: String?,
    dbParameterGroupFamilyVal: String?,
) {
    val groupRequest =
        CreateDbClusterParameterGroupRequest {
            dbClusterParameterGroupName = dbClusterGroupNameVal
            dbParameterGroupFamily = dbParameterGroupFamilyVal
            description = "Created by using the AWS SDK for Kotlin"
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbClusterParameterGroup(groupRequest)
        println("The group name is
 ${response.dbClusterParameterGroup?.dbClusterParameterGroupName}")
    }
}
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def create_parameter_group(
        self, parameter_group_name, parameter_group_family, description
    ):
        """
        Creates a DB cluster parameter group that is based on the specified
 parameter group
        family.

        :param parameter_group_name: The name of the newly created parameter
 group.
        :param parameter_group_family: The family that is used as the basis of
 the new
                                       parameter group.
        :param description: A description given to the parameter group.
        :return: Data about the newly created parameter group.
        """
        try:
            response = self.rds_client.create_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name,
                DBParameterGroupFamily=parameter_group_family,
                Description=description,
            )
        except ClientError as err:
            logger.error(
                "Couldn't create parameter group %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   // Select an engine family and create a custom DB cluster parameter group.
rds.CreateDbClusterParameterGroup(DBParameterGroupFamily='aurora-mysql8.0')
   pub async fn set_engine(&mut self, engine: &str, version: &str) -> Result<(),
ScenarioError> {
       self.engine_family = Some(engine.to_string());
       self.engine_version = Some(version.to_string());
       let create_db_cluster_parameter_group = self
           .rds
           .create_db_cluster_parameter_group(
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_CLUSTER_PARAMETER_GROUP_DESCRIPTION,
               engine,
           )
           .await;

       match create_db_cluster_parameter_group {
           Ok(CreateDbClusterParameterGroupOutput {
                db_cluster_parameter_group: None,
                ..
            }) => {
                return Err(ScenarioError::with(
                    "CreateDBClusterParameterGroup had empty response",
                ));
            }
            Err(error) => {
                if error.code() == Some("DBParameterGroupAlreadyExists") {
                    info!("Cluster Parameter Group already exists, nothing to
 do");
                } else {
                    return Err(ScenarioError::new(
                        "Could not create Cluster Parameter Group",
                        &error,
                    ));
                }
            }
            _ => {
                info!("Created Cluster Parameter Group");
            }
        }

        Ok(())
    }

    pub async fn create_db_cluster_parameter_group(
        &self,
        name: &str,
        description: &str,
        family: &str,
    ) -> Result<CreateDbClusterParameterGroupOutput,
 SdkError<CreateDBClusterParameterGroupError>>
    {
        self.inner
            .create_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .description(description)
            .db_parameter_group_family(family)
            .send()
            .await
    }

#[tokio::test]
```

```
async fn test_scenario_set_engine() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .with(
            eq("RustSDKCodeExamplesDBParameterGroup"),
            eq("Parameter Group created by Rust SDK Code Example"),
            eq("aurora-mysql"),
        )
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()

 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);

    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;

    assert_eq!(set_engine, Ok(()));
    assert_eq!(Some("aurora-mysql"), scenario.engine_family.as_deref());
    assert_eq!(Some("aurora-mysql8.0"), scenario.engine_version.as_deref());
}

#[tokio::test]
async fn test_scenario_set_engine_not_create() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .with(
            eq("RustSDKCodeExamplesDBParameterGroup"),
            eq("Parameter Group created by Rust SDK Code Example"),
            eq("aurora-mysql"),
        )
        .return_once(|_, _, _|
 Ok(CreateDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;

    assert!(set_engine.is_err());
}

#[tokio::test]
async fn test_scenario_set_engine_param_group_exists() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .withf(|_, _, _| true)
        .return_once(|_, _, _| {
            Err(SdkError::service_error(

 CreateDBClusterParameterGroupError::DbParameterGroupAlreadyExistsFault(
                    DbParameterGroupAlreadyExistsFault::builder().build(),
                ),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);

    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;

    assert!(set_engine.is_err());
}
```

- For API details, see CreateDBClusterParameterGroup in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateDBClusterSnapshot` with an AWS SDK or CLI

The following code examples show how to use `CreateDBClusterSnapshot`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Create a snapshot of a cluster.
    /// </summary>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <param name="snapshotIdentifier">Identifier for the snapshot.</param>
    /// <returns>DB snapshot object.</returns>
    public async Task<DBClusterSnapshot>
 CreateClusterSnapshotByIdentifierAsync(string dbClusterIdentifier, string
 snapshotIdentifier)
    {
        var response = await _amazonRDS.CreateDBClusterSnapshotAsync(
            new CreateDBClusterSnapshotRequest()
            {
                DBClusterIdentifier = dbClusterIdentifier,
                DBClusterSnapshotIdentifier = snapshotIdentifier,
            });

        return response.DBClusterSnapshot;
    }
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region (overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::RDS::RDSClient client(clientConfig);

           Aws::RDS::Model::CreateDBClusterSnapshotRequest request;
           request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
           request.SetDBClusterSnapshotIdentifier(snapshotID);

           Aws::RDS::Model::CreateDBClusterSnapshotOutcome outcome =
                   client.CreateDBClusterSnapshot(request);

           if (outcome.IsSuccess()) {
               std::cout << "Snapshot creation has started."
                         << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::CreateDBClusterSnapshot. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
               return false;
           }
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// CreateClusterSnapshot creates a snapshot of a DB cluster.
func (clusters *DbClusters) CreateClusterSnapshot(clusterName string,
 snapshotName string) (
 *types.DBClusterSnapshot, error) {
 output, err := clusters.AuroraClient.CreateDBClusterSnapshot(context.TODO(),
 &rds.CreateDBClusterSnapshotInput{
  DBClusterIdentifier:         aws.String(clusterName),
  DBClusterSnapshotIdentifier: aws.String(snapshotName),
 })
 if err != nil {
  log.Printf("Couldn't create snapshot %v: %v\n", snapshotName, err)
  return nil, err
 } else {
  return output.DBClusterSnapshot, nil
 }
}
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void createDBClusterSnapshot(RdsClient rdsClient, String
dbInstanceClusterIdentifier,
           String dbSnapshotIdentifier) {
       try {
           CreateDbClusterSnapshotRequest snapshotRequest =
CreateDbClusterSnapshotRequest.builder()
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .dbClusterSnapshotIdentifier(dbSnapshotIdentifier)
                   .build();

           CreateDbClusterSnapshotResponse response =
rdsClient.createDBClusterSnapshot(snapshotRequest);
           System.out.println("The Snapshot ARN is " +
response.dbClusterSnapshot().dbClusterSnapshotArn());

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createDBClusterSnapshot(
    dbInstanceClusterIdentifier: String?,
    dbSnapshotIdentifier: String?,
) {
    val snapshotRequest =
        CreateDbClusterSnapshotRequest {
            dbClusterIdentifier = dbInstanceClusterIdentifier
            dbClusterSnapshotIdentifier = dbSnapshotIdentifier
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbClusterSnapshot(snapshotRequest)
        println("The Snapshot ARN is
 ${response.dbClusterSnapshot?.dbClusterSnapshotArn}")
    }
}
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def create_cluster_snapshot(self, snapshot_id, cluster_id):
        """
        Creates a snapshot of a DB cluster.

        :param snapshot_id: The ID to give the created snapshot.
        :param cluster_id: The DB cluster to snapshot.
        :return: Data about the newly created snapshot.
        """
        try:
            response = self.rds_client.create_db_cluster_snapshot(
                DBClusterSnapshotIdentifier=snapshot_id,
 DBClusterIdentifier=cluster_id
            )
            snapshot = response["DBClusterSnapshot"]
        except ClientError as err:
            logger.error(
                "Couldn't create snapshot of %s. Here's why: %s: %s",
                cluster_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return snapshot
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   // Get a list of allowed engine versions.
rds.DescribeDbEngineVersions(Engine='aurora-mysql', DBParameterGroupFamily=<the
family used to create your parameter group in step 2>)
   // Create an Aurora DB cluster database cluster that contains a MySql
database and uses the parameter group you created.
   // Wait for DB cluster to be ready. Call rds.DescribeDBClusters and check for
Status == 'available'.
   // Get a list of instance classes available for the selected engine
and engine version. rds.DescribeOrderableDbInstanceOptions(Engine='mysql',
EngineVersion=).

   // Create a database instance in the cluster.
   // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check
for DBInstanceStatus == 'available'.
   pub async fn start_cluster_and_instance(&mut self) -> Result<(),
ScenarioError> {
       if self.password.is_none() {
           return Err(ScenarioError::with(
               "Must set Secret Password before starting a cluster",
           ));
       }
       let create_db_cluster = self
           .rds
           .create_db_cluster(
               DB_CLUSTER_IDENTIFIER,
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_ENGINE,
               self.engine_version.as_deref().expect("engine version"),
               self.username.as_deref().expect("username"),
               self.password
                   .replace(SecretString::new("".to_string()))
                   .expect("password"),
           )
           .await;
       if let Err(err) = create_db_cluster {
           return Err(ScenarioError::new(
               "Failed to create DB Cluster with cluster group",
               &err,
           ));
       }

       self.db_cluster_identifier = create_db_cluster
           .unwrap()
           .db_cluster
           .and_then(|c| c.db_cluster_identifier);

       if self.db_cluster_identifier.is_none() {
           return Err(ScenarioError::with("Created DB Cluster missing 
Identifier"));
       }

       info!(
           "Started a db cluster: {}",
           self.db_cluster_identifier
               .as_deref()
               .unwrap_or("Missing ARN")
       );

       let create_db_instance = self
           .rds
           .create_db_instance(
               self.db_cluster_identifier.as_deref().expect("cluster name"),
               DB_INSTANCE_IDENTIFIER,
               self.instance_class.as_deref().expect("instance class"),
               DB_ENGINE,
           )
           .await;
       if let Err(err) = create_db_instance {
           return Err(ScenarioError::new(
               "Failed to create Instance in DB Cluster",
               &err,
           ));
       }

       self.db_instance_identifier = create_db_instance
           .unwrap()
           .db_instance
           .and_then(|i| i.db_instance_identifier);

       // Cluster creation can take up to 20 minutes to become available
       let cluster_max_wait = Duration::from_secs(20 * 60);
       let waiter = Waiter::builder().max(cluster_max_wait).build();
       while waiter.sleep().await.is_ok() {
           let cluster = self
               .rds
               .describe_db_clusters(
                   self.db_cluster_identifier
                       .as_deref()
                       .expect("cluster identifier"),
               )
               .await;

           if let Err(err) = cluster {
               warn!(?err, "Failed to describe cluster while waiting for
ready");
               continue;
           }

           let instance = self
               .rds
               .describe_db_instance(
                   self.db_instance_identifier
                       .as_deref()
                       .expect("instance identifier"),
               )
               .await;
           if let Err(err) = instance {
               return Err(ScenarioError::new(
                   "Failed to find instance for cluster",
                   &err,
               ));
           }
```

```
           let instances_available = instance
               .unwrap()
               .db_instances()
               .iter()
               .all(|instance| instance.db_instance_status() ==
Some("Available"));

           let endpoints = self
               .rds
               .describe_db_cluster_endpoints(
                   self.db_cluster_identifier
                       .as_deref()
                       .expect("cluster identifier"),
               )
               .await;

           if let Err(err) = endpoints {
               return Err(ScenarioError::new(
                   "Failed to find endpoint for cluster",
                   &err,
               ));
           }

           let endpoints_available = endpoints
               .unwrap()
               .db_cluster_endpoints()
               .iter()
               .all(|endpoint| endpoint.status() == Some("available"));

           if instances_available && endpoints_available {
               return Ok(());
           }
       }

       Err(ScenarioError::with("timed out waiting for cluster"))
   }

   pub async fn snapshot_cluster(
       &self,
       db_cluster_identifier: &str,
       snapshot_name: &str,
   ) -> Result<CreateDbClusterSnapshotOutput,
SdkError<CreateDBClusterSnapshotError>> {
       self.inner
```

```
            .create_db_cluster_snapshot()
            .db_cluster_identifier(db_cluster_identifier)
            .db_cluster_snapshot_identifier(snapshot_name)
            .send()
            .await
    }

#[tokio::test]
async fn test_start_cluster_and_instance() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
```

```
                       .db_instance_class(class)
                       .build(),
               )
               .build())
       });

   mock_rds
       .expect_describe_db_clusters()
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .return_once(|id| {
           Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
               .build())
       });

   mock_rds
       .expect_describe_db_instance()
       .with(eq("RustSDKCodeExamplesDBInstance"))
       .return_once(|name| {
           Ok(DescribeDbInstancesOutput::builder()
               .db_instances(
                   DbInstance::builder()
                       .db_instance_identifier(name)
                       .db_instance_status("Available")
                       .build(),
               )
               .build())
       });

   mock_rds
       .expect_describe_db_cluster_endpoints()
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .return_once(|_| {
           Ok(DescribeDbClusterEndpointsOutput::builder()

.db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
               .build())
       });

   let mut scenario = AuroraScenario::new(mock_rds);
   scenario.engine_version = Some("aurora-mysql8.0".into());
   scenario.instance_class = Some("m5.large".into());
   scenario.username = Some("test username".into());
```

```
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
        assert!(scenario
            .password
            .replace(SecretString::new("BAD SECRET".into()))
            .unwrap()
            .expose_secret()
            .is_empty());
        assert_eq!(
            scenario.db_cluster_identifier,
            Some("RustSDKCodeExamplesDBCluster".into())
        );
    });
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Err(SdkError::service_error(
                CreateDBClusterError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));
    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _}) if message
 == "Failed to create DB Cluster with cluster group")
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_missing_id() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
                .db_cluster(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context:_ }) if message
 == "Created DB Cluster missing Identifier");
}

#[tokio::test]
async fn test_start_cluster_and_instance_instance_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .return_once(|_, _, _, _| {
            Err(SdkError::service_error(
                CreateDBInstanceError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db instance error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _ }) if message
 == "Failed to create Instance in DB Cluster")
}

#[tokio::test]
async fn test_start_cluster_and_instance_wait_hiccup() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
```

```
           true
       })
       .return_once(|id, _, _, _, _, _| {
           Ok(CreateDbClusterOutput::builder()

.db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
               .build())
       });

   mock_rds
       .expect_create_db_instance()
       .withf(|cluster, name, class, engine| {
           assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
           assert_eq!(name, "RustSDKCodeExamplesDBInstance");
           assert_eq!(class, "m5.large");
           assert_eq!(engine, "aurora-mysql");
           true
       })
       .return_once(|cluster, name, class, _| {
           Ok(CreateDbInstanceOutput::builder()
               .db_instance(
                   DbInstance::builder()
                       .db_cluster_identifier(cluster)
                       .db_instance_identifier(name)
                       .db_instance_class(class)
                       .build(),
               )
               .build())
       });

   mock_rds
       .expect_describe_db_clusters()
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .times(1)
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe cluster error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       })
```

```
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds.expect_describe_db_instance().return_once(|name| {
        Ok(DescribeDbInstancesOutput::builder()
            .db_instances(
                DbInstance::builder()
                    .db_instance_identifier(name)
                    .db_instance_status("Available")
                    .build(),
            )
            .build())
    });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

.db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see CreateDBClusterSnapshot in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateDBInstance` with an AWS SDK or CLI

The following code examples show how to use `CreateDBInstance`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Create an Amazon Relational Database Service (Amazon RDS) DB instance
    /// with a particular set of properties. Use the action
 DescribeDBInstancesAsync
    /// to determine when the DB instance is ready to use.
    /// </summary>
    /// <param name="dbInstanceIdentifier">DB instance identifier.</param>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <param name="dbEngine">The engine for the DB instance.</param>
    /// <param name="dbEngineVersion">Version for the DB instance.</param>
    /// <param name="instanceClass">Class for the DB instance.</param>
    /// <returns>DB instance object.</returns>
    public async Task<DBInstance> CreateDBInstanceInClusterAsync(
        string dbClusterIdentifier,
        string dbInstanceIdentifier,
        string dbEngine,
        string dbEngineVersion,
        string instanceClass)
    {
        // When creating the instance within a cluster, do not specify the name
 or size.
        var response = await _amazonRDS.CreateDBInstanceAsync(
            new CreateDBInstanceRequest()
            {
                DBClusterIdentifier = dbClusterIdentifier,
                DBInstanceIdentifier = dbInstanceIdentifier,
                Engine = dbEngine,
                EngineVersion = dbEngineVersion,
                DBInstanceClass = instanceClass
            });

        return response.DBInstance;
    }
```

- For API details, see CreateDBInstance in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);
```

```
        Aws::RDS::Model::CreateDBInstanceRequest request;
        request.SetDBInstanceIdentifier(DB_INSTANCE_IDENTIFIER);
        request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
        request.SetEngine(engineName);
        request.SetDBInstanceClass(dbInstanceClass);

        Aws::RDS::Model::CreateDBInstanceOutcome outcome =
                client.CreateDBInstance(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB instance creation has started."
                      << std::endl;
        }
        else {
            std::cerr << "Error with RDS::CreateDBInstance. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, DB_CLUSTER_IDENTIFIER,
 "",
                             client);
            return false;
        }
```

- For API details, see CreateDBInstance in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}
```

```
// CreateInstanceInCluster creates a database instance in an existing DB cluster.
 The first database that is
// created defaults to a read-write DB instance.
func (clusters *DbClusters) CreateInstanceInCluster(clusterName string,
 instanceName string,
 dbEngine string, dbInstanceClass string) (*types.DBInstance, error) {
 output, err := clusters.AuroraClient.CreateDBInstance(context.TODO(),
 &rds.CreateDBInstanceInput{
  DBInstanceIdentifier: aws.String(instanceName),
  DBClusterIdentifier:  aws.String(clusterName),
  Engine:               aws.String(dbEngine),
  DBInstanceClass:      aws.String(dbInstanceClass),
 })
 if err != nil {
  log.Printf("Couldn't create instance %v: %v\n", instanceName, err)
  return nil, err
 } else {
  return output.DBInstance, nil
 }
}
```

- For API details, see CreateDBInstance in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static String createDBInstanceCluster(RdsClient rdsClient,
            String dbInstanceIdentifier,
            String dbInstanceClusterIdentifier,
            String instanceClass) {
        try {
            CreateDbInstanceRequest instanceRequest =
 CreateDbInstanceRequest.builder()
                    .dbInstanceIdentifier(dbInstanceIdentifier)
                    .dbClusterIdentifier(dbInstanceClusterIdentifier)
                    .engine("aurora-mysql")
                    .dbInstanceClass(instanceClass)
                    .build();

            CreateDbInstanceResponse response =
 rdsClient.createDBInstance(instanceRequest);
            System.out.print("The status is " +
 response.dbInstance().dbInstanceStatus());
            return response.dbInstance().dbInstanceArn();

        } catch (RdsException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- For API details, see CreateDBInstance in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createDBInstanceCluster(
    dbInstanceIdentifierVal: String?,
    dbInstanceClusterIdentifierVal: String?,
    instanceClassVal: String?,
): String? {
    val instanceRequest =
        CreateDbInstanceRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
            dbClusterIdentifier = dbInstanceClusterIdentifierVal
            engine = "aurora-mysql"
            dbInstanceClass = instanceClassVal
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbInstance(instanceRequest)
        print("The status is ${response.dbInstance?.dbInstanceStatus}")
        return response.dbInstance?.dbInstanceArn
    }
}
```

- For API details, see CreateDBInstance in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)
    def create_instance_in_cluster(
        self, instance_id, cluster_id, db_engine, instance_class
    ):
        """
        Creates a database instance in an existing DB cluster. The first database
that is
        created defaults to a read-write DB instance.

        :param instance_id: The ID to give the newly created DB instance.
        :param cluster_id: The ID of the DB cluster where the DB instance is
created.
        :param db_engine: The database engine of a database to create in the DB
instance.
                          This must be compatible with the configured parameter
group
                          of the DB cluster.
        :param instance_class: The DB instance class for the newly created DB
instance.
        :return: Data about the newly created DB instance.
        """
        try:
            response = self.rds_client.create_db_instance(
                DBInstanceIdentifier=instance_id,
                DBClusterIdentifier=cluster_id,
                Engine=db_engine,
                DBInstanceClass=instance_class,
            )
            db_inst = response["DBInstance"]
        except ClientError as err:
            logger.error(
                "Couldn't create DB instance %s. Here's why: %s: %s",
                instance_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return db_inst
```

- For API details, see CreateDBInstance in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   // Get a list of allowed engine versions.
rds.DescribeDbEngineVersions(Engine='aurora-mysql', DBParameterGroupFamily=<the
family used to create your parameter group in step 2>)
   // Create an Aurora DB cluster database cluster that contains a MySql
database and uses the parameter group you created.
   // Wait for DB cluster to be ready. Call rds.DescribeDBClusters and check for
Status == 'available'.
   // Get a list of instance classes available for the selected engine
and engine version. rds.DescribeOrderableDbInstanceOptions(Engine='mysql',
EngineVersion=).

   // Create a database instance in the cluster.
   // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check
for DBInstanceStatus == 'available'.
   pub async fn start_cluster_and_instance(&mut self) -> Result<(),
ScenarioError> {
       if self.password.is_none() {
           return Err(ScenarioError::with(
               "Must set Secret Password before starting a cluster",
           ));
       }
       let create_db_cluster = self
           .rds
           .create_db_cluster(
               DB_CLUSTER_IDENTIFIER,
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_ENGINE,
               self.engine_version.as_deref().expect("engine version"),
               self.username.as_deref().expect("username"),
               self.password
                   .replace(SecretString::new("".to_string()))
                   .expect("password"),
           )
           .await;
       if let Err(err) = create_db_cluster {
           return Err(ScenarioError::new(
               "Failed to create DB Cluster with cluster group",
               &err,
           ));
       }

       self.db_cluster_identifier = create_db_cluster
           .unwrap()
           .db_cluster
           .and_then(|c| c.db_cluster_identifier);

       if self.db_cluster_identifier.is_none() {
           return Err(ScenarioError::with("Created DB Cluster missing
Identifier"));
       }

       info!(
           "Started a db cluster: {}",
           self.db_cluster_identifier
               .as_deref()
               .unwrap_or("Missing ARN")
       );

       let create_db_instance = self
           .rds
           .create_db_instance(
               self.db_cluster_identifier.as_deref().expect("cluster name"),
               DB_INSTANCE_IDENTIFIER,
               self.instance_class.as_deref().expect("instance class"),
               DB_ENGINE,
           )
           .await;
       if let Err(err) = create_db_instance {
           return Err(ScenarioError::new(
               "Failed to create Instance in DB Cluster",
               &err,
           ));
       }

       self.db_instance_identifier = create_db_instance
           .unwrap()
           .db_instance
           .and_then(|i| i.db_instance_identifier);

       // Cluster creation can take up to 20 minutes to become available
       let cluster_max_wait = Duration::from_secs(20 * 60);
       let waiter = Waiter::builder().max(cluster_max_wait).build();
       while waiter.sleep().await.is_ok() {
           let cluster = self
               .rds
               .describe_db_clusters(
                   self.db_cluster_identifier
                       .as_deref()
                       .expect("cluster identifier"),
               )
               .await;

           if let Err(err) = cluster {
               warn!(?err, "Failed to describe cluster while waiting for
ready");
               continue;
           }

           let instance = self
               .rds
               .describe_db_instance(
                   self.db_instance_identifier
                       .as_deref()
                       .expect("instance identifier"),
               )
               .await;
           if let Err(err) = instance {
               return Err(ScenarioError::new(
                   "Failed to find instance for cluster",
                   &err,
               ));
           }

           let instances_available = instance
               .unwrap()
               .db_instances()
               .iter()
               .all(|instance| instance.db_instance_status() ==
Some("Available"));
```

```
            let endpoints = self
                .rds
                .describe_db_cluster_endpoints(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = endpoints {
                return Err(ScenarioError::new(
                    "Failed to find endpoint for cluster",
                    &err,
                ));
            }

            let endpoints_available = endpoints
                .unwrap()
                .db_cluster_endpoints()
                .iter()
                .all(|endpoint| endpoint.status() == Some("available"));

            if instances_available && endpoints_available {
                return Ok(());
            }
        }

        Err(ScenarioError::with("timed out waiting for cluster"))
    }

    pub async fn create_db_instance(
        &self,
        cluster_name: &str,
        instance_name: &str,
        instance_class: &str,
        engine: &str,
    ) -> Result<CreateDbInstanceOutput, SdkError<CreateDBInstanceError>> {
        self.inner
            .create_db_instance()
            .db_cluster_identifier(cluster_name)
            .db_instance_identifier(instance_name)
            .db_instance_class(instance_class)
            .engine(engine)
            .send()
            .await
    }

#[tokio::test]
async fn test_start_cluster_and_instance() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|id| {
            Ok(DescribeDbClustersOutput::builder()

 .db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_describe_db_instance()
        .with(eq("RustSDKCodeExamplesDBInstance"))
        .return_once(|name| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_instance_identifier(name)
                        .db_instance_status("Available")
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
        assert!(scenario
            .password
            .replace(SecretString::new("BAD SECRET".into()))
            .unwrap()
            .expose_secret()
            .is_empty());
        assert_eq!(
            scenario.db_cluster_identifier,
            Some("RustSDKCodeExamplesDBCluster".into())
        );
    });
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Err(SdkError::service_error(
                CreateDBClusterError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _}) if message
 == "Failed to create DB Cluster with cluster group")
```

```
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_missing_id() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
                .db_cluster(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context:_ }) if message
 == "Created DB Cluster missing Identifier");
}

#[tokio::test]
async fn test_start_cluster_and_instance_instance_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
```

```
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .return_once(|_, _, _, _| {
            Err(SdkError::service_error(
                CreateDBInstanceError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db instance error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _ }) if message
 == "Failed to create Instance in DB Cluster")
}

#[tokio::test]
async fn test_start_cluster_and_instance_wait_hiccup() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
.db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
        .returning(|_| {
            Err(SdkError::service_error(
                DescribeDBClustersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
            ))
        })
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
 .db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds.expect_describe_db_instance().return_once(|name| {
        Ok(DescribeDbInstancesOutput::builder()
            .db_instances(
                DbInstance::builder()
                    .db_instance_identifier(name)
                    .db_instance_status("Available")
                    .build(),
            )
            .build())
    });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see CreateDBInstance in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteDBCluster` with an AWS SDK or CLI

The following code examples show how to use `DeleteDBCluster`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /// <summary>
   /// Delete a particular DB cluster.
   /// </summary>
   /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
   /// <returns>DB cluster object.</returns>
   public async Task<DBCluster> DeleteDBClusterByIdentifierAsync(string
dbClusterIdentifier)
   {
       var response = await _amazonRDS.DeleteDBClusterAsync(
           new DeleteDBClusterRequest()
           {
               DBClusterIdentifier = dbClusterIdentifier,
               SkipFinalSnapshot = true
           });
       return response.DBCluster;
   }
```

- For API details, see DeleteDBCluster in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region (overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::RDS::RDSClient client(clientConfig);

           Aws::RDS::Model::DeleteDBClusterRequest request;
           request.SetDBClusterIdentifier(dbClusterIdentifier);
           request.SetSkipFinalSnapshot(true);

           Aws::RDS::Model::DeleteDBClusterOutcome outcome =
                   client.DeleteDBCluster(request);

           if (outcome.IsSuccess()) {
               std::cout << "DB cluster deletion has started."
                         << std::endl;
               clusterDeleting = true;
               std::cout
                       << "Waiting for DB cluster to delete before deleting the
parameter group."
                       << std::endl;
               std::cout << "This may take a while." << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::DeleteDBCluster. "
                         << outcome.GetError().GetMessage()
```

```
                          << std::endl;
                result = false;
            }
```

- For API details, see DeleteDBCluster in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// DeleteDbCluster deletes a DB cluster without keeping a final snapshot.
func (clusters *DbClusters) DeleteDbCluster(clusterName string) error {
 _, err := clusters.AuroraClient.DeleteDBCluster(context.TODO(),
 &rds.DeleteDBClusterInput{
  DBClusterIdentifier: aws.String(clusterName),
  SkipFinalSnapshot:   true,
 })
 if err != nil {
  log.Printf("Couldn't delete DB cluster %v: %v\n", clusterName, err)
  return err
 } else {
  return nil
 }
}
```

- For API details, see DeleteDBCluster in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void deleteCluster(RdsClient rdsClient, String
dbInstanceClusterIdentifier) {
       try {
           DeleteDbClusterRequest deleteDbClusterRequest =
DeleteDbClusterRequest.builder()
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .skipFinalSnapshot(true)
                   .build();

           rdsClient.deleteDBCluster(deleteDbClusterRequest);
           System.out.println(dbInstanceClusterIdentifier + " was deleted!");

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see DeleteDBCluster in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
suspend fun deleteCluster(dbInstanceClusterIdentifier: String) {
    val deleteDbClusterRequest =
        DeleteDbClusterRequest {
            dbClusterIdentifier = dbInstanceClusterIdentifier
            skipFinalSnapshot = true
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        rdsClient.deleteDbCluster(deleteDbClusterRequest)
        println("$dbInstanceClusterIdentifier was deleted!")
    }
}
```

- For API details, see [DeleteDBCluster](#) in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def delete_db_cluster(self, cluster_name):
        """
        Deletes a DB cluster.

        :param cluster_name: The name of the DB cluster to delete.
        """
        try:
            self.rds_client.delete_db_cluster(
                DBClusterIdentifier=cluster_name, SkipFinalSnapshot=True
            )
            logger.info("Deleted DB cluster %s.", cluster_name)
        except ClientError:
            logger.exception("Couldn't delete DB cluster %s.", cluster_name)
            raise
```

- For API details, see DeleteDBCluster in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    pub async fn clean_up(self) -> Result<(), Vec<ScenarioError>> {
        let mut clean_up_errors: Vec<ScenarioError> = vec![];

        // Delete the instance. rds.DeleteDbInstance.
        let delete_db_instance = self
            .rds
            .delete_db_instance(
                self.db_instance_identifier
                    .as_deref()
                    .expect("instance identifier"),
            )
            .await;
        if let Err(err) = delete_db_instance {
            let identifier = self
                .db_instance_identifier
                .as_deref()
                .unwrap_or("Missing Instance Identifier");
            let message = format!("failed to delete db instance {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance to delete
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_instances =
self.rds.describe_db_instances().await;
                if let Err(err) = describe_db_instances {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check instance state during deletion",
                        &err,
                    ));
                    break;
                }
                let db_instances = describe_db_instances
                    .unwrap()
                    .db_instances()
                    .iter()
                    .filter(|instance| instance.db_cluster_identifier ==
self.db_cluster_identifier)
                    .cloned()
                    .collect::<Vec<DbInstance>>();

                if db_instances.is_empty() {
                    trace!("Delete Instance waited and no instances were found");
                    break;
                }
                match db_instances.first().unwrap().db_instance_status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but instances is in 
{status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB instance");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster. rds.DeleteDbCluster.
        let delete_db_cluster = self
            .rds
            .delete_db_cluster(
                self.db_cluster_identifier
                    .as_deref()
                    .expect("cluster identifier"),
            )
            .await;

        if let Err(err) = delete_db_cluster {
            let identifier = self
                .db_cluster_identifier
                .as_deref()
                .unwrap_or("Missing DB Cluster Identifier");
            let message = format!("failed to delete db cluster {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance and cluster to fully delete. 
rds.DescribeDbInstances and rds.DescribeDbClusters until both are not found.
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_clusters = self
                    .rds
                    .describe_db_clusters(
                        self.db_cluster_identifier
                            .as_deref()
                            .expect("cluster identifier"),
                    )
                    .await;
                if let Err(err) = describe_db_clusters {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check cluster state during deletion",
                        &err,
                    ));
                    break;
                }
                let describe_db_clusters = describe_db_clusters.unwrap();
                let db_clusters = describe_db_clusters.db_clusters();
                if db_clusters.is_empty() {
                    trace!("Delete cluster waited and no clusters were found");
                    break;
                }
                match db_clusters.first().unwrap().status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but clusters is in
{status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB cluster");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster parameter group.
rds.DeleteDbClusterParameterGroup.
        let delete_db_cluster_parameter_group = self
            .rds
            .delete_db_cluster_parameter_group(
                self.db_cluster_parameter_group
                    .map(|g| {
                        g.db_cluster_parameter_group_name
                            .unwrap_or_else(||
DB_CLUSTER_PARAMETER_GROUP_NAME.to_string())
                    })
                    .as_deref()
                    .expect("cluster parameter group name"),
            )
            .await;
        if let Err(error) = delete_db_cluster_parameter_group {
            clean_up_errors.push(ScenarioError::new(
                "Failed to delete the db cluster parameter group",
                &error,
            ))
        }

        if clean_up_errors.is_empty() {
            Ok(())
        } else {
            Err(clean_up_errors)
        }
    }

    pub async fn delete_db_cluster(
        &self,
        cluster_identifier: &str,
    ) -> Result<DeleteDbClusterOutput, SdkError<DeleteDBClusterError>> {
        self.inner
            .delete_db_cluster()
            .db_cluster_identifier(cluster_identifier)
            .skip_final_snapshot(true)
            .send()
            .await
    }

#[tokio::test]
async fn test_scenario_clean_up() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| Ok(DescribeDbInstancesOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| Ok(DescribeDbClustersOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_scenario_clean_up_errors() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
```

```
               .build())
       })
       .with()
       .times(1)
       .returning(|| {
           Err(SdkError::service_error(
               DescribeDBInstancesError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe db instances error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       });

   mock_rds
       .expect_delete_db_cluster()
       .with(eq("MockCluster"))
       .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

   mock_rds
       .expect_describe_db_clusters()
       .with(eq("MockCluster"))
       .times(1)
       .returning(|id| {
           Ok(DescribeDbClustersOutput::builder()
               .db_clusters(
                   DbCluster::builder()
                       .db_cluster_identifier(id)
                       .status("Deleting")
                       .build(),
               )
               .build())
       })
       .with(eq("MockCluster"))
       .times(1)
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe db clusters error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_err());
        let errs = clean_up.unwrap_err();
        assert_eq!(errs.len(), 2);
        assert_matches!(errs.get(0), Some(ScenarioError {message, context: _}) if
 message == "Failed to check instance state during deletion");
        assert_matches!(errs.get(1), Some(ScenarioError {message, context: _}) if
 message == "Failed to check cluster state during deletion");
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see DeleteDBCluster in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use DeleteDBClusterParameterGroup with an AWS SDK or CLI

The following code examples show how to use `DeleteDBClusterParameterGroup`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /// <summary>
   /// Delete a particular parameter group by name.
   /// </summary>
   /// <param name="groupName">The name of the parameter group.</param>
   /// <returns>True if successful.</returns>
   public async Task<bool> DeleteClusterParameterGroupByNameAsync(string
groupName)
   {
       var request = new DeleteDBClusterParameterGroupRequest
       {
           DBClusterParameterGroupName = groupName,
       };

       var response = await
_amazonRDS.DeleteDBClusterParameterGroupAsync(request);
       return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
   }
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);

        Aws::RDS::Model::DeleteDBClusterParameterGroupRequest request;
        request.SetDBClusterParameterGroupName(parameterGroupName);

        Aws::RDS::Model::DeleteDBClusterParameterGroupOutcome outcome =
                client.DeleteDBClusterParameterGroup(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB parameter group was successfully deleted."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::DeleteDBClusterParameterGroup. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            result = false;
        }
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// DeleteParameterGroup deletes the named DB cluster parameter group.
func (clusters *DbClusters) DeleteParameterGroup(parameterGroupName string) error
 {
 _, err := clusters.AuroraClient.DeleteDBClusterParameterGroup(context.TODO(),
  &rds.DeleteDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
  })
 if err != nil {
  log.Printf("Couldn't delete parameter group %v: %v\n", parameterGroupName, err)
  return err
 } else {
  return nil
 }
}
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void deleteDBClusterGroup(RdsClient rdsClient, String
dbClusterGroupName, String clusterDBARN)
            throws InterruptedException {
        try {
            boolean isDataDel = false;
            boolean didFind;
            String instanceARN;

            // Make sure that the database has been deleted.
            while (!isDataDel) {
                DescribeDbInstancesResponse response =
rdsClient.describeDBInstances();
                List<DBInstance> instanceList = response.dbInstances();
                int listSize = instanceList.size();
                didFind = false;
                int index = 1;
                for (DBInstance instance : instanceList) {
                    instanceARN = instance.dbInstanceArn();
                    if (instanceARN.compareTo(clusterDBARN) == 0) {
                        System.out.println(clusterDBARN + " still exists");
                        didFind = true;
                    }
                    if ((index == listSize) && (!didFind)) {
                        // Went through the entire list and did not find the
database ARN.
                        isDataDel = true;
                    }
                    Thread.sleep(sleepTime * 1000);
                    index++;
                }
            }
            DeleteDbClusterParameterGroupRequest clusterParameterGroupRequest =
 DeleteDbClusterParameterGroupRequest
                    .builder()
                    .dbClusterParameterGroupName(dbClusterGroupName)
                    .build();


 rdsClient.deleteDBClusterParameterGroup(clusterParameterGroupRequest);
            System.out.println(dbClusterGroupName + " was deleted.");

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
@Throws(InterruptedException::class)
suspend fun deleteDBClusterGroup(
    dbClusterGroupName: String,
    clusterDBARN: String,
) {
    var isDataDel = false
    var didFind: Boolean
    var instanceARN: String

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        // Make sure that the database has been deleted.
        while (!isDataDel) {
            val response = rdsClient.describeDbInstances()
            val instanceList = response.dbInstances
            val listSize = instanceList?.size
            isDataDel = false
            didFind = false
            var index = 1
            if (instanceList != null) {
                for (instance in instanceList) {
                    instanceARN = instance.dbInstanceArn.toString()
                    if (instanceARN.compareTo(clusterDBARN) == 0) {
                        println("$clusterDBARN still exists")
                        didFind = true
                    }
                    if (index == listSize && !didFind) {
                        // Went through the entire list and did not find the
 database ARN.
                        isDataDel = true
                    }
                    delay(slTime * 1000)
                    index++
                }
            }
        }
        val clusterParameterGroupRequest =
            DeleteDbClusterParameterGroupRequest {
                dbClusterParameterGroupName = dbClusterGroupName
            }

        rdsClient.deleteDbClusterParameterGroup(clusterParameterGroupRequest)
        println("$dbClusterGroupName was deleted.")
    }
}
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def delete_parameter_group(self, parameter_group_name):
        """
        Deletes a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to delete.
        :return: Data about the parameter group.
        """
        try:
            response = self.rds_client.delete_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name
            )
        except ClientError as err:
            logger.error(
                "Couldn't delete parameter group %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    pub async fn clean_up(self) -> Result<(), Vec<ScenarioError>> {
        let mut clean_up_errors: Vec<ScenarioError> = vec![];

        // Delete the instance. rds.DeleteDbInstance.
        let delete_db_instance = self
            .rds
            .delete_db_instance(
                self.db_instance_identifier
                    .as_deref()
                    .expect("instance identifier"),
            )
            .await;
        if let Err(err) = delete_db_instance {
            let identifier = self
                .db_instance_identifier
                .as_deref()
                .unwrap_or("Missing Instance Identifier");
            let message = format!("failed to delete db instance {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance to delete
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_instances =
self.rds.describe_db_instances().await;
                if let Err(err) = describe_db_instances {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check instance state during deletion",
                        &err,
                    ));
                    break;
                }
                let db_instances = describe_db_instances
                    .unwrap()
                    .db_instances()
                    .iter()
                    .filter(|instance| instance.db_cluster_identifier ==
self.db_cluster_identifier)
                    .cloned()
                    .collect::<Vec<DbInstance>>();

                if db_instances.is_empty() {
                    trace!("Delete Instance waited and no instances were found");
                    break;
                }
                match db_instances.first().unwrap().db_instance_status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but instances is in
{status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB instance");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster. rds.DeleteDbCluster.
        let delete_db_cluster = self
            .rds
            .delete_db_cluster(
                self.db_cluster_identifier
                    .as_deref()
                    .expect("cluster identifier"),
            )
            .await;

        if let Err(err) = delete_db_cluster {
            let identifier = self
                .db_cluster_identifier
                .as_deref()
                .unwrap_or("Missing DB Cluster Identifier");
            let message = format!("failed to delete db cluster {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance and cluster to fully delete.
rds.DescribeDbInstances and rds.DescribeDbClusters until both are not found.
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_clusters = self
                    .rds
                    .describe_db_clusters(
                        self.db_cluster_identifier
                            .as_deref()
                            .expect("cluster identifier"),
                    )
                    .await;
                if let Err(err) = describe_db_clusters {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check cluster state during deletion",
                        &err,
                    ));
                    break;
                }
                let describe_db_clusters = describe_db_clusters.unwrap();
                let db_clusters = describe_db_clusters.db_clusters();
                if db_clusters.is_empty() {
                    trace!("Delete cluster waited and no clusters were found");
                    break;
                }
                match db_clusters.first().unwrap().status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but clusters is in
 {status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB cluster");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster parameter group.
 rds.DeleteDbClusterParameterGroup.
        let delete_db_cluster_parameter_group = self
            .rds
            .delete_db_cluster_parameter_group(
                self.db_cluster_parameter_group
                    .map(|g| {
                        g.db_cluster_parameter_group_name
                            .unwrap_or_else(||
 DB_CLUSTER_PARAMETER_GROUP_NAME.to_string())
                    })
                    .as_deref()
                    .expect("cluster parameter group name"),
            )
            .await;
        if let Err(error) = delete_db_cluster_parameter_group {
            clean_up_errors.push(ScenarioError::new(
                "Failed to delete the db cluster parameter group",
                &error,
            ))
        }

        if clean_up_errors.is_empty() {
            Ok(())
        } else {
            Err(clean_up_errors)
        }
    }

    pub async fn delete_db_cluster_parameter_group(
        &self,
        name: &str,
    ) -> Result<DeleteDbClusterParameterGroupOutput,
 SdkError<DeleteDBClusterParameterGroupError>>
    {
        self.inner
            .delete_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .send()
            .await
    }

#[tokio::test]
async fn test_scenario_clean_up() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| Ok(DescribeDbInstancesOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
```

```
       .expect_describe_db_clusters()
       .with(eq("MockCluster"))
       .times(1)
       .returning(|id| {
           Ok(DescribeDbClustersOutput::builder()
               .db_clusters(
                   DbCluster::builder()
                       .db_cluster_identifier(id)
                       .status("Deleting")
                       .build(),
               )
               .build())
       })
       .with(eq("MockCluster"))
       .times(1)
       .returning(|_| Ok(DescribeDbClustersOutput::builder().build()));

   mock_rds
       .expect_delete_db_cluster_parameter_group()
       .with(eq("MockParamGroup"))
       .return_once(|_|
Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

   let mut scenario = AuroraScenario::new(mock_rds);
   scenario.db_cluster_identifier = Some(String::from("MockCluster"));
   scenario.db_instance_identifier = Some(String::from("MockInstance"));
   scenario.db_cluster_parameter_group = Some(
       DbClusterParameterGroup::builder()
           .db_cluster_parameter_group_name("MockParamGroup")
           .build(),
   );

   tokio::time::pause();
   let assertions = tokio::spawn(async move {
       let clean_up = scenario.clean_up().await;
       assert!(clean_up.is_ok());
   });

   tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
Describe Instances
   tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
Describe Instances
   tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
Describe Cluster
```

```
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_scenario_clean_up_errors() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| {
            Err(SdkError::service_error(
                DescribeDBInstancesError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db instances error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster()
```

```
       .with(eq("MockCluster"))
       .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

   mock_rds
       .expect_describe_db_clusters()
       .with(eq("MockCluster"))
       .times(1)
       .returning(|id| {
           Ok(DescribeDbClustersOutput::builder()
               .db_clusters(
                   DbCluster::builder()
                       .db_cluster_identifier(id)
                       .status("Deleting")
                       .build(),
               )
               .build())
       })
       .with(eq("MockCluster"))
       .times(1)
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe db clusters error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       });

   mock_rds
       .expect_delete_db_cluster_parameter_group()
       .with(eq("MockParamGroup"))
       .return_once(|_|
Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

   let mut scenario = AuroraScenario::new(mock_rds);
   scenario.db_cluster_identifier = Some(String::from("MockCluster"));
   scenario.db_instance_identifier = Some(String::from("MockInstance"));
   scenario.db_cluster_parameter_group = Some(
       DbClusterParameterGroup::builder()
           .db_cluster_parameter_group_name("MockParamGroup")
           .build(),
   );
```

```
    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_err());
        let errs = clean_up.unwrap_err();
        assert_eq!(errs.len(), 2);
        assert_matches!(errs.get(0), Some(ScenarioError {message, context: _}) if
 message == "Failed to check instance state during deletion");
        assert_matches!(errs.get(1), Some(ScenarioError {message, context: _}) if
 message == "Failed to check cluster state during deletion");
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see DeleteDBClusterParameterGroup in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteDBInstance` with an AWS SDK or CLI

The following code examples show how to use `DeleteDBInstance`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Delete a particular DB instance.
    /// </summary>
    /// <param name="dbInstanceIdentifier">DB instance identifier.</param>
    /// <returns>DB instance object.</returns>
    public async Task<DBInstance> DeleteDBInstanceByIdentifierAsync(string
dbInstanceIdentifier)
    {
        var response = await _amazonRDS.DeleteDBInstanceAsync(
            new DeleteDBInstanceRequest()
            {
                DBInstanceIdentifier = dbInstanceIdentifier,
                SkipFinalSnapshot = true,
                DeleteAutomatedBackups = true
            });

        return response.DBInstance;
    }
```

- For API details, see DeleteDBInstance in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region (overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::RDS::RDSClient client(clientConfig);

           Aws::RDS::Model::DeleteDBInstanceRequest request;
           request.SetDBInstanceIdentifier(dbInstanceIdentifier);
           request.SetSkipFinalSnapshot(true);
           request.SetDeleteAutomatedBackups(true);

           Aws::RDS::Model::DeleteDBInstanceOutcome outcome =
                   client.DeleteDBInstance(request);

           if (outcome.IsSuccess()) {
               std::cout << "DB instance deletion has started."
                         << std::endl;
               instanceDeleting = true;
               std::cout
                       << "Waiting for DB instance to delete before deleting the
parameter group."
                       << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::DeleteDBInstance. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               result = false;
           }
```

- For API details, see DeleteDBInstance in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// DeleteInstance deletes a DB instance.
func (clusters *DbClusters) DeleteInstance(instanceName string) error {
 _, err := clusters.AuroraClient.DeleteDBInstance(context.TODO(),
 &rds.DeleteDBInstanceInput{
  DBInstanceIdentifier:   aws.String(instanceName),
  SkipFinalSnapshot:      true,
  DeleteAutomatedBackups: aws.Bool(true),
 })
 if err != nil {
  log.Printf("Couldn't delete instance %v: %v\n", instanceName, err)
  return err
 } else {
  return nil
 }
}
```

- For API details, see DeleteDBInstance in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static void deleteDatabaseInstance(RdsClient rdsClient, String
 dbInstanceIdentifier) {
        try {
            DeleteDbInstanceRequest deleteDbInstanceRequest =
 DeleteDbInstanceRequest.builder()
                    .dbInstanceIdentifier(dbInstanceIdentifier)
                    .deleteAutomatedBackups(true)
                    .skipFinalSnapshot(true)
                    .build();

            DeleteDbInstanceResponse response =
 rdsClient.deleteDBInstance(deleteDbInstanceRequest);
            System.out.println("The status of the database is " +
 response.dbInstance().dbInstanceStatus());

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
```

- For API details, see DeleteDBInstance in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun deleteDBInstance(dbInstanceIdentifierVal: String) {
    val deleteDbInstanceRequest =
        DeleteDbInstanceRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
            deleteAutomatedBackups = true
            skipFinalSnapshot = true
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.deleteDbInstance(deleteDbInstanceRequest)
        print("The status of the database is
 ${response.dbInstance?.dbInstanceStatus}")
    }
}
```

- For API details, see DeleteDBInstance in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def delete_db_instance(self, instance_id):
        """
        Deletes a DB instance.

        :param instance_id: The ID of the DB instance to delete.
        :return: Data about the deleted DB instance.
        """
        try:
            response = self.rds_client.delete_db_instance(
                DBInstanceIdentifier=instance_id,
                SkipFinalSnapshot=True,
                DeleteAutomatedBackups=True,
            )
            db_inst = response["DBInstance"]
        except ClientError as err:
            logger.error(
                "Couldn't delete DB instance %s. Here's why: %s: %s",
                instance_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return db_inst
```

- For API details, see DeleteDBInstance in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    pub async fn clean_up(self) -> Result<(), Vec<ScenarioError>> {
        let mut clean_up_errors: Vec<ScenarioError> = vec![];

        // Delete the instance. rds.DeleteDbInstance.
        let delete_db_instance = self
            .rds
            .delete_db_instance(
                self.db_instance_identifier
```

```
                    .as_deref()
                    .expect("instance identifier"),
            )
            .await;
        if let Err(err) = delete_db_instance {
            let identifier = self
                .db_instance_identifier
                .as_deref()
                .unwrap_or("Missing Instance Identifier");
            let message = format!("failed to delete db instance {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance to delete
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_instances =
self.rds.describe_db_instances().await;
                if let Err(err) = describe_db_instances {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check instance state during deletion",
                        &err,
                    ));
                    break;
                }
                let db_instances = describe_db_instances
                    .unwrap()
                    .db_instances()
                    .iter()
                    .filter(|instance| instance.db_cluster_identifier ==
self.db_cluster_identifier)
                    .cloned()
                    .collect::<Vec<DbInstance>>();

                if db_instances.is_empty() {
                    trace!("Delete Instance waited and no instances were found");
                    break;
                }
                match db_instances.first().unwrap().db_instance_status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but instances is in
{status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB instance");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster. rds.DeleteDbCluster.
        let delete_db_cluster = self
            .rds
            .delete_db_cluster(
                self.db_cluster_identifier
                    .as_deref()
                    .expect("cluster identifier"),
            )
            .await;

        if let Err(err) = delete_db_cluster {
            let identifier = self
                .db_cluster_identifier
                .as_deref()
                .unwrap_or("Missing DB Cluster Identifier");
            let message = format!("failed to delete db cluster {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance and cluster to fully delete.
rds.DescribeDbInstances and rds.DescribeDbClusters until both are not found.
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_clusters = self
                    .rds
                    .describe_db_clusters(
                        self.db_cluster_identifier
                            .as_deref()
                            .expect("cluster identifier"),
                    )
                    .await;
                if let Err(err) = describe_db_clusters {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check cluster state during deletion",
                        &err,
                    ));
                    break;
```

```
               }
               let describe_db_clusters = describe_db_clusters.unwrap();
               let db_clusters = describe_db_clusters.db_clusters();
               if db_clusters.is_empty() {
                   trace!("Delete cluster waited and no clusters were found");
                   break;
               }
               match db_clusters.first().unwrap().status() {
                   Some("Deleting") => continue,
                   Some(status) => {
                       info!("Attempting to delete but clusters is in
{status}");
                       continue;
                   }
                   None => {
                       warn!("No status for DB cluster");
                       break;
                   }
               }
           }
       }

       // Delete the DB cluster parameter group.
rds.DeleteDbClusterParameterGroup.
       let delete_db_cluster_parameter_group = self
           .rds
           .delete_db_cluster_parameter_group(
               self.db_cluster_parameter_group
                   .map(|g| {
                       g.db_cluster_parameter_group_name
                           .unwrap_or_else(||
DB_CLUSTER_PARAMETER_GROUP_NAME.to_string())
                   })
                   .as_deref()
                   .expect("cluster parameter group name"),
           )
           .await;
       if let Err(error) = delete_db_cluster_parameter_group {
           clean_up_errors.push(ScenarioError::new(
               "Failed to delete the db cluster parameter group",
               &error,
           ))
       }
```

```
        if clean_up_errors.is_empty() {
            Ok(())
        } else {
            Err(clean_up_errors)
        }
    }

    pub async fn delete_db_instance(
        &self,
        instance_identifier: &str,
    ) -> Result<DeleteDbInstanceOutput, SdkError<DeleteDBInstanceError>> {
        self.inner
            .delete_db_instance()
            .db_instance_identifier(instance_identifier)
            .skip_final_snapshot(true)
            .send()
            .await
    }

#[tokio::test]
async fn test_scenario_clean_up() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| Ok(DescribeDbInstancesOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| Ok(DescribeDbClustersOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_scenario_clean_up_errors() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| {
            Err(SdkError::service_error(
                DescribeDBInstancesError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db instances error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(), 
 SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| {
            Err(SdkError::service_error(
                DescribeDBClustersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db clusters error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(), 
 SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_| 
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));
```

```
    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_err());
        let errs = clean_up.unwrap_err();
        assert_eq!(errs.len(), 2);
        assert_matches!(errs.get(0), Some(ScenarioError {message, context: _}) if
 message == "Failed to check instance state during deletion");
        assert_matches!(errs.get(1), Some(ScenarioError {message, context: _}) if
 message == "Failed to check cluster state during deletion");
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see DeleteDBInstance in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBClusterParameterGroups` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBClusterParameterGroups`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Get the description of a DB cluster parameter group by name.
    /// </summary>
    /// <param name="name">The name of the DB parameter group to describe.</
param>
    /// <returns>The parameter group description.</returns>
    public async Task<DBClusterParameterGroup?>
 DescribeCustomDBClusterParameterGroupAsync(string name)
    {
        var response = await _amazonRDS.DescribeDBClusterParameterGroupsAsync(
            new DescribeDBClusterParameterGroupsRequest()
            {
                DBClusterParameterGroupName = name
            });
        return response.DBClusterParameterGroups.FirstOrDefault();
    }
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);

        Aws::RDS::Model::DescribeDBClusterParameterGroupsRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);

        Aws::RDS::Model::DescribeDBClusterParameterGroupsOutcome outcome =
                client.DescribeDBClusterParameterGroups(request);

        if (outcome.IsSuccess()) {
            std::cout << "DB cluster parameter group named '" <<
                      CLUSTER_PARAMETER_GROUP_NAME << "' already exists." <<
 std::endl;
            dbParameterGroupFamily =
 outcome.GetResult().GetDBClusterParameterGroups()
[0].GetDBParameterGroupFamily();
        }

        else {
            std::cerr << "Error with Aurora::DescribeDBClusterParameterGroups. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetParameterGroup gets a DB cluster parameter group by name.
func (clusters *DbClusters) GetParameterGroup(parameterGroupName string) (
 *types.DBClusterParameterGroup, error) {
 output, err := clusters.AuroraClient.DescribeDBClusterParameterGroups(
  context.TODO(), &rds.DescribeDBClusterParameterGroupsInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
  })
 if err != nil {
  var notFoundError *types.DBParameterGroupNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("Parameter group %v does not exist.\n", parameterGroupName)
   err = nil
  } else {
   log.Printf("Error getting parameter group %v: %v\n", parameterGroupName, err)
  }
  return nil, err
 } else {
  return &output.DBClusterParameterGroups[0], err
 }
}
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void describeDbClusterParameterGroups(RdsClient rdsClient,
String dbClusterGroupName) {
       try {
           DescribeDbClusterParameterGroupsRequest groupsRequest =
DescribeDbClusterParameterGroupsRequest.builder()
                   .dbClusterParameterGroupName(dbClusterGroupName)
                   .maxRecords(20)
                   .build();

           List<DBClusterParameterGroup> groups =
rdsClient.describeDBClusterParameterGroups(groupsRequest)
                   .dbClusterParameterGroups();
           for (DBClusterParameterGroup group : groups) {
               System.out.println("The group name is " +
group.dbClusterParameterGroupName());
               System.out.println("The group ARN is " +
group.dbClusterParameterGroupArn());
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun describeDbClusterParameterGroups(dbClusterGroupName: String?) {
    val groupsRequest =
        DescribeDbClusterParameterGroupsRequest {
            dbClusterParameterGroupName = dbClusterGroupName
            maxRecords = 20
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.describeDbClusterParameterGroups(groupsRequest)
        response.dbClusterParameterGroups?.forEach { group ->
            println("The group name is ${group.dbClusterParameterGroupName}")
            println("The group ARN is ${group.dbClusterParameterGroupArn}")
        }
    }
}
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_parameter_group(self, parameter_group_name):
        """
        Gets a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to retrieve.
        :return: The requested parameter group.
        """
        try:
            response = self.rds_client.describe_db_cluster_parameter_groups(
                DBClusterParameterGroupName=parameter_group_name
            )
            parameter_group = response["DBClusterParameterGroups"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBParameterGroupNotFound":
                logger.info("Parameter group %s does not exist.",
 parameter_group_name)
            else:
                logger.error(
                    "Couldn't get parameter group %s. Here's why: %s: %s",
                    parameter_group_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return parameter_group
```

- For API details, see DescribeDBClusterParameterGroups in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBClusterParameters` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBClusterParameters`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Describe the cluster parameters in a parameter group.
    /// </summary>
    /// <param name="groupName">The name of the parameter group.</param>
    /// <param name="source">The optional name of the source filter.</param>
    /// <returns>The collection of parameters.</returns>
    public async Task<List<Parameter>>
 DescribeDBClusterParametersInGroupAsync(string groupName, string? source = null)
    {
        var paramList = new List<Parameter>();

        DescribeDBClusterParametersResponse response;
        var request = new DescribeDBClusterParametersRequest
        {
            DBClusterParameterGroupName = groupName,
            Source = source,
        };

        // Get the full list if there are multiple pages.
        do
        {
            response = await
_amazonRDS.DescribeDBClusterParametersAsync(request);
            paramList.AddRange(response.Parameters);

            request.Marker = response.Marker;
        }
        while (response.Marker is not null);

        return paramList;
    }
```

- For API details, see DescribeDBClusterParameters in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);
```

```
//! Routine which gets DB parameters using the 'DescribeDBClusterParameters' api.
/*!
 \sa getDBCLusterParameters()
 \param parameterGroupName: The name of the cluster parameter group.
 \param namePrefix: Prefix string to filter results by parameter name.
 \param source: A source such as 'user', ignored if empty.
 \param parametersResult: Vector of 'Parameter' objects returned by the routine.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::getDBCLusterParameters(const Aws::String
 &parameterGroupName,
                                            const Aws::String &namePrefix,
                                            const Aws::String &source,

 Aws::Vector<Aws::RDS::Model::Parameter> &parametersResult,
                                            const Aws::RDS::RDSClient &client) {
    Aws::String marker; // The marker is used for pagination.
    do {
        Aws::RDS::Model::DescribeDBClusterParametersRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        if (!marker.empty()) {
            request.SetMarker(marker);
        }
        if (!source.empty()) {
            request.SetSource(source);
        }

        Aws::RDS::Model::DescribeDBClusterParametersOutcome outcome =
                client.DescribeDBClusterParameters(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::Parameter> &parameters =
                    outcome.GetResult().GetParameters();
            for (const Aws::RDS::Model::Parameter &parameter: parameters) {
                if (!namePrefix.empty()) {
                    if (parameter.GetParameterName().find(namePrefix) == 0) {
                        parametersResult.push_back(parameter);
                    }
                }
                else {
                    parametersResult.push_back(parameter);
                }
            }

            marker = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeDBClusterParameters. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!marker.empty());

    return true;
}
```

- For API details, see [DescribeDBClusterParameters]() in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository]().

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetParameters gets the parameters that are contained in a DB cluster parameter
 group.
func (clusters *DbClusters) GetParameters(parameterGroupName string, source
 string) (
 []types.Parameter, error) {

 var output *rds.DescribeDBClusterParametersOutput
 var params []types.Parameter
 var err error
 parameterPaginator :=
 rds.NewDescribeDBClusterParametersPaginator(clusters.AuroraClient,
  &rds.DescribeDBClusterParametersInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   Source:                      aws.String(source),
  })
 for parameterPaginator.HasMorePages() {
  output, err = parameterPaginator.NextPage(context.TODO())
  if err != nil {
   log.Printf("Couldn't get paramaeters for %v: %v\n", parameterGroupName, err)
   break
  } else {
   params = append(params, output.Parameters...)
  }
 }
 return params, err
}
```

- For API details, see DescribeDBClusterParameters in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void describeDbClusterParameters(RdsClient rdsClient, String
dbCLusterGroupName, int flag) {
       try {
           DescribeDbClusterParametersRequest dbParameterGroupsRequest;
           if (flag == 0) {
               dbParameterGroupsRequest =
DescribeDbClusterParametersRequest.builder()
                       .dbClusterParameterGroupName(dbCLusterGroupName)
                       .build();
           } else {
               dbParameterGroupsRequest =
DescribeDbClusterParametersRequest.builder()
                       .dbClusterParameterGroupName(dbCLusterGroupName)
                       .source("user")
                       .build();
           }

           DescribeDbClusterParametersResponse response = rdsClient
                   .describeDBClusterParameters(dbParameterGroupsRequest);
           List<Parameter> dbParameters = response.parameters();
           String paraName;
           for (Parameter para : dbParameters) {
               // Only print out information about either auto_increment_offset
or
               // auto_increment_increment.
               paraName = para.parameterName();
               if ((paraName.compareTo("auto_increment_offset") == 0)
                       || (paraName.compareTo("auto_increment_increment ") ==
0)) {
                   System.out.println("*** The parameter name is  " + paraName);
                   System.out.println("*** The parameter value is  " +
para.parameterValue());
                   System.out.println("*** The parameter data type is " +
para.dataType());
                   System.out.println("*** The parameter description is " +
para.description());
                   System.out.println("*** The parameter allowed values  is " +
para.allowedValues());
               }
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see DescribeDBClusterParameters in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/kotlin/services/rds#code-examples).

```
suspend fun describeDbClusterParameters(
    dbCLusterGroupName: String?,
    flag: Int,
) {
    val dbParameterGroupsRequest: DescribeDbClusterParametersRequest
    dbParameterGroupsRequest =
        if (flag == 0) {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
            }
        } else {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
                source = "user"
            }
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response =
 rdsClient.describeDbClusterParameters(dbParameterGroupsRequest)
        response.parameters?.forEach { para ->
            // Only print out information about either auto_increment_offset or
 auto_increment_increment.
            val paraName = para.parameterName
            if (paraName != null) {
                if (paraName.compareTo("auto_increment_offset") == 0 ||
 paraName.compareTo("auto_increment_increment ") == 0) {
                    println("*** The parameter name is  $paraName")
                    println("*** The parameter value is  ${para.parameterValue}")
                    println("*** The parameter data type is ${para.dataType}")
                    println("*** The parameter description is
 ${para.description}")
```

```
                    println("*** The parameter allowed values  is
 ${para.allowedValues}")
                }
            }
        }
    }
}
```

- For API details, see [DescribeDBClusterParameters]() in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository]().

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_parameters(self, parameter_group_name, name_prefix="", source=None):
        """
        Gets the parameters that are contained in a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to query.
        :param name_prefix: When specified, the retrieved list of parameters is
filtered
                            to contain only parameters that start with this
prefix.
        :param source: When specified, only parameters from this source are
retrieved.
                       For example, a source of 'user' retrieves only parameters
that
                       were set by a user.
        :return: The list of requested parameters.
        """
        try:
            kwargs = {"DBClusterParameterGroupName": parameter_group_name}
            if source is not None:
                kwargs["Source"] = source
            parameters = []
            paginator =
self.rds_client.get_paginator("describe_db_cluster_parameters")
            for page in paginator.paginate(**kwargs):
                parameters += [
                    p
                    for p in page["Parameters"]
                    if p["ParameterName"].startswith(name_prefix)
                ]
        except ClientError as err:
            logger.error(
                "Couldn't get parameters for %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return parameters
```

- For API details, see DescribeDBClusterParameters in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    // Get the parameter group. rds.DescribeDbClusterParameterGroups
    // Get parameters in the group. This is a long list so you will have to 
paginate. Find the auto_increment_offset and auto_increment_increment parameters 
(by ParameterName). rds.DescribeDbClusterParameters
    // Parse the ParameterName, Description, and AllowedValues values and display 
them.
    pub async fn cluster_parameters(&self) -> 
Result<Vec<AuroraScenarioParameter>, ScenarioError> {
        let parameters_output = self
            .rds
            .describe_db_cluster_parameters(DB_CLUSTER_PARAMETER_GROUP_NAME)
            .await;

        if let Err(err) = parameters_output {
            return Err(ScenarioError::new(
                format!("Failed to retrieve parameters for 
{DB_CLUSTER_PARAMETER_GROUP_NAME}"),
                &err,
            ));
        }

        let parameters = parameters_output
            .unwrap()
            .into_iter()
            .flat_map(|p| p.parameters.unwrap_or_default().into_iter())
            .filter(|p| 
FILTER_PARAMETER_NAMES.contains(p.parameter_name().unwrap_or_default()))
            .map(AuroraScenarioParameter::from)
            .collect::<Vec<_>>();

        Ok(parameters)
    }
    pub async fn describe_db_cluster_parameters(
        &self,
        name: &str,
    ) -> Result<Vec<DescribeDbClusterParametersOutput>,
 SdkError<DescribeDBClusterParametersError>>
    {
        self.inner
            .describe_db_cluster_parameters()
            .db_cluster_parameter_group_name(name)
            .into_paginator()
            .send()
            .try_collect()
            .await
    }

#[tokio::test]
async fn test_scenario_cluster_parameters() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_cluster_parameters()
        .with(eq("RustSDKCodeExamplesDBParameterGroup"))
        .return_once(|_| {
            Ok(vec![DescribeDbClusterParametersOutput::builder()
                .parameters(Parameter::builder().parameter_name("a").build())
                .parameters(Parameter::builder().parameter_name("b").build())
                .parameters(
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .build(),
                )
                .parameters(Parameter::builder().parameter_name("c").build())
                .parameters(
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .build(),
                )
                .parameters(Parameter::builder().parameter_name("d").build())
                .build()])
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let params = scenario.cluster_parameters().await.expect("cluster params");
    let names: Vec<String> = params.into_iter().map(|p| p.name).collect();
    assert_eq!(
        names,
        vec!["auto_increment_offset", "auto_increment_increment"]
    );
}

#[tokio::test]
async fn test_scenario_cluster_parameters_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_cluster_parameters()
        .with(eq("RustSDKCodeExamplesDBParameterGroup"))
        .return_once(|_| {
            Err(SdkError::service_error(
                DescribeDBClusterParametersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_db_cluster_parameters_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let params = scenario.cluster_parameters().await;
    assert_matches!(params, Err(ScenarioError { message, context: _ }) if message
 == "Failed to retrieve parameters for RustSDKCodeExamplesDBParameterGroup");
}
```

- For API details, see DescribeDBClusterParameters in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBClusterSnapshots` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBClusterSnapshots`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /// <summary>
   /// Return a list of DB snapshots for a particular DB cluster.
   /// </summary>
   /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
   /// <returns>List of DB snapshots.</returns>
   public async Task<List<DBClusterSnapshot>>
DescribeDBClusterSnapshotsByIdentifierAsync(string dbClusterIdentifier)
   {
       var results = new List<DBClusterSnapshot>();

       DescribeDBClusterSnapshotsResponse response;
       DescribeDBClusterSnapshotsRequest request = new
DescribeDBClusterSnapshotsRequest
       {
           DBClusterIdentifier = dbClusterIdentifier
       };
       // Get the full list if there are multiple pages.
       do
       {
           response = await _amazonRDS.DescribeDBClusterSnapshotsAsync(request);
           results.AddRange(response.DBClusterSnapshots);
           request.Marker = response.Marker;
       }
       while (response.Marker is not null);
       return results;
   }
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region (overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::RDS::RDSClient client(clientConfig);

           Aws::RDS::Model::DescribeDBClusterSnapshotsRequest request;
           request.SetDBClusterSnapshotIdentifier(snapshotID);

           Aws::RDS::Model::DescribeDBClusterSnapshotsOutcome outcome =
                   client.DescribeDBClusterSnapshots(request);

           if (outcome.IsSuccess()) {
               snapshot = outcome.GetResult().GetDBClusterSnapshots()[0];
           }
           else {
               std::cerr << "Error with Aurora::DescribeDBClusterSnapshots. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
               return false;
           }
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetClusterSnapshot gets a DB cluster snapshot.
func (clusters *DbClusters) GetClusterSnapshot(snapshotName string)
 (*types.DBClusterSnapshot, error) {
 output, err := clusters.AuroraClient.DescribeDBClusterSnapshots(context.TODO(),
  &rds.DescribeDBClusterSnapshotsInput{
   DBClusterSnapshotIdentifier: aws.String(snapshotName),
  })
 if err != nil {
  log.Printf("Couldn't get snapshot %v: %v\n", snapshotName, err)
  return nil, err
 } else {
  return &output.DBClusterSnapshots[0], nil
 }
}
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void waitForSnapshotReady(RdsClient rdsClient, String 
dbSnapshotIdentifier,
           String dbInstanceClusterIdentifier) {
       try {
           boolean snapshotReady = false;
           String snapshotReadyStr;
           System.out.println("Waiting for the snapshot to become available.");

           DescribeDbClusterSnapshotsRequest snapshotsRequest = 
DescribeDbClusterSnapshotsRequest.builder()
                   .dbClusterSnapshotIdentifier(dbSnapshotIdentifier)
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .build();

           while (!snapshotReady) {
               DescribeDbClusterSnapshotsResponse response = 
rdsClient.describeDBClusterSnapshots(snapshotsRequest);
               List<DBClusterSnapshot> snapshotList = 
response.dbClusterSnapshots();
               for (DBClusterSnapshot snapshot : snapshotList) {
                   snapshotReadyStr = snapshot.status();
                   if (snapshotReadyStr.contains("available")) {
                       snapshotReady = true;
                   } else {
                       System.out.println(".");
                       Thread.sleep(sleepTime * 5000);
                   }
               }
           }

           System.out.println("The Snapshot is available!");
```

```
        } catch (RdsException | InterruptedException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun waitSnapshotReady(
    dbSnapshotIdentifier: String?,
    dbInstanceClusterIdentifier: String?,
) {
    var snapshotReady = false
    var snapshotReadyStr: String
    println("Waiting for the snapshot to become available.")

    val snapshotsRequest =
        DescribeDbClusterSnapshotsRequest {
            dbClusterSnapshotIdentifier = dbSnapshotIdentifier
            dbClusterIdentifier = dbInstanceClusterIdentifier
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        while (!snapshotReady) {
            val response = rdsClient.describeDbClusterSnapshots(snapshotsRequest)
            val snapshotList = response.dbClusterSnapshots
            if (snapshotList != null) {
                for (snapshot in snapshotList) {
                    snapshotReadyStr = snapshot.status.toString()
                    if (snapshotReadyStr.contains("available")) {
                        snapshotReady = true
                    } else {
                        println(".")
                        delay(slTime * 5000)
                    }
                }
            }
        }
    }
    println("The Snapshot is available!")
}
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)
    def get_cluster_snapshot(self, snapshot_id):
        """
        Gets a DB cluster snapshot.

        :param snapshot_id: The ID of the snapshot to retrieve.
        :return: The retrieved snapshot.
        """
        try:
            response = self.rds_client.describe_db_cluster_snapshots(
                DBClusterSnapshotIdentifier=snapshot_id
            )
            snapshot = response["DBClusterSnapshots"][0]
        except ClientError as err:
            logger.error(
                "Couldn't get DB cluster snapshot %s. Here's why: %s: %s",
                snapshot_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return snapshot
```

- For API details, see DescribeDBClusterSnapshots in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBClusters` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBClusters`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /// <summary>
   /// Returns a list of DB clusters.
   /// </summary>
   /// <param name="dbInstanceIdentifier">Optional name of a specific DB 
cluster.</param>
   /// <returns>List of DB clusters.</returns>
   public async Task<List<DBCluster>> DescribeDBClustersPagedAsync(string? 
dbClusterIdentifier = null)
   {
       var results = new List<DBCluster>();

       DescribeDBClustersResponse response;
       DescribeDBClustersRequest request = new DescribeDBClustersRequest
       {
           DBClusterIdentifier = dbClusterIdentifier
       };
       // Get the full list if there are multiple pages.
       do
       {
           response = await _amazonRDS.DescribeDBClustersAsync(request);
           results.AddRange(response.DBClusters);
           request.Marker = response.Marker;
       }
       while (response.Marker is not null);
       return results;
   }
```

- For API details, see DescribeDBClusters in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);

//! Routine which gets a DB cluster description.
/*!
 \sa describeDBCluster()
 \param dbClusterIdentifier: A DB cluster identifier.
 \param clusterResult: The 'DBCluster' object containing the description.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::describeDBCluster(const Aws::String &dbClusterIdentifier,
                                       Aws::RDS::Model::DBCluster &clusterResult,
                                       const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBClustersRequest request;
    request.SetDBClusterIdentifier(dbClusterIdentifier);

    Aws::RDS::Model::DescribeDBClustersOutcome outcome =
            client.DescribeDBClusters(request);

    bool result = true;
    if (outcome.IsSuccess()) {
        clusterResult = outcome.GetResult().GetDBClusters()[0];
    }
    else if (outcome.GetError().GetErrorType() !=
             Aws::RDS::RDSErrors::D_B_CLUSTER_NOT_FOUND_FAULT) {
        result = false;
        std::cerr << "Error with Aurora::GDescribeDBClusters. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
        // This example does not log an error if the DB cluster does not exist.
        // Instead, clusterResult is set to empty.
    else {
        clusterResult = Aws::RDS::Model::DBCluster();
    }

    return result;

}
```

- For API details, see DescribeDBClusters in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetDbCluster gets data about an Aurora DB cluster.
func (clusters *DbClusters) GetDbCluster(clusterName string) (*types.DBCluster,
 error) {
 output, err := clusters.AuroraClient.DescribeDBClusters(context.TODO(),
  &rds.DescribeDBClustersInput{
   DBClusterIdentifier: aws.String(clusterName),
  })
 if err != nil {
  var notFoundError *types.DBClusterNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("DB cluster %v does not exist.\n", clusterName)
   err = nil
  } else {
   log.Printf("Couldn't get DB cluster %v: %v\n", clusterName, err)
  }
  return nil, err
 } else {
  return &output.DBClusters[0], err
 }
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static void describeDbClusterParameters(RdsClient rdsClient, String
 dbCLusterGroupName, int flag) {
        try {
            DescribeDbClusterParametersRequest dbParameterGroupsRequest;
            if (flag == 0) {
                dbParameterGroupsRequest =
 DescribeDbClusterParametersRequest.builder()
                        .dbClusterParameterGroupName(dbCLusterGroupName)
                        .build();
            } else {
                dbParameterGroupsRequest =
 DescribeDbClusterParametersRequest.builder()
                        .dbClusterParameterGroupName(dbCLusterGroupName)
                        .source("user")
                        .build();
            }
```

```
           DescribeDbClusterParametersResponse response = rdsClient
                   .describeDBClusterParameters(dbParameterGroupsRequest);
           List<Parameter> dbParameters = response.parameters();
           String paraName;
           for (Parameter para : dbParameters) {
               // Only print out information about either auto_increment_offset
or
               // auto_increment_increment.
               paraName = para.parameterName();
               if ((paraName.compareTo("auto_increment_offset") == 0)
                       || (paraName.compareTo("auto_increment_increment ") ==
0)) {
                   System.out.println("*** The parameter name is  " + paraName);
                   System.out.println("*** The parameter value is  " +
para.parameterValue());
                   System.out.println("*** The parameter data type is " +
para.dataType());
                   System.out.println("*** The parameter description is " +
para.description());
                   System.out.println("*** The parameter allowed values  is " +
para.allowedValues());
               }
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }
```

- For API details, see DescribeDBClusters in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun describeDbClusterParameters(
    dbCLusterGroupName: String?,
    flag: Int,
) {
    val dbParameterGroupsRequest: DescribeDbClusterParametersRequest
    dbParameterGroupsRequest =
        if (flag == 0) {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
            }
        } else {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
                source = "user"
            }
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response =
 rdsClient.describeDbClusterParameters(dbParameterGroupsRequest)
        response.parameters?.forEach { para ->
            // Only print out information about either auto_increment_offset or
 auto_increment_increment.
            val paraName = para.parameterName
            if (paraName != null) {
                if (paraName.compareTo("auto_increment_offset") == 0 ||
 paraName.compareTo("auto_increment_increment ") == 0) {
                    println("*** The parameter name is  $paraName")
                    println("*** The parameter value is  ${para.parameterValue}")
                    println("*** The parameter data type is ${para.dataType}")
                    println("*** The parameter description is
 ${para.description}")
                    println("*** The parameter allowed values  is
 ${para.allowedValues}")
                }
            }
        }
    }
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_db_cluster(self, cluster_name):
        """
        Gets data about an Aurora DB cluster.

        :param cluster_name: The name of the DB cluster to retrieve.
        :return: The retrieved DB cluster.
        """
        try:
            response = self.rds_client.describe_db_clusters(
                DBClusterIdentifier=cluster_name
            )
            cluster = response["DBClusters"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBClusterNotFoundFault":
                logger.info("Cluster %s does not exist.", cluster_name)
            else:
                logger.error(
                    "Couldn't verify the existence of DB cluster %s. Here's why: 
%s: %s",
                    cluster_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return cluster
```

- For API details, see DescribeDBClusters in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    // Get a list of allowed engine versions. 
rds.DescribeDbEngineVersions(Engine='aurora-mysql', DBParameterGroupFamily=<the 
family used to create your parameter group in step 2>)
    // Create an Aurora DB cluster database cluster that contains a MySql 
database and uses the parameter group you created.
    // Wait for DB cluster to be ready. Call rds.DescribeDBClusters and check for 
Status == 'available'.
    // Get a list of instance classes available for the selected engine 
and engine version. rds.DescribeOrderableDbInstanceOptions(Engine='mysql', 
EngineVersion=).

    // Create a database instance in the cluster.
    // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check 
for DBInstanceStatus == 'available'.
```

```
   pub async fn start_cluster_and_instance(&mut self) -> Result<(),
ScenarioError> {
       if self.password.is_none() {
           return Err(ScenarioError::with(
               "Must set Secret Password before starting a cluster",
           ));
       }
       let create_db_cluster = self
           .rds
           .create_db_cluster(
               DB_CLUSTER_IDENTIFIER,
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_ENGINE,
               self.engine_version.as_deref().expect("engine version"),
               self.username.as_deref().expect("username"),
               self.password
                   .replace(SecretString::new("".to_string()))
                   .expect("password"),
           )
           .await;
       if let Err(err) = create_db_cluster {
           return Err(ScenarioError::new(
               "Failed to create DB Cluster with cluster group",
               &err,
           ));
       }

       self.db_cluster_identifier = create_db_cluster
           .unwrap()
           .db_cluster
           .and_then(|c| c.db_cluster_identifier);

       if self.db_cluster_identifier.is_none() {
           return Err(ScenarioError::with("Created DB Cluster missing
Identifier"));
       }

       info!(
           "Started a db cluster: {}",
           self.db_cluster_identifier
               .as_deref()
               .unwrap_or("Missing ARN")
       );
```

```
        let create_db_instance = self
            .rds
            .create_db_instance(
                self.db_cluster_identifier.as_deref().expect("cluster name"),
                DB_INSTANCE_IDENTIFIER,
                self.instance_class.as_deref().expect("instance class"),
                DB_ENGINE,
            )
            .await;
        if let Err(err) = create_db_instance {
            return Err(ScenarioError::new(
                "Failed to create Instance in DB Cluster",
                &err,
            ));
        }

        self.db_instance_identifier = create_db_instance
            .unwrap()
            .db_instance
            .and_then(|i| i.db_instance_identifier);

        // Cluster creation can take up to 20 minutes to become available
        let cluster_max_wait = Duration::from_secs(20 * 60);
        let waiter = Waiter::builder().max(cluster_max_wait).build();
        while waiter.sleep().await.is_ok() {
            let cluster = self
                .rds
                .describe_db_clusters(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = cluster {
                warn!(?err, "Failed to describe cluster while waiting for
 ready");
                continue;
            }

            let instance = self
                .rds
                .describe_db_instance(
                    self.db_instance_identifier
```

```
                        .as_deref()
                        .expect("instance identifier"),
                )
                .await;
            if let Err(err) = instance {
                return Err(ScenarioError::new(
                    "Failed to find instance for cluster",
                    &err,
                ));
            }

            let instances_available = instance
                .unwrap()
                .db_instances()
                .iter()
                .all(|instance| instance.db_instance_status() ==
Some("Available"));

            let endpoints = self
                .rds
                .describe_db_cluster_endpoints(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = endpoints {
                return Err(ScenarioError::new(
                    "Failed to find endpoint for cluster",
                    &err,
                ));
            }

            let endpoints_available = endpoints
                .unwrap()
                .db_cluster_endpoints()
                .iter()
                .all(|endpoint| endpoint.status() == Some("available"));

            if instances_available && endpoints_available {
                return Ok(());
            }
        }
        Err(ScenarioError::with("timed out waiting for cluster"))
    }

    pub async fn describe_db_clusters(
        &self,
        id: &str,
    ) -> Result<DescribeDbClustersOutput, SdkError<DescribeDBClustersError>> {
        self.inner
            .describe_db_clusters()
            .db_cluster_identifier(id)
            .send()
            .await
    }

#[tokio::test]
async fn test_start_cluster_and_instance() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|id| {
            Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_describe_db_instance()
        .with(eq("RustSDKCodeExamplesDBInstance"))
        .return_once(|name| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_instance_identifier(name)
                        .db_instance_status("Available")
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()
 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
        assert!(scenario
            .password
            .replace(SecretString::new("BAD SECRET".into()))
            .unwrap()
            .expose_secret()
            .is_empty());
        assert_eq!(
            scenario.db_cluster_identifier,
            Some("RustSDKCodeExamplesDBCluster".into())
        );
    });
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Err(SdkError::service_error(
                CreateDBClusterError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _}) if message
 == "Failed to create DB Cluster with cluster group")
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_missing_id() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
                .db_cluster(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context:_ }) if message
 == "Created DB Cluster missing Identifier");
}

#[tokio::test]
async fn test_start_cluster_and_instance_instance_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
```

```
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .return_once(|_, _, _, _| {
            Err(SdkError::service_error(
                CreateDBInstanceError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db instance error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _ }) if message
 == "Failed to create Instance in DB Cluster")
}

#[tokio::test]
async fn test_start_cluster_and_instance_wait_hiccup() {
    let mut mock_rds = MockRdsImpl::default();
    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .times(1)
```

```
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe cluster error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       })
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .times(1)
       .returning(|id| {
           Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
               .build())
       });

   mock_rds.expect_describe_db_instance().return_once(|name| {
       Ok(DescribeDbInstancesOutput::builder()
           .db_instances(
               DbInstance::builder()
                   .db_instance_identifier(name)
                   .db_instance_status("Available")
                   .build(),
           )
           .build())
   });

   mock_rds
       .expect_describe_db_cluster_endpoints()
       .return_once(|_| {
           Ok(DescribeDbClusterEndpointsOutput::builder()

.db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
               .build())
       });

   let mut scenario = AuroraScenario::new(mock_rds);
   scenario.engine_version = Some("aurora-mysql8.0".into());
   scenario.instance_class = Some("m5.large".into());
   scenario.username = Some("test username".into());
   scenario.password = Some(SecretString::new("test password".into()));
```

```
    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}
```

- For API details, see DescribeDBClusters in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBEngineVersions` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBEngineVersions`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Get a list of DB engine versions for a particular DB engine.
    /// </summary>
    /// <param name="engine">The name of the engine.</param>
    /// <param name="parameterGroupFamily">Optional parameter group family
 name.</param>
    /// <returns>A list of DBEngineVersions.</returns>
    public async Task<List<DBEngineVersion>>
 DescribeDBEngineVersionsForEngineAsync(string engine,
        string? parameterGroupFamily = null)
    {
        var response = await _amazonRDS.DescribeDBEngineVersionsAsync(
            new DescribeDBEngineVersionsRequest()
            {
                Engine = engine,
                DBParameterGroupFamily = parameterGroupFamily
            });
        return response.DBEngineVersions;
    }
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);


//! Routine which gets available DB engine versions for an engine name and
//! an optional parameter group family.
/*!
```

```
 \sa getDBEngineVersions()
 \param engineName: A DB engine name.
 \param parameterGroupFamily: A parameter group family name, ignored if empty.
 \param engineVersionsResult: Vector of 'DBEngineVersion' objects returned by the
 routine.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::getDBEngineVersions(const Aws::String &engineName,
                                         const Aws::String &parameterGroupFamily,

 Aws::Vector<Aws::RDS::Model::DBEngineVersion> &engineVersionsResult,
                                         const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBEngineVersionsRequest request;
    request.SetEngine(engineName);
    if (!parameterGroupFamily.empty()) {
        request.SetDBParameterGroupFamily(parameterGroupFamily);
    }

    engineVersionsResult.clear();
    Aws::String marker; // The marker is used for pagination.
    do {
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::RDS::Model::DescribeDBEngineVersionsOutcome outcome =
                client.DescribeDBEngineVersions(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::DBEngineVersion> &engineVersions =
                    outcome.GetResult().GetDBEngineVersions();

            engineVersionsResult.insert(engineVersionsResult.end(),
                                        engineVersions.begin(),
 engineVersions.end());
            marker  = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeDBEngineVersionsRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());
    return true;
}
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetEngineVersions gets database engine versions that are available for the
 specified engine
// and parameter group family.
func (clusters *DbClusters) GetEngineVersions(engine string, parameterGroupFamily
 string) (
 []types.DBEngineVersion, error) {
 output, err := clusters.AuroraClient.DescribeDBEngineVersions(context.TODO(),
  &rds.DescribeDBEngineVersionsInput{
   Engine:                 aws.String(engine),
   DBParameterGroupFamily: aws.String(parameterGroupFamily),
  })
 if err != nil {
  log.Printf("Couldn't get engine versions for %v: %v\n", engine, err)
  return nil, err
 } else {
  return output.DBEngineVersions, nil
 }
}
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   public static void describeDBEngines(RdsClient rdsClient) {
       try {
           DescribeDbEngineVersionsRequest engineVersionsRequest =
DescribeDbEngineVersionsRequest.builder()
                   .engine("aurora-mysql")
                   .defaultOnly(true)
                   .maxRecords(20)
                   .build();

           DescribeDbEngineVersionsResponse response =
rdsClient.describeDBEngineVersions(engineVersionsRequest);
           List<DBEngineVersion> engines = response.dbEngineVersions();

           // Get all DBEngineVersion objects.
           for (DBEngineVersion engineOb : engines) {
               System.out.println("The name of the DB parameter group family for
the database engine is "
                       + engineOb.dbParameterGroupFamily());
               System.out.println("The name of the database engine " +
engineOb.engine());
               System.out.println("The version number of the database engine " +
engineOb.engineVersion());
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
```

```
        }
    }
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Get a list of allowed engine versions.
suspend fun getAllowedClusterEngines(dbParameterGroupFamilyVal: String?) {
    val versionsRequest =
        DescribeDbEngineVersionsRequest {
            dbParameterGroupFamily = dbParameterGroupFamilyVal
            engine = "aurora-mysql"
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.describeDbEngineVersions(versionsRequest)
        response.dbEngineVersions?.forEach { dbEngine ->
            println("The engine version is ${dbEngine.engineVersion}")
            println("The engine description is ${dbEngine.dbEngineDescription}")
        }
    }
}
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_engine_versions(self, engine, parameter_group_family=None):
        """
        Gets database engine versions that are available for the specified engine
        and parameter group family.

        :param engine: The database engine to look up.
        :param parameter_group_family: When specified, restricts the returned
 list of
                                       engine versions to those that are
 compatible with
                                       this parameter group family.
        :return: The list of database engine versions.
        """
        try:
            kwargs = {"Engine": engine}
            if parameter_group_family is not None:
                kwargs["DBParameterGroupFamily"] = parameter_group_family
            response = self.rds_client.describe_db_engine_versions(**kwargs)
            versions = response["DBEngineVersions"]
        except ClientError as err:
            logger.error(
                "Couldn't get engine versions for %s. Here's why: %s: %s",
                engine,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return versions
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    // Get available engine families for Aurora MySql.
 rds.DescribeDbEngineVersions(Engine='aurora-mysql') and build a set of the
 'DBParameterGroupFamily' field values. I get {aurora-mysql8.0, aurora-mysql5.7}.
    pub async fn get_engines(&self) -> Result<HashMap<String, Vec<String>>,
 ScenarioError> {
        let describe_db_engine_versions =
 self.rds.describe_db_engine_versions(DB_ENGINE).await;
        trace!(versions=?describe_db_engine_versions, "full list of versions");
        if let Err(err) = describe_db_engine_versions {
            return Err(ScenarioError::new(
                "Failed to retrieve DB Engine Versions",
                &err,
            ));
        };

        let version_count = describe_db_engine_versions
            .as_ref()
            .map(|o| o.db_engine_versions().len())
            .unwrap_or_default();
        info!(version_count, "got list of versions");

        // Create a map of engine families to their available versions.
        let mut versions = HashMap::<String, Vec<String>>::new();
        describe_db_engine_versions
            .unwrap()
            .db_engine_versions()
            .iter()
            .filter_map(
                |v| match (&v.db_parameter_group_family, &v.engine_version) {
                    (Some(family), Some(version)) => Some((family.clone(),
version.clone())),
                    _ => None,
                },
            )
            .for_each(|(family, version)|
versions.entry(family).or_default().push(version));

        Ok(versions)
    }

    pub async fn describe_db_engine_versions(
        &self,
        engine: &str,
    ) -> Result<DescribeDbEngineVersionsOutput,
SdkError<DescribeDBEngineVersionsError>> {
        self.inner
            .describe_db_engine_versions()
            .engine(engine)
            .send()
            .await
    }
```

```
#[tokio::test]
async fn test_scenario_get_engines() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_engine_versions()
        .with(eq("aurora-mysql"))
        .return_once(|_| {
            Ok(DescribeDbEngineVersionsOutput::builder()
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f1")
                        .engine_version("f1a")
                        .build(),
                )
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f1")
                        .engine_version("f1b")
                        .build(),
                )
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f2")
                        .engine_version("f2a")
                        .build(),
                )
                .db_engine_versions(DbEngineVersion::builder().build())
                .build())
        });

    let scenario = AuroraScenario::new(mock_rds);

    let versions_map = scenario.get_engines().await;

    assert_eq!(
        versions_map,
        Ok(HashMap::from([
            ("f1".into(), vec!["f1a".into(), "f1b".into()]),
            ("f2".into(), vec!["f2a".into()])
        ]))
    );
}
```

```
#[tokio::test]
async fn test_scenario_get_engines_failed() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_engine_versions()
        .with(eq("aurora-mysql"))
        .return_once(|_| {
            Err(SdkError::service_error(
                DescribeDBEngineVersionsError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_db_engine_versions error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let scenario = AuroraScenario::new(mock_rds);

    let versions_map = scenario.get_engines().await;
    assert_matches!(
        versions_map,
        Err(ScenarioError { message, context: _ }) if message == "Failed to
 retrieve DB Engine Versions"
    );
}
```

- For API details, see DescribeDBEngineVersions in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeDBInstances` with an AWS SDK or CLI

The following code examples show how to use `DescribeDBInstances`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /// <summary>
   /// Returns a list of DB instances.
   /// </summary>
   /// <param name="dbInstanceIdentifier">Optional name of a specific DB 
instance.</param>
   /// <returns>List of DB instances.</returns>
   public async Task<List<DBInstance>> DescribeDBInstancesPagedAsync(string? 
dbInstanceIdentifier = null)
   {
       var results = new List<DBInstance>();
       var instancesPaginator = _amazonRDS.Paginators.DescribeDBInstances(
           new DescribeDBInstancesRequest
           {
               DBInstanceIdentifier = dbInstanceIdentifier
           });
       // Get the entire list using the paginator.
       await foreach (var instances in instancesPaginator.DBInstances)
       {
           results.Add(instances);
       }
       return results;
   }
```

- For API details, see DescribeDBInstances in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);


//! Routine which gets a DB instance description.
/*!
 \sa describeDBCluster()
 \param dbInstanceIdentifier: A DB instance identifier.
 \param instanceResult: The 'DBInstance' object containing the description.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::describeDBInstance(const Aws::String &dbInstanceIdentifier,
                                        Aws::RDS::Model::DBInstance
 &instanceResult,
                                        const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBInstancesRequest request;
    request.SetDBInstanceIdentifier(dbInstanceIdentifier);

    Aws::RDS::Model::DescribeDBInstancesOutcome outcome =
            client.DescribeDBInstances(request);

    bool result = true;
    if (outcome.IsSuccess()) {
        instanceResult = outcome.GetResult().GetDBInstances()[0];
    }
    else if (outcome.GetError().GetErrorType() !=
             Aws::RDS::RDSErrors::D_B_INSTANCE_NOT_FOUND_FAULT) {
        result = false;
        std::cerr << "Error with Aurora::DescribeDBInstances. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
        // This example does not log an error if the DB instance does not exist.
        // Instead, instanceResult is set to empty.
    else {
        instanceResult = Aws::RDS::Model::DBInstance();
    }

    return result;
}
```

- For API details, see DescribeDBInstances in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```

type DbClusters struct {
 AuroraClient *rds.Client
}




// GetInstance gets data about a DB instance.
func (clusters *DbClusters) GetInstance(instanceName string) (
 *types.DBInstance, error) {
 output, err := clusters.AuroraClient.DescribeDBInstances(context.TODO(),
  &rds.DescribeDBInstancesInput{
   DBInstanceIdentifier: aws.String(instanceName),
  })
 if err != nil {
  var notFoundError *types.DBInstanceNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("DB instance %v does not exist.\n", instanceName)
   err = nil
  } else {
   log.Printf("Couldn't get instance %v: %v\n", instanceName, err)
  }
  return nil, err
 } else {
  return &output.DBInstances[0], nil
 }
}
```

- For API details, see DescribeDBInstances in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   // Waits until the database instance is available.
   public static void waitForInstanceReady(RdsClient rdsClient, String
dbClusterIdentifier) {
       boolean instanceReady = false;
       String instanceReadyStr;
       System.out.println("Waiting for instance to become available.");
       try {
           DescribeDbClustersRequest instanceRequest =
DescribeDbClustersRequest.builder()
                   .dbClusterIdentifier(dbClusterIdentifier)
                   .build();

           while (!instanceReady) {
               DescribeDbClustersResponse response =
rdsClient.describeDBClusters(instanceRequest);
               List<DBCluster> clusterList = response.dbClusters();
```

```
                for (DBCluster cluster : clusterList) {
                    instanceReadyStr = cluster.status();
                    if (instanceReadyStr.contains("available")) {
                        instanceReady = true;
                    } else {
                        System.out.print(".");
                        Thread.sleep(sleepTime * 1000);
                    }
                }
            }
            System.out.println("Database cluster is available!");

        } catch (RdsException | InterruptedException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see DescribeDBInstances in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun waitDBAuroraInstanceReady(dbInstanceIdentifierVal: String?) {
    var instanceReady = false
    var instanceReadyStr: String
    println("Waiting for instance to become available.")
    val instanceRequest =
        DescribeDbInstancesRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
        }

    var endpoint = ""
    RdsClient { region = "us-west-2" }.use { rdsClient ->
```

```
        while (!instanceReady) {
            val response = rdsClient.describeDbInstances(instanceRequest)
            response.dbInstances?.forEach { instance ->
                instanceReadyStr = instance.dbInstanceStatus.toString()
                if (instanceReadyStr.contains("available")) {
                    endpoint = instance.endpoint?.address.toString()
                    instanceReady = true
                } else {
                    print(".")
                    delay(sleepTime * 1000)
                }
            }
        }
    }
    println("Database instance is available! The connection endpoint is
 $endpoint")
}
```

- For API details, see DescribeDBInstances in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_db_instance(self, instance_id):
        """
        Gets data about a DB instance.

        :param instance_id: The ID of the DB instance to retrieve.
        :return: The retrieved DB instance.
        """
        try:
            response = self.rds_client.describe_db_instances(
                DBInstanceIdentifier=instance_id
            )
            db_inst = response["DBInstances"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBInstanceNotFound":
                logger.info("Instance %s does not exist.", instance_id)
            else:
                logger.error(
                    "Couldn't get DB instance %s. Here's why: %s: %s",
                    instance_id,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return db_inst
```

- For API details, see DescribeDBInstances in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    pub async fn clean_up(self) -> Result<(), Vec<ScenarioError>> {
        let mut clean_up_errors: Vec<ScenarioError> = vec![];

        // Delete the instance. rds.DeleteDbInstance.
        let delete_db_instance = self
            .rds
            .delete_db_instance(
                self.db_instance_identifier
                    .as_deref()
                    .expect("instance identifier"),
            )
            .await;
        if let Err(err) = delete_db_instance {
            let identifier = self
                .db_instance_identifier
                .as_deref()
                .unwrap_or("Missing Instance Identifier");
            let message = format!("failed to delete db instance {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance to delete
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_instances =
self.rds.describe_db_instances().await;
                if let Err(err) = describe_db_instances {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check instance state during deletion",
                        &err,
                    ));
                    break;
                }
                let db_instances = describe_db_instances
                    .unwrap()
                    .db_instances()
                    .iter()
                    .filter(|instance| instance.db_cluster_identifier ==
self.db_cluster_identifier)
                    .cloned()
                    .collect::<Vec<DbInstance>>();

                if db_instances.is_empty() {
                    trace!("Delete Instance waited and no instances were found");
                    break;
                }
                match db_instances.first().unwrap().db_instance_status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but instances is in
{status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB instance");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster. rds.DeleteDbCluster.
        let delete_db_cluster = self
            .rds
            .delete_db_cluster(
                self.db_cluster_identifier
                    .as_deref()
                    .expect("cluster identifier"),
            )
            .await;

        if let Err(err) = delete_db_cluster {
            let identifier = self
                .db_cluster_identifier
                .as_deref()
                .unwrap_or("Missing DB Cluster Identifier");
            let message = format!("failed to delete db cluster {identifier}");
```

```
           clean_up_errors.push(ScenarioError::new(message, &err));
       } else {
           // Wait for the instance and cluster to fully delete.
rds.DescribeDbInstances and rds.DescribeDbClusters until both are not found.
           let waiter = Waiter::default();
           while waiter.sleep().await.is_ok() {
               let describe_db_clusters = self
                   .rds
                   .describe_db_clusters(
                       self.db_cluster_identifier
                           .as_deref()
                           .expect("cluster identifier"),
                   )
                   .await;
               if let Err(err) = describe_db_clusters {
                   clean_up_errors.push(ScenarioError::new(
                       "Failed to check cluster state during deletion",
                       &err,
                   ));
                   break;
               }
               let describe_db_clusters = describe_db_clusters.unwrap();
               let db_clusters = describe_db_clusters.db_clusters();
               if db_clusters.is_empty() {
                   trace!("Delete cluster waited and no clusters were found");
                   break;
               }
               match db_clusters.first().unwrap().status() {
                   Some("Deleting") => continue,
                   Some(status) => {
                       info!("Attempting to delete but clusters is in
{status}");
                       continue;
                   }
                   None => {
                       warn!("No status for DB cluster");
                       break;
                   }
               }
           }
       }

       // Delete the DB cluster parameter group.
rds.DeleteDbClusterParameterGroup.
```

```
        let delete_db_cluster_parameter_group = self
            .rds
            .delete_db_cluster_parameter_group(
                self.db_cluster_parameter_group
                    .map(|g| {
                        g.db_cluster_parameter_group_name
                            .unwrap_or_else(||
 DB_CLUSTER_PARAMETER_GROUP_NAME.to_string())
                    })
                    .as_deref()
                    .expect("cluster parameter group name"),
            )
            .await;
        if let Err(error) = delete_db_cluster_parameter_group {
            clean_up_errors.push(ScenarioError::new(
                "Failed to delete the db cluster parameter group",
                &error,
            ))
        }

        if clean_up_errors.is_empty() {
            Ok(())
        } else {
            Err(clean_up_errors)
        }
    }

    pub async fn describe_db_instances(
        &self,
    ) -> Result<DescribeDbInstancesOutput, SdkError<DescribeDBInstancesError>> {
        self.inner.describe_db_instances().send().await
    }

#[tokio::test]
async fn test_scenario_clean_up() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| Ok(DescribeDbInstancesOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| Ok(DescribeDbClustersOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));
```

```
    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_scenario_clean_up_errors() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| {
            Err(SdkError::service_error(
                DescribeDBInstancesError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db instances error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| {
            Err(SdkError::service_error(
                DescribeDBClustersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db clusters error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_err());
        let errs = clean_up.unwrap_err();
        assert_eq!(errs.len(), 2);
        assert_matches!(errs.get(0), Some(ScenarioError {message, context: _}) if
 message == "Failed to check instance state during deletion");
        assert_matches!(errs.get(1), Some(ScenarioError {message, context: _}) if
 message == "Failed to check cluster state during deletion");
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
```

```
    let _ = assertions.await;
}
```

- For API details, see DescribeDBInstances in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DescribeOrderableDBInstanceOptions` with an AWS SDK or CLI

The following code examples show how to use `DescribeOrderableDBInstanceOptions`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Get a list of orderable DB instance options for a specific
    /// engine and engine version.
    /// </summary>
    /// <param name="engine">Name of the engine.</param>
    /// <param name="engineVersion">Version of the engine.</param>
    /// <returns>List of OrderableDBInstanceOptions.</returns>
    public async Task<List<OrderableDBInstanceOption>>
 DescribeOrderableDBInstanceOptionsPagedAsync(string engine, string
 engineVersion)
```

```
    {
        // Use a paginator to get a list of DB instance options.
        var results = new List<OrderableDBInstanceOption>();
        var paginateInstanceOptions =
 _amazonRDS.Paginators.DescribeOrderableDBInstanceOptions(
            new DescribeOrderableDBInstanceOptionsRequest()
            {
                Engine = engine,
                EngineVersion = engineVersion,
            });
        // Get the entire list using the paginator.
        await foreach (var instanceOptions in
 paginateInstanceOptions.OrderableDBInstanceOptions)
        {
            results.Add(instanceOptions);
        }
        return results;
    }
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::RDS::RDSClient client(clientConfig);


//! Routine which gets available DB instance classes, displays the list
//! to the user, and returns the user selection.
/*!
 \sa chooseDBInstanceClass()
 \param engineName: The DB engine name.
 \param engineVersion: The DB engine version.
 \param dbInstanceClass: String for DB instance class chosen by the user.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::chooseDBInstanceClass(const Aws::String &engine,
                                           const Aws::String &engineVersion,
                                           Aws::String &dbInstanceClass,
                                           const Aws::RDS::RDSClient &client) {
    std::vector<Aws::String> instanceClasses;
    Aws::String marker; // The marker is used for pagination.
    do {
        Aws::RDS::Model::DescribeOrderableDBInstanceOptionsRequest request;
        request.SetEngine(engine);
        request.SetEngineVersion(engineVersion);
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::RDS::Model::DescribeOrderableDBInstanceOptionsOutcome outcome =
                client.DescribeOrderableDBInstanceOptions(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::OrderableDBInstanceOption>
 &options =
                    outcome.GetResult().GetOrderableDBInstanceOptions();
            for (const Aws::RDS::Model::OrderableDBInstanceOption &option:
 options) {
                const Aws::String &instanceClass = option.GetDBInstanceClass();
                if (std::find(instanceClasses.begin(), instanceClasses.end(),
                              instanceClass) == instanceClasses.end()) {
                    instanceClasses.push_back(instanceClass);
                }
            }
            marker = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeOrderableDBInstanceOptions.
 "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!marker.empty());

    std::cout << "The available DB instance classes for your database engine
 are:"
              << std::endl;
    for (int i = 0; i < instanceClasses.size(); ++i) {
        std::cout << "   " << i + 1 << ": " << instanceClasses[i] << std::endl;
    }

    int choice = askQuestionForIntRange(
            "Which DB instance class do you want to use? ",
            1, static_cast<int>(instanceClasses.size()));
    dbInstanceClass = instanceClasses[choice - 1];
    return true;
}
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}
```

```
// GetOrderableInstances uses a paginator to get DB instance options that can be
 used to create DB instances that are
// compatible with a set of specifications.
func (clusters *DbClusters) GetOrderableInstances(engine string, engineVersion
 string) (
 []types.OrderableDBInstanceOption, error) {

 var output *rds.DescribeOrderableDBInstanceOptionsOutput
 var instances []types.OrderableDBInstanceOption
 var err error
 orderablePaginator :=
 rds.NewDescribeOrderableDBInstanceOptionsPaginator(clusters.AuroraClient,
  &rds.DescribeOrderableDBInstanceOptionsInput{
   Engine:        aws.String(engine),
   EngineVersion: aws.String(engineVersion),
  })
 for orderablePaginator.HasMorePages() {
  output, err = orderablePaginator.NextPage(context.TODO())
  if err != nil {
   log.Printf("Couldn't get orderable DB instances: %v\n", err)
   break
  } else {
   instances = append(instances, output.OrderableDBInstanceOptions...)
  }
 }
 return instances, err
}
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static void describeDBEngines(RdsClient rdsClient) {
        try {
            DescribeDbEngineVersionsRequest engineVersionsRequest =
 DescribeDbEngineVersionsRequest.builder()
                    .engine("aurora-mysql")
                    .defaultOnly(true)
                    .maxRecords(20)
                    .build();

            DescribeDbEngineVersionsResponse response =
 rdsClient.describeDBEngineVersions(engineVersionsRequest);
            List<DBEngineVersion> engines = response.dbEngineVersions();

            // Get all DBEngineVersion objects.
            for (DBEngineVersion engineOb : engines) {
                System.out.println("The name of the DB parameter group family for
 the database engine is "
                        + engineOb.dbParameterGroupFamily());
                System.out.println("The name of the database engine " +
 engineOb.engine());
                System.out.println("The version number of the database engine " +
 engineOb.engineVersion());
            }

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for Java 2.x API Reference*.

PowerShell

Tools for PowerShell

Example 1: This example lists the DB engine versions that support a specific DB instance class in an AWS Region.

```
$params = @{
  Engine = 'aurora-postgresql'
  DBInstanceClass = 'db.r5.large'
  Region = 'us-east-1'
}
Get-RDSOrderableDBInstanceOption @params
```

Example 2: This example lists the DB instance classes that are supported for a specific DB engine version in an AWS Region.

```
$params = @{
  Engine = 'aurora-postgresql'
  EngineVersion = '13.6'
  Region = 'us-east-1'
}
Get-RDSOrderableDBInstanceOption @params
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS Tools for PowerShell Cmdlet Reference*.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def get_orderable_instances(self, db_engine, db_engine_version):
        """
        Gets DB instance options that can be used to create DB instances that are
        compatible with a set of specifications.

        :param db_engine: The database engine that must be supported by the DB
instance.
        :param db_engine_version: The engine version that must be supported by
the DB instance.
        :return: The list of DB instance options that can be used to create a
compatible DB instance.
        """
        try:
            inst_opts = []
            paginator = self.rds_client.get_paginator(
                "describe_orderable_db_instance_options"
            )
            for page in paginator.paginate(
                Engine=db_engine, EngineVersion=db_engine_version
            ):
                inst_opts += page["OrderableDBInstanceOptions"]
        except ClientError as err:
            logger.error(
                "Couldn't get orderable DB instances. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return inst_opts
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   pub async fn get_instance_classes(&self) -> Result<Vec<String>,
ScenarioError> {
       let describe_orderable_db_instance_options_items = self
           .rds
           .describe_orderable_db_instance_options(
               DB_ENGINE,
               self.engine_version
                   .as_ref()
                   .expect("engine version for db instance options")
                   .as_str(),
           )
           .await;

       describe_orderable_db_instance_options_items
           .map(|options| {
               options
                   .iter()
                   .map(|o|
o.db_instance_class().unwrap_or_default().to_string())
                   .collect::<Vec<String>>()
           })
           .map_err(|err| ScenarioError::new("Could not get available instance
classes", &err))
   }

   pub async fn describe_orderable_db_instance_options(
       &self,
       engine: &str,
       engine_version: &str,
   ) -> Result<Vec<OrderableDbInstanceOption>,
SdkError<DescribeOrderableDBInstanceOptionsError>>
   {
        self.inner
            .describe_orderable_db_instance_options()
            .engine(engine)
            .engine_version(engine_version)
            .into_paginator()
            .items()
            .send()
            .try_collect()
            .await
    }

#[tokio::test]
async fn test_scenario_get_instance_classes() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()

 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    mock_rds
        .expect_describe_orderable_db_instance_options()
        .with(eq("aurora-mysql"), eq("aurora-mysql8.0"))
        .return_once(|_, _| {
            Ok(vec![
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t1")
                    .build(),
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t2")
                    .build(),
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t3")
                    .build(),
            ])
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario
        .set_engine("aurora-mysql", "aurora-mysql8.0")
        .await
        .expect("set engine");

    let instance_classes = scenario.get_instance_classes().await;

    assert_eq!(
        instance_classes,
        Ok(vec!["t1".into(), "t2".into(), "t3".into()])
    );
}

#[tokio::test]
async fn test_scenario_get_instance_classes_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_orderable_db_instance_options()
        .with(eq("aurora-mysql"), eq("aurora-mysql8.0"))
        .return_once(|_, _| {
            Err(SdkError::service_error(

 DescribeOrderableDBInstanceOptionsError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_orderable_db_instance_options_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_family = Some("aurora-mysql".into());
    scenario.engine_version = Some("aurora-mysql8.0".into());

    let instance_classes = scenario.get_instance_classes().await;

    assert_matches!(
        instance_classes,
        Err(ScenarioError {message, context: _}) if message == "Could not get
 available instance classes"
    );
}
```

- For API details, see DescribeOrderableDBInstanceOptions in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ModifyDBClusterParameterGroup` with an AWS SDK or CLI

The following code examples show how to use `ModifyDBClusterParameterGroup`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Get started with DB clusters

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Modify the specified integer parameters with new values from user input.
    /// </summary>
    /// <param name="groupName">The group name for the parameters.</param>
    /// <param name="parameters">The list of integer parameters to modify.</
param>
    /// <param name="newValue">Optional int value to set for parameters.</param>
    /// <returns>The name of the group that was modified.</returns>
    public async Task<string> ModifyIntegerParametersInGroupAsync(string
 groupName, List<Parameter> parameters, int newValue = 0)
    {
        foreach (var p in parameters)
        {
            if (p.IsModifiable && p.DataType == "integer")
            {
                while (newValue == 0)
                {
                    Console.WriteLine(
                        $"Enter a new value for {p.ParameterName} from the 
allowed values {p.AllowedValues} ");

                    var choice = Console.ReadLine();
                    int.TryParse(choice, out newValue);
                }

                p.ParameterValue = newValue.ToString();
            }
        }

        var request = new ModifyDBClusterParameterGroupRequest
        {
            Parameters = parameters,
            DBClusterParameterGroupName = groupName,
        };

        var result = await 
_amazonRDS.ModifyDBClusterParameterGroupAsync(request);
        return result.DBClusterParameterGroupName;
    }
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for .NET API Reference*.

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";
```

```
    Aws::RDS::RDSClient client(clientConfig);

        Aws::RDS::Model::ModifyDBClusterParameterGroupRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        request.SetParameters(updateParameters);

        Aws::RDS::Model::ModifyDBClusterParameterGroupOutcome outcome =
                client.ModifyDBClusterParameterGroup(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB cluster parameter group was successfully
 modified."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::ModifyDBClusterParameterGroup. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// UpdateParameters updates parameters in a named DB cluster parameter group.
```

```
func (clusters *DbClusters) UpdateParameters(parameterGroupName string, params
 []types.Parameter) error {
 _, err := clusters.AuroraClient.ModifyDBClusterParameterGroup(context.TODO(),
  &rds.ModifyDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   Parameters:                  params,
  })
 if err != nil {
  log.Printf("Couldn't update parameters in %v: %v\n", parameterGroupName, err)
  return err
 } else {
  return nil
 }
}
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public static void describeDbClusterParameterGroups(RdsClient rdsClient,
 String dbClusterGroupName) {
        try {
            DescribeDbClusterParameterGroupsRequest groupsRequest =
 DescribeDbClusterParameterGroupsRequest.builder()
                    .dbClusterParameterGroupName(dbClusterGroupName)
                    .maxRecords(20)
                    .build();

            List<DBClusterParameterGroup> groups =
 rdsClient.describeDBClusterParameterGroups(groupsRequest)
                    .dbClusterParameterGroups();
            for (DBClusterParameterGroup group : groups) {
                System.out.println("The group name is " +
 group.dbClusterParameterGroupName());
                System.out.println("The group ARN is " +
 group.dbClusterParameterGroupArn());
            }

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
```

- For API details, see [ModifyDBClusterParameterGroup](#) in *AWS SDK for Java 2.x API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

```
// Modify the auto_increment_offset parameter.
suspend fun modifyDBClusterParas(dClusterGroupName: String?) {
    val parameter1 =
        Parameter {
            parameterName = "auto_increment_offset"
            applyMethod = ApplyMethod.fromValue("immediate")
            parameterValue = "5"
        }

    val paraList = ArrayList<Parameter>()
    paraList.add(parameter1)
    val groupRequest =
        ModifyDbClusterParameterGroupRequest {
            dbClusterParameterGroupName = dClusterGroupName
            parameters = paraList
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.modifyDbClusterParameterGroup(groupRequest)
        println("The parameter group ${response.dbClusterParameterGroupName} was
 successfully modified")
    }
}
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for Kotlin API reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)


    def update_parameters(self, parameter_group_name, update_parameters):
        """
        Updates parameters in a custom DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to update.
        :param update_parameters: The parameters to update in the group.
        :return: Data about the modified parameter group.
        """
        try:
            response = self.rds_client.modify_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name,
                Parameters=update_parameters,
            )
        except ClientError as err:
            logger.error(
                "Couldn't update parameters in %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for Python (Boto3) API Reference*.

Rust

SDK for Rust

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    // Modify both the auto_increment_offset and auto_increment_increment
parameters in one call in the custom parameter group. Set their ParameterValue
fields to a new allowable value. rds.ModifyDbClusterParameterGroup.
```

```
    pub async fn update_auto_increment(
        &self,
        offset: u8,
        increment: u8,
    ) -> Result<(), ScenarioError> {
        let modify_db_cluster_parameter_group = self
            .rds
            .modify_db_cluster_parameter_group(
                DB_CLUSTER_PARAMETER_GROUP_NAME,
                vec![
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .parameter_value(format!("{offset}"))
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .parameter_value(format!("{increment}"))
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                ],
            )
            .await;

        if let Err(error) = modify_db_cluster_parameter_group {
            return Err(ScenarioError::new(
                "Failed to modify cluster parameter group",
                &error,
            ));
        }

        Ok(())
    }

    pub async fn modify_db_cluster_parameter_group(
        &self,
        name: &str,
        parameters: Vec<Parameter>,
    ) -> Result<ModifyDbClusterParameterGroupOutput,
SdkError<ModifyDBClusterParameterGroupError>>
    {
        self.inner
            .modify_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .set_parameters(Some(parameters))
            .send()
            .await
    }

#[tokio::test]
async fn test_scenario_update_auto_increment() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_modify_db_cluster_parameter_group()
        .withf(|name, params| {
            assert_eq!(name, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(
                params,
                &vec![
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .parameter_value("10")
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .parameter_value("20")
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                ]
            );
            true
        })
        .return_once(|_, _|
 Ok(ModifyDbClusterParameterGroupOutput::builder().build()));

    let scenario = AuroraScenario::new(mock_rds);

    scenario
        .update_auto_increment(10, 20)
        .await
        .expect("update auto increment");
}

#[tokio::test]
async fn test_scenario_update_auto_increment_error() {
    let mut mock_rds = MockRdsImpl::default();
    mock_rds
        .expect_modify_db_cluster_parameter_group()
        .return_once(|_, _| {
            Err(SdkError::service_error(
 ModifyDBClusterParameterGroupError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "modify_db_cluster_parameter_group_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let scenario = AuroraScenario::new(mock_rds);

    let update = scenario.update_auto_increment(10, 20).await;
    assert_matches!(update, Err(ScenarioError { message, context: _}) if message
 == "Failed to modify cluster parameter group");
}
```

- For API details, see ModifyDBClusterParameterGroup in *AWS SDK for Rust API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Aurora using AWS SDKs

The following code examples show you how to implement common scenarios in Aurora with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Aurora. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- Get started with Aurora DB clusters using an AWS SDK

Get started with Aurora DB clusters using an AWS SDK

The following code examples show how to:

- Create a custom Aurora DB cluster parameter group and set parameter values.
- Create a DB cluster that uses the parameter group.
- Create a DB instance that contains a database.
- Take a snapshot of the DB cluster, then clean up resources.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
using Amazon.RDS;
using Amazon.RDS.Model;
using AuroraActions;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Logging.Debug;

namespace AuroraScenario;

/// <summary>
/// Scenario for Amazon Aurora examples.
/// </summary>
public class AuroraScenario
{
    /*
```

```
   Before running this .NET code example, set up your development environment,
including your credentials.

   This .NET example performs the following tasks:
   1.  Return a list of the available DB engine families for Aurora MySql using
the DescribeDBEngineVersionsAsync method.
   2.  Select an engine family and create a custom DB cluster parameter group
using the CreateDBClusterParameterGroupAsync method.
   3.  Get the parameter group using the DescribeDBClusterParameterGroupsAsync
method.
   4.  Get some parameters in the group using the
DescribeDBClusterParametersAsync method.
   5.  Parse and display some parameters in the group.
   6.  Modify the auto_increment_offset and auto_increment_increment parameters
       using the ModifyDBClusterParameterGroupAsync method.
   7.  Get and display the updated parameters using the
DescribeDBClusterParametersAsync method with a source of "user".
   8.  Get a list of allowed engine versions using the
DescribeDBEngineVersionsAsync method.
   9.  Create an Aurora DB cluster that contains a MySql database and uses the
parameter group.
       using the CreateDBClusterAsync method.
   10. Wait for the DB cluster to be ready using the DescribeDBClustersAsync
method.
   11. Display and select from a list of instance classes available for the
selected engine and version
       using the paginated DescribeOrderableDBInstanceOptions method.
   12. Create a database instance in the cluster using the CreateDBInstanceAsync
method.
   13. Wait for the DB instance to be ready using the DescribeDBInstances
method.
   14. Display the connection endpoint string for the new DB cluster.
   15. Create a snapshot of the DB cluster using the
CreateDBClusterSnapshotAsync method.
   16. Wait for DB snapshot to be ready using the
DescribeDBClusterSnapshotsAsync method.
   17. Delete the DB instance using the DeleteDBInstanceAsync method.
   18. Delete the DB cluster using the DeleteDBClusterAsync method.
   19. Wait for DB cluster to be deleted using the DescribeDBClustersAsync
methods.
   20. Delete the cluster parameter group using the
DeleteDBClusterParameterGroupAsync.
   */
   private static readonly string sepBar = new('-', 80);
   private static AuroraWrapper auroraWrapper = null!;
   private static ILogger logger = null!;
   private static readonly string engine = "aurora-mysql";
   static async Task Main(string[] args)
   {
       // Set up dependency injection for the Amazon Relational Database Service
(Amazon RDS).
       using var host = Host.CreateDefaultBuilder(args)
           .ConfigureLogging(logging =>
               logging.AddFilter("System", LogLevel.Debug)
                   .AddFilter<DebugLoggerProvider>("Microsoft",
LogLevel.Information)
                   .AddFilter<ConsoleLoggerProvider>("Microsoft",
LogLevel.Trace))
           .ConfigureServices((_, services) =>
               services.AddAWSService<IAmazonRDS>()
                   .AddTransient<AuroraWrapper>()
           )
           .Build();

       logger = LoggerFactory.Create(builder =>
       {
           builder.AddConsole();
       }).CreateLogger<AuroraScenario>();

       auroraWrapper = host.Services.GetRequiredService<AuroraWrapper>();

       Console.WriteLine(sepBar);
       Console.WriteLine(
           "Welcome to the Amazon Aurora: get started with DB clusters
example.");
       Console.WriteLine(sepBar);

       DBClusterParameterGroup parameterGroup = null!;
       DBCluster? newCluster = null;
       DBInstance? newInstance = null;

       try
       {
           var parameterGroupFamily = await ChooseParameterGroupFamilyAsync();

           parameterGroup = await
CreateDBParameterGroupAsync(parameterGroupFamily);
```

```
           var parameters = await
DescribeParametersInGroupAsync(parameterGroup.DBClusterParameterGroupName,
               new List<string> { "auto_increment_offset",
"auto_increment_increment" });

           await
ModifyParametersAsync(parameterGroup.DBClusterParameterGroupName, parameters);

           await
DescribeUserSourceParameters(parameterGroup.DBClusterParameterGroupName);

           var engineVersionChoice = await
ChooseDBEngineVersionAsync(parameterGroupFamily);

           var newClusterIdentifier = "Example-Cluster-" + DateTime.Now.Ticks;

           newCluster = await CreateNewCluster
           (
               parameterGroup,
               engine,
               engineVersionChoice.EngineVersion,
               newClusterIdentifier
           );

           var instanceClassChoice = await ChooseDBInstanceClass(engine,
engineVersionChoice.EngineVersion);

           var newInstanceIdentifier = "Example-Instance-" + DateTime.Now.Ticks;

           newInstance = await CreateNewInstance(
               newClusterIdentifier,
               engine,
               engineVersionChoice.EngineVersion,
               instanceClassChoice.DBInstanceClass,
               newInstanceIdentifier
           );

           DisplayConnectionString(newCluster!);
           await CreateSnapshot(newCluster!);
           await CleanupResources(newInstance, newCluster, parameterGroup);

           Console.WriteLine("Scenario complete.");
           Console.WriteLine(sepBar);
```

```
       }
       catch (Exception ex)

       {
           await CleanupResources(newInstance, newCluster, parameterGroup);
           logger.LogError(ex, "There was a problem executing the scenario.");
       }
   }

   /// <summary>
   /// Choose the Aurora DB parameter group family from a list of available
options.
   /// </summary>
   /// <returns>The selected parameter group family.</returns>
   public static async Task<string> ChooseParameterGroupFamilyAsync()
   {
       Console.WriteLine(sepBar);
       // 1. Get a list of available engines.
       var engines = await
auroraWrapper.DescribeDBEngineVersionsForEngineAsync(engine);

       Console.WriteLine($"1. The following is a list of available DB parameter
group families for engine {engine}:");

       var parameterGroupFamilies =
           engines.GroupBy(e => e.DBParameterGroupFamily).ToList();
       for (var i = 1; i <= parameterGroupFamilies.Count; i++)
       {
           var parameterGroupFamily = parameterGroupFamilies[i - 1];
           // List the available parameter group families.
           Console.WriteLine(
               $"\t{i}. Family: {parameterGroupFamily.Key}");
       }

       var choiceNumber = 0;
       while (choiceNumber < 1 || choiceNumber > parameterGroupFamilies.Count)
       {
           Console.WriteLine("2. Select an available DB parameter group family
by entering a number from the preceding list:");
           var choice = Console.ReadLine();
           Int32.TryParse(choice, out choiceNumber);
       }
       var parameterGroupFamilyChoice = parameterGroupFamilies[choiceNumber -
1];
```

```
        Console.WriteLine(sepBar);
        return parameterGroupFamilyChoice.Key;
    }

    /// <summary>
    /// Create and get information on a DB parameter group.
    /// </summary>
    /// <param name="dbParameterGroupFamily">The DBParameterGroupFamily for the
 new DB parameter group.</param>
    /// <returns>The new DBParameterGroup.</returns>
    public static async Task<DBClusterParameterGroup>
 CreateDBParameterGroupAsync(string dbParameterGroupFamily)
    {
        Console.WriteLine(sepBar);
        Console.WriteLine($"2. Create new DB parameter group with family
 {dbParameterGroupFamily}:");

        var parameterGroup = await
 auroraWrapper.CreateCustomClusterParameterGroupAsync(
            dbParameterGroupFamily,
            "ExampleParameterGroup-" + DateTime.Now.Ticks,
            "New example parameter group");

        var groupInfo =
            await
 auroraWrapper.DescribeCustomDBClusterParameterGroupAsync(parameterGroup.DBClusterParamet

        Console.WriteLine(
            $"3. New DB parameter group created: \n\t{groupInfo?.Description}, \n
\tARN {groupInfo?.DBClusterParameterGroupName}");
        Console.WriteLine(sepBar);
        return parameterGroup;
    }

    /// <summary>
    /// Get and describe parameters from a DBParameterGroup.
    /// </summary>
    /// <param name="parameterGroupName">The name of the DBParameterGroup.</
param>
    /// <param name="parameterNames">Optional specific names of parameters to
 describe.</param>
    /// <returns>The list of requested parameters.</returns>
```

```
   public static async Task<List<Parameter>>
DescribeParametersInGroupAsync(string parameterGroupName, List<string>?
parameterNames = null)
   {
       Console.WriteLine(sepBar);
       Console.WriteLine("4. Get some parameters from the group.");
       Console.WriteLine(sepBar);

       var parameters =
           await
auroraWrapper.DescribeDBClusterParametersInGroupAsync(parameterGroupName);

       var matchingParameters =
           parameters.Where(p => parameterNames == null ||
parameterNames.Contains(p.ParameterName)).ToList();

       Console.WriteLine("5. Parameter information:");
       matchingParameters.ForEach(p =>
           Console.WriteLine(
               $"\n\tParameter: {p.ParameterName}." +
               $"\n\tDescription: {p.Description}." +
               $"\n\tAllowed Values: {p.AllowedValues}." +
               $"\n\tValue: {p.ParameterValue}."));

       Console.WriteLine(sepBar);

       return matchingParameters;
   }

   /// <summary>
   /// Modify a parameter from a DBParameterGroup.
   /// </summary>
   /// <param name="parameterGroupName">Name of the DBParameterGroup.</param>
   /// <param name="parameters">The parameters to modify.</param>
   /// <returns>Async task.</returns>
   public static async Task ModifyParametersAsync(string parameterGroupName,
List<Parameter> parameters)
   {
       Console.WriteLine(sepBar);
       Console.WriteLine("6. Modify some parameters in the group.");

       await
auroraWrapper.ModifyIntegerParametersInGroupAsync(parameterGroupName,
parameters);
```

```
        Console.WriteLine(sepBar);
    }

    /// <summary>
    /// Describe the user source parameters in the group.
    /// </summary>
    /// <param name="parameterGroupName">The name of the DBParameterGroup.</
param>
    /// <returns>Async task.</returns>
    public static async Task DescribeUserSourceParameters(string
 parameterGroupName)
    {
        Console.WriteLine(sepBar);
        Console.WriteLine("7. Describe updated user source parameters in the
 group.");

        var parameters =
            await
 auroraWrapper.DescribeDBClusterParametersInGroupAsync(parameterGroupName,
 "user");

        parameters.ForEach(p =>
            Console.WriteLine(
                $"\n\tParameter: {p.ParameterName}." +
                $"\n\tDescription: {p.Description}." +
                $"\n\tAllowed Values: {p.AllowedValues}." +
                $"\n\tValue: {p.ParameterValue}."));

        Console.WriteLine(sepBar);
    }

    /// <summary>
    /// Choose a DB engine version.
    /// </summary>
    /// <param name="dbParameterGroupFamily">DB parameter group family for engine
 choice.</param>
    /// <returns>The selected engine version.</returns>
    public static async Task<DBEngineVersion> ChooseDBEngineVersionAsync(string
 dbParameterGroupFamily)
    {
        Console.WriteLine(sepBar);
        // Get a list of allowed engines.
        var allowedEngines =
            await auroraWrapper.DescribeDBEngineVersionsForEngineAsync(engine,
 dbParameterGroupFamily);

        Console.WriteLine($"Available DB engine versions for parameter group
 family {dbParameterGroupFamily}:");
        int i = 1;
        foreach (var version in allowedEngines)
        {
            Console.WriteLine(
                $"\t{i}. {version.DBEngineVersionDescription}.");
            i++;
        }

        var choiceNumber = 0;
        while (choiceNumber < 1 || choiceNumber > allowedEngines.Count)
        {
            Console.WriteLine("8. Select an available DB engine version by
 entering a number from the list above:");
            var choice = Console.ReadLine();
            Int32.TryParse(choice, out choiceNumber);
        }

        var engineChoice = allowedEngines[choiceNumber - 1];
        Console.WriteLine(sepBar);
        return engineChoice;
    }

    /// <summary>
    /// Create a new RDS DB cluster.
    /// </summary>
    /// <param name="parameterGroup">Parameter group to use for the DB cluster.</
param>
    /// <param name="engineName">Engine to use for the DB cluster.</param>
    /// <param name="engineVersion">Engine version to use for the DB cluster.</
param>
    /// <param name="clusterIdentifier">Cluster identifier to use for the DB
 cluster.</param>
    /// <returns>The new DB cluster.</returns>
    public static async Task<DBCluster?> CreateNewCluster(DBClusterParameterGroup
 parameterGroup,
        string engineName, string engineVersion, string clusterIdentifier)
    {
        Console.WriteLine(sepBar);
        Console.WriteLine($"9. Create a new DB cluster with identifier 
{clusterIdentifier}.");

        DBCluster newCluster;
        var clusters = await auroraWrapper.DescribeDBClustersPagedAsync();
        var isClusterCreated = clusters.Any(i => i.DBClusterIdentifier == 
clusterIdentifier);

        if (isClusterCreated)
        {
            Console.WriteLine("Cluster already created.");
            newCluster = clusters.First(i => i.DBClusterIdentifier == 
clusterIdentifier);
        }
        else
        {
            Console.WriteLine("Enter an admin username:");
            var username = Console.ReadLine();

            Console.WriteLine("Enter an admin password:");
            var password = Console.ReadLine();

            newCluster = await auroraWrapper.CreateDBClusterWithAdminAsync(
                "ExampleDatabase",
                clusterIdentifier,
                parameterGroup.DBClusterParameterGroupName,
                engineName,
                engineVersion,
                username!,
                password!
            );

            Console.WriteLine("10. Waiting for DB cluster to be ready...");
            while (newCluster.Status != "available")
            {
                Console.Write(".");
                Thread.Sleep(5000);
                clusters = await 
auroraWrapper.DescribeDBClustersPagedAsync(clusterIdentifier);
                newCluster = clusters.First();
            }
        }

        Console.WriteLine(sepBar);
        return newCluster;
    }

    /// <summary>
    /// Choose a DB instance class for a particular engine and engine version.
    /// </summary>
    /// <param name="engine">DB engine for DB instance choice.</param>
    /// <param name="engineVersion">DB engine version for DB instance choice.</
param>
    /// <returns>The selected orderable DB instance option.</returns>
    public static async Task<OrderableDBInstanceOption>
 ChooseDBInstanceClass(string engine, string engineVersion)
    {
        Console.WriteLine(sepBar);
        // Get a list of allowed DB instance classes.
        var allowedInstances =
            await
 auroraWrapper.DescribeOrderableDBInstanceOptionsPagedAsync(engine,
 engineVersion);

        Console.WriteLine($"Available DB instance classes for engine {engine} and
 version {engineVersion}:");
        int i = 1;


        foreach (var instance in allowedInstances)
        {
            Console.WriteLine(
                $"\t{i}. Instance class: {instance.DBInstanceClass} (storage type
 {instance.StorageType})");
            i++;
        }

        var choiceNumber = 0;
        while (choiceNumber < 1 || choiceNumber > allowedInstances.Count)
        {
            Console.WriteLine("11. Select an available DB instance class by
 entering a number from the preceding list:");
            var choice = Console.ReadLine();
            Int32.TryParse(choice, out choiceNumber);
        }

        var instanceChoice = allowedInstances[choiceNumber - 1];
        Console.WriteLine(sepBar);
        return instanceChoice;
    }

    /// <summary>
    /// Create a new DB instance.
    /// </summary>
    /// <param name="engineName">Engine to use for the DB instance.</param>
    /// <param name="engineVersion">Engine version to use for the DB instance.</
param>
    /// <param name="instanceClass">Instance class to use for the DB instance.</
param>
    /// <param name="instanceIdentifier">Instance identifier to use for the DB
 instance.</param>
    /// <returns>The new DB instance.</returns>
    public static async Task<DBInstance?> CreateNewInstance(
        string clusterIdentifier,
        string engineName,
        string engineVersion,
        string instanceClass,
        string instanceIdentifier)
    {
        Console.WriteLine(sepBar);
        Console.WriteLine($"12. Create a new DB instance with identifier
 {instanceIdentifier}.");
        bool isInstanceReady = false;
        DBInstance newInstance;
        var instances = await auroraWrapper.DescribeDBInstancesPagedAsync();
        isInstanceReady = instances.FirstOrDefault(i =>
            i.DBInstanceIdentifier == instanceIdentifier)?.DBInstanceStatus ==
 "available";

        if (isInstanceReady)
        {
            Console.WriteLine("Instance already created.");
            newInstance = instances.First(i => i.DBInstanceIdentifier ==
 instanceIdentifier);
        }
        else
        {

            newInstance = await auroraWrapper.CreateDBInstanceInClusterAsync(
                clusterIdentifier,
                instanceIdentifier,
                engineName,
                engineVersion,
                instanceClass
            );

            Console.WriteLine("13. Waiting for DB instance to be ready...");
            while (!isInstanceReady)
            {
                Console.Write(".");
                Thread.Sleep(5000);
                instances = await
 auroraWrapper.DescribeDBInstancesPagedAsync(instanceIdentifier);
                isInstanceReady = instances.FirstOrDefault()?.DBInstanceStatus ==
 "available";
                newInstance = instances.First();
            }
        }

        Console.WriteLine(sepBar);
        return newInstance;
    }

    /// <summary>
    /// Display a connection string for an Amazon RDS DB cluster.
    /// </summary>
    /// <param name="cluster">The DB cluster to use to get a connection string.</
param>
    public static void DisplayConnectionString(DBCluster cluster)
    {
        Console.WriteLine(sepBar);
        // Display the connection string.
        Console.WriteLine("14. New DB cluster connection string: ");
        Console.WriteLine(
            $"\n{engine} -h {cluster.Endpoint} -P {cluster.Port} "
            + $"-u {cluster.MasterUsername} -p [YOUR PASSWORD]\n");

        Console.WriteLine(sepBar);
    }

    /// <summary>
    /// Create a snapshot from an Amazon RDS DB cluster.
    /// </summary>
    /// <param name="cluster">DB cluster to use when creating a snapshot.</param>
    /// <returns>The snapshot object.</returns>
    public static async Task<DBClusterSnapshot> CreateSnapshot(DBCluster cluster)
```

```
   {
       Console.WriteLine(sepBar);
       // Create a snapshot.
       Console.WriteLine($"15. Creating snapshot from DB cluster
{cluster.DBClusterIdentifier}.");
       var snapshot = await
auroraWrapper.CreateClusterSnapshotByIdentifierAsync(
           cluster.DBClusterIdentifier,
           "ExampleSnapshot-" + DateTime.Now.Ticks);

       // Wait for the snapshot to be available.
       bool isSnapshotReady = false;

       Console.WriteLine($"16. Waiting for snapshot to be ready...");
       while (!isSnapshotReady)
       {
           Console.Write(".");
           Thread.Sleep(5000);
           var snapshots =
               await
auroraWrapper.DescribeDBClusterSnapshotsByIdentifierAsync(cluster.DBClusterIdentifier);
           isSnapshotReady = snapshots.FirstOrDefault()?.Status == "available";
           snapshot = snapshots.First();
       }

       Console.WriteLine(
           $"Snapshot {snapshot.DBClusterSnapshotIdentifier} status is
{snapshot.Status}.");
       Console.WriteLine(sepBar);
       return snapshot;
   }

   /// <summary>
   /// Clean up resources from the scenario.
   /// </summary>
   /// <param name="newInstance">The instance to clean up.</param>
   /// <param name="newCluster">The cluster to clean up.</param>
   /// <param name="parameterGroup">The parameter group to clean up.</param>
   /// <returns>Async Task.</returns>
   private static async Task CleanupResources(
       DBInstance? newInstance,
       DBCluster? newCluster,
       DBClusterParameterGroup? parameterGroup)
   {
       Console.WriteLine(new string('-', 80));
       Console.WriteLine($"Clean up resources.");

       if (newInstance is not null && GetYesNoResponse($"\tClean up instance 
{newInstance.DBInstanceIdentifier}? (y/n)"))
       {
           // Delete the DB instance.
           Console.WriteLine($"17. Deleting the DB instance 
{newInstance.DBInstanceIdentifier}.");
           await 
auroraWrapper.DeleteDBInstanceByIdentifierAsync(newInstance.DBInstanceIdentifier);
       }

       if (newCluster is not null && GetYesNoResponse($"\tClean up cluster 
{newCluster.DBClusterIdentifier}? (y/n)"))
       {
           // Delete the DB cluster.
           Console.WriteLine($"18. Deleting the DB cluster 
{newCluster.DBClusterIdentifier}.");
           await 
auroraWrapper.DeleteDBClusterByIdentifierAsync(newCluster.DBClusterIdentifier);

           // Wait for the DB cluster to delete.
           Console.WriteLine($"19. Waiting for the DB cluster to delete...");
           bool isClusterDeleted = false;

           while (!isClusterDeleted)
           {
               Console.Write(".");
               Thread.Sleep(5000);
               var cluster = await auroraWrapper.DescribeDBClustersPagedAsync();
               isClusterDeleted = cluster.All(i => i.DBClusterIdentifier != 
newCluster.DBClusterIdentifier);
           }

           Console.WriteLine("DB cluster deleted.");
       }

       if (parameterGroup is not null && GetYesNoResponse($"\tClean up parameter 
group? (y/n)"))
       {
           Console.WriteLine($"20. Deleting the DB parameter group 
{parameterGroup.DBClusterParameterGroupName}.");
```

```
            await
 auroraWrapper.DeleteClusterParameterGroupByNameAsync(parameterGroup.DBClusterParameterGr
            Console.WriteLine("Parameter group deleted.");
        }

        Console.WriteLine(new string('-', 80));
    }

    /// <summary>
    /// Get a yes or no response from the user.
    /// </summary>
    /// <param name="question">The question string to print on the console.</
param>
    /// <returns>True if the user responds with a yes.</returns>
    private static bool GetYesNoResponse(string question)
    {
        Console.WriteLine(question);
        var ynResponse = Console.ReadLine();
        var response = ynResponse != null &&
                       ynResponse.Equals("y",
                           StringComparison.InvariantCultureIgnoreCase);
        return response;
    }
```

Wrapper methods that are called by the scenario to manage Aurora actions.

```
using Amazon.RDS;
using Amazon.RDS.Model;

namespace AuroraActions;

/// <summary>
/// Wrapper for the Amazon Aurora cluster client operations.
/// </summary>
public class AuroraWrapper
{
    private readonly IAmazonRDS _amazonRDS;
    public AuroraWrapper(IAmazonRDS amazonRDS)
    {
        _amazonRDS = amazonRDS;
    }
    /// <summary>
    /// Get a list of DB engine versions for a particular DB engine.
    /// </summary>
    /// <param name="engine">The name of the engine.</param>
    /// <param name="parameterGroupFamily">Optional parameter group family
 name.</param>
    /// <returns>A list of DBEngineVersions.</returns>
    public async Task<List<DBEngineVersion>>
 DescribeDBEngineVersionsForEngineAsync(string engine,
        string? parameterGroupFamily = null)
    {
        var response = await _amazonRDS.DescribeDBEngineVersionsAsync(
            new DescribeDBEngineVersionsRequest()
            {
                Engine = engine,
                DBParameterGroupFamily = parameterGroupFamily
            });
        return response.DBEngineVersions;
    }

    /// <summary>
    /// Create a custom cluster parameter group.
    /// </summary>
    /// <param name="parameterGroupFamily">The family of the parameter group.</
param>
    /// <param name="groupName">The name for the new parameter group.</param>
    /// <param name="description">A description for the new parameter group.</
param>
    /// <returns>The new parameter group object.</returns>
    public async Task<DBClusterParameterGroup>
 CreateCustomClusterParameterGroupAsync(
        string parameterGroupFamily,
        string groupName,
        string description)
    {
        var request = new CreateDBClusterParameterGroupRequest
        {
            DBParameterGroupFamily = parameterGroupFamily,
            DBClusterParameterGroupName = groupName,
            Description = description,
        };
        var response = await
 _amazonRDS.CreateDBClusterParameterGroupAsync(request);
        return response.DBClusterParameterGroup;
    }

    /// <summary>
    /// Describe the cluster parameters in a parameter group.
    /// </summary>
    /// <param name="groupName">The name of the parameter group.</param>
    /// <param name="source">The optional name of the source filter.</param>
    /// <returns>The collection of parameters.</returns>
    public async Task<List<Parameter>>
 DescribeDBClusterParametersInGroupAsync(string groupName, string? source = null)
    {
        var paramList = new List<Parameter>();

        DescribeDBClusterParametersResponse response;
        var request = new DescribeDBClusterParametersRequest
        {
            DBClusterParameterGroupName = groupName,
            Source = source,
        };

        // Get the full list if there are multiple pages.
        do
        {
            response = await
 _amazonRDS.DescribeDBClusterParametersAsync(request);
            paramList.AddRange(response.Parameters);

            request.Marker = response.Marker;
        }
        while (response.Marker is not null);

        return paramList;
    }

    /// <summary>
    /// Get the description of a DB cluster parameter group by name.
    /// </summary>
    /// <param name="name">The name of the DB parameter group to describe.</
param>
    /// <returns>The parameter group description.</returns>
    public async Task<DBClusterParameterGroup?>
 DescribeCustomDBClusterParameterGroupAsync(string name)
    {
        var response = await _amazonRDS.DescribeDBClusterParameterGroupsAsync(
            new DescribeDBClusterParameterGroupsRequest()
            {
                DBClusterParameterGroupName = name
            });
        return response.DBClusterParameterGroups.FirstOrDefault();
    }

    /// <summary>
    /// Modify the specified integer parameters with new values from user input.
    /// </summary>
    /// <param name="groupName">The group name for the parameters.</param>
    /// <param name="parameters">The list of integer parameters to modify.</
param>
    /// <param name="newValue">Optional int value to set for parameters.</param>
    /// <returns>The name of the group that was modified.</returns>
    public async Task<string> ModifyIntegerParametersInGroupAsync(string
 groupName, List<Parameter> parameters, int newValue = 0)
    {
        foreach (var p in parameters)
        {
            if (p.IsModifiable && p.DataType == "integer")
            {
                while (newValue == 0)
                {
                    Console.WriteLine(
                        $"Enter a new value for {p.ParameterName} from the
 allowed values {p.AllowedValues} ");

                    var choice = Console.ReadLine();
                    int.TryParse(choice, out newValue);
                }

                p.ParameterValue = newValue.ToString();
            }
        }

        var request = new ModifyDBClusterParameterGroupRequest
        {
            Parameters = parameters,
            DBClusterParameterGroupName = groupName,
        };

        var result = await
_amazonRDS.ModifyDBClusterParameterGroupAsync(request);
        return result.DBClusterParameterGroupName;
    }


    /// <summary>
    /// Get a list of orderable DB instance options for a specific
    /// engine and engine version.
    /// </summary>
    /// <param name="engine">Name of the engine.</param>
    /// <param name="engineVersion">Version of the engine.</param>
    /// <returns>List of OrderableDBInstanceOptions.</returns>
    public async Task<List<OrderableDBInstanceOption>>
DescribeOrderableDBInstanceOptionsPagedAsync(string engine, string
engineVersion)
    {
        // Use a paginator to get a list of DB instance options.
        var results = new List<OrderableDBInstanceOption>();
        var paginateInstanceOptions =
_amazonRDS.Paginators.DescribeOrderableDBInstanceOptions(
            new DescribeOrderableDBInstanceOptionsRequest()
            {
                Engine = engine,
                EngineVersion = engineVersion,
            });
        // Get the entire list using the paginator.
        await foreach (var instanceOptions in
paginateInstanceOptions.OrderableDBInstanceOptions)
        {
            results.Add(instanceOptions);
        }
        return results;
    }

    /// <summary>
    /// Delete a particular parameter group by name.
    /// </summary>
    /// <param name="groupName">The name of the parameter group.</param>
    /// <returns>True if successful.</returns>
    public async Task<bool> DeleteClusterParameterGroupByNameAsync(string
groupName)
```

```
    {
        var request = new DeleteDBClusterParameterGroupRequest
        {
            DBClusterParameterGroupName = groupName,
        };

        var response = await
_amazonRDS.DeleteDBClusterParameterGroupAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new cluster and database.
    /// </summary>
    /// <param name="dbName">The name of the new database.</param>
    /// <param name="clusterIdentifier">The identifier of the cluster.</param>
    /// <param name="parameterGroupName">The name of the parameter group.</param>
    /// <param name="dbEngine">The engine to use for the new cluster.</param>
    /// <param name="dbEngineVersion">The version of the engine to use.</param>
    /// <param name="adminName">The admin username.</param>
    /// <param name="adminPassword">The primary admin password.</param>
    /// <returns>The cluster object.</returns>
    public async Task<DBCluster> CreateDBClusterWithAdminAsync(
        string dbName,
        string clusterIdentifier,
        string parameterGroupName,
        string dbEngine,
        string dbEngineVersion,
        string adminName,
        string adminPassword)
    {
        var request = new CreateDBClusterRequest
        {
            DatabaseName = dbName,
            DBClusterIdentifier = clusterIdentifier,
            DBClusterParameterGroupName = parameterGroupName,
            Engine = dbEngine,
            EngineVersion = dbEngineVersion,
            MasterUsername = adminName,
            MasterUserPassword = adminPassword,
        };

        var response = await _amazonRDS.CreateDBClusterAsync(request);
        return response.DBCluster;
    }

    /// <summary>
    /// Returns a list of DB instances.
    /// </summary>
    /// <param name="dbInstanceIdentifier">Optional name of a specific DB 
 instance.</param>
    /// <returns>List of DB instances.</returns>
    public async Task<List<DBInstance>> DescribeDBInstancesPagedAsync(string? 
 dbInstanceIdentifier = null)
    {
        var results = new List<DBInstance>();
        var instancesPaginator = _amazonRDS.Paginators.DescribeDBInstances(
            new DescribeDBInstancesRequest
            {
                DBInstanceIdentifier = dbInstanceIdentifier
            });
        // Get the entire list using the paginator.
        await foreach (var instances in instancesPaginator.DBInstances)
        {
            results.Add(instances);
        }
        return results;
    }

    /// <summary>
    /// Returns a list of DB clusters.
    /// </summary>
    /// <param name="dbInstanceIdentifier">Optional name of a specific DB 
 cluster.</param>
    /// <returns>List of DB clusters.</returns>
    public async Task<List<DBCluster>> DescribeDBClustersPagedAsync(string? 
 dbClusterIdentifier = null)
    {
        var results = new List<DBCluster>();

        DescribeDBClustersResponse response;
        DescribeDBClustersRequest request = new DescribeDBClustersRequest
        {
            DBClusterIdentifier = dbClusterIdentifier
        };
        // Get the full list if there are multiple pages.
        do
        {
            response = await _amazonRDS.DescribeDBClustersAsync(request);
            results.AddRange(response.DBClusters);
            request.Marker = response.Marker;
        }
        while (response.Marker is not null);
        return results;
    }

    /// <summary>
    /// Create an Amazon Relational Database Service (Amazon RDS) DB instance
    /// with a particular set of properties. Use the action
 DescribeDBInstancesAsync
    /// to determine when the DB instance is ready to use.
    /// </summary>
    /// <param name="dbInstanceIdentifier">DB instance identifier.</param>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <param name="dbEngine">The engine for the DB instance.</param>
    /// <param name="dbEngineVersion">Version for the DB instance.</param>
    /// <param name="instanceClass">Class for the DB instance.</param>
    /// <returns>DB instance object.</returns>
    public async Task<DBInstance> CreateDBInstanceInClusterAsync(
        string dbClusterIdentifier,
        string dbInstanceIdentifier,
        string dbEngine,
        string dbEngineVersion,
        string instanceClass)
    {
        // When creating the instance within a cluster, do not specify the name
 or size.
        var response = await _amazonRDS.CreateDBInstanceAsync(
            new CreateDBInstanceRequest()
            {
                DBClusterIdentifier = dbClusterIdentifier,
                DBInstanceIdentifier = dbInstanceIdentifier,
                Engine = dbEngine,
                EngineVersion = dbEngineVersion,
                DBInstanceClass = instanceClass
            });

        return response.DBInstance;
    }

    /// <summary>
    /// Create a snapshot of a cluster.
    /// </summary>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <param name="snapshotIdentifier">Identifier for the snapshot.</param>
    /// <returns>DB snapshot object.</returns>
    public async Task<DBClusterSnapshot>
 CreateClusterSnapshotByIdentifierAsync(string dbClusterIdentifier, string
 snapshotIdentifier)
    {
        var response = await _amazonRDS.CreateDBClusterSnapshotAsync(
            new CreateDBClusterSnapshotRequest()
            {
                DBClusterIdentifier = dbClusterIdentifier,
                DBClusterSnapshotIdentifier = snapshotIdentifier,
            });

        return response.DBClusterSnapshot;
    }

    /// <summary>
    /// Return a list of DB snapshots for a particular DB cluster.
    /// </summary>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <returns>List of DB snapshots.</returns>
    public async Task<List<DBClusterSnapshot>>
 DescribeDBClusterSnapshotsByIdentifierAsync(string dbClusterIdentifier)
    {
        var results = new List<DBClusterSnapshot>();

        DescribeDBClusterSnapshotsResponse response;
        DescribeDBClusterSnapshotsRequest request = new
 DescribeDBClusterSnapshotsRequest
        {
            DBClusterIdentifier = dbClusterIdentifier
        };
        // Get the full list if there are multiple pages.
        do
        {
            response = await _amazonRDS.DescribeDBClusterSnapshotsAsync(request);
            results.AddRange(response.DBClusterSnapshots);
            request.Marker = response.Marker;
        }
        while (response.Marker is not null);
        return results;
    }
    /// <summary>
    /// Delete a particular DB cluster.
    /// </summary>
    /// <param name="dbClusterIdentifier">DB cluster identifier.</param>
    /// <returns>DB cluster object.</returns>
    public async Task<DBCluster> DeleteDBClusterByIdentifierAsync(string
 dbClusterIdentifier)
    {
        var response = await _amazonRDS.DeleteDBClusterAsync(
            new DeleteDBClusterRequest()
            {
                DBClusterIdentifier = dbClusterIdentifier,
                SkipFinalSnapshot = true
            });

        return response.DBCluster;
    }

    /// <summary>
    /// Delete a particular DB instance.
    /// </summary>
    /// <param name="dbInstanceIdentifier">DB instance identifier.</param>
    /// <returns>DB instance object.</returns>
    public async Task<DBInstance> DeleteDBInstanceByIdentifierAsync(string
 dbInstanceIdentifier)
    {
        var response = await _amazonRDS.DeleteDBInstanceAsync(
            new DeleteDBInstanceRequest()
            {
                DBInstanceIdentifier = dbInstanceIdentifier,
                SkipFinalSnapshot = true,
                DeleteAutomatedBackups = true
            });

        return response.DBInstance;
    }
}
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup

- CreateDBClusterSnapshot
- CreateDBInstance
- DeleteDBCluster
- DeleteDBClusterParameterGroup
- DeleteDBInstance
- DescribeDBClusterParameterGroups
- DescribeDBClusterParameters
- DescribeDBClusterSnapshots
- DescribeDBClusters
- DescribeDBEngineVersions
- DescribeDBInstances
- DescribeOrderableDBInstanceOptions
- ModifyDBClusterParameterGroup

C++

SDK for C++

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

//! Routine which creates an Amazon Aurora DB cluster and demonstrates several
 operations
//! on that cluster.
/*!
 \sa gettingStartedWithDBClusters()
 \param clientConfiguration: AWS client configuration.
 \return bool: Successful completion.
 */
```

```
bool AwsDoc::Aurora::gettingStartedWithDBClusters(
        const Aws::Client::ClientConfiguration &clientConfig) {
    Aws::RDS::RDSClient client(clientConfig);

    printAsterisksLine();
    std::cout << "Welcome to the Amazon Relational Database Service (Amazon
 Aurora)"
              << std::endl;
    std::cout << "get started with DB clusters demo." << std::endl;
    printAsterisksLine();

    std::cout << "Checking for an existing DB cluster parameter group named '" <<
              CLUSTER_PARAMETER_GROUP_NAME << "'." << std::endl;
    Aws::String dbParameterGroupFamily("Undefined");
    bool parameterGroupFound = true;
    {
        // 1. Check if the DB cluster parameter group already exists.
        Aws::RDS::Model::DescribeDBClusterParameterGroupsRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);

        Aws::RDS::Model::DescribeDBClusterParameterGroupsOutcome outcome =
                client.DescribeDBClusterParameterGroups(request);

        if (outcome.IsSuccess()) {
            std::cout << "DB cluster parameter group named '" <<
                      CLUSTER_PARAMETER_GROUP_NAME << "' already exists." <<
 std::endl;
            dbParameterGroupFamily =
 outcome.GetResult().GetDBClusterParameterGroups()
[0].GetDBParameterGroupFamily();
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::RDS::RDSErrors::D_B_PARAMETER_GROUP_NOT_FOUND_FAULT) {
            std::cout << "DB cluster parameter group named '" <<
                      CLUSTER_PARAMETER_GROUP_NAME << "' does not exist." <<
 std::endl;
            parameterGroupFound = false;
        }
        else {
            std::cerr << "Error with Aurora::DescribeDBClusterParameterGroups. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
```

```
   }

   if (!parameterGroupFound) {
       Aws::Vector<Aws::RDS::Model::DBEngineVersion> engineVersions;

       // 2. Get available parameter group families for the specified engine.
       if (!getDBEngineVersions(DB_ENGINE, NO_PARAMETER_GROUP_FAMILY,
                                engineVersions, client)) {
           return false;
       }

       std::cout << "Getting available parameter group families for " <<
DB_ENGINE
                 << "."
                 << std::endl;
       std::vector<Aws::String> families;
       for (const Aws::RDS::Model::DBEngineVersion &version: engineVersions) {
           Aws::String family = version.GetDBParameterGroupFamily();
           if (std::find(families.begin(), families.end(), family) ==
               families.end()) {
               families.push_back(family);
               std::cout << "  " << families.size() << ": " << family <<
std::endl;
           }
       }

       int choice = askQuestionForIntRange("Which family do you want to use? ",
1,
                                           static_cast<int>(families.size()));
       dbParameterGroupFamily = families[choice - 1];
   }
   if (!parameterGroupFound) {
       // 3.  Create a DB cluster parameter group.
       Aws::RDS::Model::CreateDBClusterParameterGroupRequest request;
       request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
       request.SetDBParameterGroupFamily(dbParameterGroupFamily);
       request.SetDescription("Example cluster parameter group.");

       Aws::RDS::Model::CreateDBClusterParameterGroupOutcome outcome =
               client.CreateDBClusterParameterGroup(request);

       if (outcome.IsSuccess()) {
           std::cout << "The DB cluster parameter group was successfully
created."
```

```
                        << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::CreateDBClusterParameterGroup. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    }

    printAsterisksLine();
    std::cout << "Let's set some parameter values in your cluster parameter
 group."
              << std::endl;

    Aws::Vector<Aws::RDS::Model::Parameter> autoIncrementParameters;
    // 4.  Get the parameters in the DB cluster parameter group.
    if (!getDBCLusterParameters(CLUSTER_PARAMETER_GROUP_NAME,
 AUTO_INCREMENT_PREFIX,
                                NO_SOURCE,
                                autoIncrementParameters,
                                client)) {
        cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
        return false;
    }

    Aws::Vector<Aws::RDS::Model::Parameter> updateParameters;

    for (Aws::RDS::Model::Parameter &autoIncParameter: autoIncrementParameters) {
        if (autoIncParameter.GetIsModifiable() &&
            (autoIncParameter.GetDataType() == "integer")) {
            std::cout << "The " << autoIncParameter.GetParameterName()
                      << " is described as: " <<
                      autoIncParameter.GetDescription() << "." << std::endl;
            if (autoIncParameter.ParameterValueHasBeenSet()) {
                std::cout << "The current value is "
                          << autoIncParameter.GetParameterValue()
                          << "." << std::endl;
            }
            std::vector<int> splitValues = splitToInts(
                    autoIncParameter.GetAllowedValues(), '-');
            if (splitValues.size() == 2) {
                int newValue = askQuestionForIntRange(
                        Aws::String("Enter a new value between ") +
                        autoIncParameter.GetAllowedValues() + ": ",
                        splitValues[0], splitValues[1]);
                autoIncParameter.SetParameterValue(std::to_string(newValue));
                updateParameters.push_back(autoIncParameter);

            }
            else {
                std::cerr << "Error parsing " <<
 autoIncParameter.GetAllowedValues()
                          << std::endl;
            }
        }
    }

    {
        // 5.  Modify the auto increment parameters in the DB cluster parameter
 group.
        Aws::RDS::Model::ModifyDBClusterParameterGroupRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        request.SetParameters(updateParameters);

        Aws::RDS::Model::ModifyDBClusterParameterGroupOutcome outcome =
                client.ModifyDBClusterParameterGroup(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB cluster parameter group was successfully
 modified."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::ModifyDBClusterParameterGroup. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "You can get a list of parameters you've set by specifying a
 source of 'user'."
            << std::endl;

    Aws::Vector<Aws::RDS::Model::Parameter> userParameters;
    // 6.  Display the modified parameters in the DB cluster parameter group.
    if (!getDBCLusterParameters(CLUSTER_PARAMETER_GROUP_NAME, NO_NAME_PREFIX,
 "user",
                                userParameters,
                                client)) {
        cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
        return false;
    }

    for (const auto &userParameter: userParameters) {
        std::cout << "  " << userParameter.GetParameterName() << ", " <<
                  userParameter.GetDescription() << ", parameter value - "
                  << userParameter.GetParameterValue() << std::endl;
    }

    printAsterisksLine();
    std::cout << "Checking for an existing DB Cluster." << std::endl;

    Aws::RDS::Model::DBCluster dbCluster;
    // 7.  Check if the DB cluster already exists.
    if (!describeDBCluster(DB_CLUSTER_IDENTIFIER, dbCluster, client)) {
        cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
        return false;
    }

    Aws::String engineVersionName;
    Aws::String engineName;
    if (dbCluster.DBClusterIdentifierHasBeenSet()) {
        std::cout << "The DB cluster already exists." << std::endl;
        engineVersionName = dbCluster.GetEngineVersion();
        engineName = dbCluster.GetEngine();

    }
    else {
        std::cout << "Let's create a DB cluster." << std::endl;
        const Aws::String administratorName = askQuestion(
                "Enter an administrator username for the database: ");
        const Aws::String administratorPassword = askQuestion(
                "Enter a password for the administrator (at least 8 characters):
 ");
        Aws::Vector<Aws::RDS::Model::DBEngineVersion> engineVersions;

        // 8.  Get a list of engine versions for the parameter group family.
        if (!getDBEngineVersions(DB_ENGINE, dbParameterGroupFamily,
 engineVersions,
                                 client)) {
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
            return false;
        }

        std::cout << "The available engines for your parameter group family are:"
                  << std::endl;

        int index = 1;
        for (const Aws::RDS::Model::DBEngineVersion &engineVersion:
 engineVersions) {
            std::cout << "  " << index << ": " <<
 engineVersion.GetEngineVersion()
                      << std::endl;
            ++index;
        }
        int choice = askQuestionForIntRange("Which engine do you want to use? ",
 1,

 static_cast<int>(engineVersions.size()));
        const Aws::RDS::Model::DBEngineVersion engineVersion =
 engineVersions[choice -
                                                                              1];

        engineName = engineVersion.GetEngine();
        engineVersionName = engineVersion.GetEngineVersion();
        std::cout << "Creating a DB cluster named '" << DB_CLUSTER_IDENTIFIER
                  << "' and database '" << DB_NAME << "'.\n"
                  << "The DB cluster is configured to use your custom cluster
 parameter group '"
                  << CLUSTER_PARAMETER_GROUP_NAME << "', and \n"
                  << "selected engine version " <<
 engineVersion.GetEngineVersion()
                  << ".\nThis typically takes several minutes." << std::endl;

        Aws::RDS::Model::CreateDBClusterRequest request;
        request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        request.SetEngine(engineName);
        request.SetEngineVersion(engineVersionName);
        request.SetMasterUsername(administratorName);
        request.SetMasterUserPassword(administratorPassword);

        Aws::RDS::Model::CreateDBClusterOutcome outcome =
                client.CreateDBCluster(request);

        if (outcome.IsSuccess()) {
            std::cout << "The DB cluster creation has started."
                      << std::endl;
        }
        else {
            std::cerr << "Error with Aurora::CreateDBCluster. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, "", "", client);
            return false;
        }
    }

    std::cout << "Waiting for the DB cluster to become available." << std::endl;

    int counter = 0;
    // 11. Wait for the DB cluster to become available.
    do {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        ++counter;
        if (counter > 900) {
            std::cerr << "Wait for cluster to become available timed out ofter "
                      << counter
                      << " seconds." << std::endl;
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                             DB_CLUSTER_IDENTIFIER, "", client);
            return false;
        }

        dbCluster = Aws::RDS::Model::DBCluster();
        if (!describeDBCluster(DB_CLUSTER_IDENTIFIER, dbCluster, client)) {
            cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                             DB_CLUSTER_IDENTIFIER, "", client);
            return false;
        }

        if ((counter % 20) == 0) {
            std::cout << "Current DB cluster status is '"
                      << dbCluster.GetStatus()
                      << "' after " << counter << " seconds." << std::endl;
        }
    } while (dbCluster.GetStatus() != "available");
```

```
   if (dbCluster.GetStatus() == "available") {
       std::cout << "The DB cluster has been created." << std::endl;
   }

   printAsterisksLine();
   Aws::RDS::Model::DBInstance dbInstance;
   // 11.  Check if the DB instance already exists.
   if (!describeDBInstance(DB_INSTANCE_IDENTIFIER, dbInstance, client)) {
       cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, DB_CLUSTER_IDENTIFIER, "",
                        client);
       return false;
   }

   if (dbInstance.DbInstancePortHasBeenSet()) {
       std::cout << "The DB instance already exists." << std::endl;
   }
   else {
       std::cout << "Let's create a DB instance." << std::endl;

       Aws::String dbInstanceClass;
       // 12.  Get a list of instance classes.
       if (!chooseDBInstanceClass(engineName,
                                  engineVersionName,
                                  dbInstanceClass,
                                  client)) {
           cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, DB_CLUSTER_IDENTIFIER,
"",
                            client);
           return false;
       }

       std::cout << "Creating a DB instance named '" << DB_INSTANCE_IDENTIFIER
                 << "' with selected DB instance class '" << dbInstanceClass
                 << "'.\nThis typically takes several minutes." << std::endl;

       // 13. Create a DB instance.
       Aws::RDS::Model::CreateDBInstanceRequest request;
       request.SetDBInstanceIdentifier(DB_INSTANCE_IDENTIFIER);
       request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
       request.SetEngine(engineName);
       request.SetDBInstanceClass(dbInstanceClass);

       Aws::RDS::Model::CreateDBInstanceOutcome outcome =
               client.CreateDBInstance(request);

       if (outcome.IsSuccess()) {
           std::cout << "The DB instance creation has started."
                     << std::endl;
       }
       else {
           std::cerr << "Error with RDS::CreateDBInstance. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
           cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME, DB_CLUSTER_IDENTIFIER,
"",
                            client);
           return false;
       }
   }

   std::cout << "Waiting for the DB instance to become available." << std::endl;

   counter = 0;
   // 14. Wait for the DB instance to become available.
   do {
       std::this_thread::sleep_for(std::chrono::seconds(1));
       ++counter;
       if (counter > 900) {
           std::cerr << "Wait for instance to become available timed out ofter "
                     << counter
                     << " seconds." << std::endl;
           cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                            DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
           return false;
       }

       dbInstance = Aws::RDS::Model::DBInstance();
       if (!describeDBInstance(DB_INSTANCE_IDENTIFIER, dbInstance, client)) {
           cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                            DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
           return false;
       }

       if ((counter % 20) == 0) {
           std::cout << "Current DB instance status is '"
```

```
                     << dbInstance.GetDBInstanceStatus()
                     << "' after " << counter << " seconds." << std::endl;
       }
   } while (dbInstance.GetDBInstanceStatus() != "available");

   if (dbInstance.GetDBInstanceStatus() == "available") {
       std::cout << "The DB instance has been created." << std::endl;
   }

   // 15. Display the connection string that can be used to connect a 'mysql'
shell to the database.
   displayConnection(dbCluster);

   printAsterisksLine();

   if (askYesNoQuestion(
           "Do you want to create a snapshot of your DB cluster (y/n)? ")) {
       Aws::String snapshotID(DB_CLUSTER_IDENTIFIER + "-" +
                              Aws::String(Aws::Utils::UUID::RandomUUID()));
       {
           std::cout << "Creating a snapshot named " << snapshotID << "." <<
std::endl;
           std::cout << "This typically takes a few minutes." << std::endl;

           // 16. Create a snapshot of the DB cluster. (CreateDBClusterSnapshot)
           Aws::RDS::Model::CreateDBClusterSnapshotRequest request;
           request.SetDBClusterIdentifier(DB_CLUSTER_IDENTIFIER);
           request.SetDBClusterSnapshotIdentifier(snapshotID);

           Aws::RDS::Model::CreateDBClusterSnapshotOutcome outcome =
                   client.CreateDBClusterSnapshot(request);

           if (outcome.IsSuccess()) {
               std::cout << "Snapshot creation has started."
                         << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::CreateDBClusterSnapshot. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
               return false;
```

```
            }
        }

        std::cout << "Waiting for the snapshot to become available." <<
std::endl;

        Aws::RDS::Model::DBClusterSnapshot snapshot;
        counter = 0;
        do {
            std::this_thread::sleep_for(std::chrono::seconds(1));
            ++counter;
            if (counter > 600) {
                std::cerr << "Wait for snapshot to be available timed out ofter "
                          << counter
                          << " seconds." << std::endl;
                cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                 DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
                return false;
            }

            // 17. Wait for the snapshot to become available.
            Aws::RDS::Model::DescribeDBClusterSnapshotsRequest request;
            request.SetDBClusterSnapshotIdentifier(snapshotID);

            Aws::RDS::Model::DescribeDBClusterSnapshotsOutcome outcome =
                    client.DescribeDBClusterSnapshots(request);

            if (outcome.IsSuccess()) {
                snapshot = outcome.GetResult().GetDBClusterSnapshots()[0];
            }
            else {
                std::cerr << "Error with Aurora::DescribeDBClusterSnapshots. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                 DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
client);
                return false;
            }

            if ((counter % 20) == 0) {
                std::cout << "Current snapshot status is '"
                          << snapshot.GetStatus()
                          << "' after " << counter << " seconds." << std::endl;
            }
        } while (snapshot.GetStatus() != "available");

        if (snapshot.GetStatus() != "available") {
            std::cout << "A snapshot has been created." << std::endl;
        }
    }

    printAsterisksLine();

    bool result = true;
    if (askYesNoQuestion(
            "Do you want to delete the DB cluster, DB instance, and parameter
 group (y/n)? ")) {
        result = cleanUpResources(CLUSTER_PARAMETER_GROUP_NAME,
                                  DB_CLUSTER_IDENTIFIER, DB_INSTANCE_IDENTIFIER,
                                  client);
    }

    return result;
}

//! Routine which gets a DB cluster description.
/*!
 \sa describeDBCluster()
 \param dbClusterIdentifier: A DB cluster identifier.
 \param clusterResult: The 'DBCluster' object containing the description.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::describeDBCluster(const Aws::String &dbClusterIdentifier,
                                       Aws::RDS::Model::DBCluster &clusterResult,
                                       const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBClustersRequest request;
    request.SetDBClusterIdentifier(dbClusterIdentifier);

    Aws::RDS::Model::DescribeDBClustersOutcome outcome =
            client.DescribeDBClusters(request);

    bool result = true;
    if (outcome.IsSuccess()) {
        clusterResult = outcome.GetResult().GetDBClusters()[0];
    }
    else if (outcome.GetError().GetErrorType() !=
             Aws::RDS::RDSErrors::D_B_CLUSTER_NOT_FOUND_FAULT) {
        result = false;
        std::cerr << "Error with Aurora::GDescribeDBClusters. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
        // This example does not log an error if the DB cluster does not exist.
        // Instead, clusterResult is set to empty.
    else {
        clusterResult = Aws::RDS::Model::DBCluster();
    }

    return result;

}


//! Routine which gets DB parameters using the 'DescribeDBClusterParameters' api.
/*!
 \sa getDBCLusterParameters()
 \param parameterGroupName: The name of the cluster parameter group.
 \param namePrefix: Prefix string to filter results by parameter name.
 \param source: A source such as 'user', ignored if empty.
 \param parametersResult: Vector of 'Parameter' objects returned by the routine.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::getDBCLusterParameters(const Aws::String
 &parameterGroupName,
                                            const Aws::String &namePrefix,
                                            const Aws::String &source,

 Aws::Vector<Aws::RDS::Model::Parameter> &parametersResult,
                                            const Aws::RDS::RDSClient &client) {
    Aws::String marker; // The marker is used for pagination.
    do {
        Aws::RDS::Model::DescribeDBClusterParametersRequest request;
        request.SetDBClusterParameterGroupName(CLUSTER_PARAMETER_GROUP_NAME);
        if (!marker.empty()) {
            request.SetMarker(marker);
        }
        if (!source.empty()) {
            request.SetSource(source);
        }

        Aws::RDS::Model::DescribeDBClusterParametersOutcome outcome =
                client.DescribeDBClusterParameters(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::Parameter> &parameters =
                    outcome.GetResult().GetParameters();
            for (const Aws::RDS::Model::Parameter &parameter: parameters) {
                if (!namePrefix.empty()) {
                    if (parameter.GetParameterName().find(namePrefix) == 0) {
                        parametersResult.push_back(parameter);
                    }
                }
                else {
                    parametersResult.push_back(parameter);
                }
            }

            marker = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeDBClusterParameters. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!marker.empty());

    return true;
}


//! Routine which gets available DB engine versions for an engine name and
//! an optional parameter group family.
/*!
 \sa getDBEngineVersions()
 \param engineName: A DB engine name.
 \param parameterGroupFamily: A parameter group family name, ignored if empty.
 \param engineVersionsResult: Vector of 'DBEngineVersion' objects returned by the
 routine.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
```

```
bool AwsDoc::Aurora::getDBEngineVersions(const Aws::String &engineName,
                                         const Aws::String &parameterGroupFamily,

 Aws::Vector<Aws::RDS::Model::DBEngineVersion> &engineVersionsResult,
                                         const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBEngineVersionsRequest request;
    request.SetEngine(engineName);
    if (!parameterGroupFamily.empty()) {
        request.SetDBParameterGroupFamily(parameterGroupFamily);
    }

    engineVersionsResult.clear();
    Aws::String marker; // The marker is used for pagination.
    do {
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::RDS::Model::DescribeDBEngineVersionsOutcome outcome =
                client.DescribeDBEngineVersions(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::DBEngineVersion> &engineVersions =
                    outcome.GetResult().GetDBEngineVersions();

            engineVersionsResult.insert(engineVersionsResult.end(),
                                        engineVersions.begin(),
 engineVersions.end());
            marker  = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeDBEngineVersionsRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());

    return true;
}


//! Routine which gets a DB instance description.
/*!
 \sa describeDBCluster()
 \param dbInstanceIdentifier: A DB instance identifier.
 \param instanceResult: The 'DBInstance' object containing the description.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::describeDBInstance(const Aws::String &dbInstanceIdentifier,
                                        Aws::RDS::Model::DBInstance
 &instanceResult,
                                        const Aws::RDS::RDSClient &client) {
    Aws::RDS::Model::DescribeDBInstancesRequest request;
    request.SetDBInstanceIdentifier(dbInstanceIdentifier);

    Aws::RDS::Model::DescribeDBInstancesOutcome outcome =
            client.DescribeDBInstances(request);

    bool result = true;
    if (outcome.IsSuccess()) {
        instanceResult = outcome.GetResult().GetDBInstances()[0];
    }
    else if (outcome.GetError().GetErrorType() !=
             Aws::RDS::RDSErrors::D_B_INSTANCE_NOT_FOUND_FAULT) {
        result = false;
        std::cerr << "Error with Aurora::DescribeDBInstances. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
        // This example does not log an error if the DB instance does not exist.
        // Instead, instanceResult is set to empty.
    else {
        instanceResult = Aws::RDS::Model::DBInstance();
    }

    return result;
}


//! Routine which gets available DB instance classes, displays the list
//! to the user, and returns the user selection.
/*!
 \sa chooseDBInstanceClass()
 \param engineName: The DB engine name.
 \param engineVersion: The DB engine version.
 \param dbInstanceClass: String for DB instance class chosen by the user.
 \param client: 'RDSClient' instance.
 \return bool: Successful completion.
 */
bool AwsDoc::Aurora::chooseDBInstanceClass(const Aws::String &engine,
                                           const Aws::String &engineVersion,
                                           Aws::String &dbInstanceClass,
                                           const Aws::RDS::RDSClient &client) {
    std::vector<Aws::String> instanceClasses;
    Aws::String marker; // The marker is used for pagination.
    do {
        Aws::RDS::Model::DescribeOrderableDBInstanceOptionsRequest request;
        request.SetEngine(engine);
        request.SetEngineVersion(engineVersion);
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::RDS::Model::DescribeOrderableDBInstanceOptionsOutcome outcome =
                client.DescribeOrderableDBInstanceOptions(request);

        if (outcome.IsSuccess()) {
            const Aws::Vector<Aws::RDS::Model::OrderableDBInstanceOption>
 &options =
                    outcome.GetResult().GetOrderableDBInstanceOptions();
            for (const Aws::RDS::Model::OrderableDBInstanceOption &option:
 options) {
                const Aws::String &instanceClass = option.GetDBInstanceClass();
                if (std::find(instanceClasses.begin(), instanceClasses.end(),
                              instanceClass) == instanceClasses.end()) {
                    instanceClasses.push_back(instanceClass);
                }
            }
            marker = outcome.GetResult().GetMarker();
        }
        else {
            std::cerr << "Error with Aurora::DescribeOrderableDBInstanceOptions.
 "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            return false;
        }
    } while (!marker.empty());

    std::cout << "The available DB instance classes for your database engine
 are:"
```

```
              << std::endl;
    for (int i = 0; i < instanceClasses.size(); ++i) {
        std::cout << "   " << i + 1 << ": " << instanceClasses[i] << std::endl;
    }

    int choice = askQuestionForIntRange(
            "Which DB instance class do you want to use? ",
            1, static_cast<int>(instanceClasses.size()));
    dbInstanceClass = instanceClasses[choice - 1];
    return true;
}

//! Routine which deletes resources created by the scenario.
/*!
\sa cleanUpResources()
\param parameterGroupName: A parameter group name, this may be empty.
\param dbInstanceIdentifier: A DB instance identifier, this may be empty.
\param client: 'RDSClient' instance.
\return bool: Successful completion.
*/
bool AwsDoc::Aurora::cleanUpResources(const Aws::String &parameterGroupName,
                                      const Aws::String &dbClusterIdentifier,
                                      const Aws::String &dbInstanceIdentifier,
                                      const Aws::RDS::RDSClient &client) {
    bool result = true;
    bool instanceDeleting = false;
    bool clusterDeleting = false;
    if (!dbInstanceIdentifier.empty()) {
        {
            // 18. Delete the DB instance.
            Aws::RDS::Model::DeleteDBInstanceRequest request;
            request.SetDBInstanceIdentifier(dbInstanceIdentifier);
            request.SetSkipFinalSnapshot(true);
            request.SetDeleteAutomatedBackups(true);

            Aws::RDS::Model::DeleteDBInstanceOutcome outcome =
                    client.DeleteDBInstance(request);

            if (outcome.IsSuccess()) {
                std::cout << "DB instance deletion has started."
                          << std::endl;
                instanceDeleting = true;
                std::cout
```

```
                       << "Waiting for DB instance to delete before deleting the
parameter group."
                       << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::DeleteDBInstance. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               result = false;
           }
       }
   }

   if (!dbClusterIdentifier.empty()) {
       {
           // 19. Delete the DB cluster.
           Aws::RDS::Model::DeleteDBClusterRequest request;
           request.SetDBClusterIdentifier(dbClusterIdentifier);
           request.SetSkipFinalSnapshot(true);

           Aws::RDS::Model::DeleteDBClusterOutcome outcome =
                   client.DeleteDBCluster(request);

           if (outcome.IsSuccess()) {
               std::cout << "DB cluster deletion has started."
                         << std::endl;
               clusterDeleting = true;
               std::cout
                       << "Waiting for DB cluster to delete before deleting the
parameter group."
                       << std::endl;
               std::cout << "This may take a while." << std::endl;
           }
           else {
               std::cerr << "Error with Aurora::DeleteDBCluster. "
                         << outcome.GetError().GetMessage()
                         << std::endl;
               result = false;
           }
       }
   }
   int counter = 0;

   while (clusterDeleting || instanceDeleting) {
       // 20. Wait for the DB cluster and instance to be deleted.
       std::this_thread::sleep_for(std::chrono::seconds(1));
       ++counter;
       if (counter > 800) {
           std::cerr << "Wait for instance to delete timed out ofter " <<
counter
                     << " seconds." << std::endl;
           return false;
       }

       Aws::RDS::Model::DBInstance dbInstance = Aws::RDS::Model::DBInstance();
       if (instanceDeleting) {
           if (!describeDBInstance(dbInstanceIdentifier, dbInstance, client)) {
               return false;
           }
           instanceDeleting = dbInstance.DBInstanceIdentifierHasBeenSet();
       }

       Aws::RDS::Model::DBCluster dbCluster = Aws::RDS::Model::DBCluster();
       if (clusterDeleting) {
           if (!describeDBCluster(dbClusterIdentifier, dbCluster, client)) {
               return false;
           }

           clusterDeleting = dbCluster.DBClusterIdentifierHasBeenSet();
       }

       if ((counter % 20) == 0) {
           if (instanceDeleting) {
               std::cout << "Current DB instance status is '"
                         << dbInstance.GetDBInstanceStatus() << "." <<
std::endl;
           }

           if (clusterDeleting) {
               std::cout << "Current DB cluster status is '"
                         << dbCluster.GetStatus() << "." << std::endl;
           }
       }
   }

   if (!parameterGroupName.empty()) {
       // 21. Delete the DB cluster parameter group.
       Aws::RDS::Model::DeleteDBClusterParameterGroupRequest request;
       request.SetDBClusterParameterGroupName(parameterGroupName);

       Aws::RDS::Model::DeleteDBClusterParameterGroupOutcome outcome =
               client.DeleteDBClusterParameterGroup(request);

       if (outcome.IsSuccess()) {
           std::cout << "The DB parameter group was successfully deleted."
                     << std::endl;
       }
       else {
           std::cerr << "Error with Aurora::DeleteDBClusterParameterGroup. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
           result = false;
       }
   }

   return result;
}
```

- For API details, see the following topics in *AWS SDK for C++ API Reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster
  - DeleteDBClusterParameterGroup
  - DeleteDBInstance
  - DescribeDBClusterParameterGroups
  - DescribeDBClusterParameters
  - DescribeDBClusterSnapshots
  - DescribeDBClusters
  - DescribeDBEngineVersions
  - DescribeDBInstances
  - DescribeOrderableDBInstanceOptions
  - ModifyDBClusterParameterGroup

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
// GetStartedClusters is an interactive example that shows you how to use the AWS
 SDK for Go
// with Amazon Aurora to do the following:
//
// 1. Create a custom DB cluster parameter group and set parameter values.
// 2. Create an Aurora DB cluster that is configured to use the parameter group.
// 3. Create a DB instance in the DB cluster that contains a database.
// 4. Take a snapshot of the DB cluster.
// 5. Delete the DB instance, DB cluster, and parameter group.
type GetStartedClusters struct {
 sdkConfig  aws.Config
 dbClusters actions.DbClusters
 questioner demotools.IQuestioner
 helper     IScenarioHelper
 isTestRun  bool
}

// NewGetStartedClusters constructs a GetStartedClusters instance from a
 configuration.
// It uses the specified config to get an Amazon Relational Database Service
 (Amazon RDS)
// client and create wrappers for the actions used in the scenario.
func NewGetStartedClusters(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedClusters {
 auroraClient := rds.NewFromConfig(sdkConfig)
 return GetStartedClusters{
  sdkConfig:  sdkConfig,
  dbClusters: actions.DbClusters{AuroraClient: auroraClient},
  questioner: questioner,
  helper:     helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedClusters) Run(dbEngine string, parameterGroupName
 string,
 clusterName string, dbName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the Amazon Aurora DB Cluster demo.")
 log.Println(strings.Repeat("-", 88))

 parameterGroup := scenario.CreateParameterGroup(dbEngine, parameterGroupName)
 scenario.SetUserParameters(parameterGroupName)
 cluster := scenario.CreateCluster(clusterName, dbEngine, dbName, parameterGroup)
 scenario.helper.Pause(5)
 dbInstance := scenario.CreateInstance(cluster)
 scenario.DisplayConnection(cluster)
 scenario.CreateSnapshot(clusterName)
 scenario.Cleanup(dbInstance, cluster, parameterGroup)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// CreateParameterGroup shows how to get available engine versions for a
 specified
// database engine and create a DB cluster parameter group that is compatible
 with a
// selected engine family.
func (scenario GetStartedClusters) CreateParameterGroup(dbEngine string,
 parameterGroupName string) *types.DBClusterParameterGroup {

 log.Printf("Checking for an existing DB cluster parameter group named %v.\n",
  parameterGroupName)
 parameterGroup, err := scenario.dbClusters.GetParameterGroup(parameterGroupName)
 if err != nil {
  panic(err)
 }
 if parameterGroup == nil {
  log.Printf("Getting available database engine versions for %v.\n", dbEngine)
  engineVersions, err := scenario.dbClusters.GetEngineVersions(dbEngine, "")
  if err != nil {
   panic(err)
  }

  familySet := map[string]struct{}{}
  for _, family := range engineVersions {
   familySet[*family.DBParameterGroupFamily] = struct{}{}
  }
  var families []string
  for family := range familySet {
   families = append(families, family)
  }
  sort.Strings(families)
  familyIndex := scenario.questioner.AskChoice("Which family do you want to use?
\n", families)
  log.Println("Creating a DB cluster parameter group.")
  _, err = scenario.dbClusters.CreateParameterGroup(
   parameterGroupName, families[familyIndex], "Example parameter group.")
  if err != nil {
   panic(err)
  }
  parameterGroup, err = scenario.dbClusters.GetParameterGroup(parameterGroupName)
  if err != nil {
   panic(err)
  }
 }
 log.Printf("Parameter group %v:\n", *parameterGroup.DBParameterGroupFamily)
 log.Printf("\tName: %v\n", *parameterGroup.DBClusterParameterGroupName)
 log.Printf("\tARN: %v\n", *parameterGroup.DBClusterParameterGroupArn)
 log.Printf("\tFamily: %v\n", *parameterGroup.DBParameterGroupFamily)
 log.Printf("\tDescription: %v\n", *parameterGroup.Description)
 log.Println(strings.Repeat("-", 88))
 return parameterGroup

}

// SetUserParameters shows how to get the parameters contained in a custom
 parameter
// group and update some of the parameter values in the group.
```

```
func (scenario GetStartedClusters) SetUserParameters(parameterGroupName string) {
 log.Println("Let's set some parameter values in your parameter group.")
 dbParameters, err := scenario.dbClusters.GetParameters(parameterGroupName, "")
 if err != nil {
  panic(err)
 }
 var updateParams []types.Parameter
 for _, dbParam := range dbParameters {
  if strings.HasPrefix(*dbParam.ParameterName, "auto_increment") &&
   dbParam.IsModifiable && *dbParam.DataType == "integer" {
   log.Printf("The %v parameter is described as:\n\t%v",
    *dbParam.ParameterName, *dbParam.Description)
   rangeSplit := strings.Split(*dbParam.AllowedValues, "-")
   lower, _ := strconv.Atoi(rangeSplit[0])
   upper, _ := strconv.Atoi(rangeSplit[1])
   newValue := scenario.questioner.AskInt(
    fmt.Sprintf("Enter a value between %v and %v:", lower, upper),
    demotools.InIntRange{Lower: lower, Upper: upper})
   dbParam.ParameterValue = aws.String(strconv.Itoa(newValue))
   updateParams = append(updateParams, dbParam)
  }
 }
 err = scenario.dbClusters.UpdateParameters(parameterGroupName, updateParams)
 if err != nil {
  panic(err)
 }
 log.Println("You can get a list of parameters you've set by specifying a source
 of 'user'.")
 userParameters, err := scenario.dbClusters.GetParameters(parameterGroupName,
 "user")
 if err != nil {
  panic(err)
 }
 log.Println("Here are the parameters you've set:")
 for _, param := range userParameters {
  log.Printf("\t%v: %v\n", *param.ParameterName, *param.ParameterValue)
 }
 log.Println(strings.Repeat("-", 88))
}

// CreateCluster shows how to create an Aurora DB cluster that contains a
 database
// of a specified type. The database is also configured to use a custom DB
 cluster
```

```
// parameter group.
func (scenario GetStartedClusters) CreateCluster(clusterName string, dbEngine
 string,
 dbName string, parameterGroup *types.DBClusterParameterGroup) *types.DBCluster {

 log.Println("Checking for an existing DB cluster.")
 cluster, err := scenario.dbClusters.GetDbCluster(clusterName)
 if err != nil {
  panic(err)
 }
 if cluster == nil {
  adminUsername := scenario.questioner.Ask(
   "Enter an administrator user name for the database: ", demotools.NotEmpty{})
  adminPassword := scenario.questioner.Ask(
   "Enter a password for the administrator (at least 8 characters): ",
 demotools.NotEmpty{})
  engineVersions, err := scenario.dbClusters.GetEngineVersions(dbEngine,
 *parameterGroup.DBParameterGroupFamily)
  if err != nil {
   panic(err)
  }
  var engineChoices []string
  for _, engine := range engineVersions {
   engineChoices = append(engineChoices, *engine.EngineVersion)
  }
  log.Println("The available engines for your parameter group are:")
  engineIndex := scenario.questioner.AskChoice("Which engine do you want to use?
\n", engineChoices)
  log.Printf("Creating DB cluster %v and database %v.\n", clusterName, dbName)
  log.Printf("The DB cluster is configured to use\nyour custom parameter group %v
\n",
   *parameterGroup.DBClusterParameterGroupName)
  log.Printf("and selected engine %v.\n", engineChoices[engineIndex])
  log.Println("This typically takes several minutes.")
  cluster, err = scenario.dbClusters.CreateDbCluster(
   clusterName, *parameterGroup.DBClusterParameterGroupName, dbName, dbEngine,
   engineChoices[engineIndex], adminUsername, adminPassword)
  if err != nil {
   panic(err)
  }
  for *cluster.Status != "available" {
   scenario.helper.Pause(30)
   cluster, err = scenario.dbClusters.GetDbCluster(clusterName)
   if err != nil {
    panic(err)
   }
   log.Println("Cluster created and available.")
  }
 }
 log.Println("Cluster data:")
 log.Printf("\tDBClusterIdentifier: %v\n", *cluster.DBClusterIdentifier)
 log.Printf("\tARN: %v\n", *cluster.DBClusterArn)
 log.Printf("\tStatus: %v\n", *cluster.Status)
 log.Printf("\tEngine: %v\n", *cluster.Engine)
 log.Printf("\tEngine version: %v\n", *cluster.EngineVersion)
 log.Printf("\tDBClusterParameterGroup: %v\n", *cluster.DBClusterParameterGroup)
 log.Printf("\tEngineMode: %v\n", *cluster.EngineMode)
 log.Println(strings.Repeat("-", 88))
 return cluster
}

// CreateInstance shows how to create a DB instance in an existing Aurora DB
 cluster.
// A new DB cluster contains no DB instances, so you must add one. The first DB
 instance
// that is added to a DB cluster defaults to a read-write DB instance.
func (scenario GetStartedClusters) CreateInstance(cluster *types.DBCluster)
 *types.DBInstance {
 log.Println("Checking for an existing database instance.")
 dbInstance, err := scenario.dbClusters.GetInstance(*cluster.DBClusterIdentifier)
 if err != nil {
  panic(err)
 }
 if dbInstance == nil {
  log.Println("Let's create a database instance in your DB cluster.")
  log.Println("First, choose a DB instance type:")
  instOpts, err := scenario.dbClusters.GetOrderableInstances(
   *cluster.Engine, *cluster.EngineVersion)
  if err != nil {
   panic(err)
  }
  var instChoices []string
  for _, opt := range instOpts {
   instChoices = append(instChoices, *opt.DBInstanceClass)
  }
  instIndex := scenario.questioner.AskChoice(
   "Which DB instance class do you want to use?\n", instChoices)
  log.Println("Creating a database instance. This typically takes several
 minutes.")
  dbInstance, err = scenario.dbClusters.CreateInstanceInCluster(
   *cluster.DBClusterIdentifier, *cluster.DBClusterIdentifier, *cluster.Engine,
   instChoices[instIndex])
  if err != nil {
   panic(err)
  }
  for *dbInstance.DBInstanceStatus != "available" {
   scenario.helper.Pause(30)
   dbInstance, err =
 scenario.dbClusters.GetInstance(*cluster.DBClusterIdentifier)
   if err != nil {
    panic(err)
   }
  }
 }
 log.Println("Instance data:")
 log.Printf("\tDBInstanceIdentifier: %v\n", *dbInstance.DBInstanceIdentifier)
 log.Printf("\tARN: %v\n", *dbInstance.DBInstanceArn)
 log.Printf("\tStatus: %v\n", *dbInstance.DBInstanceStatus)
 log.Printf("\tEngine: %v\n", *dbInstance.Engine)
 log.Printf("\tEngine version: %v\n", *dbInstance.EngineVersion)
 log.Println(strings.Repeat("-", 88))
 return dbInstance
}

// DisplayConnection displays connection information about an Aurora DB cluster
 and tips
// on how to connect to it.
func (scenario GetStartedClusters) DisplayConnection(cluster *types.DBCluster) {
 log.Println(
  "You can now connect to your database using your favorite MySql client.\n" +
   "One way to connect is by using the 'mysql' shell on an Amazon EC2 instance\n"
 +
   "that is running in the same VPC as your database cluster. Pass the endpoint,
\n" +
   "port, and administrator user name to 'mysql' and enter your password\n" +
   "when prompted:")
 log.Printf("\n\tmysql -h %v -P %v -u %v -p\n",
  *cluster.Endpoint, *cluster.Port, *cluster.MasterUsername)
 log.Println("For more information, see the User Guide for Aurora:\n" +
  "\thttps://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/
CHAP_GettingStartedAurora.CreatingConnecting.Aurora.html#CHAP_GettingStartedAurora.Aurora
```

```
 log.Println(strings.Repeat("-", 88))
}

// CreateSnapshot shows how to create a DB cluster snapshot and wait until it's
 available.
func (scenario GetStartedClusters) CreateSnapshot(clusterName string) {
 if scenario.questioner.AskBool(
  "Do you want to create a snapshot of your DB cluster (y/n)? ", "y") {
  snapshotId := fmt.Sprintf("%v-%v", clusterName, scenario.helper.UniqueId())
  log.Printf("Creating a snapshot named %v. This typically takes a few minutes.
\n", snapshotId)
  snapshot, err := scenario.dbClusters.CreateClusterSnapshot(clusterName,
 snapshotId)
  if err != nil {
   panic(err)
  }
  for *snapshot.Status != "available" {
   scenario.helper.Pause(30)
   snapshot, err = scenario.dbClusters.GetClusterSnapshot(snapshotId)
   if err != nil {
    panic(err)
   }
  }
  log.Println("Snapshot data:")
  log.Printf("\tDBClusterSnapshotIdentifier: %v\n",
 *snapshot.DBClusterSnapshotIdentifier)
  log.Printf("\tARN: %v\n", *snapshot.DBClusterSnapshotArn)
  log.Printf("\tStatus: %v\n", *snapshot.Status)
  log.Printf("\tEngine: %v\n", *snapshot.Engine)
  log.Printf("\tEngine version: %v\n", *snapshot.EngineVersion)
  log.Printf("\tDBClusterIdentifier: %v\n", *snapshot.DBClusterIdentifier)
  log.Printf("\tSnapshotCreateTime: %v\n", *snapshot.SnapshotCreateTime)
  log.Println(strings.Repeat("-", 88))
 }
}

// Cleanup shows how to clean up a DB instance, DB cluster, and DB cluster
 parameter group.
// Before the DB cluster parameter group can be deleted, all associated DB
 instances and
// DB clusters must first be deleted.
func (scenario GetStartedClusters) Cleanup(dbInstance *types.DBInstance, cluster
 *types.DBCluster,
 parameterGroup *types.DBClusterParameterGroup) {
```

```
 if scenario.questioner.AskBool(
  "\nDo you want to delete the database instance, DB cluster, and parameter group
 (y/n)? ", "y") {
  log.Printf("Deleting database instance %v.\n",
 *dbInstance.DBInstanceIdentifier)
  err := scenario.dbClusters.DeleteInstance(*dbInstance.DBInstanceIdentifier)
  if err != nil {
   panic(err)
  }
  log.Printf("Deleting database cluster %v.\n", *cluster.DBClusterIdentifier)
  err = scenario.dbClusters.DeleteDbCluster(*cluster.DBClusterIdentifier)
  if err != nil {
   panic(err)
  }
  log.Println(
   "Waiting for the DB instance and DB cluster to delete. This typically takes
 several minutes.")
  for dbInstance != nil || cluster != nil {
   scenario.helper.Pause(30)
   if dbInstance != nil {
    dbInstance, err =
 scenario.dbClusters.GetInstance(*dbInstance.DBInstanceIdentifier)
    if err != nil {
     panic(err)
    }
   }
   if cluster != nil {
    cluster, err = scenario.dbClusters.GetDbCluster(*cluster.DBClusterIdentifier)
    if err != nil {
     panic(err)
    }
   }
  }
  log.Printf("Deleting parameter group %v.",
 *parameterGroup.DBClusterParameterGroupName)
  err =
 scenario.dbClusters.DeleteParameterGroup(*parameterGroup.DBClusterParameterGroupName)
  if err != nil {
   panic(err)
  }
 }
}
```

Define functions that are called by the scenario to manage Aurora actions.

```
type DbClusters struct {
 AuroraClient *rds.Client
}



// GetParameterGroup gets a DB cluster parameter group by name.
func (clusters *DbClusters) GetParameterGroup(parameterGroupName string) (
 *types.DBClusterParameterGroup, error) {
 output, err := clusters.AuroraClient.DescribeDBClusterParameterGroups(
  context.TODO(), &rds.DescribeDBClusterParameterGroupsInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
  })
 if err != nil {
  var notFoundError *types.DBParameterGroupNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("Parameter group %v does not exist.\n", parameterGroupName)
   err = nil
  } else {
   log.Printf("Error getting parameter group %v: %v\n", parameterGroupName, err)
  }
  return nil, err
 } else {
  return &output.DBClusterParameterGroups[0], err
 }
}



// CreateParameterGroup creates a DB cluster parameter group that is based on the
 specified
// parameter group family.
func (clusters *DbClusters) CreateParameterGroup(
 parameterGroupName string, parameterGroupFamily string, description string) (
 *types.DBClusterParameterGroup, error) {

 output, err :=
 clusters.AuroraClient.CreateDBClusterParameterGroup(context.TODO(),
  &rds.CreateDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   DBParameterGroupFamily:      aws.String(parameterGroupFamily),
   Description:                 aws.String(description),
  })
 if err != nil {
  log.Printf("Couldn't create parameter group %v: %v\n", parameterGroupName, err)
  return nil, err
 } else {
  return output.DBClusterParameterGroup, err
 }
}



// DeleteParameterGroup deletes the named DB cluster parameter group.
func (clusters *DbClusters) DeleteParameterGroup(parameterGroupName string) error
 {
 _, err := clusters.AuroraClient.DeleteDBClusterParameterGroup(context.TODO(),
  &rds.DeleteDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
  })
 if err != nil {
  log.Printf("Couldn't delete parameter group %v: %v\n", parameterGroupName, err)
  return err
 } else {
  return nil
 }
}



// GetParameters gets the parameters that are contained in a DB cluster parameter
 group.
func (clusters *DbClusters) GetParameters(parameterGroupName string, source
 string) (
 []types.Parameter, error) {

 var output *rds.DescribeDBClusterParametersOutput
 var params []types.Parameter
 var err error
 parameterPaginator :=
 rds.NewDescribeDBClusterParametersPaginator(clusters.AuroraClient,
  &rds.DescribeDBClusterParametersInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   Source:                      aws.String(source),
  })
 for parameterPaginator.HasMorePages() {
  output, err = parameterPaginator.NextPage(context.TODO())
  if err != nil {
   log.Printf("Couldn't get paramaeters for %v: %v\n", parameterGroupName, err)
   break
  } else {
   params = append(params, output.Parameters...)
  }
 }
 return params, err
}



// UpdateParameters updates parameters in a named DB cluster parameter group.
func (clusters *DbClusters) UpdateParameters(parameterGroupName string, params
 []types.Parameter) error {
 _, err := clusters.AuroraClient.ModifyDBClusterParameterGroup(context.TODO(),
  &rds.ModifyDBClusterParameterGroupInput{
   DBClusterParameterGroupName: aws.String(parameterGroupName),
   Parameters:                  params,
  })
 if err != nil {
  log.Printf("Couldn't update parameters in %v: %v\n", parameterGroupName, err)
  return err
 } else {
  return nil
 }
}



// GetDbCluster gets data about an Aurora DB cluster.
func (clusters *DbClusters) GetDbCluster(clusterName string) (*types.DBCluster,
 error) {
 output, err := clusters.AuroraClient.DescribeDBClusters(context.TODO(),
  &rds.DescribeDBClustersInput{
   DBClusterIdentifier: aws.String(clusterName),
  })
 if err != nil {
  var notFoundError *types.DBClusterNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("DB cluster %v does not exist.\n", clusterName)
   err = nil
  } else {
   log.Printf("Couldn't get DB cluster %v: %v\n", clusterName, err)
  }
  return nil, err
 } else {
  return &output.DBClusters[0], err
 }
}



// CreateDbCluster creates a DB cluster that is configured to use the specified
 parameter group.
// The newly created DB cluster contains a database that uses the specified
 engine and
// engine version.
func (clusters *DbClusters) CreateDbCluster(clusterName string,
 parameterGroupName string,
 dbName string, dbEngine string, dbEngineVersion string, adminName string,
 adminPassword string) (
 *types.DBCluster, error) {

 output, err := clusters.AuroraClient.CreateDBCluster(context.TODO(),
 &rds.CreateDBClusterInput{
  DBClusterIdentifier:         aws.String(clusterName),
  Engine:                      aws.String(dbEngine),
  DBClusterParameterGroupName: aws.String(parameterGroupName),
  DatabaseName:                aws.String(dbName),
  EngineVersion:               aws.String(dbEngineVersion),
  MasterUserPassword:          aws.String(adminPassword),
  MasterUsername:              aws.String(adminName),
 })
 if err != nil {
  log.Printf("Couldn't create DB cluster %v: %v\n", clusterName, err)
  return nil, err
 } else {
  return output.DBCluster, err
 }
}



// DeleteDbCluster deletes a DB cluster without keeping a final snapshot.
```

```
func (clusters *DbClusters) DeleteDbCluster(clusterName string) error {
 _, err := clusters.AuroraClient.DeleteDBCluster(context.TODO(),
 &rds.DeleteDBClusterInput{
  DBClusterIdentifier: aws.String(clusterName),
  SkipFinalSnapshot:   true,
 })
 if err != nil {
  log.Printf("Couldn't delete DB cluster %v: %v\n", clusterName, err)
  return err
 } else {
  return nil
 }
}



// CreateClusterSnapshot creates a snapshot of a DB cluster.
func (clusters *DbClusters) CreateClusterSnapshot(clusterName string,
 snapshotName string) (
 *types.DBClusterSnapshot, error) {
 output, err := clusters.AuroraClient.CreateDBClusterSnapshot(context.TODO(),
 &rds.CreateDBClusterSnapshotInput{
  DBClusterIdentifier:         aws.String(clusterName),
  DBClusterSnapshotIdentifier: aws.String(snapshotName),
 })
 if err != nil {
  log.Printf("Couldn't create snapshot %v: %v\n", snapshotName, err)
  return nil, err
 } else {
  return output.DBClusterSnapshot, nil
 }
}



// GetClusterSnapshot gets a DB cluster snapshot.
func (clusters *DbClusters) GetClusterSnapshot(snapshotName string)
 (*types.DBClusterSnapshot, error) {
 output, err := clusters.AuroraClient.DescribeDBClusterSnapshots(context.TODO(),
  &rds.DescribeDBClusterSnapshotsInput{
   DBClusterSnapshotIdentifier: aws.String(snapshotName),
  })
 if err != nil {
  log.Printf("Couldn't get snapshot %v: %v\n", snapshotName, err)
  return nil, err
 } else {
  return &output.DBClusterSnapshots[0], nil
 }
}



// CreateInstanceInCluster creates a database instance in an existing DB cluster.
 The first database that is
// created defaults to a read-write DB instance.
func (clusters *DbClusters) CreateInstanceInCluster(clusterName string,
 instanceName string,
 dbEngine string, dbInstanceClass string) (*types.DBInstance, error) {
 output, err := clusters.AuroraClient.CreateDBInstance(context.TODO(),
 &rds.CreateDBInstanceInput{
  DBInstanceIdentifier: aws.String(instanceName),
  DBClusterIdentifier:  aws.String(clusterName),
  Engine:               aws.String(dbEngine),
  DBInstanceClass:      aws.String(dbInstanceClass),
 })
 if err != nil {
  log.Printf("Couldn't create instance %v: %v\n", instanceName, err)
  return nil, err
 } else {
  return output.DBInstance, nil
 }
}



// GetInstance gets data about a DB instance.
func (clusters *DbClusters) GetInstance(instanceName string) (
 *types.DBInstance, error) {
 output, err := clusters.AuroraClient.DescribeDBInstances(context.TODO(),
  &rds.DescribeDBInstancesInput{
   DBInstanceIdentifier: aws.String(instanceName),
  })
 if err != nil {
  var notFoundError *types.DBInstanceNotFoundFault
  if errors.As(err, &notFoundError) {
   log.Printf("DB instance %v does not exist.\n", instanceName)
   err = nil
  } else {
   log.Printf("Couldn't get instance %v: %v\n", instanceName, err)
  }
  return nil, err
 } else {
  return &output.DBInstances[0], nil
 }
}



// DeleteInstance deletes a DB instance.
func (clusters *DbClusters) DeleteInstance(instanceName string) error {
 _, err := clusters.AuroraClient.DeleteDBInstance(context.TODO(),
 &rds.DeleteDBInstanceInput{
  DBInstanceIdentifier:   aws.String(instanceName),
  SkipFinalSnapshot:      true,
  DeleteAutomatedBackups: aws.Bool(true),
 })
 if err != nil {
  log.Printf("Couldn't delete instance %v: %v\n", instanceName, err)
  return err
 } else {
  return nil
 }
}



// GetEngineVersions gets database engine versions that are available for the
 specified engine
// and parameter group family.
func (clusters *DbClusters) GetEngineVersions(engine string, parameterGroupFamily
 string) (
 []types.DBEngineVersion, error) {
 output, err := clusters.AuroraClient.DescribeDBEngineVersions(context.TODO(),
  &rds.DescribeDBEngineVersionsInput{
   Engine:                 aws.String(engine),
   DBParameterGroupFamily: aws.String(parameterGroupFamily),
  })
 if err != nil {
  log.Printf("Couldn't get engine versions for %v: %v\n", engine, err)
  return nil, err
 } else {
  return output.DBEngineVersions, nil
 }
}



// GetOrderableInstances uses a paginator to get DB instance options that can be
 used to create DB instances that are
// compatible with a set of specifications.
func (clusters *DbClusters) GetOrderableInstances(engine string, engineVersion
 string) (
 []types.OrderableDBInstanceOption, error) {

 var output *rds.DescribeOrderableDBInstanceOptionsOutput
 var instances []types.OrderableDBInstanceOption
 var err error
 orderablePaginator :=
 rds.NewDescribeOrderableDBInstanceOptionsPaginator(clusters.AuroraClient,
  &rds.DescribeOrderableDBInstanceOptionsInput{
   Engine:        aws.String(engine),
   EngineVersion: aws.String(engineVersion),
  })
 for orderablePaginator.HasMorePages() {
  output, err = orderablePaginator.NextPage(context.TODO())
  if err != nil {
   log.Printf("Couldn't get orderable DB instances: %v\n", err)
   break
  } else {
   instances = append(instances, output.OrderableDBInstanceOptions...)
  }
 }
 return instances, err
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster

- DeleteDBClusterParameterGroup
- DeleteDBInstance
- DescribeDBClusterParameterGroups
- DescribeDBClusterParameters
- DescribeDBClusterSnapshots
- DescribeDBClusters
- DescribeDBEngineVersions
- DescribeDBInstances
- DescribeOrderableDBInstanceOptions
- ModifyDBClusterParameterGroup

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
 * Before running this Java (v2) code example, set up your development
 * environment, including your credentials.
 *
 * For more information, see the following documentation topic:
 *
 * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 * This example requires an AWS Secrets Manager secret that contains the
 * database credentials. If you do not create a
 * secret, this example will not work. For details, see:
 *
 * https://docs.aws.amazon.com/secretsmanager/latest/userguide/integrating_how-
services-use-secrets_RS.html
 *
 * This Java example performs the following tasks:
 *
 * 1. Gets available engine families for Amazon Aurora MySQL-Compatible Edition
 * by calling the DescribeDbEngineVersions(Engine='aurora-mysql') method.
 * 2. Selects an engine family and creates a custom DB cluster parameter group
 * by invoking the describeDBClusterParameters method.
 * 3. Gets the parameter groups by invoking the describeDBClusterParameterGroups
 * method.
 * 4. Gets parameters in the group by invoking the describeDBClusterParameters
 * method.
 * 5. Modifies the auto_increment_offset parameter by invoking the
 * modifyDbClusterParameterGroupRequest method.
 * 6. Gets and displays the updated parameters.
 * 7. Gets a list of allowed engine versions by invoking the
 * describeDbEngineVersions method.
 * 8. Creates an Aurora DB cluster database cluster that contains a MySQL
 * database.
 * 9. Waits for DB instance to be ready.
 * 10. Gets a list of instance classes available for the selected engine.
 * 11. Creates a database instance in the cluster.
 * 12. Waits for DB instance to be ready.
 * 13. Creates a snapshot.
 * 14. Waits for DB snapshot to be ready.
 * 15. Deletes the DB cluster.
 * 16. Deletes the DB cluster group.
 */
public class AuroraScenario {
    public static long sleepTime = 20;
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = "\n" +
                "Usage:\n" +
                "    <dbClusterGroupName> <dbParameterGroupFamily>
 <dbInstanceClusterIdentifier> <dbInstanceIdentifier> <dbName>
 <dbSnapshotIdentifier><secretName>"
                +
                "Where:\n" +
                "    dbClusterGroupName - The name of the DB cluster parameter
 group. \n" +
                "    dbParameterGroupFamily - The DB cluster parameter group
 family name (for example, aurora-mysql5.7). \n"
                +
                "    dbInstanceClusterIdentifier - The instance cluster 
 identifier value.\n" +
                "    dbInstanceIdentifier - The database instance identifier.\n" 
 +
                "    dbName - The database name.\n" +
                "    dbSnapshotIdentifier - The snapshot identifier.\n" +
                "    secretName - The name of the AWS Secrets Manager secret that 
 contains the database credentials\"\n";
        ;

        if (args.length != 7) {
            System.out.println(usage);
            System.exit(1);
        }

        String dbClusterGroupName = args[0];
        String dbParameterGroupFamily = args[1];
        String dbInstanceClusterIdentifier = args[2];
        String dbInstanceIdentifier = args[3];
        String dbName = args[4];
        String dbSnapshotIdentifier = args[5];
        String secretName = args[6];

        // Retrieve the database credentials using AWS Secrets Manager.
        Gson gson = new Gson();
        User user = gson.fromJson(String.valueOf(getSecretValues(secretName)), 
 User.class);
        String username = user.getUsername();
        String userPassword = user.getPassword();

        Region region = Region.US_WEST_2;
        RdsClient rdsClient = RdsClient.builder()
                .region(region)
                .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the Amazon Aurora example scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("1. Return a list of the available DB engines");
        describeDBEngines(rdsClient);
        System.out.println(DASHES);
```

```
        System.out.println(DASHES);
        System.out.println("2. Create a custom parameter group");
        createDBClusterParameterGroup(rdsClient, dbClusterGroupName,
dbParameterGroupFamily);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. Get the parameter group");
        describeDbClusterParameterGroups(rdsClient, dbClusterGroupName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Get the parameters in the group");
        describeDbClusterParameters(rdsClient, dbClusterGroupName, 0);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Modify the auto_increment_offset parameter");
        modifyDBClusterParas(rdsClient, dbClusterGroupName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Display the updated parameter value");
        describeDbClusterParameters(rdsClient, dbClusterGroupName, -1);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Get a list of allowed engine versions");
        getAllowedEngines(rdsClient, dbParameterGroupFamily);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("8. Create an Aurora DB cluster database");
        String arnClusterVal = createDBCluster(rdsClient, dbClusterGroupName,
dbName, dbInstanceClusterIdentifier,
                username, userPassword);
        System.out.println("The ARN of the cluster is " + arnClusterVal);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("9. Wait for DB instance to be ready");
        waitForInstanceReady(rdsClient, dbInstanceClusterIdentifier);
        System.out.println(DASHES);
        System.out.println(DASHES);
        System.out.println("10. Get a list of instance classes available for the
selected engine");
        String instanceClass = getListInstanceClasses(rdsClient);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("11. Create a database instance in the cluster.");
        String clusterDBARN = createDBInstanceCluster(rdsClient,
dbInstanceIdentifier, dbInstanceClusterIdentifier,
                instanceClass);
        System.out.println("The ARN of the database is " + clusterDBARN);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("12. Wait for DB instance to be ready");
        waitDBInstanceReady(rdsClient, dbInstanceIdentifier);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("13. Create a snapshot");
        createDBClusterSnapshot(rdsClient, dbInstanceClusterIdentifier,
dbSnapshotIdentifier);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("14. Wait for DB snapshot to be ready");
        waitForSnapshotReady(rdsClient, dbSnapshotIdentifier,
dbInstanceClusterIdentifier);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("14. Delete the DB instance");
        deleteDatabaseInstance(rdsClient, dbInstanceIdentifier);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("15. Delete the DB cluster");
        deleteCluster(rdsClient, dbInstanceClusterIdentifier);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("16. Delete the DB cluster group");
        deleteDBClusterGroup(rdsClient, dbClusterGroupName, clusterDBARN);
```

```
       System.out.println(DASHES);

       System.out.println(DASHES);
       System.out.println("The Scenario has successfully completed.");
       System.out.println(DASHES);
       rdsClient.close();
   }

   private static SecretsManagerClient getSecretClient() {
       Region region = Region.US_WEST_2;
       return SecretsManagerClient.builder()
               .region(region)

.credentialsProvider(EnvironmentVariableCredentialsProvider.create())
               .build();
   }

   private static String getSecretValues(String secretName) {
       SecretsManagerClient secretClient = getSecretClient();
       GetSecretValueRequest valueRequest = GetSecretValueRequest.builder()
               .secretId(secretName)
               .build();

       GetSecretValueResponse valueResponse =
secretClient.getSecretValue(valueRequest);
       return valueResponse.secretString();
   }

   public static void deleteDBClusterGroup(RdsClient rdsClient, String
dbClusterGroupName, String clusterDBARN)
           throws InterruptedException {
       try {
           boolean isDataDel = false;
           boolean didFind;
           String instanceARN;

           // Make sure that the database has been deleted.
           while (!isDataDel) {
               DescribeDbInstancesResponse response =
rdsClient.describeDBInstances();
               List<DBInstance> instanceList = response.dbInstances();
               int listSize = instanceList.size();
               didFind = false;
               int index = 1;
               for (DBInstance instance : instanceList) {
                   instanceARN = instance.dbInstanceArn();
                   if (instanceARN.compareTo(clusterDBARN) == 0) {
                       System.out.println(clusterDBARN + " still exists");
                       didFind = true;
                   }
                   if ((index == listSize) && (!didFind)) {
                       // Went through the entire list and did not find the
database ARN.
                       isDataDel = true;
                   }
                   Thread.sleep(sleepTime * 1000);
                   index++;
               }
           }

           DeleteDbClusterParameterGroupRequest clusterParameterGroupRequest =
DeleteDbClusterParameterGroupRequest
                   .builder()
                   .dbClusterParameterGroupName(dbClusterGroupName)
                   .build();


rdsClient.deleteDBClusterParameterGroup(clusterParameterGroupRequest);
           System.out.println(dbClusterGroupName + " was deleted.");

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void deleteCluster(RdsClient rdsClient, String
dbInstanceClusterIdentifier) {
       try {
           DeleteDbClusterRequest deleteDbClusterRequest =
DeleteDbClusterRequest.builder()
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .skipFinalSnapshot(true)
                   .build();

           rdsClient.deleteDBCluster(deleteDbClusterRequest);
           System.out.println(dbInstanceClusterIdentifier + " was deleted!");
       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void deleteDatabaseInstance(RdsClient rdsClient, String
dbInstanceIdentifier) {
       try {
           DeleteDbInstanceRequest deleteDbInstanceRequest =
DeleteDbInstanceRequest.builder()
                   .dbInstanceIdentifier(dbInstanceIdentifier)
                   .deleteAutomatedBackups(true)
                   .skipFinalSnapshot(true)
                   .build();

           DeleteDbInstanceResponse response =
rdsClient.deleteDBInstance(deleteDbInstanceRequest);
           System.out.println("The status of the database is " +
response.dbInstance().dbInstanceStatus());

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void waitForSnapshotReady(RdsClient rdsClient, String
dbSnapshotIdentifier,
           String dbInstanceClusterIdentifier) {
       try {
           boolean snapshotReady = false;
           String snapshotReadyStr;
           System.out.println("Waiting for the snapshot to become available.");

           DescribeDbClusterSnapshotsRequest snapshotsRequest =
DescribeDbClusterSnapshotsRequest.builder()
                   .dbClusterSnapshotIdentifier(dbSnapshotIdentifier)
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .build();

           while (!snapshotReady) {
               DescribeDbClusterSnapshotsResponse response =
rdsClient.describeDBClusterSnapshots(snapshotsRequest);
```

```
               List<DBClusterSnapshot> snapshotList =
response.dbClusterSnapshots();
               for (DBClusterSnapshot snapshot : snapshotList) {
                   snapshotReadyStr = snapshot.status();
                   if (snapshotReadyStr.contains("available")) {
                       snapshotReady = true;
                   } else {
                       System.out.println(".");
                       Thread.sleep(sleepTime * 5000);
                   }
               }
           }

           System.out.println("The Snapshot is available!");

       } catch (RdsException | InterruptedException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void createDBClusterSnapshot(RdsClient rdsClient, String
dbInstanceClusterIdentifier,
           String dbSnapshotIdentifier) {
       try {
           CreateDbClusterSnapshotRequest snapshotRequest =
CreateDbClusterSnapshotRequest.builder()
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .dbClusterSnapshotIdentifier(dbSnapshotIdentifier)
                   .build();

           CreateDbClusterSnapshotResponse response =
rdsClient.createDBClusterSnapshot(snapshotRequest);
           System.out.println("The Snapshot ARN is " +
response.dbClusterSnapshot().dbClusterSnapshotArn());

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void waitDBInstanceReady(RdsClient rdsClient, String
dbInstanceIdentifier) {
       boolean instanceReady = false;
       String instanceReadyStr;
       System.out.println("Waiting for instance to become available.");
       try {
           DescribeDbInstancesRequest instanceRequest = 
DescribeDbInstancesRequest.builder()
                   .dbInstanceIdentifier(dbInstanceIdentifier)
                   .build();

           String endpoint = "";
           while (!instanceReady) {
               DescribeDbInstancesResponse response = 
rdsClient.describeDBInstances(instanceRequest);
               List<DBInstance> instanceList = response.dbInstances();
               for (DBInstance instance : instanceList) {
                   instanceReadyStr = instance.dbInstanceStatus();
                   if (instanceReadyStr.contains("available")) {
                       endpoint = instance.endpoint().address();
                       instanceReady = true;
                   } else {
                       System.out.print(".");
                       Thread.sleep(sleepTime * 1000);
                   }
               }
           }
           System.out.println("Database instance is available! The connection 
endpoint is " + endpoint);

       } catch (RdsException | InterruptedException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }

   public static String createDBInstanceCluster(RdsClient rdsClient,
           String dbInstanceIdentifier,
           String dbInstanceClusterIdentifier,
           String instanceClass) {
       try {
           CreateDbInstanceRequest instanceRequest = 
CreateDbInstanceRequest.builder()
                   .dbInstanceIdentifier(dbInstanceIdentifier)
                   .dbClusterIdentifier(dbInstanceClusterIdentifier)
                   .engine("aurora-mysql")
```

```
                   .dbInstanceClass(instanceClass)
                   .build();

           CreateDbInstanceResponse response =
rdsClient.createDBInstance(instanceRequest);
           System.out.print("The status is " +
response.dbInstance().dbInstanceStatus());
           return response.dbInstance().dbInstanceArn();

       } catch (RdsException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
       return "";
   }

   public static String getListInstanceClasses(RdsClient rdsClient) {
       try {
           DescribeOrderableDbInstanceOptionsRequest optionsRequest =
DescribeOrderableDbInstanceOptionsRequest
                   .builder()
                   .engine("aurora-mysql")
                   .maxRecords(20)
                   .build();

           DescribeOrderableDbInstanceOptionsResponse response = rdsClient
                   .describeOrderableDBInstanceOptions(optionsRequest);
           List<OrderableDBInstanceOption> instanceOptions =
response.orderableDBInstanceOptions();
           String instanceClass = "";
           for (OrderableDBInstanceOption instanceOption : instanceOptions) {
               instanceClass = instanceOption.dbInstanceClass();
               System.out.println("The instance class is " +
instanceOption.dbInstanceClass());
               System.out.println("The engine version is " +
instanceOption.engineVersion());
           }
           return instanceClass;

       } catch (RdsException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
       return "";
   }

   // Waits until the database instance is available.
   public static void waitForInstanceReady(RdsClient rdsClient, String
dbClusterIdentifier) {
       boolean instanceReady = false;
       String instanceReadyStr;
       System.out.println("Waiting for instance to become available.");
       try {
           DescribeDbClustersRequest instanceRequest =
DescribeDbClustersRequest.builder()
                   .dbClusterIdentifier(dbClusterIdentifier)
                   .build();

           while (!instanceReady) {
               DescribeDbClustersResponse response =
rdsClient.describeDBClusters(instanceRequest);
               List<DBCluster> clusterList = response.dbClusters();
               for (DBCluster cluster : clusterList) {
                   instanceReadyStr = cluster.status();
                   if (instanceReadyStr.contains("available")) {
                       instanceReady = true;
                   } else {
                       System.out.print(".");
                       Thread.sleep(sleepTime * 1000);
                   }
               }
           }
           System.out.println("Database cluster is available!");

       } catch (RdsException | InterruptedException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }

   public static String createDBCluster(RdsClient rdsClient, String
dbParameterGroupFamily, String dbName,
           String dbClusterIdentifier, String userName, String password) {
       try {
           CreateDbClusterRequest clusterRequest =
CreateDbClusterRequest.builder()
                   .databaseName(dbName)
                   .dbClusterIdentifier(dbClusterIdentifier)
                   .dbClusterParameterGroupName(dbParameterGroupFamily)
                   .engine("aurora-mysql")
                   .masterUsername(userName)
                   .masterUserPassword(password)
                   .build();

           CreateDbClusterResponse response =
rdsClient.createDBCluster(clusterRequest);
           return response.dbCluster().dbClusterArn();

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
       return "";
   }

   // Get a list of allowed engine versions.
   public static void getAllowedEngines(RdsClient rdsClient, String
dbParameterGroupFamily) {
       try {
           DescribeDbEngineVersionsRequest versionsRequest =
DescribeDbEngineVersionsRequest.builder()
                   .dbParameterGroupFamily(dbParameterGroupFamily)
                   .engine("aurora-mysql")
                   .build();

           DescribeDbEngineVersionsResponse response =
rdsClient.describeDBEngineVersions(versionsRequest);
           List<DBEngineVersion> dbEngines = response.dbEngineVersions();
           for (DBEngineVersion dbEngine : dbEngines) {
               System.out.println("The engine version is " +
dbEngine.engineVersion());
               System.out.println("The engine description is " +
dbEngine.dbEngineDescription());
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   // Modify the auto_increment_offset parameter.
   public static void modifyDBClusterParas(RdsClient rdsClient, String
dClusterGroupName) {
       try {
           Parameter parameter1 = Parameter.builder()
                   .parameterName("auto_increment_offset")
                   .applyMethod("immediate")
                   .parameterValue("5")
                   .build();

           List<Parameter> paraList = new ArrayList<>();
           paraList.add(parameter1);
           ModifyDbClusterParameterGroupRequest groupRequest =
ModifyDbClusterParameterGroupRequest.builder()
                   .dbClusterParameterGroupName(dClusterGroupName)
                   .parameters(paraList)
                   .build();

           ModifyDbClusterParameterGroupResponse response =
rdsClient.modifyDBClusterParameterGroup(groupRequest);
           System.out.println(
                   "The parameter group " +
response.dbClusterParameterGroupName() + " was successfully modified");

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void describeDbClusterParameters(RdsClient rdsClient, String
dbCLusterGroupName, int flag) {
       try {
           DescribeDbClusterParametersRequest dbParameterGroupsRequest;
           if (flag == 0) {
               dbParameterGroupsRequest =
DescribeDbClusterParametersRequest.builder()
                       .dbClusterParameterGroupName(dbCLusterGroupName)
                       .build();
           } else {
               dbParameterGroupsRequest =
DescribeDbClusterParametersRequest.builder()
                       .dbClusterParameterGroupName(dbCLusterGroupName)
                       .source("user")
                       .build();
           }

           DescribeDbClusterParametersResponse response = rdsClient
                   .describeDBClusterParameters(dbParameterGroupsRequest);
           List<Parameter> dbParameters = response.parameters();
           String paraName;
           for (Parameter para : dbParameters) {
               // Only print out information about either auto_increment_offset
or
               // auto_increment_increment.
               paraName = para.parameterName();
               if ((paraName.compareTo("auto_increment_offset") == 0)
                       || (paraName.compareTo("auto_increment_increment ") ==
0)) {
                   System.out.println("*** The parameter name is  " + paraName);
                   System.out.println("*** The parameter value is  " +
para.parameterValue());
                   System.out.println("*** The parameter data type is " +
para.dataType());
                   System.out.println("*** The parameter description is " +
para.description());
                   System.out.println("*** The parameter allowed values  is " +
para.allowedValues());
               }
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void describeDbClusterParameterGroups(RdsClient rdsClient,
String dbClusterGroupName) {
       try {
           DescribeDbClusterParameterGroupsRequest groupsRequest =
DescribeDbClusterParameterGroupsRequest.builder()
                   .dbClusterParameterGroupName(dbClusterGroupName)
                   .maxRecords(20)
                   .build();

           List<DBClusterParameterGroup> groups =
rdsClient.describeDBClusterParameterGroups(groupsRequest)
                   .dbClusterParameterGroups();
           for (DBClusterParameterGroup group : groups) {
               System.out.println("The group name is " +
group.dbClusterParameterGroupName());
               System.out.println("The group ARN is " +
group.dbClusterParameterGroupArn());
           }

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void createDBClusterParameterGroup(RdsClient rdsClient, String
dbClusterGroupName,
           String dbParameterGroupFamily) {
       try {
           CreateDbClusterParameterGroupRequest groupRequest =
CreateDbClusterParameterGroupRequest.builder()
                   .dbClusterParameterGroupName(dbClusterGroupName)
                   .dbParameterGroupFamily(dbParameterGroupFamily)
                   .description("Created by using the AWS SDK for Java")
                   .build();

           CreateDbClusterParameterGroupResponse response =
rdsClient.createDBClusterParameterGroup(groupRequest);
           System.out.println("The group name is " +
response.dbClusterParameterGroup().dbClusterParameterGroupName());

       } catch (RdsException e) {
           System.out.println(e.getLocalizedMessage());
           System.exit(1);
       }
   }

   public static void describeDBEngines(RdsClient rdsClient) {
       try {
           DescribeDbEngineVersionsRequest engineVersionsRequest =
DescribeDbEngineVersionsRequest.builder()
                   .engine("aurora-mysql")
                   .defaultOnly(true)
                   .maxRecords(20)
                   .build();
```

```
            DescribeDbEngineVersionsResponse response =
 rdsClient.describeDBEngineVersions(engineVersionsRequest);
            List<DBEngineVersion> engines = response.dbEngineVersions();

            // Get all DBEngineVersion objects.
            for (DBEngineVersion engineOb : engines) {
                System.out.println("The name of the DB parameter group family for
 the database engine is "
                        + engineOb.dbParameterGroupFamily());
                System.out.println("The name of the database engine " +
 engineOb.engine());
                System.out.println("The version number of the database engine " +
 engineOb.engineVersion());
            }

        } catch (RdsException e) {
            System.out.println(e.getLocalizedMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster
  - DeleteDBClusterParameterGroup
  - DeleteDBInstance
  - DescribeDBClusterParameterGroups
  - DescribeDBClusterParameters
  - DescribeDBClusterSnapshots
  - DescribeDBClusters
  - DescribeDBEngineVersions
  - DescribeDBInstances

- DescribeOrderableDBInstanceOptions
- ModifyDBClusterParameterGroup

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
Before running this Kotlin code example, set up your development environment,
 including your credentials.

For more information, see the following documentation topic:

https://docs.aws.amazon.com/sdk-for-kotlin/latest/developer-guide/setup.html

This example requires an AWS Secrets Manager secret that contains the database
 credentials. If you do not create a
secret, this example will not work. For more details, see:

https://docs.aws.amazon.com/secretsmanager/latest/userguide/integrating_how-
services-use-secrets_RS.html

This Kotlin example performs the following tasks:

1. Returns a list of the available DB engines.
2. Creates a custom DB parameter group.
3. Gets the parameter groups.
4. Gets the parameters in the group.
5. Modifies the auto_increment_increment parameter.
6. Displays the updated parameter value.
7. Gets a list of allowed engine versions.
8. Creates an Aurora DB cluster database.
9. Waits for DB instance to be ready.
10. Gets a list of instance classes available for the selected engine.
```

```
11. Creates a database instance in the cluster.
12. Waits for the database instance in the cluster to be ready.
13. Creates a snapshot.
14. Waits for DB snapshot to be ready.
15. Deletes the DB instance.
16. Deletes the DB cluster.
17. Deletes the DB cluster group.
 */

var slTime: Long = 20

suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
            <dbClusterGroupName> <dbParameterGroupFamily>
 <dbInstanceClusterIdentifier> <dbName> <dbSnapshotIdentifier> <secretName>
        Where:
            dbClusterGroupName - The database group name.
            dbParameterGroupFamily - The database parameter group name.
            dbInstanceClusterIdentifier - The database instance identifier.
            dbName -  The database name.
            dbSnapshotIdentifier - The snapshot identifier.
            secretName - The name of the AWS Secrets Manager secret that contains
 the database credentials.
    """

    if (args.size != 7) {
        println(usage)
        exitProcess(1)
    }

    val dbClusterGroupName = args[0]
    val dbParameterGroupFamily = args[1]
    val dbInstanceClusterIdentifier = args[2]
    val dbInstanceIdentifier = args[3]
    val dbName = args[4]
    val dbSnapshotIdentifier = args[5]
    val secretName = args[6]

    val gson = Gson()
    val user = gson.fromJson(getSecretValues(secretName).toString(),
 User::class.java)
    val username = user.username
    val userPassword = user.password
```

```
   println("1. Return a list of the available DB engines")
   describeAuroraDBEngines()

   println("2. Create a custom parameter group")
   createDBClusterParameterGroup(dbClusterGroupName, dbParameterGroupFamily)

   println("3. Get the parameter group")
   describeDbClusterParameterGroups(dbClusterGroupName)

   println("4. Get the parameters in the group")
   describeDbClusterParameters(dbClusterGroupName, 0)

   println("5. Modify the auto_increment_offset parameter")
   modifyDBClusterParas(dbClusterGroupName)

   println("6. Display the updated parameter value")
   describeDbClusterParameters(dbClusterGroupName, -1)

   println("7. Get a list of allowed engine versions")
   getAllowedClusterEngines(dbParameterGroupFamily)

   println("8. Create an Aurora DB cluster database")
   val arnClusterVal = createDBCluster(dbClusterGroupName, dbName,
dbInstanceClusterIdentifier, username, userPassword)
   println("The ARN of the cluster is $arnClusterVal")

   println("9. Wait for DB instance to be ready")
   waitForClusterInstanceReady(dbInstanceClusterIdentifier)

   println("10. Get a list of instance classes available for the selected
engine")
   val instanceClass = getListInstanceClasses()

   println("11. Create a database instance in the cluster.")
   val clusterDBARN = createDBInstanceCluster(dbInstanceIdentifier,
dbInstanceClusterIdentifier, instanceClass)
   println("The ARN of the database is $clusterDBARN")

   println("12. Wait for DB instance to be ready")
   waitDBAuroraInstanceReady(dbInstanceIdentifier)

   println("13. Create a snapshot")
   createDBClusterSnapshot(dbInstanceClusterIdentifier, dbSnapshotIdentifier)
```

```
    println("14. Wait for DB snapshot to be ready")
    waitSnapshotReady(dbSnapshotIdentifier, dbInstanceClusterIdentifier)

    println("15. Delete the DB instance")
    deleteDBInstance(dbInstanceIdentifier)

    println("16. Delete the DB cluster")
    deleteCluster(dbInstanceClusterIdentifier)

    println("17. Delete the DB cluster group")
    if (clusterDBARN != null) {
        deleteDBClusterGroup(dbClusterGroupName, clusterDBARN)
    }
    println("The Scenario has successfully completed.")
}

@Throws(InterruptedException::class)
suspend fun deleteDBClusterGroup(
    dbClusterGroupName: String,
    clusterDBARN: String,
) {
    var isDataDel = false
    var didFind: Boolean
    var instanceARN: String

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        // Make sure that the database has been deleted.
        while (!isDataDel) {
            val response = rdsClient.describeDbInstances()
            val instanceList = response.dbInstances
            val listSize = instanceList?.size
            isDataDel = false
            didFind = false
            var index = 1
            if (instanceList != null) {
                for (instance in instanceList) {
                    instanceARN = instance.dbInstanceArn.toString()
                    if (instanceARN.compareTo(clusterDBARN) == 0) {
                        println("$clusterDBARN still exists")
                        didFind = true
                    }
                    if (index == listSize && !didFind) {
                        // Went through the entire list and did not find the
 database ARN.
                        isDataDel = true
                    }
                    delay(slTime * 1000)
                    index++
                }
            }
        }
        val clusterParameterGroupRequest =
            DeleteDbClusterParameterGroupRequest {
                dbClusterParameterGroupName = dbClusterGroupName
            }

        rdsClient.deleteDbClusterParameterGroup(clusterParameterGroupRequest)
        println("$dbClusterGroupName was deleted.")
    }
}

suspend fun deleteCluster(dbInstanceClusterIdentifier: String) {
    val deleteDbClusterRequest =
        DeleteDbClusterRequest {
            dbClusterIdentifier = dbInstanceClusterIdentifier
            skipFinalSnapshot = true
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        rdsClient.deleteDbCluster(deleteDbClusterRequest)
        println("$dbInstanceClusterIdentifier was deleted!")
    }
}

suspend fun deleteDBInstance(dbInstanceIdentifierVal: String) {
    val deleteDbInstanceRequest =
        DeleteDbInstanceRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
            deleteAutomatedBackups = true
            skipFinalSnapshot = true
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.deleteDbInstance(deleteDbInstanceRequest)
        print("The status of the database is
 ${response.dbInstance?.dbInstanceStatus}")
```

```
    }
}

suspend fun waitSnapshotReady(
    dbSnapshotIdentifier: String?,
    dbInstanceClusterIdentifier: String?,
) {
    var snapshotReady = false
    var snapshotReadyStr: String
    println("Waiting for the snapshot to become available.")

    val snapshotsRequest =
        DescribeDbClusterSnapshotsRequest {
            dbClusterSnapshotIdentifier = dbSnapshotIdentifier
            dbClusterIdentifier = dbInstanceClusterIdentifier
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        while (!snapshotReady) {
            val response = rdsClient.describeDbClusterSnapshots(snapshotsRequest)
            val snapshotList = response.dbClusterSnapshots
            if (snapshotList != null) {
                for (snapshot in snapshotList) {
                    snapshotReadyStr = snapshot.status.toString()
                    if (snapshotReadyStr.contains("available")) {
                        snapshotReady = true
                    } else {
                        println(".")
                        delay(slTime * 5000)
                    }
                }
            }
        }
    }
    println("The Snapshot is available!")
}

suspend fun createDBClusterSnapshot(
    dbInstanceClusterIdentifier: String?,
    dbSnapshotIdentifier: String?,
) {
    val snapshotRequest =
        CreateDbClusterSnapshotRequest {
            dbClusterIdentifier = dbInstanceClusterIdentifier
            dbClusterSnapshotIdentifier = dbSnapshotIdentifier
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbClusterSnapshot(snapshotRequest)
        println("The Snapshot ARN is
 ${response.dbClusterSnapshot?.dbClusterSnapshotArn}")
    }
}

suspend fun waitDBAuroraInstanceReady(dbInstanceIdentifierVal: String?) {
    var instanceReady = false
    var instanceReadyStr: String
    println("Waiting for instance to become available.")
    val instanceRequest =
        DescribeDbInstancesRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
        }

    var endpoint = ""
    RdsClient { region = "us-west-2" }.use { rdsClient ->
        while (!instanceReady) {
            val response = rdsClient.describeDbInstances(instanceRequest)
            response.dbInstances?.forEach { instance ->
                instanceReadyStr = instance.dbInstanceStatus.toString()
                if (instanceReadyStr.contains("available")) {
                    endpoint = instance.endpoint?.address.toString()
                    instanceReady = true
                } else {
                    print(".")
                    delay(sleepTime * 1000)
                }
            }
        }
    }
    println("Database instance is available! The connection endpoint is
 $endpoint")
}

suspend fun createDBInstanceCluster(
    dbInstanceIdentifierVal: String?,
    dbInstanceClusterIdentifierVal: String?,
    instanceClassVal: String?,
): String? {
    val instanceRequest =
        CreateDbInstanceRequest {
            dbInstanceIdentifier = dbInstanceIdentifierVal
            dbClusterIdentifier = dbInstanceClusterIdentifierVal
            engine = "aurora-mysql"
            dbInstanceClass = instanceClassVal
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbInstance(instanceRequest)
        print("The status is ${response.dbInstance?.dbInstanceStatus}")
        return response.dbInstance?.dbInstanceArn
    }
}

suspend fun getListInstanceClasses(): String {
    val optionsRequest =
        DescribeOrderableDbInstanceOptionsRequest {
            engine = "aurora-mysql"
            maxRecords = 20
        }
    var instanceClass = ""
    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response =
 rdsClient.describeOrderableDbInstanceOptions(optionsRequest)
        response.orderableDbInstanceOptions?.forEach { instanceOption ->
            instanceClass = instanceOption.dbInstanceClass.toString()
            println("The instance class is ${instanceOption.dbInstanceClass}")
            println("The engine version is ${instanceOption.engineVersion}")
        }
    }
    return instanceClass
}

// Waits until the database instance is available.
suspend fun waitForClusterInstanceReady(dbClusterIdentifierVal: String?) {
    var instanceReady = false
    var instanceReadyStr: String
    println("Waiting for instance to become available.")

    val instanceRequest =
        DescribeDbClustersRequest {
            dbClusterIdentifier = dbClusterIdentifierVal
        }
    RdsClient { region = "us-west-2" }.use { rdsClient ->
        while (!instanceReady) {
            val response = rdsClient.describeDbClusters(instanceRequest)
            response.dbClusters?.forEach { cluster ->
                instanceReadyStr = cluster.status.toString()
                if (instanceReadyStr.contains("available")) {
                    instanceReady = true
                } else {
                    print(".")
                    delay(sleepTime * 1000)
                }
            }
        }
    }
    println("Database cluster is available!")
}

suspend fun createDBCluster(
    dbParameterGroupFamilyVal: String?,
    dbName: String?,
    dbClusterIdentifierVal: String?,
    userName: String?,
    password: String?,
): String? {
    val clusterRequest =
        CreateDbClusterRequest {
            databaseName = dbName
            dbClusterIdentifier = dbClusterIdentifierVal
            dbClusterParameterGroupName = dbParameterGroupFamilyVal
            engine = "aurora-mysql"
            masterUsername = userName
            masterUserPassword = password
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbCluster(clusterRequest)
        return response.dbCluster?.dbClusterArn
    }
}

// Get a list of allowed engine versions.
suspend fun getAllowedClusterEngines(dbParameterGroupFamilyVal: String?) {
    val versionsRequest =
        DescribeDbEngineVersionsRequest {
            dbParameterGroupFamily = dbParameterGroupFamilyVal
            engine = "aurora-mysql"
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.describeDbEngineVersions(versionsRequest)
        response.dbEngineVersions?.forEach { dbEngine ->
            println("The engine version is ${dbEngine.engineVersion}")
            println("The engine description is ${dbEngine.dbEngineDescription}")
        }
    }
}

// Modify the auto_increment_offset parameter.
suspend fun modifyDBClusterParas(dClusterGroupName: String?) {
    val parameter1 =
        Parameter {
            parameterName = "auto_increment_offset"
            applyMethod = ApplyMethod.fromValue("immediate")
            parameterValue = "5"
        }

    val paraList = ArrayList<Parameter>()
    paraList.add(parameter1)
    val groupRequest =
        ModifyDbClusterParameterGroupRequest {
            dbClusterParameterGroupName = dClusterGroupName
            parameters = paraList
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.modifyDbClusterParameterGroup(groupRequest)
        println("The parameter group ${response.dbClusterParameterGroupName} was
 successfully modified")
    }
}

suspend fun describeDbClusterParameters(
    dbCLusterGroupName: String?,
    flag: Int,
) {
    val dbParameterGroupsRequest: DescribeDbClusterParametersRequest
    dbParameterGroupsRequest =
        if (flag == 0) {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
            }
        } else {
            DescribeDbClusterParametersRequest {
                dbClusterParameterGroupName = dbCLusterGroupName
                source = "user"
            }
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response =
 rdsClient.describeDbClusterParameters(dbParameterGroupsRequest)
        response.parameters?.forEach { para ->
            // Only print out information about either auto_increment_offset or
 auto_increment_increment.
            val paraName = para.parameterName
            if (paraName != null) {
                if (paraName.compareTo("auto_increment_offset") == 0 ||
 paraName.compareTo("auto_increment_increment ") == 0) {
                    println("*** The parameter name is  $paraName")
                    println("*** The parameter value is  ${para.parameterValue}")
                    println("*** The parameter data type is ${para.dataType}")
                    println("*** The parameter description is
 ${para.description}")
                    println("*** The parameter allowed values  is
 ${para.allowedValues}")
                }
            }
        }
    }
}

suspend fun describeDbClusterParameterGroups(dbClusterGroupName: String?) {
    val groupsRequest =
        DescribeDbClusterParameterGroupsRequest {
            dbClusterParameterGroupName = dbClusterGroupName
            maxRecords = 20
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.describeDbClusterParameterGroups(groupsRequest)
        response.dbClusterParameterGroups?.forEach { group ->
```

```
            println("The group name is ${group.dbClusterParameterGroupName}")
            println("The group ARN is ${group.dbClusterParameterGroupArn}")
        }
    }
}

suspend fun createDBClusterParameterGroup(
    dbClusterGroupNameVal: String?,
    dbParameterGroupFamilyVal: String?,
) {
    val groupRequest =
        CreateDbClusterParameterGroupRequest {
            dbClusterParameterGroupName = dbClusterGroupNameVal
            dbParameterGroupFamily = dbParameterGroupFamilyVal
            description = "Created by using the AWS SDK for Kotlin"
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.createDbClusterParameterGroup(groupRequest)
        println("The group name is
 ${response.dbClusterParameterGroup?.dbClusterParameterGroupName}")
    }
}

suspend fun describeAuroraDBEngines() {
    val engineVersionsRequest =
        DescribeDbEngineVersionsRequest {
            engine = "aurora-mysql"
            defaultOnly = true
            maxRecords = 20
        }

    RdsClient { region = "us-west-2" }.use { rdsClient ->
        val response = rdsClient.describeDbEngineVersions(engineVersionsRequest)
        response.dbEngineVersions?.forEach { engineOb ->
            println("The name of the DB parameter group family for the database
 engine is ${engineOb.dbParameterGroupFamily}")
            println("The name of the database engine ${engineOb.engine}")
            println("The version number of the database engine
 ${engineOb.engineVersion}")
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster
  - DeleteDBClusterParameterGroup
  - DeleteDBInstance
  - DescribeDBClusterParameterGroups
  - DescribeDBClusterParameters
  - DescribeDBClusterSnapshots
  - DescribeDBClusters
  - DescribeDBEngineVersions
  - DescribeDBInstances
  - DescribeOrderableDBInstanceOptions
  - ModifyDBClusterParameterGroup

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
class AuroraClusterScenario:
    """Runs a scenario that shows how to get started using Aurora DB clusters."""

    def __init__(self, aurora_wrapper):
        """
        :param aurora_wrapper: An object that wraps Aurora DB cluster actions.
        """
        self.aurora_wrapper = aurora_wrapper

    def create_parameter_group(self, db_engine, parameter_group_name):
        """
        Shows how to get available engine versions for a specified database
 engine and
        create a DB cluster parameter group that is compatible with a selected
 engine family.

        :param db_engine: The database engine to use as a basis.
        :param parameter_group_name: The name given to the newly created
 parameter group.
        :return: The newly created parameter group.
        """
        print(
            f"Checking for an existing DB cluster parameter group named
 {parameter_group_name}."
        )
        parameter_group =
 self.aurora_wrapper.get_parameter_group(parameter_group_name)
        if parameter_group is None:
            print(f"Getting available database engine versions for {db_engine}.")
            engine_versions = self.aurora_wrapper.get_engine_versions(db_engine)
            families = list({ver["DBParameterGroupFamily"] for ver in
 engine_versions})
            family_index = q.choose("Which family do you want to use? ",
 families)
            print(f"Creating a DB cluster parameter group.")
            self.aurora_wrapper.create_parameter_group(
                parameter_group_name, families[family_index], "Example parameter
 group."
            )
            parameter_group = self.aurora_wrapper.get_parameter_group(
                parameter_group_name
            )
        print(f"Parameter group
 {parameter_group['DBClusterParameterGroupName']}:")
        pp(parameter_group)
        print("-" * 88)
        return parameter_group
    def set_user_parameters(self, parameter_group_name):
        """
        Shows how to get the parameters contained in a custom parameter group and
        update some of the parameter values in the group.

        :param parameter_group_name: The name of the parameter group to query and
 modify.
        """
        print("Let's set some parameter values in your parameter group.")
        auto_inc_parameters = self.aurora_wrapper.get_parameters(
            parameter_group_name, name_prefix="auto_increment"
        )
        update_params = []
        for auto_inc in auto_inc_parameters:
            if auto_inc["IsModifiable"] and auto_inc["DataType"] == "integer":
                print(f"The {auto_inc['ParameterName']} parameter is described
 as:")
                print(f"\t{auto_inc['Description']}")
                param_range = auto_inc["AllowedValues"].split("-")
                auto_inc["ParameterValue"] = str(
                    q.ask(
                        f"Enter a value between {param_range[0]} and
 {param_range[1]}: ",
                        q.is_int,
                        q.in_range(int(param_range[0]), int(param_range[1])),
                    )
                )
                update_params.append(auto_inc)
        self.aurora_wrapper.update_parameters(parameter_group_name,
 update_params)
        print(
            "You can get a list of parameters you've set by specifying a source
 of 'user'."
        )
        user_parameters = self.aurora_wrapper.get_parameters(
            parameter_group_name, source="user"
        )
        pp(user_parameters)
        print("-" * 88)

    def create_cluster(self, cluster_name, db_engine, db_name, parameter_group):
        """
        Shows how to create an Aurora DB cluster that contains a database of a
 specified
```

```
        type. The database is also configured to use a custom DB cluster
parameter group.

        :param cluster_name: The name given to the newly created DB cluster.
        :param db_engine: The engine of the created database.
        :param db_name: The name given to the created database.
        :param parameter_group: The parameter group that is associated with the
DB cluster.
        :return: The newly created DB cluster.
        """
        print("Checking for an existing DB cluster.")
        cluster = self.aurora_wrapper.get_db_cluster(cluster_name)
        if cluster is None:
            admin_username = q.ask(
                "Enter an administrator user name for the database: ",
q.non_empty
            )
            admin_password = q.ask(
                "Enter a password for the administrator (at least 8 characters):
",
                q.non_empty,
            )
            engine_versions = self.aurora_wrapper.get_engine_versions(
                db_engine, parameter_group["DBParameterGroupFamily"]
            )
            engine_choices = [ver["EngineVersionDescription"] for ver in
engine_versions]
            print("The available engines for your parameter group are:")
            engine_index = q.choose("Which engine do you want to use? ",
engine_choices)
            print(
                f"Creating DB cluster {cluster_name} and database {db_name}.\n"
                f"The DB cluster is configured to use\n"
                f"your custom parameter group
{parameter_group['DBClusterParameterGroupName']}\n"
                f"and selected engine {engine_choices[engine_index]}.\n"
                f"This typically takes several minutes."
            )
            cluster = self.aurora_wrapper.create_db_cluster(
                cluster_name,
                parameter_group["DBClusterParameterGroupName"],
                db_name,
                db_engine,
                engine_versions[engine_index]["EngineVersion"],
                admin_username,
                admin_password,
            )
            while cluster.get("Status") != "available":
                wait(30)
                cluster = self.aurora_wrapper.get_db_cluster(cluster_name)
            print("Cluster created and available.\n")
        print("Cluster data:")
        pp(cluster)
        print("-" * 88)
        return cluster

    def create_instance(self, cluster):
        """
        Shows how to create a DB instance in an existing Aurora DB cluster. A new 
DB cluster
        contains no DB instances, so you must add one. The first DB instance that 
is added
        to a DB cluster defaults to a read-write DB instance.

        :param cluster: The DB cluster where the DB instance is added.
        :return: The newly created DB instance.
        """
        print("Checking for an existing database instance.")
        cluster_name = cluster["DBClusterIdentifier"]
        db_inst = self.aurora_wrapper.get_db_instance(cluster_name)
        if db_inst is None:
            print("Let's create a database instance in your DB cluster.")
            print("First, choose a DB instance type:")
            inst_opts = self.aurora_wrapper.get_orderable_instances(
                cluster["Engine"], cluster["EngineVersion"]
            )
            inst_choices = list({opt["DBInstanceClass"] + ", storage type: " + 
opt["StorageType"] for opt in inst_opts})
            inst_index = q.choose(
                "Which DB instance class do you want to use? ", inst_choices
            )
            print(
                f"Creating a database instance. This typically takes several 
minutes."
            )
            db_inst = self.aurora_wrapper.create_instance_in_cluster(
                cluster_name, cluster_name, cluster["Engine"], 
inst_opts[inst_index]["DBInstanceClass"]
```

```
            )
            while db_inst.get("DBInstanceStatus") != "available":
                wait(30)
                db_inst = self.aurora_wrapper.get_db_instance(cluster_name)
        print("Instance data:")
        pp(db_inst)
        print("-" * 88)
        return db_inst

    @staticmethod
    def display_connection(cluster):
        """
        Displays connection information about an Aurora DB cluster and tips on
 how to
        connect to it.

        :param cluster: The DB cluster to display.
        """
        print(
            "You can now connect to your database using your favorite MySql
 client.\n"
            "One way to connect is by using the 'mysql' shell on an Amazon EC2
 instance\n"
            "that is running in the same VPC as your database cluster. Pass the
 endpoint,\n"
            "port, and administrator user name to 'mysql' and enter your password
\n"
            "when prompted:\n"
        )
        print(
            f"\n\tmysql -h {cluster['Endpoint']} -P {cluster['Port']} -u
 {cluster['MasterUsername']} -p\n"
        )
        print(
            "For more information, see the User Guide for Aurora:\n"
            "\thttps://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/
CHAP_GettingStartedAurora.CreatingConnecting.Aurora.html#CHAP_GettingStartedAurora.Aurora
        )
        print("-" * 88)

    def create_snapshot(self, cluster_name):
        """
        Shows how to create a DB cluster snapshot and wait until it's available.
```

```
       :param cluster_name: The name of a DB cluster to snapshot.
       """
       if q.ask(
           "Do you want to create a snapshot of your DB cluster (y/n)? ",
q.is_yesno
       ):
           snapshot_id = f"{cluster_name}-{uuid.uuid4()}"
           print(
               f"Creating a snapshot named {snapshot_id}. This typically takes a
few minutes."
           )
           snapshot = self.aurora_wrapper.create_cluster_snapshot(
               snapshot_id, cluster_name
           )
           while snapshot.get("Status") != "available":
               wait(30)
               snapshot = self.aurora_wrapper.get_cluster_snapshot(snapshot_id)
           pp(snapshot)
           print("-" * 88)

   def cleanup(self, db_inst, cluster, parameter_group):
       """
       Shows how to clean up a DB instance, DB cluster, and DB cluster parameter
group.
       Before the DB cluster parameter group can be deleted, all associated DB
instances and
       DB clusters must first be deleted.

       :param db_inst: The DB instance to delete.
       :param cluster: The DB cluster to delete.
       :param parameter_group: The DB cluster parameter group to delete.
       """
       cluster_name = cluster["DBClusterIdentifier"]
       parameter_group_name = parameter_group["DBClusterParameterGroupName"]
       if q.ask(
           "\nDo you want to delete the database instance, DB cluster, and
parameter "
           "group (y/n)? ",
           q.is_yesno,
       ):
           print(f"Deleting database instance
{db_inst['DBInstanceIdentifier']}.")

self.aurora_wrapper.delete_db_instance(db_inst["DBInstanceIdentifier"])
```

```
           print(f"Deleting database cluster {cluster_name}.")
           self.aurora_wrapper.delete_db_cluster(cluster_name)
           print(
               "Waiting for the DB instance and DB cluster to delete.\n"
               "This typically takes several minutes."
           )
           while db_inst is not None or cluster is not None:
               wait(30)
               if db_inst is not None:
                   db_inst = self.aurora_wrapper.get_db_instance(
                       db_inst["DBInstanceIdentifier"]
                   )
               if cluster is not None:
                   cluster = self.aurora_wrapper.get_db_cluster(
                       cluster["DBClusterIdentifier"]
                   )
           print(f"Deleting parameter group {parameter_group_name}.")
           self.aurora_wrapper.delete_parameter_group(parameter_group_name)

   def run_scenario(self, db_engine, parameter_group_name, cluster_name, 
db_name):
       print("-" * 88)
       print(
           "Welcome to the Amazon Relational Database Service (Amazon RDS) get 
started\n"
           "with Aurora DB clusters demo."
       )
       print("-" * 88)

       parameter_group = self.create_parameter_group(db_engine, 
parameter_group_name)
       self.set_user_parameters(parameter_group_name)
       cluster = self.create_cluster(cluster_name, db_engine, db_name, 
parameter_group)
       wait(5)
       db_inst = self.create_instance(cluster)
       self.display_connection(cluster)
       self.create_snapshot(cluster_name)
       self.cleanup(db_inst, cluster, parameter_group)

       print("\nThanks for watching!")
       print("-" * 88)
```

```
if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")
    try:
        scenario = AuroraClusterScenario(AuroraWrapper.from_client())
        scenario.run_scenario(
            "aurora-mysql",
            "doc-example-cluster-parameter-group",
            "doc-example-aurora",
            "docexampledb",
        )
    except Exception:
        logging.exception("Something went wrong with the demo.")
```

Define functions that are called by the scenario to manage Aurora actions.

```
class AuroraWrapper:
    """Encapsulates Aurora DB cluster actions."""

    def __init__(self, rds_client):
        """
        :param rds_client: A Boto3 Amazon Relational Database Service (Amazon
 RDS) client.
        """
        self.rds_client = rds_client

    @classmethod
    def from_client(cls):
        """
        Instantiates this class from a Boto3 client.
        """
        rds_client = boto3.client("rds")
        return cls(rds_client)



    def get_parameter_group(self, parameter_group_name):
        """
        Gets a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to retrieve.
        :return: The requested parameter group.
        """
        try:
            response = self.rds_client.describe_db_cluster_parameter_groups(
                DBClusterParameterGroupName=parameter_group_name
            )
            parameter_group = response["DBClusterParameterGroups"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBParameterGroupNotFound":
                logger.info("Parameter group %s does not exist.",
parameter_group_name)
            else:
                logger.error(
                    "Couldn't get parameter group %s. Here's why: %s: %s",
                    parameter_group_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return parameter_group


    def create_parameter_group(
        self, parameter_group_name, parameter_group_family, description
    ):
        """
        Creates a DB cluster parameter group that is based on the specified
parameter group
        family.

        :param parameter_group_name: The name of the newly created parameter
group.
        :param parameter_group_family: The family that is used as the basis of
the new
                                       parameter group.
        :param description: A description given to the parameter group.
        :return: Data about the newly created parameter group.
        """
        try:
            response = self.rds_client.create_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name,
                DBParameterGroupFamily=parameter_group_family,
                Description=description,
            )
        except ClientError as err:
            logger.error(
                "Couldn't create parameter group %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def delete_parameter_group(self, parameter_group_name):
        """
        Deletes a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to delete.
        :return: Data about the parameter group.
        """
        try:
            response = self.rds_client.delete_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name
            )
        except ClientError as err:
            logger.error(
                "Couldn't delete parameter group %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def get_parameters(self, parameter_group_name, name_prefix="", source=None):
        """
        Gets the parameters that are contained in a DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to query.
        :param name_prefix: When specified, the retrieved list of parameters is
filtered
                            to contain only parameters that start with this
prefix.
        :param source: When specified, only parameters from this source are
retrieved.
```

```
                       For example, a source of 'user' retrieves only parameters
that
                       were set by a user.
        :return: The list of requested parameters.
        """
        try:
            kwargs = {"DBClusterParameterGroupName": parameter_group_name}
            if source is not None:
                kwargs["Source"] = source
            parameters = []
            paginator =
self.rds_client.get_paginator("describe_db_cluster_parameters")
            for page in paginator.paginate(**kwargs):
                parameters += [
                    p
                    for p in page["Parameters"]
                    if p["ParameterName"].startswith(name_prefix)
                ]
        except ClientError as err:
            logger.error(
                "Couldn't get parameters for %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return parameters


    def update_parameters(self, parameter_group_name, update_parameters):
        """
        Updates parameters in a custom DB cluster parameter group.

        :param parameter_group_name: The name of the parameter group to update.
        :param update_parameters: The parameters to update in the group.
        :return: Data about the modified parameter group.
        """
        try:
            response = self.rds_client.modify_db_cluster_parameter_group(
                DBClusterParameterGroupName=parameter_group_name,
                Parameters=update_parameters,
            )
        except ClientError as err:
            logger.error(
                "Couldn't update parameters in %s. Here's why: %s: %s",
                parameter_group_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def get_db_cluster(self, cluster_name):
        """
        Gets data about an Aurora DB cluster.

        :param cluster_name: The name of the DB cluster to retrieve.
        :return: The retrieved DB cluster.
        """
        try:
            response = self.rds_client.describe_db_clusters(
                DBClusterIdentifier=cluster_name
            )
            cluster = response["DBClusters"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBClusterNotFoundFault":
                logger.info("Cluster %s does not exist.", cluster_name)
            else:
                logger.error(
                    "Couldn't verify the existence of DB cluster %s. Here's why:
%s: %s",
                    cluster_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return cluster


    def create_db_cluster(
        self,
        cluster_name,
        parameter_group_name,
        db_name,
        db_engine,
        db_engine_version,
        admin_name,
        admin_password,
    ):
        """
        Creates a DB cluster that is configured to use the specified parameter
group.
        The newly created DB cluster contains a database that uses the specified
engine and
        engine version.

        :param cluster_name: The name of the DB cluster to create.
        :param parameter_group_name: The name of the parameter group to associate
with
                                       the DB cluster.
        :param db_name: The name of the database to create.
        :param db_engine: The database engine of the database that is created,
such as MySql.
        :param db_engine_version: The version of the database engine.
        :param admin_name: The user name of the database administrator.
        :param admin_password: The password of the database administrator.
        :return: The newly created DB cluster.
        """
        try:
            response = self.rds_client.create_db_cluster(
                DatabaseName=db_name,
                DBClusterIdentifier=cluster_name,
                DBClusterParameterGroupName=parameter_group_name,
                Engine=db_engine,
                EngineVersion=db_engine_version,
                MasterUsername=admin_name,
                MasterUserPassword=admin_password,
            )
            cluster = response["DBCluster"]
        except ClientError as err:
            logger.error(
                "Couldn't create database %s. Here's why: %s: %s",
                db_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return cluster


    def delete_db_cluster(self, cluster_name):
        """
        Deletes a DB cluster.

        :param cluster_name: The name of the DB cluster to delete.
        """
        try:
            self.rds_client.delete_db_cluster(
                DBClusterIdentifier=cluster_name, SkipFinalSnapshot=True
            )
            logger.info("Deleted DB cluster %s.", cluster_name)
        except ClientError:
            logger.exception("Couldn't delete DB cluster %s.", cluster_name)
            raise


    def create_cluster_snapshot(self, snapshot_id, cluster_id):
        """
        Creates a snapshot of a DB cluster.

        :param snapshot_id: The ID to give the created snapshot.
        :param cluster_id: The DB cluster to snapshot.
        :return: Data about the newly created snapshot.
        """
        try:
            response = self.rds_client.create_db_cluster_snapshot(
                DBClusterSnapshotIdentifier=snapshot_id,
 DBClusterIdentifier=cluster_id
            )
            snapshot = response["DBClusterSnapshot"]
        except ClientError as err:
            logger.error(
                "Couldn't create snapshot of %s. Here's why: %s: %s",
                cluster_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return snapshot
```

```
   def get_cluster_snapshot(self, snapshot_id):
       """
       Gets a DB cluster snapshot.

       :param snapshot_id: The ID of the snapshot to retrieve.
       :return: The retrieved snapshot.
       """
       try:
           response = self.rds_client.describe_db_cluster_snapshots(
               DBClusterSnapshotIdentifier=snapshot_id
           )
           snapshot = response["DBClusterSnapshots"][0]
       except ClientError as err:
           logger.error(
               "Couldn't get DB cluster snapshot %s. Here's why: %s: %s",
               snapshot_id,
               err.response["Error"]["Code"],
               err.response["Error"]["Message"],
           )
           raise
       else:
           return snapshot


   def create_instance_in_cluster(
       self, instance_id, cluster_id, db_engine, instance_class
   ):
       """
       Creates a database instance in an existing DB cluster. The first database 
that is
       created defaults to a read-write DB instance.

       :param instance_id: The ID to give the newly created DB instance.
       :param cluster_id: The ID of the DB cluster where the DB instance is 
created.
       :param db_engine: The database engine of a database to create in the DB 
instance.
                         This must be compatible with the configured parameter 
group
                         of the DB cluster.
       :param instance_class: The DB instance class for the newly created DB 
instance.
       :return: Data about the newly created DB instance.
```

```
        """
        try:
            response = self.rds_client.create_db_instance(
                DBInstanceIdentifier=instance_id,
                DBClusterIdentifier=cluster_id,
                Engine=db_engine,
                DBInstanceClass=instance_class,
            )
            db_inst = response["DBInstance"]
        except ClientError as err:
            logger.error(
                "Couldn't create DB instance %s. Here's why: %s: %s",
                instance_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return db_inst


    def get_engine_versions(self, engine, parameter_group_family=None):
        """
        Gets database engine versions that are available for the specified engine
        and parameter group family.

        :param engine: The database engine to look up.
        :param parameter_group_family: When specified, restricts the returned
 list of
                                       engine versions to those that are
 compatible with
                                       this parameter group family.
        :return: The list of database engine versions.
        """
        try:
            kwargs = {"Engine": engine}
            if parameter_group_family is not None:
                kwargs["DBParameterGroupFamily"] = parameter_group_family
            response = self.rds_client.describe_db_engine_versions(**kwargs)
            versions = response["DBEngineVersions"]
        except ClientError as err:
            logger.error(
                "Couldn't get engine versions for %s. Here's why: %s: %s",
                engine,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return versions


    def get_orderable_instances(self, db_engine, db_engine_version):
        """
        Gets DB instance options that can be used to create DB instances that are
        compatible with a set of specifications.

        :param db_engine: The database engine that must be supported by the DB
instance.
        :param db_engine_version: The engine version that must be supported by
the DB instance.
        :return: The list of DB instance options that can be used to create a
compatible DB instance.
        """
        try:
            inst_opts = []
            paginator = self.rds_client.get_paginator(
                "describe_orderable_db_instance_options"
            )
            for page in paginator.paginate(
                Engine=db_engine, EngineVersion=db_engine_version
            ):
                inst_opts += page["OrderableDBInstanceOptions"]
        except ClientError as err:
            logger.error(
                "Couldn't get orderable DB instances. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return inst_opts


    def get_db_instance(self, instance_id):
        """
        Gets data about a DB instance.
        :param instance_id: The ID of the DB instance to retrieve.
        :return: The retrieved DB instance.
        """
        try:
            response = self.rds_client.describe_db_instances(
                DBInstanceIdentifier=instance_id
            )
            db_inst = response["DBInstances"][0]
        except ClientError as err:
            if err.response["Error"]["Code"] == "DBInstanceNotFound":
                logger.info("Instance %s does not exist.", instance_id)
            else:
                logger.error(
                    "Couldn't get DB instance %s. Here's why: %s: %s",
                    instance_id,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        else:
            return db_inst


    def delete_db_instance(self, instance_id):
        """
        Deletes a DB instance.

        :param instance_id: The ID of the DB instance to delete.
        :return: Data about the deleted DB instance.
        """
        try:
            response = self.rds_client.delete_db_instance(
                DBInstanceIdentifier=instance_id,
                SkipFinalSnapshot=True,
                DeleteAutomatedBackups=True,
            )
            db_inst = response["DBInstance"]
        except ClientError as err:
            logger.error(
                "Couldn't delete DB instance %s. Here's why: %s: %s",
                instance_id,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
```

```
                raise
            else:
                return db_inst
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster
  - DeleteDBClusterParameterGroup
  - DeleteDBInstance
  - DescribeDBClusterParameterGroups
  - DescribeDBClusterParameters
  - DescribeDBClusterSnapshots
  - DescribeDBClusters
  - DescribeDBEngineVersions
  - DescribeDBInstances
  - DescribeOrderableDBInstanceOptions
  - ModifyDBClusterParameterGroup

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

A library containing the scenario-specific functions for the Aurora scenario.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use phf::{phf_set, Set};
use secrecy::SecretString;
use std::{collections::HashMap, fmt::Display, time::Duration};

use aws_sdk_rds::{
    error::ProvideErrorMetadata,

 operation::create_db_cluster_parameter_group::CreateDbClusterParameterGroupOutput,
    types::{DbCluster, DbClusterParameterGroup, DbClusterSnapshot, DbInstance,
 Parameter},
};
use sdk_examples_test_utils::waiter::Waiter;
use tracing::{info, trace, warn};

const DB_ENGINE: &str = "aurora-mysql";
const DB_CLUSTER_PARAMETER_GROUP_NAME: &str =
 "RustSDKCodeExamplesDBParameterGroup";
const DB_CLUSTER_PARAMETER_GROUP_DESCRIPTION: &str =
    "Parameter Group created by Rust SDK Code Example";
const DB_CLUSTER_IDENTIFIER: &str = "RustSDKCodeExamplesDBCluster";
const DB_INSTANCE_IDENTIFIER: &str = "RustSDKCodeExamplesDBInstance";

static FILTER_PARAMETER_NAMES: Set<&'static str> = phf_set! {
    "auto_increment_offset",
    "auto_increment_increment",
};

#[derive(Debug, PartialEq, Eq)]
struct MetadataError {
    message: Option<String>,
    code: Option<String>,
}

impl MetadataError {
    fn from(err: &dyn ProvideErrorMetadata) -> Self {
        MetadataError {
            message: err.message().map(String::from),
            code: err.code().map(String::from),
        }
    }
}

impl Display for MetadataError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        let display = match (&self.message, &self.code) {
            (None, None) => "Unknown".to_string(),
            (None, Some(code)) => format!("({code})"),
            (Some(message), None) => message.to_string(),
            (Some(message), Some(code)) => format!("{message} ({code})"),
        };
        write!(f, "{display}")
    }
}

#[derive(Debug, PartialEq, Eq)]
pub struct ScenarioError {
    message: String,
    context: Option<MetadataError>,
}

impl ScenarioError {
    pub fn with(message: impl Into<String>) -> Self {
        ScenarioError {
            message: message.into(),
            context: None,
        }
    }

    pub fn new(message: impl Into<String>, err: &dyn ProvideErrorMetadata) ->
 Self {
        ScenarioError {
            message: message.into(),
            context: Some(MetadataError::from(err)),
        }
    }
}

impl std::error::Error for ScenarioError {}
impl Display for ScenarioError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match &self.context {
            Some(c) => write!(f, "{}: {}", self.message, c),
            None => write!(f, "{}", self.message),
        }
    }
}

// Parse the ParameterName, Description, and AllowedValues values and display
 them.
#[derive(Debug)]
pub struct AuroraScenarioParameter {
    name: String,
    allowed_values: String,
    current_value: String,
}

impl Display for AuroraScenarioParameter {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "{}: {} (allowed: {})",
            self.name, self.current_value, self.allowed_values
        )
    }
}

impl From<aws_sdk_rds::types::Parameter> for AuroraScenarioParameter {
    fn from(value: aws_sdk_rds::types::Parameter) -> Self {
        AuroraScenarioParameter {
            name: value.parameter_name.unwrap_or_default(),
            allowed_values: value.allowed_values.unwrap_or_default(),
            current_value: value.parameter_value.unwrap_or_default(),
        }
    }
}

pub struct AuroraScenario {
    rds: crate::rds::Rds,
    engine_family: Option<String>,
    engine_version: Option<String>,
    instance_class: Option<String>,
    db_cluster_parameter_group: Option<DbClusterParameterGroup>,
    db_cluster_identifier: Option<String>,
    db_instance_identifier: Option<String>,
    username: Option<String>,
    password: Option<SecretString>,
}
```

```
impl AuroraScenario {
    pub fn new(client: crate::rds::Rds) -> Self {
        AuroraScenario {
            rds: client,
            engine_family: None,
            engine_version: None,
            instance_class: None,
            db_cluster_parameter_group: None,
            db_cluster_identifier: None,
            db_instance_identifier: None,
            username: None,
            password: None,
        }
    }

    // snippet-start:[rust.aurora.get_engines.usage]
    // Get available engine families for Aurora MySql.
 rds.DescribeDbEngineVersions(Engine='aurora-mysql') and build a set of the
 'DBParameterGroupFamily' field values. I get {aurora-mysql8.0, aurora-mysql5.7}.
    pub async fn get_engines(&self) -> Result<HashMap<String, Vec<String>>,
 ScenarioError> {
        let describe_db_engine_versions =
 self.rds.describe_db_engine_versions(DB_ENGINE).await;
        trace!(versions=?describe_db_engine_versions, "full list of versions");

        if let Err(err) = describe_db_engine_versions {
            return Err(ScenarioError::new(
                "Failed to retrieve DB Engine Versions",
                &err,
            ));
        };

        let version_count = describe_db_engine_versions
            .as_ref()
            .map(|o| o.db_engine_versions().len())
            .unwrap_or_default();
        info!(version_count, "got list of versions");

        // Create a map of engine families to their available versions.
        let mut versions = HashMap::<String, Vec<String>>::new();
        describe_db_engine_versions
            .unwrap()
            .db_engine_versions()
            .iter()
            .filter_map(
                |v| match (&v.db_parameter_group_family, &v.engine_version) {
                    (Some(family), Some(version)) => Some((family.clone(),
version.clone())),
                    _ => None,
                },
            )
            .for_each(|(family, version)|
versions.entry(family).or_default().push(version));

        Ok(versions)
    }
    // snippet-end:[rust.aurora.get_engines.usage]

    // snippet-start:[rust.aurora.get_instance_classes.usage]
    pub async fn get_instance_classes(&self) -> Result<Vec<String>,
ScenarioError> {
        let describe_orderable_db_instance_options_items = self
            .rds
            .describe_orderable_db_instance_options(
                DB_ENGINE,
                self.engine_version
                    .as_ref()
                    .expect("engine version for db instance options")
                    .as_str(),
            )
            .await;

        describe_orderable_db_instance_options_items
            .map(|options| {
                options
                    .iter()
                    .map(|o|
o.db_instance_class().unwrap_or_default().to_string())
                    .collect::<Vec<String>>()
            })
            .map_err(|err| ScenarioError::new("Could not get available instance
classes", &err))
    }
    // snippet-end:[rust.aurora.get_instance_classes.usage]

    // snippet-start:[rust.aurora.set_engine.usage]
```

```
   // Select an engine family and create a custom DB cluster parameter group.
rds.CreateDbClusterParameterGroup(DBParameterGroupFamily='aurora-mysql8.0')
   pub async fn set_engine(&mut self, engine: &str, version: &str) -> Result<(),
ScenarioError> {
       self.engine_family = Some(engine.to_string());
       self.engine_version = Some(version.to_string());
       let create_db_cluster_parameter_group = self
           .rds
           .create_db_cluster_parameter_group(
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_CLUSTER_PARAMETER_GROUP_DESCRIPTION,
               engine,
           )
           .await;

       match create_db_cluster_parameter_group {
           Ok(CreateDbClusterParameterGroupOutput {
               db_cluster_parameter_group: None,
               ..
           }) => {
               return Err(ScenarioError::with(
                   "CreateDBClusterParameterGroup had empty response",
               ));
           }
           Err(error) => {
               if error.code() == Some("DBParameterGroupAlreadyExists") {
                   info!("Cluster Parameter Group already exists, nothing to
do");
               } else {
                   return Err(ScenarioError::new(
                       "Could not create Cluster Parameter Group",
                       &error,
                   ));
               }
           }
           _ => {
               info!("Created Cluster Parameter Group");
           }
       }

       Ok(())
   }
   // snippet-end:[rust.aurora.set_engine.usage]
```

```
    pub fn set_instance_class(&mut self, instance_class: Option<String>) {
        self.instance_class = instance_class;
    }

    pub fn set_login(&mut self, username: Option<String>, password:
 Option<SecretString>) {
        self.username = username;
        self.password = password;
    }

    pub async fn connection_string(&self) -> Result<String, ScenarioError> {
        let cluster = self.get_cluster().await?;
        let endpoint = cluster.endpoint().unwrap_or_default();
        let port = cluster.port().unwrap_or_default();
        let username = cluster.master_username().unwrap_or_default();
        Ok(format!("mysql -h {endpoint} -P {port} -u {username} -p"))
    }

    // snippet-start:[rust.aurora.get_cluster.usage]
    pub async fn get_cluster(&self) -> Result<DbCluster, ScenarioError> {
        let describe_db_clusters_output = self
            .rds
            .describe_db_clusters(
                self.db_cluster_identifier
                    .as_ref()
                    .expect("cluster identifier")
                    .as_str(),
            )
            .await;
        if let Err(err) = describe_db_clusters_output {
            return Err(ScenarioError::new("Failed to get cluster", &err));
        }

        let db_cluster = describe_db_clusters_output
            .unwrap()
            .db_clusters
            .and_then(|output| output.first().cloned());

        db_cluster.ok_or_else(|| ScenarioError::with("Did not find the cluster"))
    }
    // snippet-end:[rust.aurora.get_cluster.usage]

    // snippet-start:[rust.aurora.cluster_parameters.usage]
    // Get the parameter group. rds.DescribeDbClusterParameterGroups
```

```
   // Get parameters in the group. This is a long list so you will have to
paginate. Find the auto_increment_offset and auto_increment_increment parameters
(by ParameterName). rds.DescribeDbClusterParameters
   // Parse the ParameterName, Description, and AllowedValues values and display
them.
   pub async fn cluster_parameters(&self) ->
Result<Vec<AuroraScenarioParameter>, ScenarioError> {
       let parameters_output = self
           .rds
           .describe_db_cluster_parameters(DB_CLUSTER_PARAMETER_GROUP_NAME)
           .await;

       if let Err(err) = parameters_output {
           return Err(ScenarioError::new(
               format!("Failed to retrieve parameters for
{DB_CLUSTER_PARAMETER_GROUP_NAME}"),
               &err,
           ));
       }

       let parameters = parameters_output
           .unwrap()
           .into_iter()
           .flat_map(|p| p.parameters.unwrap_or_default().into_iter())
           .filter(|p|
FILTER_PARAMETER_NAMES.contains(p.parameter_name().unwrap_or_default()))
           .map(AuroraScenarioParameter::from)
           .collect::<Vec<_>>();

       Ok(parameters)
   }
   // snippet-end:[rust.aurora.cluster_parameters.usage]

   // snippet-start:[rust.aurora.update_auto_increment.usage]
   // Modify both the auto_increment_offset and auto_increment_increment
parameters in one call in the custom parameter group. Set their ParameterValue
fields to a new allowable value. rds.ModifyDbClusterParameterGroup.
   pub async fn update_auto_increment(
       &self,
       offset: u8,
       increment: u8,
   ) -> Result<(), ScenarioError> {
       let modify_db_cluster_parameter_group = self
           .rds
```

```
            .modify_db_cluster_parameter_group(
                DB_CLUSTER_PARAMETER_GROUP_NAME,
                vec![
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .parameter_value(format!("{offset}"))
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .parameter_value(format!("{increment}"))
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                ],
            )
            .await;

        if let Err(error) = modify_db_cluster_parameter_group {
            return Err(ScenarioError::new(
                "Failed to modify cluster parameter group",
                &error,
            ));
        }

        Ok(())
    }
    // snippet-end:[rust.aurora.update_auto_increment.usage]

    // snippet-start:[rust.aurora.start_cluster_and_instance.usage]
    // Get a list of allowed engine versions.
 rds.DescribeDbEngineVersions(Engine='aurora-mysql', DBParameterGroupFamily=<the
 family used to create your parameter group in step 2>)
    // Create an Aurora DB cluster database cluster that contains a MySql
 database and uses the parameter group you created.
    // Wait for DB cluster to be ready. Call rds.DescribeDBClusters and check for
 Status == 'available'.
    // Get a list of instance classes available for the selected engine
 and engine version. rds.DescribeOrderableDbInstanceOptions(Engine='mysql',
 EngineVersion=).

    // Create a database instance in the cluster.
    // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check
 for DBInstanceStatus == 'available'.
```

```
   pub async fn start_cluster_and_instance(&mut self) -> Result<(),
ScenarioError> {
       if self.password.is_none() {
           return Err(ScenarioError::with(
               "Must set Secret Password before starting a cluster",
           ));
       }
       let create_db_cluster = self
           .rds
           .create_db_cluster(
               DB_CLUSTER_IDENTIFIER,
               DB_CLUSTER_PARAMETER_GROUP_NAME,
               DB_ENGINE,
               self.engine_version.as_deref().expect("engine version"),
               self.username.as_deref().expect("username"),
               self.password
                   .replace(SecretString::new("".to_string()))
                   .expect("password"),
           )
           .await;
       if let Err(err) = create_db_cluster {
           return Err(ScenarioError::new(
               "Failed to create DB Cluster with cluster group",
               &err,
           ));
       }

       self.db_cluster_identifier = create_db_cluster
           .unwrap()
           .db_cluster
           .and_then(|c| c.db_cluster_identifier);

       if self.db_cluster_identifier.is_none() {
           return Err(ScenarioError::with("Created DB Cluster missing
Identifier"));
       }

       info!(
           "Started a db cluster: {}",
           self.db_cluster_identifier
               .as_deref()
               .unwrap_or("Missing ARN")
       );
```

```
        let create_db_instance = self
            .rds
            .create_db_instance(
                self.db_cluster_identifier.as_deref().expect("cluster name"),
                DB_INSTANCE_IDENTIFIER,
                self.instance_class.as_deref().expect("instance class"),
                DB_ENGINE,
            )
            .await;
        if let Err(err) = create_db_instance {
            return Err(ScenarioError::new(
                "Failed to create Instance in DB Cluster",
                &err,
            ));
        }

        self.db_instance_identifier = create_db_instance
            .unwrap()
            .db_instance
            .and_then(|i| i.db_instance_identifier);

        // Cluster creation can take up to 20 minutes to become available
        let cluster_max_wait = Duration::from_secs(20 * 60);
        let waiter = Waiter::builder().max(cluster_max_wait).build();
        while waiter.sleep().await.is_ok() {
            let cluster = self
                .rds
                .describe_db_clusters(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = cluster {
                warn!(?err, "Failed to describe cluster while waiting for
ready");
                continue;
            }

            let instance = self
                .rds
                .describe_db_instance(
                    self.db_instance_identifier
```

```
                        .as_deref()
                        .expect("instance identifier"),
                )
                .await;
            if let Err(err) = instance {
                return Err(ScenarioError::new(
                    "Failed to find instance for cluster",
                    &err,
                ));
            }

            let instances_available = instance
                .unwrap()
                .db_instances()
                .iter()
                .all(|instance| instance.db_instance_status() ==
Some("Available"));

            let endpoints = self
                .rds
                .describe_db_cluster_endpoints(
                    self.db_cluster_identifier
                        .as_deref()
                        .expect("cluster identifier"),
                )
                .await;

            if let Err(err) = endpoints {
                return Err(ScenarioError::new(
                    "Failed to find endpoint for cluster",
                    &err,
                ));
            }

            let endpoints_available = endpoints
                .unwrap()
                .db_cluster_endpoints()
                .iter()
                .all(|endpoint| endpoint.status() == Some("available"));

            if instances_available && endpoints_available {
                return Ok(());
            }
        }
```

```
       Err(ScenarioError::with("timed out waiting for cluster"))
   }
   // snippet-end:[rust.aurora.start_cluster_and_instance.usage]

   // snippet-start:[rust.aurora.snapshot.usage]
   // Create a snapshot of the DB cluster. rds.CreateDbClusterSnapshot.
   // Wait for the snapshot to create. rds.DescribeDbClusterSnapshots until
Status == 'available'.
   pub async fn snapshot(&self, name: &str) -> Result<DbClusterSnapshot,
ScenarioError> {
       let id = self.db_cluster_identifier.as_deref().unwrap_or_default();
       let snapshot = self
           .rds
           .snapshot_cluster(id, format!("{id}_{name}").as_str())
           .await;
       match snapshot {
           Ok(output) => match output.db_cluster_snapshot {
               Some(snapshot) => Ok(snapshot),
               None => Err(ScenarioError::with("Missing Snapshot")),
           },
           Err(err) => Err(ScenarioError::new("Failed to create snapshot",
&err)),
       }
   }
   // snippet-end:[rust.aurora.snapshot.usage]

   // snippet-start:[rust.aurora.clean_up.usage]
   pub async fn clean_up(self) -> Result<(), Vec<ScenarioError>> {
       let mut clean_up_errors: Vec<ScenarioError> = vec![];

       // Delete the instance. rds.DeleteDbInstance.
       let delete_db_instance = self
           .rds
           .delete_db_instance(
               self.db_instance_identifier
                   .as_deref()
                   .expect("instance identifier"),
           )
           .await;
       if let Err(err) = delete_db_instance {
           let identifier = self
               .db_instance_identifier
               .as_deref()
               .unwrap_or("Missing Instance Identifier");
           let message = format!("failed to delete db instance {identifier}");
           clean_up_errors.push(ScenarioError::new(message, &err));
       } else {
           // Wait for the instance to delete
           let waiter = Waiter::default();
           while waiter.sleep().await.is_ok() {
               let describe_db_instances =
self.rds.describe_db_instances().await;
               if let Err(err) = describe_db_instances {
                   clean_up_errors.push(ScenarioError::new(
                       "Failed to check instance state during deletion",
                       &err,
                   ));
                   break;
               }
               let db_instances = describe_db_instances
                   .unwrap()
                   .db_instances()
                   .iter()
                   .filter(|instance| instance.db_cluster_identifier ==
self.db_cluster_identifier)
                   .cloned()
                   .collect::<Vec<DbInstance>>();

               if db_instances.is_empty() {
                   trace!("Delete Instance waited and no instances were found");
                   break;
               }
               match db_instances.first().unwrap().db_instance_status() {
                   Some("Deleting") => continue,
                   Some(status) => {
                       info!("Attempting to delete but instances is in
{status}");
                       continue;
                   }
                   None => {
                       warn!("No status for DB instance");
                       break;
                   }
               }
           }
       }
```

```
        // Delete the DB cluster. rds.DeleteDbCluster.
        let delete_db_cluster = self
            .rds
            .delete_db_cluster(
                self.db_cluster_identifier
                    .as_deref()
                    .expect("cluster identifier"),
            )
            .await;

        if let Err(err) = delete_db_cluster {
            let identifier = self
                .db_cluster_identifier
                .as_deref()
                .unwrap_or("Missing DB Cluster Identifier");
            let message = format!("failed to delete db cluster {identifier}");
            clean_up_errors.push(ScenarioError::new(message, &err));
        } else {
            // Wait for the instance and cluster to fully delete.
rds.DescribeDbInstances and rds.DescribeDbClusters until both are not found.
            let waiter = Waiter::default();
            while waiter.sleep().await.is_ok() {
                let describe_db_clusters = self
                    .rds
                    .describe_db_clusters(
                        self.db_cluster_identifier
                            .as_deref()
                            .expect("cluster identifier"),
                    )
                    .await;
                if let Err(err) = describe_db_clusters {
                    clean_up_errors.push(ScenarioError::new(
                        "Failed to check cluster state during deletion",
                        &err,
                    ));
                    break;
                }
                let describe_db_clusters = describe_db_clusters.unwrap();
                let db_clusters = describe_db_clusters.db_clusters();
                if db_clusters.is_empty() {
                    trace!("Delete cluster waited and no clusters were found");
                    break;
                }
                match db_clusters.first().unwrap().status() {
                    Some("Deleting") => continue,
                    Some(status) => {
                        info!("Attempting to delete but clusters is in
 {status}");
                        continue;
                    }
                    None => {
                        warn!("No status for DB cluster");
                        break;
                    }
                }
            }
        }

        // Delete the DB cluster parameter group.
 rds.DeleteDbClusterParameterGroup.
        let delete_db_cluster_parameter_group = self
            .rds
            .delete_db_cluster_parameter_group(
                self.db_cluster_parameter_group
                    .map(|g| {
                        g.db_cluster_parameter_group_name
                            .unwrap_or_else(||
 DB_CLUSTER_PARAMETER_GROUP_NAME.to_string())
                    })
                    .as_deref()
                    .expect("cluster parameter group name"),
            )
            .await;
        if let Err(error) = delete_db_cluster_parameter_group {
            clean_up_errors.push(ScenarioError::new(
                "Failed to delete the db cluster parameter group",
                &error,
            ))
        }

        if clean_up_errors.is_empty() {
            Ok(())
        } else {
            Err(clean_up_errors)
        }
    }
    // snippet-end:[rust.aurora.clean_up.usage]
}
```

```
#[cfg(test)]
pub mod tests;
```

Tests for the library using automocks around the RDS Client wrapper.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use crate::rds::MockRdsImpl;

use super::*;

use std::io::{Error, ErrorKind};

use assert_matches::assert_matches;
use aws_sdk_rds::{
    error::SdkError,
    operation::{
        create_db_cluster::{CreateDBClusterError, CreateDbClusterOutput},
        create_db_cluster_parameter_group::CreateDBClusterParameterGroupError,
        create_db_cluster_snapshot::{CreateDBClusterSnapshotError,
 CreateDbClusterSnapshotOutput},
        create_db_instance::{CreateDBInstanceError, CreateDbInstanceOutput},
        delete_db_cluster::DeleteDbClusterOutput,
        delete_db_cluster_parameter_group::DeleteDbClusterParameterGroupOutput,
        delete_db_instance::DeleteDbInstanceOutput,
        describe_db_cluster_endpoints::DescribeDbClusterEndpointsOutput,
        describe_db_cluster_parameters::{
            DescribeDBClusterParametersError, DescribeDbClusterParametersOutput,
        },
        describe_db_clusters::{DescribeDBClustersError,
 DescribeDbClustersOutput},
        describe_db_engine_versions::{
            DescribeDBEngineVersionsError, DescribeDbEngineVersionsOutput,
        },
        describe_db_instances::{DescribeDBInstancesError,
 DescribeDbInstancesOutput},

 describe_orderable_db_instance_options::DescribeOrderableDBInstanceOptionsError,
        modify_db_cluster_parameter_group::{
            ModifyDBClusterParameterGroupError,
 ModifyDbClusterParameterGroupOutput,
        },
    },
    types::{
        error::DbParameterGroupAlreadyExistsFault, DbClusterEndpoint,
 DbEngineVersion,
        OrderableDbInstanceOption,
    },
};
use aws_smithy_runtime_api::http::{Response, StatusCode};
use aws_smithy_types::body::SdkBody;
use mockall::predicate::eq;
use secrecy::ExposeSecret;

// snippet-start:[rust.aurora.set_engine.test]
#[tokio::test]
async fn test_scenario_set_engine() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .with(
            eq("RustSDKCodeExamplesDBParameterGroup"),
            eq("Parameter Group created by Rust SDK Code Example"),
            eq("aurora-mysql"),
        )
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()

 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);

    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;

    assert_eq!(set_engine, Ok(()));
    assert_eq!(Some("aurora-mysql"), scenario.engine_family.as_deref());
    assert_eq!(Some("aurora-mysql8.0"), scenario.engine_version.as_deref());
}
```

```
#[tokio::test]
async fn test_scenario_set_engine_not_create() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .with(
            eq("RustSDKCodeExamplesDBParameterGroup"),
            eq("Parameter Group created by Rust SDK Code Example"),
            eq("aurora-mysql"),
        )
        .return_once(|_, _, _|
 Ok(CreateDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);

    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;

    assert!(set_engine.is_err());
}

#[tokio::test]
async fn test_scenario_set_engine_param_group_exists() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .withf(|_, _, _| true)
        .return_once(|_, _, _| {
            Err(SdkError::service_error(

 CreateDBClusterParameterGroupError::DbParameterGroupAlreadyExistsFault(
                    DbParameterGroupAlreadyExistsFault::builder().build(),
                ),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);

    let set_engine = scenario.set_engine("aurora-mysql", "aurora-
mysql8.0").await;
```

```
    assert!(set_engine.is_err());
}
// snippet-end:[rust.aurora.set_engine.test]

// snippet-start:[rust.aurora.get_engines.test]
#[tokio::test]
async fn test_scenario_get_engines() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_engine_versions()
        .with(eq("aurora-mysql"))
        .return_once(|_| {
            Ok(DescribeDbEngineVersionsOutput::builder()
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f1")
                        .engine_version("f1a")
                        .build(),
                )
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f1")
                        .engine_version("f1b")
                        .build(),
                )
                .db_engine_versions(
                    DbEngineVersion::builder()
                        .db_parameter_group_family("f2")
                        .engine_version("f2a")
                        .build(),
                )
                .db_engine_versions(DbEngineVersion::builder().build())
                .build())
        });

    let scenario = AuroraScenario::new(mock_rds);

    let versions_map = scenario.get_engines().await;

    assert_eq!(
        versions_map,
        Ok(HashMap::from([
            ("f1".into(), vec!["f1a".into(), "f1b".into()]),
            ("f2".into(), vec!["f2a".into()])
        ]))
    );
}

#[tokio::test]
async fn test_scenario_get_engines_failed() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_engine_versions()
        .with(eq("aurora-mysql"))
        .return_once(|_| {
            Err(SdkError::service_error(
                DescribeDBEngineVersionsError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_db_engine_versions error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let scenario = AuroraScenario::new(mock_rds);

    let versions_map = scenario.get_engines().await;
    assert_matches!(
        versions_map,
        Err(ScenarioError { message, context: _ }) if message == "Failed to
 retrieve DB Engine Versions"
    );
}
// snippet-end:[rust.aurora.get_engines.test]

// snippet-start:[rust.aurora.get_instance_classes.test]
#[tokio::test]
async fn test_scenario_get_instance_classes() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()
 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    mock_rds
        .expect_describe_orderable_db_instance_options()
        .with(eq("aurora-mysql"), eq("aurora-mysql8.0"))
        .return_once(|_, _| {
            Ok(vec![
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t1")
                    .build(),
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t2")
                    .build(),
                OrderableDbInstanceOption::builder()
                    .db_instance_class("t3")
                    .build(),
            ])
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario
        .set_engine("aurora-mysql", "aurora-mysql8.0")
        .await
        .expect("set engine");

    let instance_classes = scenario.get_instance_classes().await;

    assert_eq!(
        instance_classes,
        Ok(vec!["t1".into(), "t2".into(), "t3".into()])
    );
}

#[tokio::test]
async fn test_scenario_get_instance_classes_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_orderable_db_instance_options()
        .with(eq("aurora-mysql"), eq("aurora-mysql8.0"))
        .return_once(|_, _| {
            Err(SdkError::service_error(
 DescribeOrderableDBInstanceOptionsError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_orderable_db_instance_options_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_family = Some("aurora-mysql".into());
    scenario.engine_version = Some("aurora-mysql8.0".into());

    let instance_classes = scenario.get_instance_classes().await;

    assert_matches!(
        instance_classes,
        Err(ScenarioError {message, context: _}) if message == "Could not get
 available instance classes"
    );
}
// snippet-end:[rust.aurora.get_instance_classes.test]

// snippet-start:[rust.aurora.get_cluster.test]
#[tokio::test]
async fn test_scenario_get_cluster() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let cluster = scenario.get_cluster().await;

    assert!(cluster.is_ok());
```

```
}

#[tokio::test]
async fn test_scenario_get_cluster_missing_cluster() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()

 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| Ok(DescribeDbClustersOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let cluster = scenario.get_cluster().await;

    assert_matches!(cluster, Err(ScenarioError { message, context: _ }) if
 message == "Did not find the cluster");
}

#[tokio::test]
async fn test_scenario_get_cluster_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster_parameter_group()
        .return_once(|_, _, _| {
            Ok(CreateDbClusterParameterGroupOutput::builder()

 .db_cluster_parameter_group(DbClusterParameterGroup::builder().build())
                .build())
        });

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Err(SdkError::service_error(
                DescribeDBClustersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_db_clusters_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let cluster = scenario.get_cluster().await;

    assert_matches!(cluster, Err(ScenarioError { message, context: _ }) if
 message == "Failed to get cluster");
}
// snippet-end:[rust.aurora.get_cluster.test]

#[tokio::test]
async fn test_scenario_connection_string() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .endpoint("test_endpoint")
                        .port(3306)
                        .master_username("test_username")
                        .build(),
                )
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let connection_string = scenario.connection_string().await;

    assert_eq!(
        connection_string,
        Ok("mysql -h test_endpoint -P 3306 -u test_username -p".into())
    );
}

// snippet-start:[rust.aurora.cluster_parameters.test]
#[tokio::test]
async fn test_scenario_cluster_parameters() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_cluster_parameters()
        .with(eq("RustSDKCodeExamplesDBParameterGroup"))
        .return_once(|_| {
            Ok(vec![DescribeDbClusterParametersOutput::builder()
                .parameters(Parameter::builder().parameter_name("a").build())
                .parameters(Parameter::builder().parameter_name("b").build())
                .parameters(
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .build(),
                )
                .parameters(Parameter::builder().parameter_name("c").build())
                .parameters(
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .build(),
                )
                .parameters(Parameter::builder().parameter_name("d").build())
                .build()])
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());

    let params = scenario.cluster_parameters().await.expect("cluster params");
    let names: Vec<String> = params.into_iter().map(|p| p.name).collect();
    assert_eq!(
        names,
        vec!["auto_increment_offset", "auto_increment_increment"]
    );
}

#[tokio::test]
```

```
async fn test_scenario_cluster_parameters_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_describe_db_cluster_parameters()
        .with(eq("RustSDKCodeExamplesDBParameterGroup"))
        .return_once(|_| {
            Err(SdkError::service_error(
                DescribeDBClusterParametersError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe_db_cluster_parameters_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("RustSDKCodeExamplesDBCluster".into());
    let params = scenario.cluster_parameters().await;
    assert_matches!(params, Err(ScenarioError { message, context: _ }) if message
 == "Failed to retrieve parameters for RustSDKCodeExamplesDBParameterGroup");
}
// snippet-end:[rust.aurora.cluster_parameters.test]

// snippet-start:[rust.aurora.update_auto_increment.test]
#[tokio::test]
async fn test_scenario_update_auto_increment() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_modify_db_cluster_parameter_group()
        .withf(|name, params| {
            assert_eq!(name, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(
                params,
                &vec![
                    Parameter::builder()
                        .parameter_name("auto_increment_offset")
                        .parameter_value("10")
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                    Parameter::builder()
                        .parameter_name("auto_increment_increment")
                        .parameter_value("20")
                        .apply_method(aws_sdk_rds::types::ApplyMethod::Immediate)
                        .build(),
                ]
            );
            true
        })
        .return_once(|_, _|
 Ok(ModifyDbClusterParameterGroupOutput::builder().build()));

    let scenario = AuroraScenario::new(mock_rds);

    scenario
        .update_auto_increment(10, 20)
        .await
        .expect("update auto increment");
}

#[tokio::test]
async fn test_scenario_update_auto_increment_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_modify_db_cluster_parameter_group()
        .return_once(|_, _| {
            Err(SdkError::service_error(

 ModifyDBClusterParameterGroupError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "modify_db_cluster_parameter_group_error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let scenario = AuroraScenario::new(mock_rds);

    let update = scenario.update_auto_increment(10, 20).await;
    assert_matches!(update, Err(ScenarioError { message, context: _}) if message
 == "Failed to modify cluster parameter group");
}
// snippet-end:[rust.aurora.update_auto_increment.test]
// snippet-start:[rust.aurora.start_cluster_and_instance.test]
#[tokio::test]
async fn test_start_cluster_and_instance() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_create_db_instance()
        .withf(|cluster, name, class, engine| {
            assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
            assert_eq!(name, "RustSDKCodeExamplesDBInstance");
            assert_eq!(class, "m5.large");
            assert_eq!(engine, "aurora-mysql");
            true
        })
        .return_once(|cluster, name, class, _| {
            Ok(CreateDbInstanceOutput::builder()
                .db_instance(
                    DbInstance::builder()
                        .db_cluster_identifier(cluster)
                        .db_instance_identifier(name)
                        .db_instance_class(class)
                        .build(),
                )
                .build())
        });
    mock_rds
        .expect_describe_db_clusters()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|id| {
            Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });

    mock_rds
        .expect_describe_db_instance()
        .with(eq("RustSDKCodeExamplesDBInstance"))
        .return_once(|name| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_instance_identifier(name)
                        .db_instance_status("Available")
                        .build(),
                )
                .build())
        });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .with(eq("RustSDKCodeExamplesDBCluster"))
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

.db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
        assert!(scenario
            .password
            .replace(SecretString::new("BAD SECRET".into()))
            .unwrap()
            .expose_secret()
            .is_empty());
        assert_eq!(
            scenario.db_cluster_identifier,
            Some("RustSDKCodeExamplesDBCluster".into())
        );
    });
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Err(SdkError::service_error(
                CreateDBClusterError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db cluster error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _}) if message
 == "Failed to create DB Cluster with cluster group")
}
```

```
#[tokio::test]
async fn test_start_cluster_and_instance_cluster_create_missing_id() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .return_once(|_, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()
                .db_cluster(DbCluster::builder().build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context:_ }) if message
 == "Created DB Cluster missing Identifier");
}

#[tokio::test]
async fn test_start_cluster_and_instance_instance_create_error() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
                .build())
        });
    mock_rds
        .expect_create_db_instance()
        .return_once(|_, _, _, _| {
            Err(SdkError::service_error(
                CreateDBInstanceError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create db instance error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    let create = scenario.start_cluster_and_instance().await;
    assert_matches!(create, Err(ScenarioError { message, context: _ }) if message
 == "Failed to create Instance in DB Cluster")
}

#[tokio::test]
async fn test_start_cluster_and_instance_wait_hiccup() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_create_db_cluster()
        .withf(|id, params, engine, version, username, password| {
            assert_eq!(id, "RustSDKCodeExamplesDBCluster");
            assert_eq!(params, "RustSDKCodeExamplesDBParameterGroup");
            assert_eq!(engine, "aurora-mysql");
            assert_eq!(version, "aurora-mysql8.0");
            assert_eq!(username, "test username");
            assert_eq!(password.expose_secret(), "test password");
            true
        })
        .return_once(|id, _, _, _, _, _| {
            Ok(CreateDbClusterOutput::builder()

 .db_cluster(DbCluster::builder().db_cluster_identifier(id).build())
```

```
               .build())
       });

   mock_rds
       .expect_create_db_instance()
       .withf(|cluster, name, class, engine| {
           assert_eq!(cluster, "RustSDKCodeExamplesDBCluster");
           assert_eq!(name, "RustSDKCodeExamplesDBInstance");
           assert_eq!(class, "m5.large");
           assert_eq!(engine, "aurora-mysql");
           true
       })
       .return_once(|cluster, name, class, _| {
           Ok(CreateDbInstanceOutput::builder()
               .db_instance(
                   DbInstance::builder()
                       .db_cluster_identifier(cluster)
                       .db_instance_identifier(name)
                       .db_instance_class(class)
                       .build(),
               )
               .build())
       });

   mock_rds
       .expect_describe_db_clusters()
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .times(1)
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe cluster error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       })
       .with(eq("RustSDKCodeExamplesDBCluster"))
       .times(1)
       .returning(|id| {
           Ok(DescribeDbClustersOutput::builder()

.db_clusters(DbCluster::builder().db_cluster_identifier(id).build())
```

```
                .build())
        });

    mock_rds.expect_describe_db_instance().return_once(|name| {
        Ok(DescribeDbInstancesOutput::builder()
            .db_instances(
                DbInstance::builder()
                    .db_instance_identifier(name)
                    .db_instance_status("Available")
                    .build(),
            )
            .build())
    });

    mock_rds
        .expect_describe_db_cluster_endpoints()
        .return_once(|_| {
            Ok(DescribeDbClusterEndpointsOutput::builder()

 .db_cluster_endpoints(DbClusterEndpoint::builder().status("available").build())
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.engine_version = Some("aurora-mysql8.0".into());
    scenario.instance_class = Some("m5.large".into());
    scenario.username = Some("test username".into());
    scenario.password = Some(SecretString::new("test password".into()));

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let create = scenario.start_cluster_and_instance().await;
        assert!(create.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::advance(Duration::from_secs(1)).await;
    tokio::time::resume();
    let _ = assertions.await;
}
// snippet-end:[rust.aurora.start_cluster_and_instance.test]

// snippet-start:[rust.aurora.clean_up.test]
#[tokio::test]
```

```
async fn test_scenario_clean_up() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| Ok(DescribeDbInstancesOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
                        .status("Deleting")
                        .build(),
                )
                .build())
        })
        .with(eq("MockCluster"))
        .times(1)
        .returning(|_| Ok(DescribeDbClustersOutput::builder().build()));

    mock_rds
        .expect_delete_db_cluster_parameter_group()
        .with(eq("MockParamGroup"))
        .return_once(|_|
 Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some(String::from("MockCluster"));
    scenario.db_instance_identifier = Some(String::from("MockInstance"));
    scenario.db_cluster_parameter_group = Some(
        DbClusterParameterGroup::builder()
            .db_cluster_parameter_group_name("MockParamGroup")
            .build(),
    );

    tokio::time::pause();
    let assertions = tokio::spawn(async move {
        let clean_up = scenario.clean_up().await;
        assert!(clean_up.is_ok());
    });

    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}

#[tokio::test]
async fn test_scenario_clean_up_errors() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_delete_db_instance()
        .with(eq("MockInstance"))
        .return_once(|_| Ok(DeleteDbInstanceOutput::builder().build()));

    mock_rds
        .expect_describe_db_instances()
        .with()
        .times(1)
        .returning(|| {
            Ok(DescribeDbInstancesOutput::builder()
                .db_instances(
                    DbInstance::builder()
                        .db_cluster_identifier("MockCluster")
                        .db_instance_status("Deleting")
                        .build(),
                )
                .build())
        })
        .with()
        .times(1)
        .returning(|| {
            Err(SdkError::service_error(
                DescribeDBInstancesError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "describe db instances error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
            ))
        });

    mock_rds
        .expect_delete_db_cluster()
        .with(eq("MockCluster"))
        .return_once(|_| Ok(DeleteDbClusterOutput::builder().build()));

    mock_rds
        .expect_describe_db_clusters()
        .with(eq("MockCluster"))
        .times(1)
        .returning(|id| {
            Ok(DescribeDbClustersOutput::builder()
                .db_clusters(
                    DbCluster::builder()
                        .db_cluster_identifier(id)
```

```
                       .status("Deleting")
                       .build(),
               )
               .build())
       })
       .with(eq("MockCluster"))
       .times(1)
       .returning(|_| {
           Err(SdkError::service_error(
               DescribeDBClustersError::unhandled(Box::new(Error::new(
                   ErrorKind::Other,
                   "describe db clusters error",
               ))),
               Response::new(StatusCode::try_from(400).unwrap(),
SdkBody::empty()),
           ))
       });

   mock_rds
       .expect_delete_db_cluster_parameter_group()
       .with(eq("MockParamGroup"))
       .return_once(|_|
Ok(DeleteDbClusterParameterGroupOutput::builder().build()));

   let mut scenario = AuroraScenario::new(mock_rds);
   scenario.db_cluster_identifier = Some(String::from("MockCluster"));
   scenario.db_instance_identifier = Some(String::from("MockInstance"));
   scenario.db_cluster_parameter_group = Some(
       DbClusterParameterGroup::builder()
           .db_cluster_parameter_group_name("MockParamGroup")
           .build(),
   );

   tokio::time::pause();
   let assertions = tokio::spawn(async move {
       let clean_up = scenario.clean_up().await;
       assert!(clean_up.is_err());
       let errs = clean_up.unwrap_err();
       assert_eq!(errs.len(), 2);
       assert_matches!(errs.get(0), Some(ScenarioError {message, context: _}) if
message == "Failed to check instance state during deletion");
       assert_matches!(errs.get(1), Some(ScenarioError {message, context: _}) if
message == "Failed to check cluster state during deletion");
   });
```

```
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Instances
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for first
 Describe Cluster
    tokio::time::advance(Duration::from_secs(1)).await; // Wait for second
 Describe Cluster
    tokio::time::resume();
    let _ = assertions.await;
}
// snippet-end:[rust.aurora.clean_up.test]

// snippet-start:[rust.aurora.snapshot.test]
#[tokio::test]
async fn test_scenario_snapshot() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_snapshot_cluster()
        .with(eq("MockCluster"), eq("MockCluster_MockSnapshot"))
        .times(1)
        .return_once(|_, _| {
            Ok(CreateDbClusterSnapshotOutput::builder()
                .db_cluster_snapshot(
                    DbClusterSnapshot::builder()
                        .db_cluster_identifier("MockCluster")

 .db_cluster_snapshot_identifier("MockCluster_MockSnapshot")
                        .build(),
                )
                .build())
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("MockCluster".into());
    let create_snapshot = scenario.snapshot("MockSnapshot").await;
    assert!(create_snapshot.is_ok());
}

#[tokio::test]
async fn test_scenario_snapshot_error() {
    let mut mock_rds = MockRdsImpl::default();
    mock_rds
        .expect_snapshot_cluster()
        .with(eq("MockCluster"), eq("MockCluster_MockSnapshot"))
        .times(1)
        .return_once(|_, _| {
            Err(SdkError::service_error(
                CreateDBClusterSnapshotError::unhandled(Box::new(Error::new(
                    ErrorKind::Other,
                    "create snapshot error",
                ))),
                Response::new(StatusCode::try_from(400).unwrap(),
 SdkBody::empty()),
            ))
        });

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("MockCluster".into());
    let create_snapshot = scenario.snapshot("MockSnapshot").await;
    assert_matches!(create_snapshot, Err(ScenarioError { message, context: _}) if
 message == "Failed to create snapshot");
}

#[tokio::test]
async fn test_scenario_snapshot_invalid() {
    let mut mock_rds = MockRdsImpl::default();

    mock_rds
        .expect_snapshot_cluster()
        .with(eq("MockCluster"), eq("MockCluster_MockSnapshot"))
        .times(1)
        .return_once(|_, _|
 Ok(CreateDbClusterSnapshotOutput::builder().build()));

    let mut scenario = AuroraScenario::new(mock_rds);
    scenario.db_cluster_identifier = Some("MockCluster".into());
    let create_snapshot = scenario.snapshot("MockSnapshot").await;
    assert_matches!(create_snapshot, Err(ScenarioError { message, context: _}) if
 message == "Missing Snapshot");
}
// snippet-end:[rust.aurora.snapshot.test]
```

A binary to run the scenario from front to end, using inquirer so that the user can make some decisions.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use std::fmt::Display;

use anyhow::anyhow;
use aurora_code_examples::{
    aurora_scenario::{AuroraScenario, ScenarioError},
    rds::Rds as RdsClient,
};
use aws_sdk_rds::Client;
use inquire::{validator::StringValidator, CustomUserError};
use secrecy::SecretString;
use tracing::warn;

#[derive(Default, Debug)]
struct Warnings(Vec<String>);

impl Warnings {
    fn new() -> Self {
        Warnings(Vec::with_capacity(5))
    }

    fn push(&mut self, warning: &str, error: ScenarioError) {
        let formatted = format!("{warning}: {error}");
        warn!("{formatted}");
        self.0.push(formatted);
    }

    fn is_empty(&self) -> bool {
        self.0.is_empty()
    }
}

impl Display for Warnings {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        writeln!(f, "Warnings:")?;
        for warning in &self.0 {
            writeln!(f, "{: >4}- {warning}", "")?;
        }
        Ok(())
    }
}

fn select(
    prompt: &str,
    choices: Vec<String>,
    error_message: &str,
) -> Result<String, anyhow::Error> {
    inquire::Select::new(prompt, choices)
        .prompt()
        .map_err(|error| anyhow!("{error_message}: {error}"))
}

// Prepare the Aurora Scenario. Prompt for several settings that are optional to
 the Scenario, but that the user should choose for the demo.
// This includes the engine, engine version, and instance class.
async fn prepare_scenario(rds: RdsClient) -> Result<AuroraScenario,
 anyhow::Error> {
    let mut scenario = AuroraScenario::new(rds);

    // Get available engine families for Aurora MySql.
 rds.DescribeDbEngineVersions(Engine='aurora-mysql') and build a set of the
 'DBParameterGroupFamily' field values. I get {aurora-mysql8.0, aurora-mysql5.7}.
    let available_engines = scenario.get_engines().await;
    if let Err(error) = available_engines {
        return Err(anyhow!("Failed to get available engines: {}", error));
    }
    let available_engines = available_engines.unwrap();

    // Select an engine family and create a custom DB cluster parameter group.
 rds.CreateDbClusterParameterGroup(DBParameterGroupFamily='aurora-mysql8.0')
    let engine = select(
        "Select an Aurora engine family",
        available_engines.keys().cloned().collect::<Vec<String>>(),
        "Invalid engine selection",
    )?;

    let version = select(
        format!("Select an Aurora engine version for {engine}").as_str(),
        available_engines.get(&engine).cloned().unwrap_or_default(),
        "Invalid engine version selection",
    )?;
    let set_engine = scenario.set_engine(engine.as_str(),
 version.as_str()).await;
    if let Err(error) = set_engine {
        return Err(anyhow!("Could not set engine: {}", error));
    }

    let instance_classes = scenario.get_instance_classes().await;
    match instance_classes {
        Ok(classes) => {
            let instance_class = select(
                format!("Select an Aurora instance class for {engine}").as_str(),
                classes,
                "Invalid instance class selection",
            )?;
            scenario.set_instance_class(Some(instance_class))
        }
        Err(err) => return Err(anyhow!("Failed to get instance classes for
 engine: {err}")),
    }

    Ok(scenario)
}

// Prepare the cluster, creating a custom parameter group overriding some group
 parameters based on user input.
async fn prepare_cluster(scenario: &mut AuroraScenario, warnings: &mut Warnings)
 -> Result<(), ()> {
    show_parameters(scenario, warnings).await;

    let offset = prompt_number_or_default(warnings, "auto_increment_offset", 5);
    let increment = prompt_number_or_default(warnings,
 "auto_increment_increment", 3);

    // Modify both the auto_increment_offset and auto_increment_increment
 parameters in one call in the custom parameter group. Set their ParameterValue
 fields to a new allowable value. rds.ModifyDbClusterParameterGroup.
    let update_auto_increment = scenario.update_auto_increment(offset,
 increment).await;

    if let Err(error) = update_auto_increment {
        warnings.push("Failed to update auto increment", error);
        return Err(());
    }
```

```
   // Get and display the updated parameters. Specify Source of 'user' to get
just the modified parameters. rds.DescribeDbClusterParameters(Source='user')
   show_parameters(scenario, warnings).await;

   let username = inquire::Text::new("Username for the database (default
'testuser')")
       .with_default("testuser")
       .with_initial_value("testuser")
       .prompt();

   if let Err(error) = username {
       warnings.push(
           "Failed to get username, using default",
           ScenarioError::with(format!("Error from inquirer: {error}")),
       );
       return Err(());
   }
   let username = username.unwrap();

   let password = inquire::Text::new("Password for the database (minimum 8
characters)")
       .with_validator(|i: &str| {
           if i.len() >= 8 {
               Ok(inquire::validator::Validation::Valid)
           } else {
               Ok(inquire::validator::Validation::Invalid(
                   "Password must be at least 8 characters".into(),
               ))
           }
       })
       .prompt();

   let password: Option<SecretString> = match password {
       Ok(password) => Some(SecretString::from(password)),
       Err(error) => {
           warnings.push(
               "Failed to get password, using none (and not starting a DB)",
               ScenarioError::with(format!("Error from inquirer: {error}")),
           );
           return Err(());
       }
   };

   scenario.set_login(Some(username), password);
```

```
    Ok(())
}

// Start a single instance in the cluster,
async fn run_instance(scenario: &mut AuroraScenario) -> Result<(), ScenarioError>
 {
    // Create an Aurora DB cluster database cluster that contains a MySql
 database and uses the parameter group you created.
    // Create a database instance in the cluster.
    // Wait for DB instance to be ready. Call rds.DescribeDbInstances and check
 for DBInstanceStatus == 'available'.
    scenario.start_cluster_and_instance().await?;

    let connection_string = scenario.connection_string().await?;

    println!("Database ready: {connection_string}",);

    let _ = inquire::Text::new("Use the database with the connection string. When
 you're finished, press enter key to continue.").prompt();

    // Create a snapshot of the DB cluster. rds.CreateDbClusterSnapshot.
    // Wait for the snapshot to create. rds.DescribeDbClusterSnapshots until
 Status == 'available'.
    let snapshot_name = inquire::Text::new("Provide a name for the snapshot")
        .prompt()
        .unwrap_or(String::from("ScenarioRun"));
    let snapshot = scenario.snapshot(snapshot_name.as_str()).await?;
    println!(
        "Snapshot is available: {}",
        snapshot.db_cluster_snapshot_arn().unwrap_or("Missing ARN")
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    tracing_subscriber::fmt::init();
    let sdk_config = aws_config::from_env().load().await;
    let client = Client::new(&sdk_config);
    let rds = RdsClient::new(client);
    let mut scenario = prepare_scenario(rds).await?;
    // At this point, the scenario has things in AWS and needs to get cleaned up.
    let mut warnings = Warnings::new();

    if prepare_cluster(&mut scenario, &mut warnings).await.is_ok() {
        println!("Configured database cluster, starting an instance.");
        if let Err(err) = run_instance(&mut scenario).await {
            warnings.push("Problem running instance", err);
        }
    }

    // Clean up the instance, cluster, and parameter group, waiting for the
 instance and cluster to delete before moving on.
    let clean_up = scenario.clean_up().await;
    if let Err(errors) = clean_up {
        for error in errors {
            warnings.push("Problem cleaning up scenario", error);
        }
    }

    if warnings.is_empty() {
        Ok(())
    } else {
        println!("There were problems running the scenario:");
        println!("{warnings}");
        Err(anyhow!("There were problems running the scenario"))
    }
}

#[derive(Clone)]
struct U8Validator {}
impl StringValidator for U8Validator {
    fn validate(&self, input: &str) -> Result<inquire::validator::Validation,
 CustomUserError> {
        if input.parse::<u8>().is_err() {
            Ok(inquire::validator::Validation::Invalid(
                "Can't parse input as number".into(),
            ))
        } else {
            Ok(inquire::validator::Validation::Valid)
        }
    }
}

async fn show_parameters(scenario: &AuroraScenario, warnings: &mut Warnings) {
    let parameters = scenario.cluster_parameters().await;

    match parameters {
        Ok(parameters) => {
            println!("Current parameters");
            for parameter in parameters {
                println!("\t{parameter}");
            }
        }
        Err(error) => warnings.push("Could not find cluster parameters", error),
    }
}

fn prompt_number_or_default(warnings: &mut Warnings, name: &str, default: u8) ->
 u8 {
    let input = inquire::Text::new(format!("Updated {name}:").as_str())
        .with_validator(U8Validator {})
        .prompt();

    match input {
        Ok(increment) => match increment.parse::<u8>() {
            Ok(increment) => increment,
            Err(error) => {
                warnings.push(
                    format!("Invalid updated {name} (using {default}
 instead)").as_str(),
                    ScenarioError::with(format!("{error}")),
                );
                default
            }
        },
        Err(error) => {
            warnings.push(
                format!("Invalid updated {name} (using {default}
 instead)").as_str(),
                ScenarioError::with(format!("{error}")),
            );
            default
        }
    }
}
```

A wrapper around the Amazon RDS service that allows automocking for tests.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

use aws_sdk_rds::{
    error::SdkError,
    operation::{
        create_db_cluster::{CreateDBClusterError, CreateDbClusterOutput},
        create_db_cluster_parameter_group::CreateDBClusterParameterGroupError,
        create_db_cluster_parameter_group::CreateDbClusterParameterGroupOutput,
        create_db_cluster_snapshot::{CreateDBClusterSnapshotError,
 CreateDbClusterSnapshotOutput},
        create_db_instance::{CreateDBInstanceError, CreateDbInstanceOutput},
        delete_db_cluster::{DeleteDBClusterError, DeleteDbClusterOutput},
        delete_db_cluster_parameter_group::{
            DeleteDBClusterParameterGroupError,
 DeleteDbClusterParameterGroupOutput,
        },
        delete_db_instance::{DeleteDBInstanceError, DeleteDbInstanceOutput},
        describe_db_cluster_endpoints::{
            DescribeDBClusterEndpointsError, DescribeDbClusterEndpointsOutput,
        },
        describe_db_cluster_parameters::{
            DescribeDBClusterParametersError, DescribeDbClusterParametersOutput,
        },
        describe_db_clusters::{DescribeDBClustersError,
 DescribeDbClustersOutput},
        describe_db_engine_versions::{
            DescribeDBEngineVersionsError, DescribeDbEngineVersionsOutput,
        },
        describe_db_instances::{DescribeDBInstancesError,
 DescribeDbInstancesOutput},

 describe_orderable_db_instance_options::DescribeOrderableDBInstanceOptionsError,
        modify_db_cluster_parameter_group::{
            ModifyDBClusterParameterGroupError,
 ModifyDbClusterParameterGroupOutput,
        },
    },
    types::{OrderableDbInstanceOption, Parameter},
    Client as RdsClient,
};
use secrecy::{ExposeSecret, SecretString};
```

```
#[cfg(test)]
use mockall::automock;

#[cfg(test)]
pub use MockRdsImpl as Rds;
#[cfg(not(test))]
pub use RdsImpl as Rds;

pub struct RdsImpl {
    pub inner: RdsClient,
}

#[cfg_attr(test, automock)]
impl RdsImpl {
    pub fn new(inner: RdsClient) -> Self {
        RdsImpl { inner }
    }

    // snippet-start:[rust.aurora.describe_db_engine_versions.wrapper]
    pub async fn describe_db_engine_versions(
        &self,
        engine: &str,
    ) -> Result<DescribeDbEngineVersionsOutput,
 SdkError<DescribeDBEngineVersionsError>> {
        self.inner
            .describe_db_engine_versions()
            .engine(engine)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.describe_db_engine_versions.wrapper]

    // snippet-start:[rust.aurora.describe_orderable_db_instance_options.wrapper]
    pub async fn describe_orderable_db_instance_options(
        &self,
        engine: &str,
        engine_version: &str,
    ) -> Result<Vec<OrderableDbInstanceOption>,
 SdkError<DescribeOrderableDBInstanceOptionsError>>
    {
        self.inner
            .describe_orderable_db_instance_options()
            .engine(engine)
            .engine_version(engine_version)
            .into_paginator()
            .items()
            .send()
            .try_collect()
            .await
    }
    // snippet-end:[rust.aurora.describe_orderable_db_instance_options.wrapper]

    // snippet-start:[rust.aurora.create_db_cluster_parameter_group.wrapper]
    pub async fn create_db_cluster_parameter_group(
        &self,
        name: &str,
        description: &str,
        family: &str,
    ) -> Result<CreateDbClusterParameterGroupOutput,
 SdkError<CreateDBClusterParameterGroupError>>
    {
        self.inner
            .create_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .description(description)
            .db_parameter_group_family(family)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.create_db_cluster_parameter_group.wrapper]

    // snippet-start:[rust.aurora.describe_db_clusters.wrapper]
    pub async fn describe_db_clusters(
        &self,
        id: &str,
    ) -> Result<DescribeDbClustersOutput, SdkError<DescribeDBClustersError>> {
        self.inner
            .describe_db_clusters()
            .db_cluster_identifier(id)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.describe_db_clusters.wrapper]

    // snippet-start:[rust.aurora.describe_db_cluster_parameters.wrapper]
    pub async fn describe_db_cluster_parameters(
        &self,
        name: &str,
    ) -> Result<Vec<DescribeDbClusterParametersOutput>,
 SdkError<DescribeDBClusterParametersError>>
    {
        self.inner
            .describe_db_cluster_parameters()
            .db_cluster_parameter_group_name(name)
            .into_paginator()
            .send()
            .try_collect()
            .await
    }
    // snippet-end:[rust.aurora.describe_db_cluster_parameters.wrapper]

    // snippet-start:[rust.aurora.modify_db_cluster_parameter_group.wrapper]
    pub async fn modify_db_cluster_parameter_group(
        &self,
        name: &str,
        parameters: Vec<Parameter>,
    ) -> Result<ModifyDbClusterParameterGroupOutput,
 SdkError<ModifyDBClusterParameterGroupError>>
    {
        self.inner
            .modify_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .set_parameters(Some(parameters))
            .send()
            .await
    }
    // snippet-end:[rust.aurora.modify_db_cluster_parameter_group.wrapper]

    // snippet-start:[rust.aurora.create_db_cluster.wrapper]
    pub async fn create_db_cluster(
        &self,
        name: &str,
        parameter_group: &str,
        engine: &str,
        version: &str,
        username: &str,
        password: SecretString,
    ) -> Result<CreateDbClusterOutput, SdkError<CreateDBClusterError>> {
        self.inner
            .create_db_cluster()
            .db_cluster_identifier(name)
            .db_cluster_parameter_group_name(parameter_group)
            .engine(engine)
            .engine_version(version)
            .master_username(username)
            .master_user_password(password.expose_secret())
            .send()
            .await
    }
    // snippet-end:[rust.aurora.create_db_cluster.wrapper]

    // snippet-start:[rust.aurora.create_db_instance.wrapper]
    pub async fn create_db_instance(
        &self,
        cluster_name: &str,
        instance_name: &str,
        instance_class: &str,
        engine: &str,
    ) -> Result<CreateDbInstanceOutput, SdkError<CreateDBInstanceError>> {
        self.inner
            .create_db_instance()
            .db_cluster_identifier(cluster_name)
            .db_instance_identifier(instance_name)
            .db_instance_class(instance_class)
            .engine(engine)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.create_db_instance.wrapper]

    // snippet-start:[rust.aurora.describe_db_instance.wrapper]
    pub async fn describe_db_instance(
        &self,
        instance_identifier: &str,
    ) -> Result<DescribeDbInstancesOutput, SdkError<DescribeDBInstancesError>> {
        self.inner
            .describe_db_instances()
            .db_instance_identifier(instance_identifier)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.describe_db_instance.wrapper]

    // snippet-start:[rust.aurora.create_db_cluster_snapshot.wrapper]
    pub async fn snapshot_cluster(
        &self,
        db_cluster_identifier: &str,
        snapshot_name: &str,
    ) -> Result<CreateDbClusterSnapshotOutput,
 SdkError<CreateDBClusterSnapshotError>> {
        self.inner
            .create_db_cluster_snapshot()
            .db_cluster_identifier(db_cluster_identifier)
            .db_cluster_snapshot_identifier(snapshot_name)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.create_db_cluster_snapshot.wrapper]

    // snippet-start:[rust.aurora.describe_db_instances.wrapper]
    pub async fn describe_db_instances(
        &self,
    ) -> Result<DescribeDbInstancesOutput, SdkError<DescribeDBInstancesError>> {
        self.inner.describe_db_instances().send().await
    }
    // snippet-end:[rust.aurora.describe_db_instances.wrapper]

    // snippet-start:[rust.aurora.describe_db_cluster_endpoints.wrapper]
    pub async fn describe_db_cluster_endpoints(
        &self,
        cluster_identifier: &str,
    ) -> Result<DescribeDbClusterEndpointsOutput,
 SdkError<DescribeDBClusterEndpointsError>> {
        self.inner
            .describe_db_cluster_endpoints()
            .db_cluster_identifier(cluster_identifier)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.describe_db_cluster_endpoints.wrapper]

    // snippet-start:[rust.aurora.delete_db_instance.wrapper]
    pub async fn delete_db_instance(
        &self,
        instance_identifier: &str,
    ) -> Result<DeleteDbInstanceOutput, SdkError<DeleteDBInstanceError>> {
        self.inner
            .delete_db_instance()
            .db_instance_identifier(instance_identifier)
            .skip_final_snapshot(true)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.delete_db_instance.wrapper]

    // snippet-start:[rust.aurora.delete_db_cluster.wrapper]
    pub async fn delete_db_cluster(
        &self,
        cluster_identifier: &str,
    ) -> Result<DeleteDbClusterOutput, SdkError<DeleteDBClusterError>> {
        self.inner
            .delete_db_cluster()
            .db_cluster_identifier(cluster_identifier)
            .skip_final_snapshot(true)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.delete_db_cluster.wrapper]

    // snippet-start:[rust.aurora.delete_db_cluster_parameter_group.wrapper]
    pub async fn delete_db_cluster_parameter_group(
        &self,
        name: &str,
    ) -> Result<DeleteDbClusterParameterGroupOutput,
 SdkError<DeleteDBClusterParameterGroupError>>
    {
        self.inner
            .delete_db_cluster_parameter_group()
            .db_cluster_parameter_group_name(name)
            .send()
            .await
    }
    // snippet-end:[rust.aurora.delete_db_cluster_parameter_group.wrapper]
}
```

The Cargo.toml with dependencies used in this scenario.

```
[package]
name = "aurora-code-examples"
authors = [
  "David Souther <dpsouth@amazon.com>",
]
edition = "2021"
version = "0.1.0"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
anyhow = "1.0.75"
assert_matches = "1.5.0"
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-smithy-types = { version = "1.0.1" }
aws-smithy-runtime-api = { version = "1.0.1" }
aws-sdk-rds = { version = "1.3.0" }
inquire = "0.6.2"
mockall = "0.11.4"
phf = { version = "0.11.2", features = ["std", "macros"] }
sdk-examples-test-utils = { path = "../../test-utils" }
secrecy = "0.8.0"
tokio = { version = "1.20.1", features = ["full", "test-util"] }
tracing = "0.1.37"
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
```

- For API details, see the following topics in *AWS SDK for Rust API reference*.
  - CreateDBCluster
  - CreateDBClusterParameterGroup
  - CreateDBClusterSnapshot
  - CreateDBInstance
  - DeleteDBCluster
  - DeleteDBClusterParameterGroup
  - DeleteDBInstance
  - DescribeDBClusterParameterGroups
  - DescribeDBClusterParameters
  - DescribeDBClusterSnapshots
  - DescribeDBClusters
  - DescribeDBEngineVersions
  - DescribeDBInstances

- DescribeOrderableDBInstanceOptions
- ModifyDBClusterParameterGroup

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Cross-service examples for Aurora using AWS SDKs

The following sample applications use AWS SDKs to combine Aurora with other AWS services. Each example includes a link to GitHub, where you can find instructions on how to set up and run the application.

Examples

- Create a lending library REST API
- Create an Aurora Serverless work item tracker

Create a lending library REST API

The following code example shows how to create a lending library where patrons can borrow and return books by using a REST API backed by an Amazon Aurora database.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with the Amazon Relational Database Service (Amazon RDS) API and AWS Chalice to create a REST API backed by an Amazon Aurora database. The web service is fully serverless and represents a simple lending library where patrons can borrow and return books. Learn how to:

- Create and manage a serverless Aurora database cluster.
- Use AWS Secrets Manager to manage database credentials.
- Implement a data storage layer that uses Amazon RDS to move data into and out of the database.
- Use AWS Chalice to deploy a serverless REST API to Amazon API Gateway and AWS Lambda.

- Use the Requests package to send requests to the web service.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- Aurora
- Lambda
- Secrets Manager

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create an Aurora Serverless work item tracker

The following code examples show how to create a web application that tracks work items in an Amazon Aurora Serverless database and uses Amazon Simple Email Service (Amazon SES) to send reports.

.NET

AWS SDK for .NET

Shows how to use the AWS SDK for .NET to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with a RESTful .NET backend.

- Integrate a React web application with AWS services.
- List, add, update, and delete items in an Aurora table.
- Send an email report of filtered work items using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

C++

SDK for C++

Shows how to create a web application that tracks and reports on work items stored in an Amazon Aurora Serverless database.

For complete source code and instructions on how to set up a C++ REST API that queries Amazon Aurora Serverless data and for use by a React application, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Java

SDK for Java 2.x

Shows how to create a web application that tracks and reports on work items stored in an Amazon RDS database.

For complete source code and instructions on how to set up a Spring REST API that queries Amazon Aurora Serverless data and for use by a React application, see the full example on GitHub.

For complete source code and instructions on how to set up and run an example that uses the JDBC API, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

JavaScript

SDK for JavaScript (v3)

Shows how to use the AWS SDK for JavaScript (v3) to create a web application that tracks work items in an Amazon Aurora database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with an Express Node.js backend.

- Integrate a React.js web application with AWS services.
- List, add, and update items in an Aurora table.
- Send an email report of filtered work items by using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Kotlin

SDK for Kotlin

Shows how to create a web application that tracks and reports on work items stored in an Amazon RDS database.

For complete source code and instructions on how to set up a Spring REST API that queries Amazon Aurora Serverless data and for use by a React application, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

PHP

SDK for PHP

Shows how to use the AWS SDK for PHP to create a web application that tracks work items in an Amazon RDS database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses a front end built with React.js to interact with a RESTful PHP backend.

- Integrate a React.js web application with AWS services.
- List, add, update, and delete items in an Amazon RDS table.
- Send an email report of filtered work items using Amazon SES.
- Deploy and manage example resources with the included AWS CloudFormation script.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) to create a REST service that tracks work items in an Amazon Aurora Serverless database and emails reports by using Amazon Simple Email Service (Amazon SES). This example uses the Flask web framework to handle HTTP routing and integrates with a React webpage to present a fully functional web application.

- Build a Flask REST service that integrates with AWS services.
- Read, write, and update work items that are stored in an Aurora Serverless database.
- Create an AWS Secrets Manager secret that contains database credentials and use it to authenticate calls to the database.
- Use Amazon SES to send email reports of work items.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Aurora
- Amazon RDS
- Amazon RDS Data Service
- Amazon SES

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Best practices with Amazon Aurora

Following, you can find information on general best practices and options for using or migrating data to an Amazon Aurora DB cluster.

Some of the best practices for Amazon Aurora are specific to a particular database engine. For more information about Aurora best practices specific to a database engine, see the following.

Database engine	Best practices
Amazon Aurora MySQL	See Best practices with Amazon Aurora MySQL
Amazon Aurora PostgreSQL	See Best practices with Amazon Aurora PostgreSQL

Note

For common recommendations for Aurora, see Viewing and responding to Amazon Aurora recommendations.

Topics

- Basic operational guidelines for Amazon Aurora
- DB instance RAM recommendations
- AWS database drivers
- Monitoring Amazon Aurora
- Working with DB parameter groups and DB cluster parameter groups
- Amazon Aurora best practices video

Basic operational guidelines for Amazon Aurora

The following are basic operational guidelines that everyone should follow when working with Amazon Aurora. The Amazon RDS Service Level Agreement requires that you follow these guidelines:

- Monitor your memory, CPU, and storage usage. You can set up Amazon CloudWatch to notify you when usage patterns change or when you approach the capacity of your deployment. This way, you can maintain system performance and availability.
- If your client application is caching the Domain Name Service (DNS) data of your DB instances, set a time-to-live (TTL) value of less than 30 seconds. The underlying IP address of a DB instance can change after a failover. Thus, caching the DNS data for an extended time can lead to connection failures if your application tries to connect to an IP address that no longer is in service. Aurora DB clusters with multiple read replicas can experience connection failures also when connections use the reader endpoint and one of the read replica instances is in maintenance or is deleted.
- Test failover for your DB cluster to understand how long the process takes for your use case. Testing failover can help you ensure that the application that accesses your DB cluster can automatically connect to the new DB cluster after failover.

DB instance RAM recommendations

To optimize performance, allocate enough RAM so that your working set resides almost completely in memory. To determine whether your working set is almost all in memory, examine the following metrics in Amazon CloudWatch:

- `VolumeReadIOPS` – This metric measures the average number of read I/O operations from a cluster volume, reported at 5-minute intervals. The value of `VolumeReadIOPS` should be small and stable. In some cases, you might find your read I/O is spiking or is higher than usual. If so, investigate the DB instances in your DB cluster to see which DB instances are causing the increased I/O.

  > **Tip**
  >
  > If your Aurora MySQL cluster uses parallel query, you might see an increase in `VolumeReadIOPS` values. Parallel queries don't use the buffer pool. Thus, although the queries are fast, this optimized processing can result in an increase in read operations and associated charges.

- `BufferCacheHitRatio` – This metric measures the percentage of requests that are served by the buffer cache of a DB instance in your DB cluster. This metric gives you an insight into the amount of data that is being served from memory.

A high hit ratio indicates that your DB instance has enough memory available. A low hit ratio indicates that your queries on this DB instance are frequently going to disk. Investigate your workload to see which queries are causing this behavior.

If, after investigating your workload, you find that you need more memory, consider scaling up the DB instance class to a class with more RAM. After doing so, you can investigate the metrics discussed preceding and continue to scale up as necessary. If your Aurora cluster is larger than 40 TB, don't use db.t2, db.t3, or db.t4g instance classes.

For more information, see Amazon CloudWatch metrics for Amazon Aurora.

AWS database drivers

We recommend the AWS suite of drivers for application connectivity. The drivers have been designed to provide support for faster switchover and failover times, and authentication with AWS Secrets Manager, AWS Identity and Access Management (IAM), and Federated Identity. The AWS drivers rely on monitoring DB cluster status and being aware of the cluster topology to determine the new writer. This approach reduces switchover and failover times to single-digit seconds, compared to tens of seconds for open-source drivers.

As new service features are introduced, the goal of the AWS suite of drivers is to have built-in support for these service features.

For more information, see Connecting to Aurora DB clusters with the AWS drivers.

Monitoring Amazon Aurora

Amazon Aurora provides various metrics and insights that you can monitor to determine the health and performance of your Aurora DB cluster. You can use various tools, such as the AWS Management Console, AWS CLI, and CloudWatch API, to view Aurora metrics. You can view the combined Performance Insights and CloudWatch metrics in the Performance Insights dashboard and monitor your DB instance. To use this monitoring view, Performance Insights must be turned on for your DB instance. For information about this monitoring view, see Viewing combined metrics in the Amazon RDS console.

You can create a performance analysis report for a specific time period and view the insights identified and the recommendations to resolve the issues. For more information see, Creating a performance analysis report.

Working with DB parameter groups and DB cluster parameter groups

We recommend that you try out DB parameter group and DB cluster parameter group changes on a test DB cluster before applying parameter group changes to your production DB cluster. Improperly setting DB engine parameters can have unintended adverse effects, including degraded performance and system instability.

Always use caution when modifying DB engine parameters, and back up your DB cluster before modifying a DB parameter group. For information about backing up your DB cluster, see Backing up and restoring an Amazon Aurora DB cluster.

Amazon Aurora best practices video

The AWS Online Tech Talks channel on YouTube includes a video presentation on best practices for creating and configuring an Amazon Aurora DB cluster to be more secure and highly available. See Amazon Aurora best practices for high availability.

Performing a proof of concept with Amazon Aurora

Following, you can find an explanation of how to set up and run a proof of concept for Aurora. A *proof of concept* is an investigation that you do to see if Aurora is a good fit with your application. The proof of concept can help you understand Aurora features in the context of your own database applications and how Aurora compares with your current database environment. It can also show what level of effort you need to move data, port SQL code, tune performance, and adapt your current management procedures.

In this topic, you can find an overview and a step-by-step outline of the high-level procedures and decisions involved in running a proof of concept, listed following. For detailed instructions, you can follow links to the full documentation for specific subjects.

Overview of an Aurora proof of concept

When you conduct a proof of concept for Amazon Aurora, you learn what it takes to port your existing data and SQL applications to Aurora. You exercise the important aspects of Aurora at scale, using a volume of data and activity that's representative of your production environment. The objective is to feel confident that the strengths of Aurora match up well with the challenges that cause you to outgrow your previous database infrastructure. At the end of a proof of concept, you have a solid plan to do larger-scale performance benchmarking and application testing. At this point, you understand the biggest work items on your way to a production deployment.

The following advice about best practices can help you avoid common mistakes that cause problems during benchmarking. However, this topic doesn't cover the step-by-step process of performing benchmarks and doing performance tuning. Those procedures vary depending on your workload and the Aurora features that you use. For detailed information, consult performance-related documentation such as Managing performance and scaling for Aurora DB clusters, Amazon Aurora MySQL performance enhancements, Managing Amazon Aurora PostgreSQL, and Monitoring DB load with Performance Insights on Amazon Aurora.

The information in this topic applies mainly to applications where your organization writes the code and designs the schema and that support the MySQL and PostgreSQL open-source database engines. If you're testing a commercial application or code generated by an application framework, you might not have the flexibility to apply all of the guidelines. In such cases, check with your AWS representative to see if there are Aurora best practices or case studies for your type of application.

1. Identify your objectives

When you evaluate Aurora as part of a proof of concept, you choose what measurements to make and how to evaluate the success of the exercise.

You must ensure that all of the functionality of your application is compatible with Aurora. Because Aurora major versions are wire-compatible with corresponding major versions of MySQL and PostgreSQL, most applications developed for those engines are also compatible with Aurora. However, you must still validate compatibility on a per-application basis.

For example, some of the configuration choices that you make when you set up an Aurora cluster influence whether you can or should use particular database features. You might start with the most general-purpose kind of Aurora cluster, known as *provisioned*. You might then decide if a specialized configuration such as serverless or parallel query offers benefits for your workload.

Use the following questions to help identify and quantify your objectives:

- Does Aurora support all functional use cases of your workload?
- What dataset size or load level do you want? Can you scale to that level?
- What are your specific query throughput or latency requirements? Can you reach them?
- What is the minimum acceptable amount of planned or unplanned downtime for your workload? Can you achieve it?
- What are the necessary metrics for operational efficiency? Can you accurately monitor them?
- Does Aurora support your specific business goals, such as cost reduction, increase in deployment, or provisioning speed? Do you have a way to quantify these goals?
- Can you meet all security and compliance requirements for your workload?

Take some time to build knowledge about Aurora database engines and platform capabilities, and review the service documentation. Take note of all the features that can help you achieve your desired outcomes. One of these might be workload consolidation, described in the AWS Database Blog post How to plan and optimize Amazon Aurora with MySQL compatibility for consolidated workloads. Another might be demand-based scaling, described in Using Amazon Aurora Auto Scaling with Aurora Replicas in the *Amazon Aurora User Guide.* Others might be performance gains or simplified database operations.

2. Understand your workload characteristics

Evaluate Aurora in the context of your intended use case. Aurora is a good choice for online transaction processing (OLTP) workloads. You can also run reports on the cluster that holds the real-time OLTP data without provisioning a separate data warehouse cluster. You can recognize if your use case falls into these categories by checking for the following characteristics:

- High concurrency, with dozens, hundreds, or thousands of simultaneous clients.
- Large volume of low-latency queries (milliseconds to seconds).
- Short, real-time transactions.
- Highly selective query patterns, with index-based lookups.
- For HTAP, analytical queries that can take advantage of Aurora parallel query.

One of the key factors affecting your database choices is the velocity of the data. *High velocity* involves data being inserted and updated very frequently. Such a system might have thousands of connections and hundreds of thousands of simultaneous queries reading from and writing to a database. Queries in high-velocity systems usually affect a relatively small number of rows, and typically access multiple columns in the same row.

Aurora is designed to handle high-velocity data. Depending on the workload, an Aurora cluster with a single r4.16xlarge DB instance can process more than 600,000 `SELECT` statements per second. Again depending on workload, such a cluster can process 200,000 `INSERT`, `UPDATE`, and `DELETE` statements per second. Aurora is a row store database and is ideally suited for high-volume, high-throughput, and highly parallelized OLTP workloads.

Aurora can also run reporting queries on the same cluster that handles the OLTP workload. Aurora supports up to 15 replicas, each of which is on average within 10–20 milliseconds of the primary instance. Analysts can query OLTP data in real time without copying the data to a separate data warehouse cluster. With Aurora clusters using the parallel query feature, you can offload much of the processing, filtering, and aggregation work to the massively distributed Aurora storage subsystem.

Use this planning phase to familiarize yourself with the capabilities of Aurora, other AWS services, the AWS Management Console, and the AWS CLI. Also, check how these work with the other tooling that you plan to use in the proof of concept.

3. Practice with the AWS Management Console or AWS CLI

As a next step, practice with the AWS Management Console or the AWS CLI, to become familiar with these tools and with Aurora.

Practice with the AWS Management Console

The following initial activities with Aurora database clusters are mainly so you can familiarize yourself with the AWS Management Console environment and practice setting up and modifying Aurora clusters. If you use the MySQL-compatible and PostgreSQL-compatible database engines with Amazon RDS, you can build on that knowledge when you use Aurora.

By taking advantage of the Aurora shared storage model and features such as replication and snapshots, you can treat entire database clusters as another kind of object that you freely manipulate. You can set up, tear down, and change the capacity of Aurora clusters frequently during the proof of concept. You aren't locked into early choices about capacity, database settings, and physical data layout.

To get started, set up an empty Aurora cluster. Choose the **provisioned** capacity type and **regional** location for your initial experiments.

Connect to that cluster using a client program such as a SQL command-line application. Initially, you connect using the cluster endpoint. You connect to that endpoint to perform any write operations, such as data definition language (DDL) statements and extract, transform, load (ETL) processes. Later in the proof of concept, you connect query-intensive sessions using the reader endpoint, which distributes the query workload among multiple DB instances in the cluster.

Scale the cluster out by adding more Aurora Replicas. For those procedures, see Replication with Amazon Aurora. Scale the DB instances up or down by changing the AWS instance class. Understand how Aurora simplifies these kinds of operations, so that if your initial estimates for system capacity are inaccurate, you can adjust later without starting over.

Create a snapshot and restore it to a different cluster.

Examine cluster metrics to see activity over time, and how the metrics apply to the DB instances in the cluster.

It's useful to become familiar with how to do these things through the AWS Management Console in the beginning. After you understand what you can do with Aurora, you can progress to automating those operations using the AWS CLI. In the following sections, you can find more details about the procedures and best practices for these activities during the proof-of-concept period.

Practice with the AWS CLI

We recommend automating deployment and management procedures, even in a proof-of-concept setting. To do so, become familiar with the AWS CLI if you're not already. If you use the MySQL-compatible and PostgreSQL-compatible database engines with Amazon RDS, you can build on that knowledge when you use Aurora.

Aurora typically involves groups of DB instances arranged in clusters. Thus, many operations involve determining which DB instances are associated with a cluster and then performing administrative operations in a loop for all the instances.

For example, you might automate steps such as creating Aurora clusters, then scaling them up with larger instance classes or scaling them out with additional DB instances. Doing so helps you to repeat any stages in your proof of concept and explore what-if scenarios with different kinds or configurations of Aurora clusters.

Learn the capabilities and limitations of infrastructure deployment tools such as AWS CloudFormation. You might find activities that you do in a proof-of-concept context aren't suitable for production use. For example, the AWS CloudFormation behavior for modification is to create a new instance and delete the current one, including its data. For more details on this behavior, see Update behaviors of stack resources in the *AWS CloudFormation User Guide.*

4. Create your Aurora cluster

With Aurora, you can explore what-if scenarios by adding DB instances to the cluster and scaling up the DB instances to more powerful instance classes. You can also create clusters with different configuration settings to run the same workload side by side. With Aurora, you have a lot of flexibility to set up, tear down, and reconfigure DB clusters. Given this, it's helpful to practice these techniques in the early stages of the proof-of-concept process. For the general procedures to create Aurora clusters, see Creating an Amazon Aurora DB cluster.

Where practical, start with a cluster using the following settings. Skip this step only if you have certain specific use cases in mind. For example, you might skip this step if your use case requires a specialized kind of Aurora cluster. Or you might skip it if you need a particular combination of database engine and version.

- Turn off **Easy create**. For the proof of concept, we recommend that you be aware of all the settings you choose so that you can create identical or slightly different clusters later.
- Use a recent DB engine version. These combinations of database engine and version have wide compatibility with other Aurora features and substantial customer usage for production applications.
  - Aurora MySQL version 3.x (MySQL 8.0 compatibility)
  - Aurora PostgreSQL version 15.x or 16.x
- Choose the **Dev/Test** template. This choice isn't significant for your proof-of-concept activities.
- For **DB instance class**, choose **Memory optimized classes** and one of the **xlarge** instance classes. You can adjust the instance class up or down later.
- Under **Multi-AZ Deployment**, choose **Create an Aurora Replica or Reader node in a different AZ**. Many of the most useful aspects of Aurora involve clusters of multiple DB instances. It makes sense to always start with at least two DB instances in any new cluster. Using a different Availability Zone for the second DB instance helps to test different high availability scenarios.
- When you pick names for the DB instances, use a generic naming convention. Don't refer to any cluster DB instance as the "writer," because different DB instances assume those roles as needed. We recommend using something like `clustername-az-serialnumber`, for example `myprodappdb-a-01`. These pieces uniquely identify the DB instance and its placement.
- Set the backup retention high for the Aurora cluster. With a long retention period, you can do point-in-time recovery (PITR) for a period up to 35 days. You can reset your database to a known state after running tests involving DDL and data manipulation language (DML) statements. You can also recover if you delete or change data by mistake.
- Turn on additional recovery, logging, and monitoring features at cluster creation. Turn on all the choices that are available under **Backtrack**, **Performance Insights**, **Monitoring**, and **Log exports**. With these features enabled, you can test the suitability of features such as backtracking, Enhanced Monitoring, or Performance Insights for your workload. You can also easily investigate performance and perform troubleshooting during the proof of concept.

5. Set up your schema

On the Aurora cluster, set up databases, tables, indexes, foreign keys, and other schema objects for your application. If you're moving from another MySQL-compatible or PostgreSQL-compatible database system, expect this stage to be simple and straightforward. You use the same SQL syntax and command line or other client applications that you're familiar with for your database engine.

To issue SQL statements on your cluster, find its cluster endpoint and supply that value as the connection parameter to your client application. You can find the cluster endpoint on the **Connectivity** tab of the detail page of your cluster. The cluster endpoint is the one labeled **Writer**. The other endpoint, labeled **Reader**, represents a read-only connection that you can supply to end users who run reports or other read-only queries. For help with any issues around connecting to your cluster, see Connecting to an Amazon Aurora DB cluster.

If you're porting your schema and data from a different database system, expect to make some schema changes at this point. These schema changes are to match the SQL syntax and capabilities available in Aurora. You might leave out certain columns, constraints, triggers, or other schema objects at this point. Doing so can be useful particularly if these objects require rework for Aurora compatibility and aren't significant for your objectives with the proof of concept.

If you're migrating from a database system with a different underlying engine than Aurora's, consider using the AWS Schema Conversion Tool (AWS SCT) to simplify the process. For details, see the AWS Schema Conversion Tool User Guide. For general details about migration and porting activities, see the Migrating Your Databases to Amazon Aurora AWS whitepaper.

During this stage, you can evaluate whether there are inefficiencies in your schema setup, for example in your indexing strategy or other table structures such as partitioned tables. Such inefficiencies can be amplified when you deploy your application on a cluster with multiple DB instances and a heavy workload. Consider whether you can fine-tune such performance aspects now, or during later activities such as a full benchmark test.

6. Import your data

During the proof of concept, you bring across the data, or a representative sample, from your former database system. If practical, set up at least some data in each of your tables. Doing so helps to test compatibility of all data types and schema features. After you have exercised the basic Aurora features, scale up the amount of data. By the time you finish the proof of concept, you should test your ETL tools, queries, and overall workload with a dataset that's big enough to draw accurate conclusions.

You can use several techniques to import either physical or logical backup data to Aurora. For details, see Migrating data to an Amazon Aurora MySQL DB cluster or Migrating data to Amazon Aurora with PostgreSQL compatibility depending on the database engine you're using in the proof of concept.

Experiment with the ETL tools and technologies that you're considering. See which one best meets your needs. Consider both throughput and flexibility. For example, some ETL tools perform a one-time transfer, and others involve ongoing replication from the old system to Aurora.

If you're migrating from a MySQL-compatible system to Aurora MySQL, you can use the native data transfer tools. The same applies if you're migrating from a PostgreSQL-compatible system to Aurora PostgreSQL. If you're migrating from a database system that uses a different underlying engine than Aurora does, you can experiment with the AWS Database Migration Service (AWS DMS). For details about AWS DMS, see the AWS Database Migration Service User Guide.

For details about migration and porting activities, see the AWS whitepaper Aurora migration handbook.

7. Port your SQL code

Trying out SQL and associated applications requires different levels of effort depending on different cases. In particular, the level of effort depends on whether you move from a MySQL-compatible or PostgreSQL-compatible system or another kind.

- If you're moving from RDS for MySQL or RDS for PostgreSQL, the SQL changes are small enough that you can try the original SQL code with Aurora and manually incorporate needed changes.
- Similarly, if you move from an on-premises database compatible with MySQL or PostgreSQL, you can try the original SQL code and manually incorporate changes.
- If you're coming from a different commercial database, the required SQL changes are more extensive. In this case, consider using the AWS SCT.

During this stage, you can evaluate whether there are inefficiencies in your schema setup, for example in your indexing strategy or other table structures such as partitioned tables. Consider whether you can fine-tune such performance aspects now, or during later activities such as a full benchmark test.

You can verify the database connection logic in your application. To take advantage of Aurora distributed processing, you might need to use separate connections for read and write operations, and use relatively short sessions for query operations. For information about connections, see 9. Connect to Aurora.

Consider if you had to make compromises and tradeoffs to work around issues in your production database. Build time into the proof-of-concept schedule to make improvements to your schema design and queries. To judge if you can achieve easy wins in performance, operating cost, and scalability, try the original and modified applications side by side on different Aurora clusters.

For details about migration and porting activities, see the AWS whitepaper Aurora migration handbook.

8. Specify configuration settings

You can also review your database configuration parameters as part of the Aurora proof-of-concept exercise. You might already have MySQL or PostgreSQL configuration settings tuned for performance and scalability in your current environment. The Aurora storage subsystem is adapted and tuned for a distributed cloud-based environment with a high-speed storage subsystem. As a result, many former database engine settings don't apply. We recommend conducting your initial experiments with the default Aurora configuration settings. Reapply settings from your current environment only if you encounter performance and scalability bottlenecks. If you're interested, you can look more deeply into this subject in Introducing the Aurora storage engine on the AWS Database Blog.

Aurora makes it easy to reuse the optimal configuration settings for a particular application or use case. Instead of editing a separate configuration file for each DB instance, you manage sets of parameters that you assign to entire clusters or specific DB instances. For example, the time zone setting applies to all DB instances in the cluster, and you can adjust the page cache size setting for each DB instance.

You start with one of the default parameter sets, and apply changes to only the parameters that you need to fine-tune. For details about working with parameter groups, see Amazon Aurora DB cluster and DB instance parameters. For the configuration settings that are or aren't applicable to Aurora clusters, see Aurora MySQL configuration parameters or Amazon Aurora PostgreSQL parameters depending on your database engine.

9. Connect to Aurora

As you find when doing your initial schema and data setup and running sample queries, you can connect to different endpoints in an Aurora cluster. The endpoint to use depends on whether the operation is a read such as `SELECT` statement, or a write such as a `CREATE` or `INSERT` statement. As you increase the workload on an Aurora cluster and experiment with Aurora features, it's important for your application to assign each operation to the appropriate endpoint.

By using the cluster endpoint for write operations, you always connect to a DB instance in the cluster that has read/write capability. By default, only one DB instance in an Aurora cluster has read/write capability. This DB instance is called the *primary instance*. If the original primary instance becomes unavailable, Aurora activates a failover mechanism and a different DB instance takes over as the primary.

Similarly, by directing `SELECT` statements to the reader endpoint, you spread the work of processing queries among the DB instances in the cluster. Each reader connection is assigned to a different DB instance using round-robin DNS resolution. Doing most of the query work on the read-only DB Aurora Replicas reduces the load on the primary instance, freeing it to handle DDL and DML statements.

Using these endpoints reduces the dependency on hard-coded hostnames, and helps your application to recover more quickly from DB instance failures.

> **Note**
>
> Aurora also has custom endpoints that you create. Those endpoints usually aren't needed during a proof of concept.

The Aurora Replicas are subject to a replica lag, even though that lag is usually 10 to 20 milliseconds. You can monitor the replication lag and decide whether it is within the range of your data consistency requirements. In some cases, your read queries might require strong read consistency (read-after-write consistency). In these cases, you can continue using the cluster endpoint for them and not the reader endpoint.

To take full advantage of Aurora capabilities for distributed parallel execution, you might need to change the connection logic. Your objective is to avoid sending all read requests to the primary instance. The read-only Aurora Replicas are standing by, with all the same data, ready to handle `SELECT` statements. Code your application logic to use the appropriate endpoint for each kind of operation. Follow these general guidelines:

- Avoid using a single hard-coded connection string for all database sessions.
- If practical, enclose write operations such as DDL and DML statements in functions in your client application code. That way, you can make different kinds of operations use specific connections.
- Make separate functions for query operations. Aurora assigns each new connection to the reader endpoint to a different Aurora Replica to balance the load for read-intensive applications.

- For operations involving sets of queries, close and reopen the connection to the reader endpoint when each set of related queries is finished. Use connection pooling if that feature is available in your software stack. Directing queries to different connections helps Aurora to distribute the read workload among the DB instances in the cluster.

For general information about connection management and endpoints for Aurora, see Connecting to an Amazon Aurora DB cluster. For a deep dive on this subject, see Aurora MySQL database administrator's handbook – Connection management.

10. Run your workload

After the schema, data, and configuration settings are in place, you can begin exercising the cluster by running your workload. Use a workload in the proof of concept that mirrors the main aspects of your production workload. We recommend always making decisions about performance using real-world tests and workloads rather than synthetic benchmarks such as sysbench or TPC-C. Wherever practical, gather measurements based on your own schema, query patterns, and usage volume.

As much as practical, replicate the actual conditions under which the application will run. For example, you typically run your application code on Amazon EC2 instances in the same AWS Region and the same virtual private cloud (VPC) as the Aurora cluster. If your production application runs on multiple EC2 instances spanning multiple Availability Zones, set up your proof-of-concept environment in the same way. For more information on AWS Regions, see Regions and Availability Zones in the *Amazon RDS User Guide.* To learn more about the Amazon VPC service, see What is Amazon VPC? in the *Amazon VPC User Guide.*

After you've verified that the basic features of your application work and you can access the data through Aurora, you can exercise aspects of the Aurora cluster. Some features you might want to try are concurrent connections with load balancing, concurrent transactions, and automatic replication.

By this point, the data transfer mechanisms should be familiar, and so you can run tests with a larger proportion of sample data.

This stage is when you can see the effects of changing configuration settings such as memory limits and connection limits. Revisit the procedures that you explored in 8. Specify configuration settings.

You can also experiment with mechanisms such as creating and restoring snapshots. For example, you can create clusters with different AWS instance classes, numbers of AWS Replicas, and so on.

Then on each cluster, you can restore the same snapshot containing your schema and all your data. For the details of that cycle, see Creating a DB cluster snapshot and Restoring from a DB cluster snapshot.

11. Measure performance

Best practices in this area are designed to ensure that all the right tools and processes are set up to quickly isolate abnormal behaviors during workload operations. They're also set up to see that you can reliably identify any applicable causes.

You can always see the current state of your cluster, or examine trends over time, by examining the **Monitoring** tab. This tab is available from the console detail page for each Aurora cluster or DB instance. It displays metrics from the Amazon CloudWatch monitoring service in the form of charts. You can filter the metrics by name, by DB instance, and by time period.

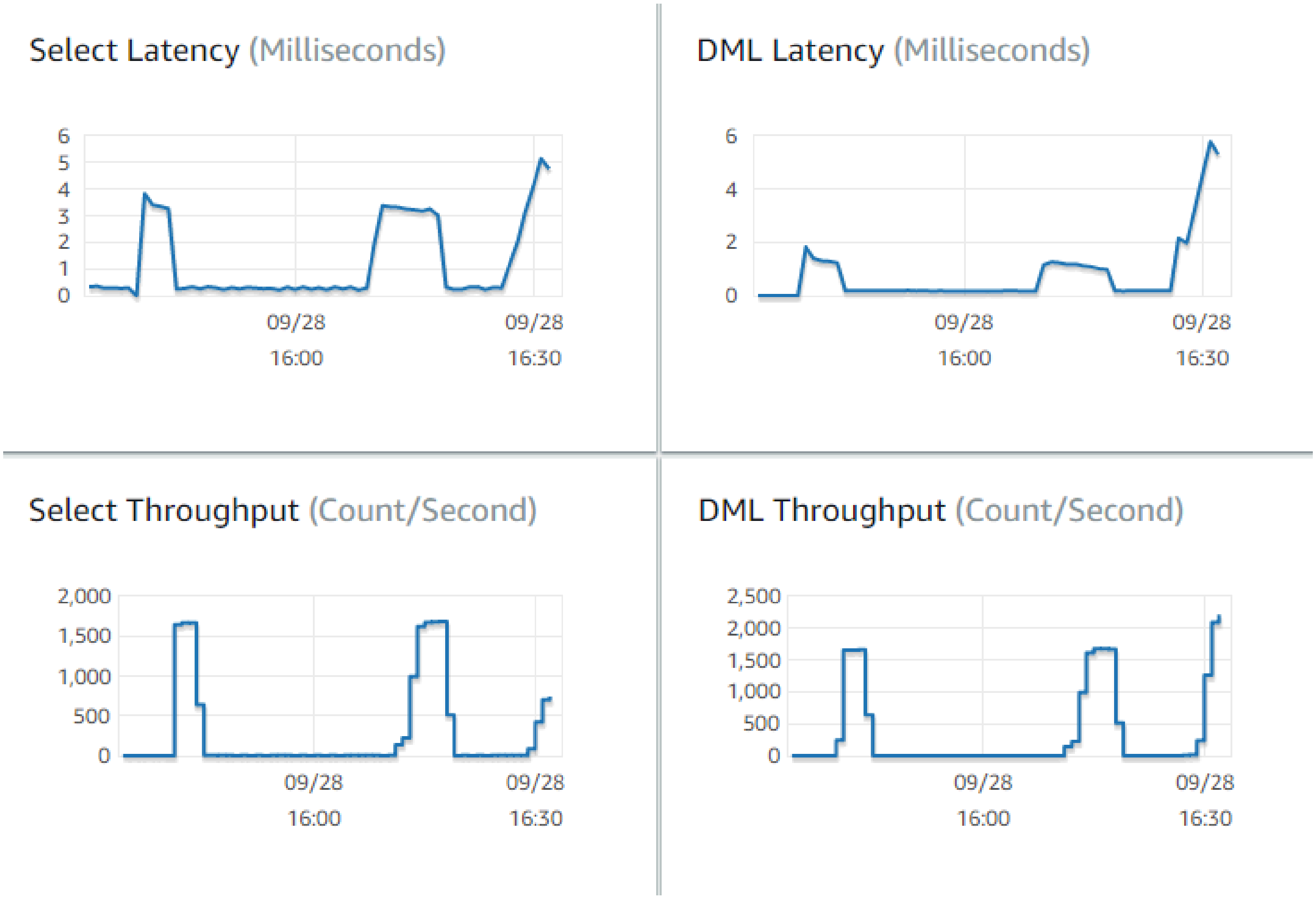
To have more choices on the **Monitoring** tab, enable Enhanced Monitoring and Performance Insights in the cluster settings. You can also enable those choices later if you didn't choose them when setting up the cluster.

To measure performance, you rely mostly on the charts showing activity for the whole Aurora cluster. You can verify whether the Aurora Replicas have similar load and response times. You can also see how the work is split up between the read/write primary instance and the read-only Aurora Replicas. If there is some imbalance between the DB instances or an issue affecting only one DB instance, you can examine the **Monitoring** tab for that specific instance.

After the environment and the actual workload are set up to emulate your production application, you can measure how well Aurora performs. The most important questions to answer are as follows:

- How many queries per second is Aurora processing? You can examine the **Throughput** metrics to see the figures for various kinds of operations.
- How long does it take, on average for Aurora to process a given query? You can examine the **Latency** metrics to see the figures for various kinds of operations.

To do so, look at the **Monitoring** tab for a given Aurora cluster in the Amazon RDS console as illustrated following.



If you can, establish baseline values for these metrics in your current environment. If that's not practical, construct a baseline on the Aurora cluster by executing a workload equivalent to your production application. For example, run your Aurora workload with a similar number of simultaneous users and queries. Then observe how the values change as you experiment with different instance classes, cluster size, configuration settings, and so on.

If the throughput numbers are lower than you expect, investigate further the factors affecting database performance for your workload. Similarly, if the latency numbers are higher than you expect, further investigate. To do so, monitor the secondary metrics for the DB server (CPU, memory, and so on). You can see whether the DB instances are close to their limits. You can also see how much extra capacity your DB instances have to handle more concurrent queries, queries against larger tables, and so on.

Tip

To detect metric values that fall outside the expected ranges, set up CloudWatch alarms.

When evaluating the ideal Aurora cluster size and capacity, you can find the configuration that achieves peak application performance without over-provisioning resources. One important factor is finding the appropriate size for the DB instances in the Aurora cluster. Start by selecting an instance size that has similar CPU and memory capacity to your current production environment. Collect throughput and latency numbers for the workload at that instance size. Then, scale the instance up to the next larger size. See if the throughput and latency numbers improve. Also scale the instance size down, and see if the latency and throughput numbers remain the same. Your goal is to get the highest throughput, with the lowest latency, on the smallest instance possible.

Tip

Size your Aurora clusters and associated DB instances with enough existing capacity to handle sudden, unpredictable traffic spikes. For mission-critical databases, leave at least 20 percent spare CPU and memory capacity.

Run performance tests long enough to measure database performance in a warm, steady state. You might need to run the workload for many minutes or even a few hours before reaching this steady state. It's normal at the beginning of a run to have some variance. This variance happens because each Aurora Replica warms up its caches based on the `SELECT` queries that it handles.

Aurora performs best with transactional workloads involving multiple concurrent users and queries. To ensure that you're driving enough load for optimal performance, run benchmarks that use multithreading, or run multiple instances of the performance tests concurrently. Measure performance with hundreds or even thousands of concurrent client threads. Simulate the number of concurrent threads that you expect in your production environment. You might also perform additional stress tests with more threads to measure Aurora scalability.

12. Exercise Aurora high availability

Many of the main Aurora features involve high availability. These features include automatic replication, automatic failover, automatic backups with point-in-time restore, and ability to add

DB instances to the cluster. The safety and reliability from features like these are important for mission-critical applications.

To evaluate these features requires a certain mindset. In earlier activities, such as performance measurement, you observe how the system performs when everything works correctly. Testing high availability requires you to think through worst-case behavior. You must consider various kinds of failures, even if such conditions are rare. You might intentionally introduce problems to make sure that the system recovers correctly and quickly.

> **Tip**
>
> For a proof of concept, set up all the DB instances in an Aurora cluster with the same AWS instance class. Doing so makes it possible to try out Aurora availability features without major changes to performance and scalability as you take DB instances offline to simulate failures.

We recommend using at least two instances in each Aurora cluster. The DB instances in an Aurora cluster can span up to three Availability Zones (AZs). Locate each of the first two or three DB instances in a different AZ. When you begin using larger clusters, spread your DB instances across all of the AZs in your AWS Region. Doing so increases fault tolerance capability. Even if a problem affects an entire AZ, Aurora can fail over to a DB instance in a different AZ. If you run a cluster with more than three instances, distribute the DB instances as evenly as you can over all three AZs.

> **Tip**
>
> The storage for an Aurora cluster is independent from the DB instances. The storage for each Aurora cluster always spans three AZs.
> When you test high availability features, always use DB instances with identical capacity in your test cluster. Doing so avoids unpredictable changes in performance, latency, and so on whenever one DB instance takes over for another.

To learn how to simulate failure conditions to test high availability features, see Testing Amazon Aurora MySQL using fault injection queries.

As part of your proof-of-concept exercise, one objective is to find the ideal number of DB instances and the optimal instance class for those DB instances. Doing so requires balancing the requirements of high availability and performance.

For Aurora, the more DB instances that you have in a cluster, the greater the benefits for high availability. Having more DB instances also improves scalability of read-intensive applications. Aurora can distribute multiple connections for `SELECT` queries among the read-only Aurora Replicas.

On the other hand, limiting the number of DB instances reduces the replication traffic from the primary node. The replication traffic consumes network bandwidth, which is another aspect of overall performance and scalability. Thus, for write-intensive OLTP applications, prefer to have a smaller number of large DB instances rather than many small DB instances.

In a typical Aurora cluster, one DB instance (the primary instance) handles all the DDL and DML statements. The other DB instances (the Aurora Replicas) handle only `SELECT` statements. Although the DB instances don't do exactly the same amount of work, we recommend using the same instance class for all the DB instances in the cluster. That way, if a failure happens and Aurora promotes one of the read-only DB instances to be the new primary instance, the primary instance has the same capacity as before.

If you need to use DB instances of different capacities in the same cluster, set up failover tiers for the DB instances. These tiers determine the order in which Aurora Replicas are promoted by the failover mechanism. Put DB instances that are a lot larger or smaller than the others into a lower failover tier. Doing so ensures that they are chosen last for promotion.

Exercise the data recovery features of Aurora, such as automatic point-in-time restore, manual snapshots and restore, and cluster backtracking. If appropriate, copy snapshots to other AWS Regions and restore into other AWS Regions to mimic DR scenarios.

Investigate your organization's requirements for restore time objective (RTO), restore point objective (RPO), and geographic redundancy. Most organizations group these items under the broad category of disaster recovery. Evaluate the Aurora high availability features described in this section in the context of your disaster recovery process to ensure that your RTO and RPO requirements are met.

13. What to do next

At the end of a successful proof-of-concept process, you confirm that Aurora is a suitable solution for you based on the anticipated workload. Throughout the preceding process, you've checked how Aurora works in a realistic operational environment and measured it against your success criteria.

After you get your database environment up and running with Aurora, you can move on to more detailed evaluation steps, leading to your final migration and production deployment. Depending on your situation, these other steps might or might not be included in the proof-of-concept process. For details about migration and porting activities, see the AWS whitepaper Aurora migration handbook.

In another next step, consider the security configurations relevant for your workload and designed to meet your security requirements in a production environment. Plan what controls to put in place to protect access to the Aurora cluster master user credentials. Define the roles and responsibilities of database users to control access to data stored in the Aurora cluster. Take into account database access requirements for applications, scripts, and third-party tools or services. Explore AWS services and features such as AWS Secrets Manager and AWS Identity and Access Management (IAM) authentication.

At this point, you should understand the procedures and best practices for running benchmark tests with Aurora. You might find you need to do additional performance tuning. For details, see Managing performance and scaling for Aurora DB clusters, Amazon Aurora MySQL performance enhancements, Managing Amazon Aurora PostgreSQL, and Monitoring DB load with Performance Insights on Amazon Aurora. If you do additional tuning, make sure that you're familiar with the metrics that you gathered during the proof of concept. For a next step, you might create new clusters with different choices for configuration settings, database engine, and database version. Or you might create specialized kinds of Aurora clusters to match the needs of specific use cases.

For example, you can explore Aurora parallel query clusters for hybrid transaction/analytical processing (HTAP) applications. If wide geographic distribution is crucial for disaster recovery or to minimize latency, you can explore Aurora global databases. If your workload is intermittent or you're using Aurora in a development/test scenario, you can explore Aurora Serverless clusters.

Your production clusters might also need to handle high volumes of incoming connections. To learn those techniques, see the AWS whitepaper Aurora MySQL database administrator's handbook – Connection management.

If, after the proof of concept, you decide that your use case is not suited for Aurora, consider these other AWS services:

- For purely analytic use cases, workloads benefit from a columnar storage format and other features more suitable to OLAP workloads. AWS services that address such use cases include the following:
  - Amazon Redshift

  - Amazon EMR
  - Amazon Athena
- Many workloads benefit from a combination of Aurora with one or more of these services. You can move data between these services by using these:
  - AWS Glue
  - AWS DMS
  - Importing from Amazon S3, as described in the *Amazon Aurora User Guide*
  - Exporting to Amazon S3, as described in the *Amazon Aurora User Guide*
  - Many other popular ETL tools

Security in Amazon Aurora

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon Aurora (Aurora), see AWS services in scope by compliance program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon Aurora. The following topics show you how to configure Amazon Aurora to meet your security and compliance objectives. You also learn how to use other AWS services that help you monitor and secure your Amazon Aurora resources.

You can manage access to your Amazon Aurora resources and your databases on a DB cluster. The method you use to manage access depends on what type of task the user needs to perform with Amazon Aurora:

- Run your DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service for the greatest possible network access control. For more information about creating a DB cluster in a VPC, see Amazon VPC VPCs and Amazon Aurora.
- Use AWS Identity and Access Management (IAM) policies to assign permissions that determine who is allowed to manage Amazon Aurora resources. For example, you can use IAM to determine who is allowed to create, describe, modify, and delete DB clusters, tag resources, or modify security groups.

  To review IAM policy examples, see Identity-based policy examples for Amazon Aurora.

- Use security groups to control what IP addresses or Amazon EC2 instances can connect to your databases on a DB cluster. When you first create a DB cluster, its firewall prevents any database access except through rules specified by an associated security group.
- Use Secure Socket Layer (SSL) or Transport Layer Security (TLS) connections with DB clusters running the Aurora MySQL or Aurora PostgreSQL. For more information on using SSL/TLS with a DB cluster, see Using SSL/TLS to encrypt a connection to a DB cluster.
- Use Amazon Aurora encryption to secure your DB clusters and snapshots at rest. Amazon Aurora encryption uses the industry standard AES-256 encryption algorithm to encrypt your data on the server that hosts your DB cluster. For more information, see Encrypting Amazon Aurora resources.
- Use the security features of your DB engine to control who can log in to the databases on a DB cluster. These features work just as if the database was on your local network.

  For information about security with Aurora MySQL, see Security with Amazon Aurora MySQL. For information about security with Aurora PostgreSQL, see Security with Amazon Aurora PostgreSQL.

Aurora is part of the managed database service Amazon Relational Database Service (Amazon RDS). Amazon RDS is a web service that makes it easier to set up, operate, and scale a relational database in the cloud. If you are not already familiar with Amazon RDS, see the *Amazon RDS user guide*.

Aurora includes a high-performance storage subsystem. Its MySQL- and PostgreSQL-compatible database engines are customized to take advantage of that fast distributed storage. Aurora also automates and standardizes database clustering and replication, which are typically among the most challenging aspects of database configuration and administration.

For both Amazon RDS and Aurora, you can access the RDS API programmatically, and you can use the AWS CLI to access the RDS API interactively. Some RDS API operations and AWS CLI commands apply to both Amazon RDS and Aurora, while others apply to either Amazon RDS or Aurora. For information about RDS API operations, see Amazon RDS API reference. For more information about the AWS CLI, see AWS Command Line Interface reference for Amazon RDS.

> **Note**
>
> You have to configure security only for your use cases. You don't have to configure security access for processes that Amazon Aurora manages. These include creating backups, automatic failover, and other processes.

For more information on managing access to Amazon Aurora resources and your databases on a DB cluster, see the following topics.

Topics

- Database authentication with Amazon Aurora
- Password management with Amazon Aurora and AWS Secrets Manager
- Data protection in Amazon RDS
- Identity and access management for Amazon Aurora
- Logging and monitoring in Amazon Aurora
- Compliance validation for Amazon Aurora
- Resilience in Amazon Aurora
- Infrastructure security in Amazon Aurora
- Amazon RDS API and interface VPC endpoints (AWS PrivateLink)
- Security best practices for Amazon Aurora
- Controlling access with security groups
- Master user account privileges
- Using service-linked roles for Amazon Aurora
- Amazon VPC VPCs and Amazon Aurora

Database authentication with Amazon Aurora

Amazon Aurora supports several ways to authenticate database users.

Password authentication is available by default for all DB clusters. For Aurora MySQL and Aurora PostgreSQL, you can also add either or both IAM database authentication and Kerberos authentication for the same DB cluster.

Password, Kerberos, and IAM database authentication use different methods of authenticating to the database. Therefore, a specific user can log in to a database using only one authentication method.

For PostgreSQL, use only one of the following role settings for a user of a specific database:

- To use IAM database authentication, assign the `rds_iam` role to the user.
- To use Kerberos authentication, assign the `rds_ad` role to the user.
- To use password authentication, don't assign either the `rds_iam` or `rds_ad` roles to the user.

Don't assign both the `rds_iam` and `rds_ad` roles to a user of a PostgreSQL database either directly or indirectly by nested grant access. If the `rds_iam` role is added to the master user, IAM authentication takes precedence over password authentication so the master user has to log in as an IAM user.

> ⚠ **Important**
>
> We strongly recommend that you do not use the master user directly in your applications. Instead, adhere to the best practice of using a database user created with the minimal privileges required for your application.

Topics

- Password authentication
- IAM database authentication
- Kerberos authentication

Password authentication

With *password authentication,* your database performs all administration of user accounts. You create users with SQL statements such as `CREATE USER`, with the appropriate clause required by the DB engine for specifying passwords. For example, in MySQL the statement is `CREATE USER` *`name`* `IDENTIFIED BY` *`password`*, while in PostgreSQL, the statement is `CREATE USER` *`name`* `WITH PASSWORD` *`password`*.

With password authentication, your database controls and authenticates user accounts. If a DB engine has strong password management features, they can enhance security. Database authentication might be easier to administer using password authentication when you have small user communities. Because clear text passwords are generated in this case, integrating with AWS Secrets Manager can enhance security.

For information about using Secrets Manager with Amazon Aurora, see Creating a basic secret and Rotating secrets for supported Amazon RDS databases in the *AWS Secrets Manager User Guide*. For information about programmatically retrieving your secrets in your custom applications, see Retrieving the secret value in the *AWS Secrets Manager User Guide*.

IAM database authentication

You can authenticate to your DB cluster using AWS Identity and Access Management (IAM) database authentication. With this authentication method, you don't need to use a password when you connect to a DB cluster. Instead, you use an authentication token.

For more information about IAM database authentication, including information about availability for specific DB engines, see IAM database authentication.

Kerberos authentication

Amazon Aurora supports external authentication of database users using Kerberos and Microsoft Active Directory. Kerberos is a network authentication protocol that uses tickets and symmetric-key cryptography to eliminate the need to transmit passwords over the network. Kerberos has been built into Active Directory and is designed to authenticate users to network resources, such as databases.

Amazon Aurora support for Kerberos and Active Directory provides the benefits of single sign-on and centralized authentication of database users. You can keep your user credentials in Active Directory. Active Directory provides a centralized place for storing and managing credentials for multiple DB clusters.

You can make it possible for your database users to authenticate against DB clusters in two ways. They can use credentials stored either in AWS Directory Service for Microsoft Active Directory or in your on-premises Active Directory.

Aurora PostgreSQL doesn't support selective authentication type in forest trust, only forest wide authentication.

Aurora supports Kerberos authentication for Aurora MySQL and Aurora PostgreSQL DB clusters. For more information about Kerberos authentication for Aurora MySQL, see Using Kerberos authentication for Aurora MySQL.

With Kerberos authentication, Aurora PostgreSQL DB clusters support one- and two-way forest trust relationships. For more information, see Using Kerberos authentication with Aurora PostgreSQL.

Password management with Amazon Aurora and AWS Secrets Manager

Amazon Aurora integrates with Secrets Manager to manage master user passwords for your DB clusters.

Topics

Region and version availability

Feature availability and support varies across specific versions of each database engine and across AWS Regions. For more information about version and Region availability with Secrets Manager integration with Amazon Aurora, see Supported Regions and Aurora DB engines for Secrets Manager integration.

Limitations for Secrets Manager integration with Amazon Aurora

Managing master user passwords with Secrets Manager isn't supported for the following features:

- Amazon RDS Blue/Green Deployments
- DB clusters that are part of an Aurora global database
- Aurora Serverless v1 DB clusters
- Aurora MySQL cross-Region read replicas
- Managing master user password with Secrets Manager for a read replica

Overview of managing master user passwords with AWS Secrets Manager

With AWS Secrets Manager, you can replace hard-coded credentials in your code, including database passwords, with an API call to Secrets Manager to retrieve the secret programmatically. For more information about Secrets Manager, see the AWS Secrets Manager User Guide.

When you store database secrets in Secrets Manager, your AWS account incurs charges. For information about pricing, see AWS Secrets Manager Pricing.

You can specify that Aurora manages the master user password in Secrets Manager for an Amazon Aurora DB cluster when you perform one of the following operations:

- Create the DB cluster
- Modify the DB cluster
- Restore the DB cluster from Amazon S3 (Aurora MySQL only)

When you specify that Aurora manages the master user password in Secrets Manager, Aurora generates the password and stores it in Secrets Manager. You can interact directly with the secret to retrieve the credentials for the master user. You can also specify a customer managed key to encrypt the secret, or use the KMS key that is provided by Secrets Manager.

Aurora manages the settings for the secret and rotates the secret every seven days by default. You can modify some of the settings, such as the rotation schedule. If you delete a DB cluster that manages a secret in Secrets Manager, the secret and its associated metadata are also deleted.

To connect to a DB cluster with the credentials in a secret, you can retrieve the secret from Secrets Manager. For more information, see Retrieve secrets from AWS Secrets Manager and Connect to a SQL database with credentials in an AWS Secrets Manager secret in the *AWS Secrets Manager User Guide*.

Benefits of managing master user passwords with Secrets Manager

Managing Aurora master user passwords with Secrets Manager provides the following benefits:

- Aurora automatically generates database credentials.
- Aurora automatically stores and manages database credentials in AWS Secrets Manager.

- Aurora rotates database credentials regularly, without requiring application changes.
- Secrets Manager secures database credentials from human access and plain text view.
- Secrets Manager allows retrieval of database credentials in secrets for database connections.
- Secrets Manager allows fine-grained control of access to database credentials in secrets using IAM.
- You can optionally separate database encryption from credentials encryption with different KMS keys.
- You can eliminate manual management and rotation of database credentials.
- You can monitor database credentials easily with AWS CloudTrail and Amazon CloudWatch.

For more information about the benefits of Secrets Manager, see the AWS Secrets Manager User Guide.

Permissions required for Secrets Manager integration

Users must have the required permissions to perform operations related to Secrets Manager integration. You can create IAM policies that grant permissions to perform specific API operations on the specified resources they need. You can then attach those policies to the IAM permission sets or roles that require those permissions. For more information, see Identity and access management for Amazon Aurora.

For create, modify, or restore operations, the user who specifies that Aurora manages the master user password in Secrets Manager must have permissions to perform the following operations:

- `kms:DescribeKey`
- `secretsmanager:CreateSecret`
- `secretsmanager:TagResource`

For create, modify, or restore operations, the user who specifies the customer managed key to encrypt the secret in Secrets Manager must have permissions to perform the following operations:

- `kms:Decrypt`
- `kms:GenerateDataKey`
- `kms:CreateGrant`

For modify operations, the user who rotates the master user password in Secrets Manager must have permissions to perform the following operation:

- `secretsmanager:RotateSecret`

Enforcing Aurora management of the master user password in AWS Secrets Manager

You can use IAM condition keys to enforce Aurora management of the master user password in AWS Secrets Manager. The following policy doesn't allow users to create or restore DB instances or DB clusters unless the master user password is managed by Aurora in Secrets Manager.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": ["rds:CreateDBInstance", "rds:CreateDBCluster",
 "rds:RestoreDBInstanceFromS3", "rds:RestoreDBClusterFromS3"],
            "Resource": "*",
            "Condition": {
                "Bool": {
                    "rds:ManageMasterUserPassword": false
                }
            }
        }
    ]
}
```

Note

This policy enforces password management in AWS Secrets Manager at creation. However, you can still disable Secrets Manager integration and manually set a master password by modifying the cluster.
To prevent this, include `rds:ModifyDBInstance`, `rds:ModifyDBCluster` in the action block of the policy. Be aware, this prevents the user from applying any further modifications to existing clusters that don't have Secrets Manager integration enabled.

For more information about using condition keys in IAM policies, see Policy condition keys for Aurora and Example policies: Using condition keys.

Managing the master user password for a DB cluster with Secrets Manager

You can configure Aurora management of the master user password in Secrets Manager when you perform the following actions:

- Creating an Amazon Aurora DB cluster
- Modifying an Amazon Aurora DB cluster
- Migrating data from an external MySQL database to an Amazon Aurora MySQL DB cluster

You can use the RDS console, the AWS CLI, or the RDS API to perform these actions.

Console

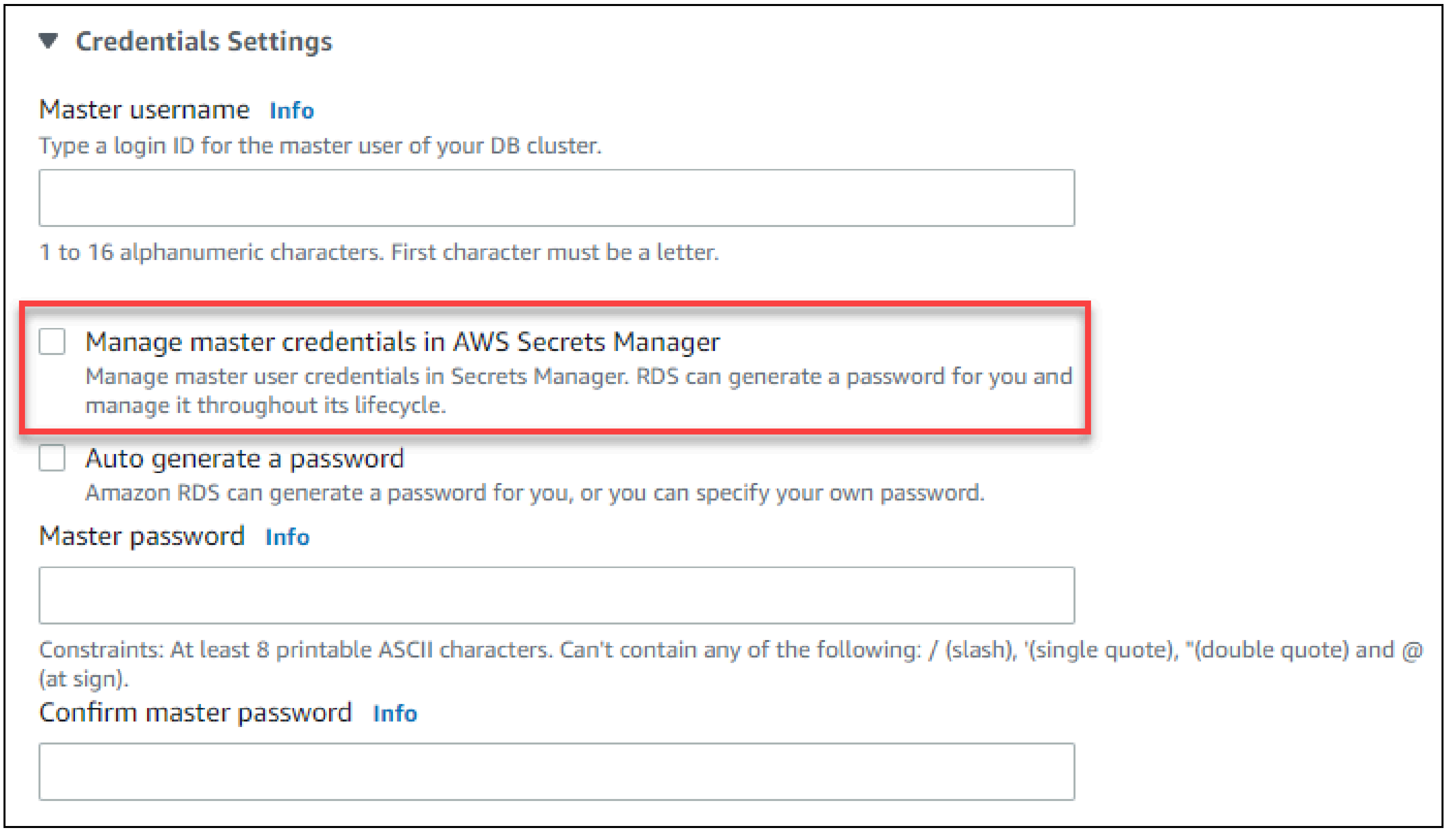
Follow the instructions for creating or modifying a DB cluster with the RDS console:

- Creating a DB cluster
- Modifying a DB instance in a DB cluster

  In the RDS console, you can modify any DB instance to specify the master user password management settings for the entire DB cluster.
- Restoring an Amazon Aurora MySQL DB cluster from an Amazon S3 bucket

When you use the RDS console to perform one of these operations, you can specify that the master user password is managed by Aurora in Secrets Manager. To do so when you are creating or restoring a DB cluster, select **Manage master credentials in AWS Secrets Manager** in **Credential settings**. When you are modifying a DB cluster, select **Manage master credentials in AWS Secrets Manager** in **Settings**.

The following image is an example of the **Manage master credentials in AWS Secrets Manager** setting when you are creating or restoring a DB cluster.



▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB cluster.

1 to 16 alphanumeric characters. First character must be a letter.

□ Manage master credentials in AWS Secrets Manager

Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

□ Auto generate a password

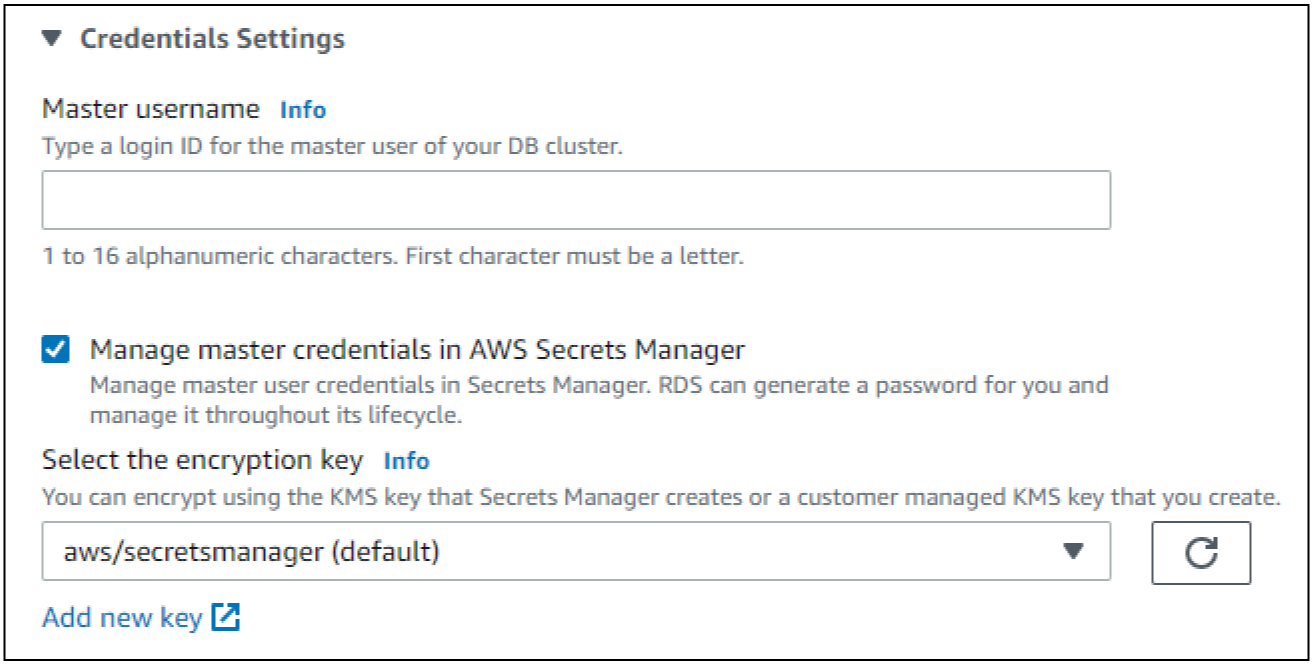
Amazon RDS can generate a password for you, or you can specify your own password.

Master password Info

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm master password Info

When you select this option, Aurora generates the master user password and manages it throughout its lifecycle in Secrets Manager.



▼ Credentials Settings

Master username Info

Type a login ID for the master user of your DB cluster.

1 to 16 alphanumeric characters. First character must be a letter.

☑ Manage master credentials in AWS Secrets Manager

Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

Select the encryption key Info

You can encrypt using the KMS key that Secrets Manager creates or a customer managed KMS key that you create.

aws/secretsmanager (default) ▼

Add new key

You can choose to encrypt the secret with a KMS key that Secrets Manager provides or with a customer managed key that you create. After Aurora is managing the database credentials for a DB cluster, you can't change the KMS key that is used to encrypt the secret.

You can choose other settings to meet your requirements.

For more information about the available settings when you are creating a DB cluster, see Settings for Aurora DB clusters. For more information about the available settings when you are modifying a DB cluster, see Settings for Amazon Aurora.

AWS CLI

To specify that Aurora manages the master user password in Secrets Manager, specify the `--manage-master-user-password` option in one of the following commands:

- create-db-cluster
- modify-db-cluster
- restore-db-cluster-from-s3

When you specify the `--manage-master-user-password` option in these commands, Aurora generates the master user password and manages it throughout its lifecycle in Secrets Manager.

To encrypt the secret, you can specify a customer managed key or use the default KMS key that is provided by Secrets Manager. Use the `--master-user-secret-kms-key-id` option to specify a customer managed key. The AWS KMS key identifier is the key ARN, key ID, alias ARN, or alias name for the KMS key. To use a KMS key in a different AWS account, specify the key ARN or alias ARN. After Aurora is managing the database credentials for a DB cluster, you can't change the KMS key that is used to encrypt the secret.

You can choose other settings to meet your requirements.

For more information about the available settings when you are creating a DB cluster, see Settings for Aurora DB clusters. For more information about the available settings when you are modifying a DB cluster, see Settings for Amazon Aurora.

This example creates a DB cluster and specifies that Aurora manages the password in Secrets Manager. The secret is encrypted using the KMS key that is provided by Secrets Manager.

Example

For Linux, macOS, or Unix:

```
aws rds create-db-cluster \
     --db-cluster-identifier sample-cluster \
     --engine aurora-mysql \
     --engine-version 8.0 \
     --master-username admin \
     --manage-master-user-password
```

For Windows:

```
aws rds create-db-cluster ^
     --db-cluster-identifier sample-cluster ^
     --engine aurora-mysql ^
     --engine-version 8.0 ^
     --master-username admin ^
     --manage-master-user-password
```

RDS API

To specify that Aurora manages the master user password in Secrets Manager, set the `ManageMasterUserPassword` parameter to `true` in one of the following operations:

- CreateDBCluster
- ModifyDBCluster
- RestoreDBClusterFromS3

When you set the `ManageMasterUserPassword` parameter to `true` in one of these operations, Aurora generates the master user password and manages it throughout its lifecycle in Secrets Manager.

To encrypt the secret, you can specify a customer managed key or use the default KMS key that is provided by Secrets Manager. Use the `MasterUserSecretKmsKeyId` parameter to specify a customer managed key. The AWS KMS key identifier is the key ARN, key ID, alias ARN, or alias name for the KMS key. To use a KMS key in a different AWS account, specify the key ARN or alias ARN. After Aurora is managing the database credentials for a DB cluster, you can't change the KMS key that is used to encrypt the secret.

Rotating the master user password secret for a DB cluster

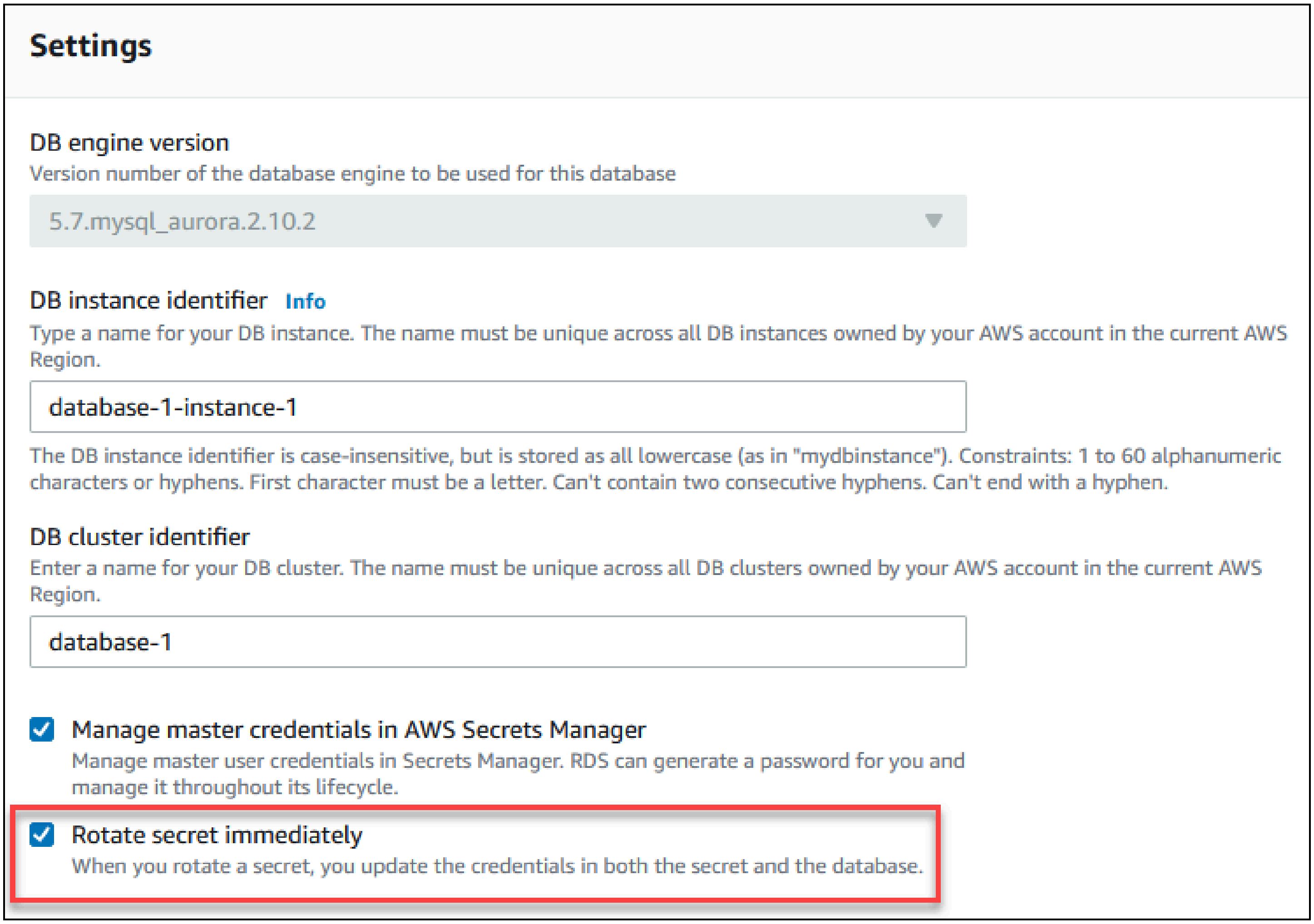
When Aurora rotates a master user password secret, Secrets Manager generates a new secret version for the existing secret. The new version of the secret contains the new master user password. Aurora changes the master user password for the DB cluster to match the password for the new secret version.

You can rotate a secret immediately instead of waiting for a scheduled rotation. To rotate a master user password secret in Secrets Manager, modify the DB cluster. For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

You can rotate a master user password secret immediately with the RDS console, the AWS CLI, or the RDS API. The new password is always 28 characters long and contains atleast one upper and lowercase character, one number, and one punctuation.

Console

To rotate a master user password secret using the RDS console, modify the DB cluster and select **Rotate secret immediately** in **Settings**.



Settings

DB engine version
Version number of the database engine to be used for this database

5.7.mysql_aurora.2.10.2 ▼

DB instance identifier Info
Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

database-1-instance-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

DB cluster identifier
Enter a name for your DB cluster. The name must be unique across all DB clusters owned by your AWS account in the current AWS Region.

database-1

☑ Manage master credentials in AWS Secrets Manager
Manage master user credentials in Secrets Manager. RDS can generate a password for you and manage it throughout its lifecycle.

☑ Rotate secret immediately
When you rotate a secret, you update the credentials in both the secret and the database.

Follow the instructions for modifying a DB cluster with the RDS console in Modifying the DB cluster by using the console, CLI, and API. You must choose **Apply immediately** on the confirmation page.

AWS CLI

To rotate a master user password secret using the AWS CLI, use the modify-db-cluster command and specify the `--rotate-master-user-password` option. You must specify the `--apply-immediately` option when you rotate the master password.

This example rotates a master user password secret.

Example

For Linux, macOS, or Unix:

```
aws rds modify-db-cluster \
    --db-cluster-identifier mydbcluster \
    --rotate-master-user-password \
    --apply-immediately
```

For Windows:

```
aws rds modify-db-cluster ^
    --db-cluster-identifier mydbcluster ^
    --rotate-master-user-password ^
    --apply-immediately
```

RDS API

You can rotate a master user password secret using the ModifyDBCluster operation and setting the `RotateMasterUserPassword` parameter to `true`. You must set the `ApplyImmediately` parameter to `true` when you rotate the master password.

Viewing the details about a secret for a DB cluster

You can retrieve your secrets using the console (https://console.aws.amazon.com/secretsmanager/) or the AWS CLI (get-secret-value Secrets Manager command).

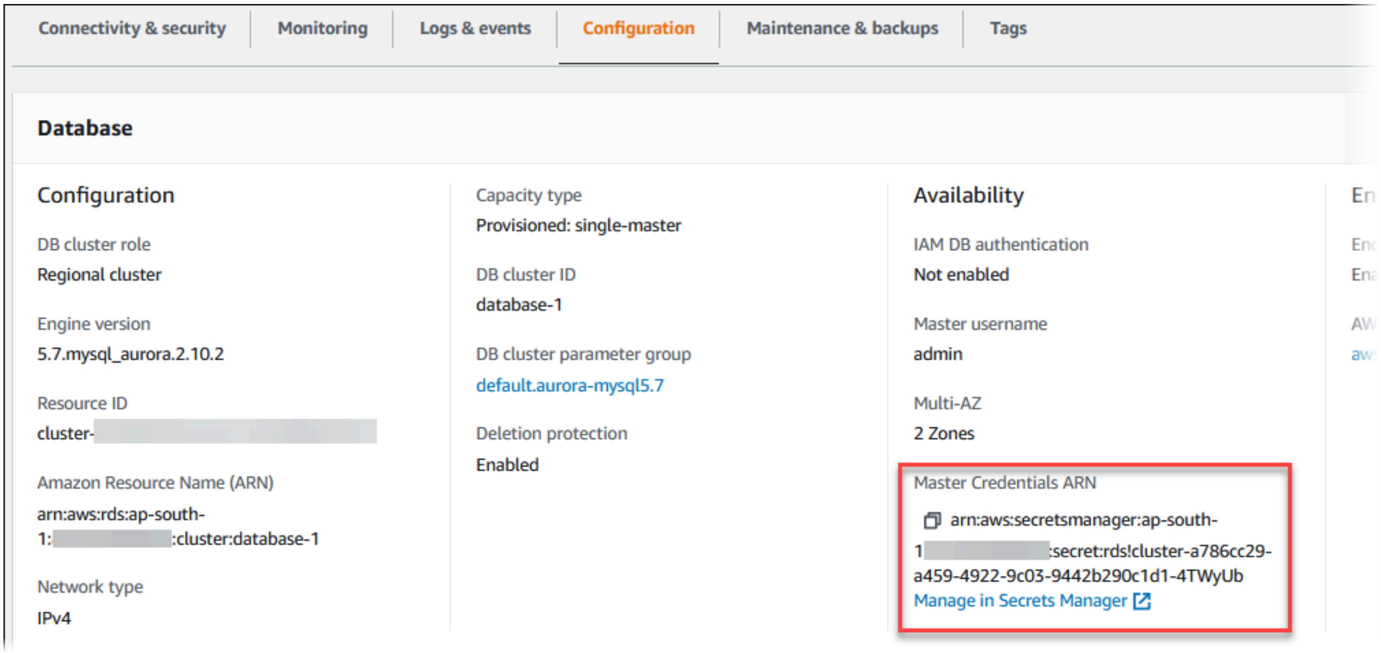
You can find the Amazon Resource Name (ARN) of a secret managed by Aurora in Secrets Manager with the RDS console, the AWS CLI, or the RDS API.

Console

To view the details about a secret managed by Aurora in Secrets Manager

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the DB cluster to show its details.
4. Choose the **Configuration** tab.

   In **Master Credentials ARN**, you can view the secret ARN.



Database

Configuration

DB cluster role
Regional cluster

Engine version
5.7.mysql_aurora.2.10.2

Resource ID
cluster-[redacted]

Amazon Resource Name (ARN)
arn:aws:rds:ap-south-1:[redacted]:cluster:database-1

Network type
IPv4

Capacity type
Provisioned: single-master

DB cluster ID
database-1

DB cluster parameter group
default.aurora-mysql5.7

Deletion protection
Enabled

Availability

IAM DB authentication
Not enabled

Master username
admin

Multi-AZ
2 Zones

Master Credentials ARN
arn:aws:secretsmanager:ap-south-1[redacted]:secret:rds!cluster-a786cc29-a459-4922-9c03-9442b290c1d1-4TWyUb
Manage in Secrets Manager

En

Enc
Ena

AW
aws

You can follow the **Manage in Secrets Manager** link to view and manage the secret in the Secrets Manager console.

AWS CLI

You can use the RDS AWS CLI describe-db-clusters command to find the following information about a secret managed by Aurora in Secrets Manager:

- `SecretArn` – The ARN of the secret
- `SecretStatus` – The status of the secret

  The possible status values include the following:

  - `creating` – The secret is being created.
  - `active` – The secret is available for normal use and rotation.
  - `rotating` – The secret is being rotated.
  - `impaired` – The secret can be used to access database credentials, but it can't be rotated. A secret might have this status if, for example, permissions are changed so that RDS can no longer access the secret or the KMS key for the secret.

    When a secret has this status, you can correct the condition that caused the status. If you correct the condition that caused status, the status remains `impaired` until the next rotation. Alternatively, you can modify the DB cluster to turn off automatic management of database credentials, and then modify the DB cluster again to turn on automatic management of database credentials. To modify the DB cluster, use the `--manage-master-user-password` option in the modify-db-cluster command.

- `KmsKeyId` – The ARN of the KMS key that is used to encrypt the secret

Specify the `--db-cluster-identifier` option to show output for a specific DB cluster. This example shows the output for a secret that is used by a DB cluster.

Example

```
aws rds describe-db-clusters --db-cluster-identifier mydbcluster
```

The following sample shows the output for a secret:

```
"MasterUserSecret": {
                "SecretArn": "arn:aws:secretsmanager:eu-west-1:123456789012:secret:rds!
cluster-033d7456-2c96-450d-9d48-f5de3025e51c-xmJRDx",
                "SecretStatus": "active",
                "KmsKeyId": "arn:aws:kms:eu-
west-1:123456789012:key/0987dcba-09fe-87dc-65ba-ab0987654321"
            }
```

When you have the secret ARN, you can view details about the secret using the get-secret-value Secrets Manager CLI command.

This example shows the details for the secret in the previous sample output.

Example

For Linux, macOS, or Unix:

```
aws secretsmanager get-secret-value \
    --secret-id 'arn:aws:secretsmanager:eu-west-1:123456789012:secret:rds!
cluster-033d7456-2c96-450d-9d48-f5de3025e51c-xmJRDx'
```

For Windows:

```
aws secretsmanager get-secret-value ^
    --secret-id 'arn:aws:secretsmanager:eu-west-1:123456789012:secret:rds!
cluster-033d7456-2c96-450d-9d48-f5de3025e51c-xmJRDx'
```

RDS API

You can view the ARN, status, and KMS key for a secret managed by Aurora in Secrets Manager using the DescribeDBClusters RDS operation and setting the `DBClusterIdentifier` parameter to a DB cluster identifier. Details about the secret are included in the output.

When you have the secret ARN, you can view details about the secret using the GetSecretValue Secrets Manager operation.

Data protection in Amazon RDS

The AWS shared responsibility model applies to data protection in Amazon Relational Database Service. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Amazon RDS or other AWS services using the console, API, AWS CLI, or AWS

SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Topics

- Protecting data using encryption
- Internetwork traffic privacy

Protecting data using encryption

You can enable encryption for database resources. You can also encrypt connections to DB clusters.

Topics

- Encrypting Amazon Aurora resources
- AWS KMS key management
- Using SSL/TLS to encrypt a connection to a DB cluster
- Rotating your SSL/TLS certificate

Encrypting Amazon Aurora resources

Amazon Aurora can encrypt your Amazon Aurora DB clusters. Data that is encrypted at rest includes the underlying storage for DB clusters, its automated backups, read replicas, and snapshots.

Amazon Aurora encrypted DB clusters use the industry standard AES-256 encryption algorithm to encrypt your data on the server that hosts your Amazon Aurora DB clusters. After your data is encrypted, Amazon Aurora handles authentication of access and decryption of your data transparently with a minimal impact on performance. You don't need to modify your database client applications to use encryption.

> **Note**
>
> For encrypted and unencrypted DB clusters, data that is in transit between the source and the read replicas is encrypted, even when replicating across AWS Regions.

Topics

Overview of encrypting Amazon Aurora resources

Amazon Aurora encrypted DB clusters provide an additional layer of data protection by securing your data from unauthorized access to the underlying storage. You can use Amazon Aurora encryption to increase data protection of your applications deployed in the cloud, and to fulfill compliance requirements for encryption at rest.

For an Amazon Aurora encrypted DB cluster, all DB instances, logs, backups, and snapshots are encrypted. You can also encrypt a read replica of an Amazon Aurora encrypted cluster. Amazon Aurora uses an AWS Key Management Service key to encrypt these resources. For more information about KMS keys, see AWS KMS keys in the *AWS Key Management Service Developer Guide* and AWS KMS key management. Each DB instance in the DB cluster is encrypted using the same KMS key as the DB cluster. If you copy an encrypted snapshot, you can use a different KMS key to encrypt the target snapshot than the one that was used to encrypt the source snapshot.

You can use an AWS managed key, or you can create customer managed keys. To manage the customer managed keys used for encrypting and decrypting your Amazon Aurora resources, you use the AWS Key Management Service (AWS KMS). AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud. Using AWS KMS, you can create customer managed keys and define the policies that control how these customer managed keys can be used. AWS KMS supports CloudTrail, so you can audit KMS key usage to verify that customer managed keys are being used appropriately. You can use your customer managed keys with Amazon Aurora and supported AWS services such as Amazon S3, Amazon EBS, and Amazon Redshift. For a list of services that are integrated with AWS KMS, see AWS Service Integration.

Encrypting an Amazon Aurora DB cluster

To encrypt a new DB cluster, choose **Enable encryption** on the console. For information on creating a DB cluster, see Creating an Amazon Aurora DB cluster.

If you use the create-db-cluster AWS CLI command to create an encrypted DB cluster, set the `--storage-encrypted` parameter. If you use the CreateDBCluster API operation, set the `StorageEncrypted` parameter to true.

When you create an encrypted DB cluster, you can choose a customer managed key or the AWS managed key for Amazon Aurora to encrypt your DB cluster. If you don't specify the key identifier for a customer managed key, Amazon Aurora uses the AWS managed key for your new DB cluster. Amazon Aurora creates an AWS managed key for Amazon Aurora for your AWS account. Your AWS account has a different AWS managed key for Amazon Aurora for each AWS Region.

For more information about KMS keys, see AWS KMS keys in the *AWS Key Management Service Developer Guide*.

Once you have created an encrypted DB cluster, you can't change the KMS key used by that DB cluster. Therefore, be sure to determine your KMS key requirements before you create your encrypted DB cluster.

If you use the AWS CLI `create-db-cluster` command to create an encrypted DB cluster with a customer managed key, set the `--kms-key-id` parameter to any key identifier for the KMS key. If you use the Amazon RDS API `CreateDBInstance` operation, set the `KmsKeyId` parameter to any key identifier for the KMS key. To use a customer managed key in a different AWS account, specify the key ARN or alias ARN.

⚠ **Important**

Amazon Aurora can lose access to the KMS key for a DB cluster when you disable the KMS key. In these cases, the encrypted DB cluster shortly goes into `inaccessible-encryption-credentials-recoverable` state. The DB cluster remains in this state for seven days, during which the instance is stopped. API calls made to the DB cluster during this time might not succeed. To recover the DB cluster, enable the KMS key and restart this DB cluster. Enable the KMS key from the AWS Management Console. Restart the DB cluster using the AWS CLI command start-db-cluster or AWS Management Console.
If the DB cluster isn't recovered within seven days, it goes into the terminal `inaccessible-encryption-credentials` state. In this state, the DB cluster is not usable anymore and you can only restore the DB cluster from a backup. We strongly recommend that you always turn on backups for encrypted DB clusters to guard against the loss of encrypted data in your databases.
During the creation of a DB cluster, Aurora checks if the calling principal has access to the KMS key and generates a grant from the KMS key that it uses for the entire lifetime of the

DB cluster. Revoking the calling principals access to the KMS key does not affect a running database. When using KMS keys in cross-account scenarios, such as copying a snapshot to another account, the KMS key needs to be shared with the other account. If you create a DB cluster from the snapshot without specifying a different KMS key, the new cluster uses the KMS key from the source account. Revoking access to the key after you create the DB cluster does not affect the cluster. However, disabling the key impacts all DB clusters encrypted with that key. To prevent this, specify a different key during the snapshot copy operation.

Determining whether encryption is turned on for a DB cluster

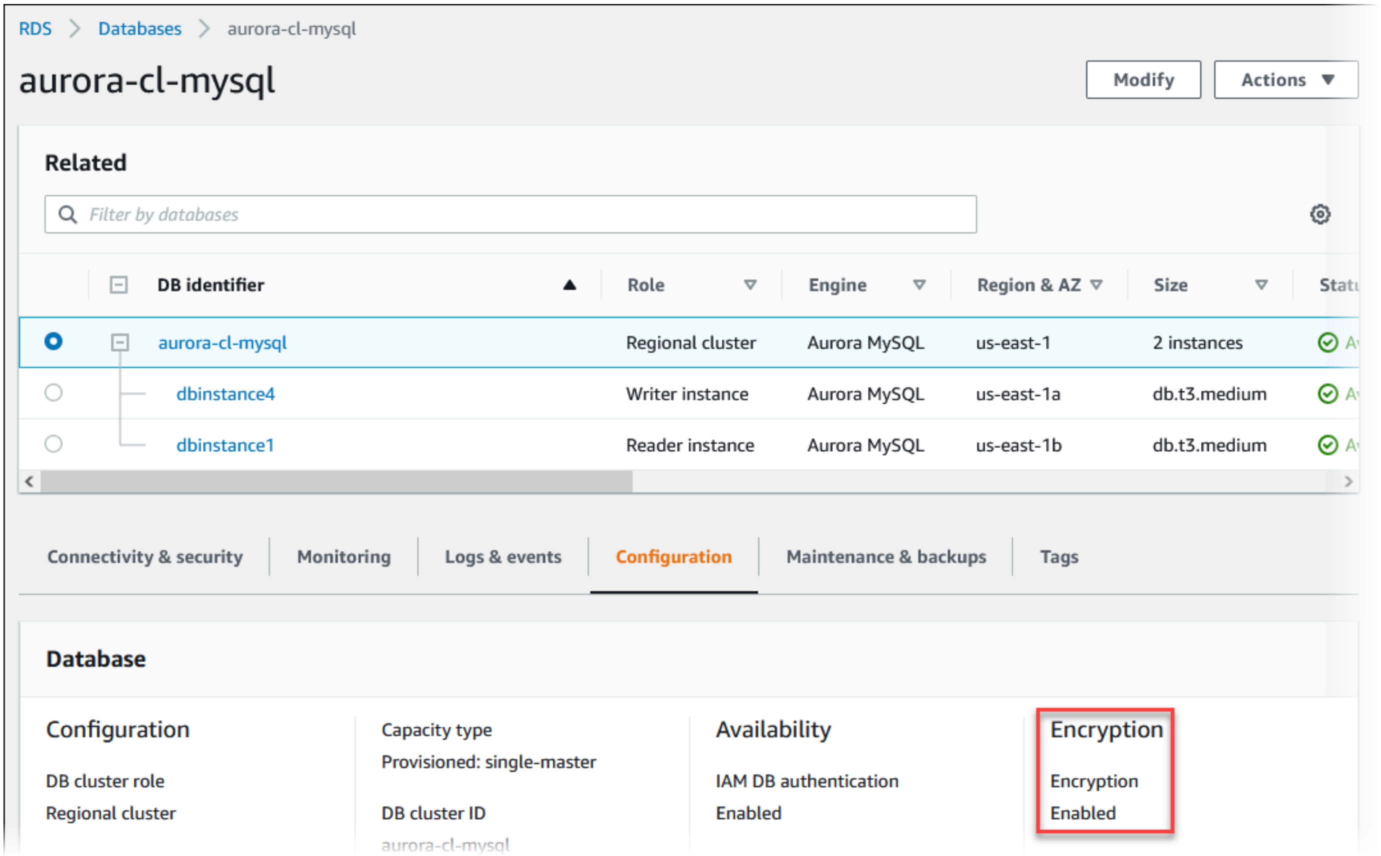
You can use the AWS Management Console, AWS CLI, or RDS API to determine whether encryption at rest is turned on for a DB cluster.

Console

To determine whether encryption at rest is turned on for a DB cluster

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the name of the DB cluster that you want to check to view its details.
4. Choose the **Configuration** tab and check the **Encryption** value.

   It shows either **Enabled** or **Not enabled**.



RDS > Databases > aurora-cl-mysql

aurora-cl-mysql

Related

	DB identifier	Role	Engine	Region & AZ	Size	Statu
●	aurora-cl-mysql	Regional cluster	Aurora MySQL	us-east-1	2 instances	✓ A
○	dbinstance4	Writer instance	Aurora MySQL	us-east-1a	db.t3.medium	✓ A
○	dbinstance1	Reader instance	Aurora MySQL	us-east-1b	db.t3.medium	✓ A

Database

Configuration

DB cluster role
Regional cluster

Capacity type
Provisioned: single-master

DB cluster ID
aurora-cl-mysql

Availability

IAM DB authentication
Enabled

Encryption

Encryption
Enabled

AWS CLI

To determine whether encryption at rest is turned on for a DB cluster by using the AWS CLI, call the describe-db-clusters command with the following option:

- `--db-cluster-identifier` – The name of the DB cluster.

The following example uses a query to return either `TRUE` or `FALSE` regarding encryption at rest for the `mydb` DB cluster.

Example

```
aws rds describe-db-clusters --db-cluster-identifier mydb --query "*[].
{StorageEncrypted:StorageEncrypted}" --output text
```

RDS API

To determine whether encryption at rest is turned on for a DB cluster by using the Amazon RDS API, call the DescribeDBClusters operation with the following parameter:

- `DBClusterIdentifier` – The name of the DB cluster.

Availability of Amazon Aurora encryption

Amazon Aurora encryption is currently available for all database engines and storage types.

> **Note**
>
> Amazon Aurora encryption is not available for the db.t2.micro DB instance class.

Encryption in transit

AWS provides secure and private connectivity between DB instances of all types. In addition, some instance types use the offload capabilities of the underlying Nitro System hardware to automatically encrypt in-transit traffic between instances. This encryption uses Authenticated Encryption with Associated Data (AEAD) algorithms, with 256-bit encryption. There is no impact on network performance. To support this additional in-transit traffic encryption between instances, the following requirements must be met:

- The instances use the following instance types:
  - **General purpose**: M6i, M6id, M6in, M6idn, M7g
  - **Memory optimized**: R6i, R6id, R6in, R6idn, R7g, X2idn, X2iedn, X2iezn
- The instances are in the same AWS Region.
- The instances are in the same VPC or peered VPCs, and the traffic does not pass through a virtual network device or service, such as a load balancer or a transit gateway.

Limitations of Amazon Aurora encrypted DB clusters

The following limitations exist for Amazon Aurora encrypted DB clusters:

- You can't turn off encryption on an encrypted DB cluster.
- You can't create an encrypted snapshot of an unencrypted DB cluster.
- A snapshot of an encrypted DB cluster must be encrypted using the same KMS key as the DB cluster.

- You can't convert an unencrypted DB cluster to an encrypted one. However, you can restore an unencrypted snapshot to an encrypted Aurora DB cluster. To do this, specify a KMS key when you restore from the unencrypted snapshot.
- You can't create an encrypted Aurora Replica from an unencrypted Aurora DB cluster. You can't create an unencrypted Aurora Replica from an encrypted Aurora DB cluster.
- To copy an encrypted snapshot from one AWS Region to another, you must specify the KMS key in the destination AWS Region. This is because KMS keys are specific to the AWS Region that they are created in.

  The source snapshot remains encrypted throughout the copy process. Amazon Aurora uses envelope encryption to protect data during the copy process. For more information about envelope encryption, see Envelope encryption in the *AWS Key Management Service Developer Guide*.
- You can't unencrypt an encrypted DB cluster. However, you can export data from an encrypted DB cluster and import the data into an unencrypted DB cluster.

AWS KMS key management

Amazon Aurora automatically integrates with AWS Key Management Service (AWS KMS) for key management. Amazon Aurora uses envelope encryption. For more information about envelope encryption, see Envelope encryption in the *AWS Key Management Service Developer Guide*.

You can use two types of AWS KMS keys to encrypt your DB clusters.

- If you want full control over a KMS key, you must create a *customer managed key*. For more information about customer managed keys, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

  You can't share a snapshot that has been encrypted using the AWS managed key of the AWS account that shared the snapshot.
- *AWS managed keys* are KMS keys in your account that are created, managed, and used on your behalf by an AWS service that is integrated with AWS KMS. By default, the RDS AWS managed key (`aws/rds`) is used for encryption. You can't manage, rotate, or delete the RDS AWS managed key. For more information about AWS managed keys, see AWS managed keys in the *AWS Key Management Service Developer Guide*.

To manage KMS keys used for Amazon Aurora encrypted DB clusters, use the AWS Key Management Service (AWS KMS) in the AWS KMS console, the AWS CLI, or the AWS KMS API. To view audit logs of every action taken with an AWS managed or customer managed key, use AWS CloudTrail. For more information about key rotation, see Rotating AWS KMS keys.

> ⚠ **Important**
>
> If you turn off or revoke permissions to a KMS key used by an RDS database, RDS puts your database into a terminal state when access to the KMS key is required. This change could be immediate, or deferred, depending on the use case that required access to the KMS key. In this state, the DB cluster is no longer available, and the current state of the database can't be recovered. To restore the DB cluster, you must re-enable access to the KMS key for RDS, and then restore the DB cluster from the latest available backup.

Authorizing use of a customer managed key

When Aurora uses a customer managed key in cryptographic operations, it acts on behalf of the user who is creating or changing the Aurora resource.

To create an Aurora resource using a customer managed key, a user must have permissions to call the following operations on the customer managed key:

- kms:CreateGrant
- kms:DescribeKey

You can specify these required permissions in a key policy, or in an IAM policy if the key policy allows it.

You can make the IAM policy stricter in various ways. For example, if you want to allow the customer managed key to be used only for requests that originate in Aurora, you can use the kms:ViaService condition key with the `rds.<region>.amazonaws.com` value. Also, you can use the keys or values in the Amazon RDS encryption context as a condition for using the customer managed key for encryption.

For more information, see Allowing users in other accounts to use a KMS key in the *AWS Key Management Service Developer Guide* and Key policies in AWS KMS.

Amazon RDS encryption context

When Aurora uses your KMS key, or when Amazon EBS uses the KMS key on behalf of Aurora, the service specifies an encryption context. The encryption context is additional authenticated data (AAD) that AWS KMS uses to ensure data integrity. When an encryption context is specified for an encryption operation, the service must specify the same encryption context for the decryption operation. Otherwise, decryption fails. The encryption context is also written to your AWS CloudTrail logs to help you understand why a given KMS key was used. Your CloudTrail logs might contain many entries describing the use of a KMS key, but the encryption context in each log entry can help you determine the reason for that particular use.

At minimum, Aurora always uses the DB instance ID for the encryption context, as in the following JSON-formatted example:

```
{ "aws:rds:db-id": "db-CQYSMDPBRZ7BPMH7Y3RTDG5QY" }
```

This encryption context can help you identify the DB instance for which your KMS key was used.

When your KMS key is used for a specific DB instance and a specific Amazon EBS volume, both the DB instance ID and the Amazon EBS volume ID are used for the encryption context, as in the following JSON-formatted example:

```
{
  "aws:rds:db-id": "db-BRG7VYS3SVIFQW7234EJQOM5RQ",
  "aws:ebs:id": "vol-ad8c6542"
}
```

Using SSL/TLS to encrypt a connection to a DB cluster

You can use Secure Socket Layer (SSL) or Transport Layer Security (TLS) from your application to encrypt a connection to a DB cluster running Aurora MySQL or Aurora PostgreSQL.

SSL/TLS connections provide a layer of security by encrypting data that moves between your client and DB cluster. Optionally, your SSL/TLS connection can perform server identity verification by validating the server certificate installed on your database. To require server identity verification, follow this general process:

1. Choose the **certificate authority (CA)** that signs the **DB server certificate,** for your database. For more information about certificate authorities, see Certificate authorities.

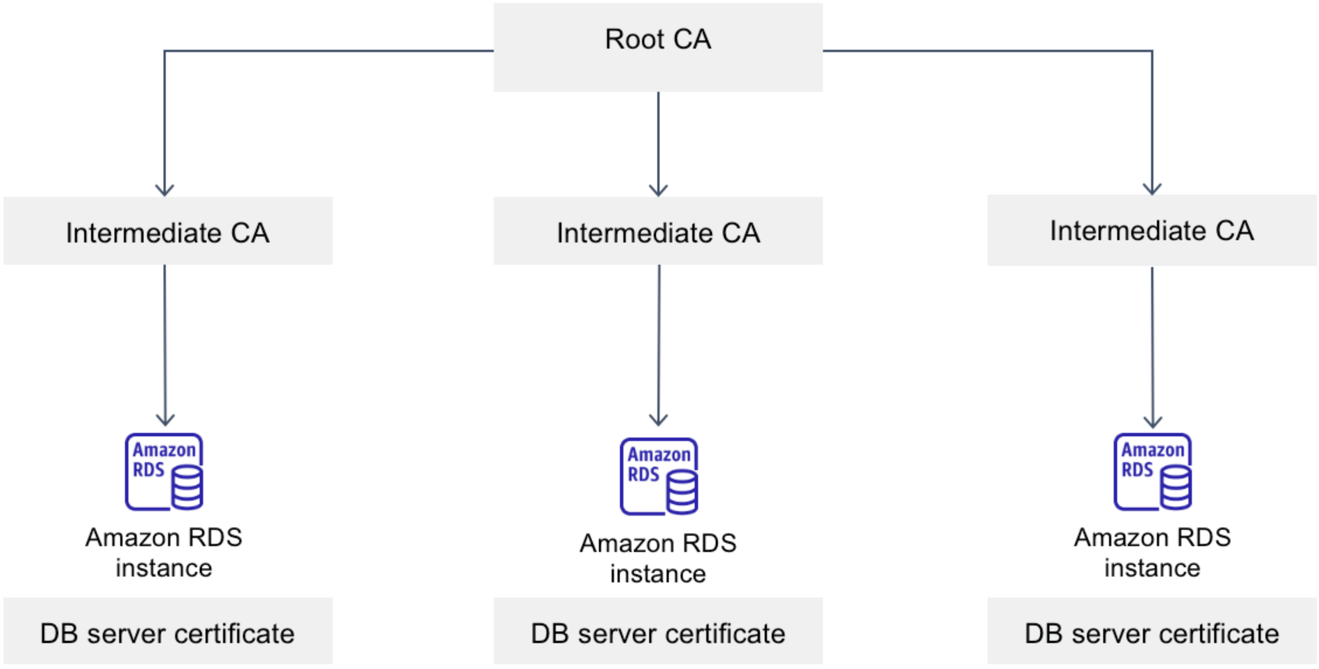
2. Download a certificate bundle to use when you are connecting to the database. To download a certificate bundle, see Certificate bundles for all AWS Regions and Certificate bundles for specific AWS Regions.

   > **Note**
   >
   > All certificates are only available for download using SSL/TLS connections.

3. Connect to the database using your DB engine's process for implementing SSL/TLS connections. Each DB engine has its own process for implementing SSL/TLS. To learn how to implement SSL/TLS for your database, follow the link that corresponds to your DB engine:
   - Security with Amazon Aurora MySQL
   - Security with Amazon Aurora PostgreSQL

Certificate authorities

The **certificate authority (CA)** is the certificate that identifies the root CA at the top of the certificate chain. The CA signs the **DB server certificate,** which is installed on each DB instance. The DB server certificate identifies the DB instance as a trusted server.



Amazon RDS provides the following CAs to sign the DB server certificate for a database.

Certificate authority (CA)	Description
rds-ca-2019	Uses a certificate authority with RSA 2048 private key algorithm and SHA256 signing algorithm. This CA expires in 2024 and doesn't support automatic server certificate rotation. If you are using this CA and want to keep the same standard, we recommend that you switch to the rds-ca-rsa2048-g1 CA.
rds-ca-rsa2048-g1	Uses a certificate authority with RSA 2048 private key algorithm and SHA256 signing algorithm in most AWS Regions. In the AWS GovCloud (US) Regions, this CA uses a certificate authority with RSA 2048 private key algorithm and SHA384 signing algorithm. This CA remains valid for longer than the rds-ca-20 19 CA. This CA supports automatic server certificate rotation.
rds-ca-rsa4096-g1	Uses a certificate authority with RSA 4096 private key algorithm and SHA384 signing algorithm. This CA supports automatic server certificate rotation.
rds-ca-ecc384-g1	Uses a certificate authority with ECC 384 private key algorithm and SHA384 signing algorithm. This CA supports automatic server certificate rotation.

Note

If you are using the AWS CLI, you can see the validities of the certificate authorities listed above by using describe-certificates.

These CA certificates are included in the regional and global certificate bundle. When you use the rds-ca-rsa2048-g1, rds-ca-rsa4096-g1, or rds-ca-ecc384-g1 CA with a database, RDS manages the

DB server certificate on the database. RDS rotates the DB server certificate automatically before it expires.

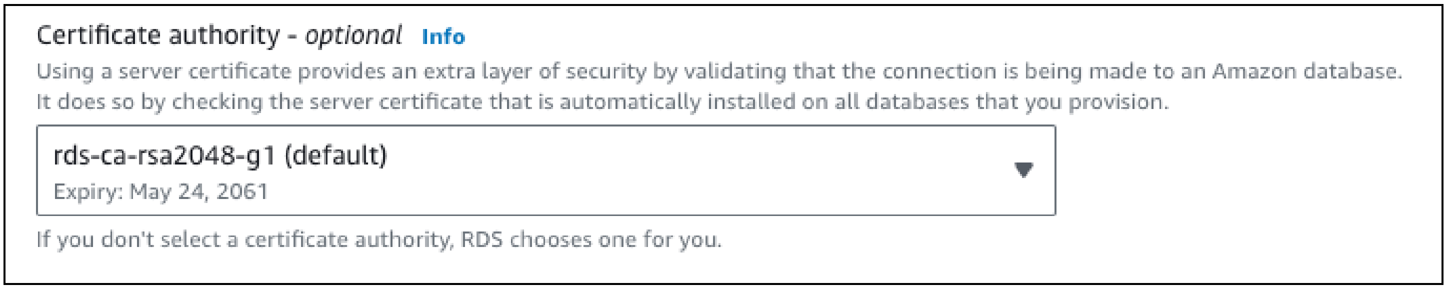
Setting the CA for your database

You can set the CA for a database when you perform the following tasks:

- Create an Aurora DB cluster – You can set the CA for a DB instance in an Aurora cluster when you create the first DB instance in the DB cluster using the AWS CLI or RDS API. Currently, you can't set the CA for the DB instances in a DB cluster when you create the DB cluster using the RDS console. For instructions, see Creating an Amazon Aurora DB cluster.
- Modify a DB instance – You can set the CA for a DB instance in a DB cluster by modifying it. For instructions, see Modifying a DB instance in a DB cluster.

> **Note**
>
> The default CA is set to rds-ca-rsa2048-g1. You can override the default CA for your AWS account by using the modify-certificates command.

The available CAs depend on the DB engine and DB engine version. When you use the AWS Management Console, you can choose the CA using the **Certificate authority** setting, as shown in the following image.



Certificate authority - *optional* Info

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-rsa2048-g1 (default)
Expiry: May 24, 2061

If you don't select a certificate authority, RDS chooses one for you.

The console only shows the CAs that are available for the DB engine and DB engine version. If you're using the AWS CLI, you can set the CA for a DB instance using the create-db-instance or modify-db-instance command.

If you're using the AWS CLI, you can see the available CAs for your account by using the describe-certificates command. This command also shows the expiration date for each CA in `ValidTill` in the output. You can find the CAs that are available for a specific DB engine and DB engine version using the describe-db-engine-versions command.

The following example shows the CAs available for the default RDS for PostgreSQL DB engine version.

```
aws rds describe-db-engine-versions --default-only --engine postgres
```

Your output is similar to the following. The available CAs are listed in `SupportedCACertificateIdentifiers`. The output also shows whether the DB engine version supports rotating the certificate without restart in `SupportsCertificateRotationWithoutRestart`.

```
{
    "DBEngineVersions": [
        {
            "Engine": "postgres",
            "MajorEngineVersion": "13",
            "EngineVersion": "13.4",
            "DBParameterGroupFamily": "postgres13",
            "DBEngineDescription": "PostgreSQL",
            "DBEngineVersionDescription": "PostgreSQL 13.4-R1",
            "ValidUpgradeTarget": [],
            "SupportsLogExportsToCloudwatchLogs": false,
            "SupportsReadReplica": true,
            "SupportedFeatureNames": [
                "Lambda"
            ],
            "Status": "available",
            "SupportsParallelQuery": false,
            "SupportsGlobalDatabases": false,
            "SupportsBabelfish": false,
            "SupportsCertificateRotationWithoutRestart": true,
            "SupportedCACertificateIdentifiers": [
                "rds-ca-2019",
                "rds-ca-rsa2048-g1",
                "rds-ca-ecc384-g1",
                "rds-ca-rsa4096-g1"
            ]
        }
    ]
}
```

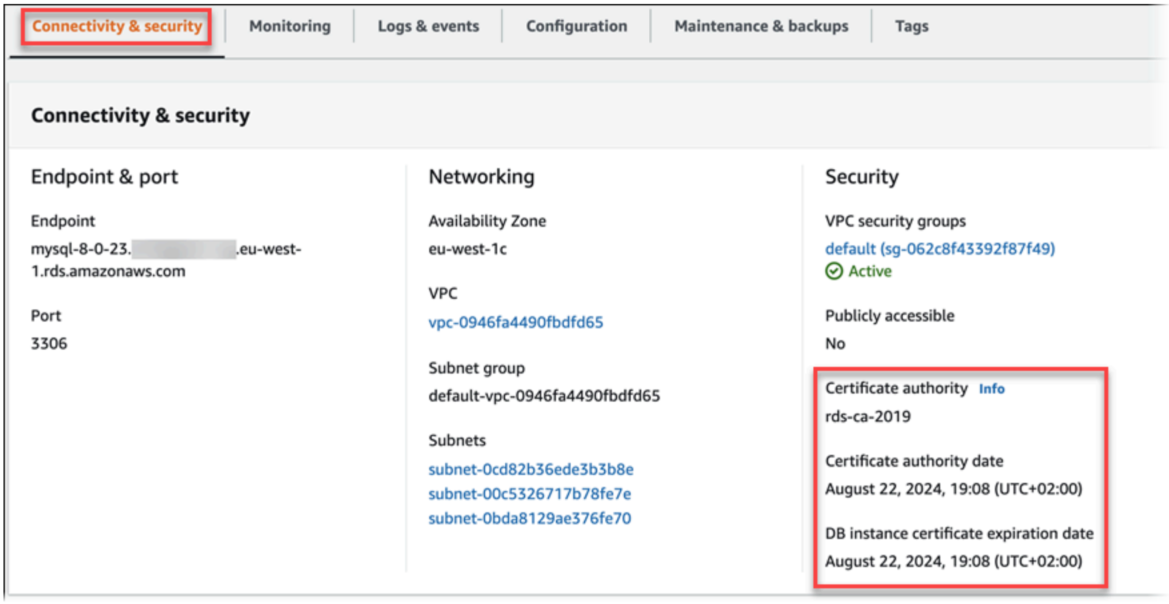
DB server certificate validities

The validity of DB server certificate depends on the DB engine and DB engine version. If the DB engine version supports rotating the certificate without restart, the validity of the DB server certificate is 1 year. Otherwise the validity is 3 years.

For more information about DB server certificate rotation, see Automatic server certificate rotation.

Viewing the CA for your DB instance

You can view the details about the CA for a database by viewing the **Connectivity & security** tab in the console, as in the following image.



Connectivity & security

Endpoint & port

Endpoint
mysql-8-0-23.[redacted].eu-west-1.rds.amazonaws.com

Port
3306

Networking

Availability Zone
eu-west-1c

VPC
vpc-0946fa4490fbdfd65

Subnet group
default-vpc-0946fa4490fbdfd65

Subnets
subnet-0cd82b36ede3b3b8e
subnet-00c5326717b78fe7e
subnet-0bda8129ae376fe70

Security

VPC security groups
default (sg-062c8f43392f87f49)
Active

Publicly accessible
No

Certificate authority Info
rds-ca-2019

Certificate authority date
August 22, 2024, 19:08 (UTC+02:00)

DB instance certificate expiration date
August 22, 2024, 19:08 (UTC+02:00)

If you're using the AWS CLI, you can view the details about the CA for a DB instance by using the describe-db-instances command.

To check the contents of your CA certificate bundle, use the following command:

```
keytool -printcert -v -file global-bundle.pem
```

Certificate bundles for all AWS Regions

To get a certificate bundle for all AWS Regions, download it from https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem.

The bundle contains both the `rds-ca-2019` intermediate and root certificates. The bundle also contains the `rds-ca-rsa2048-g1`, `rds-ca-rsa4096-g1`, and `rds-ca-ecc384-g1` root CA certificates. Your application trust store only needs to register the root CA certificate.

If your application is on Microsoft Windows and requires a PKCS7 file, you can download the PKCS7 certificate bundle from https://truststore.pki.rds.amazonaws.com/global/global-bundle.p7b.

> **Note**
>
> Amazon RDS Proxy and Aurora Serverless v1 use certificates from the AWS Certificate Manager (ACM). If you're using RDS Proxy, you don't need to download Amazon RDS certificates or update applications that use RDS Proxy connections. For more information, see Using TLS/SSL with RDS Proxy.
> If you're using Aurora Serverless v1, downloading Amazon RDS certificates isn't required. For more information, see Using TLS/SSL with Aurora Serverless v1.

Certificate bundles for specific AWS Regions

The bundle contains both the `rds-ca-2019` intermediate and root certificates. The bundle also contains the `rds-ca-rsa2048-g1`, `rds-ca-rsa4096-g1`, and `rds-ca-ecc384-g1` root CA certificates. Your application trust store only needs to register the root CA certificate.

To get a certificate bundle for an AWS Region, download it from the link for the AWS Region in the following table.

AWS Region	Certificate bundle (PEM)	Certificate bundle (PKCS7)
US East (N. Virginia)	us-east-1-bundle.pem	us-east-1-bundle.p7b
US East (Ohio)	us-east-2-bundle.pem	us-east-2-bundle.p7b
US West (N. California)	us-west-1-bundle.pem	us-west-1-bundle.p7b
US West (Oregon)	us-west-2-bundle.pem	us-west-2-bundle.p7b
Africa (Cape Town)	af-south-1-bundle.pem	af-south-1-bundle.p7b
Asia Pacific (Hong Kong)	ap-east-1-bundle.pem	ap-east-1-bundle.p7b

AWS Region	Certificate bundle (PEM)	Certificate bundle (PKCS7)
Asia Pacific (Hyderabad)	ap-south-2-bundle.pem	ap-south-2-bundle.p7b
Asia Pacific (Jakarta)	ap-southeast-3-bundle.pem	ap-southeast-3-bundle.p7b
Asia Pacific (Melbourne)	ap-southeast-4-bundle.pem	ap-southeast-4-bundle.p7b
Asia Pacific (Mumbai)	ap-south-1-bundle.pem	ap-south-1-bundle.p7b
Asia Pacific (Osaka)	ap-northeast-3-bundle.pem	ap-northeast-3-bundle.p7b
Asia Pacific (Tokyo)	ap-northeast-1-bundle.pem	ap-northeast-1-bundle.p7b
Asia Pacific (Seoul)	ap-northeast-2-bundle.pem	ap-northeast-2-bundle.p7b
Asia Pacific (Singapore)	ap-southeast-1-bundle.pem	ap-southeast-1-bundle.p7b
Asia Pacific (Sydney)	ap-southeast-2-bundle.pem	ap-southeast-2-bundle.p7b
Canada (Central)	ca-central-1-bundle.pem	ca-central-1-bundle.p7b
Canada West (Calgary)	ca-west-1-bundle.pem	ca-west-1-bundle.p7b
Europe (Frankfurt)	eu-central-1-bundle.pem	eu-central-1-bundle.p7b
Europe (Ireland)	eu-west-1-bundle.pem	eu-west-1-bundle.p7b
Europe (London)	eu-west-2-bundle.pem	eu-west-2-bundle.p7b
Europe (Milan)	eu-south-1-bundle.pem	eu-south-1-bundle.p7b
Europe (Paris)	eu-west-3-bundle.pem	eu-west-3-bundle.p7b
Europe (Spain)	eu-south-2-bundle.pem	eu-south-2-bundle.p7b
Europe (Stockholm)	eu-north-1-bundle.pem	eu-north-1-bundle.p7b
Europe (Zurich)	eu-central-2-bundle.pem	eu-central-2-bundle.p7b
Israel (Tel Aviv)	il-central-1-bundle.pem	il-central-1-bundle.p7b

AWS Region	Certificate bundle (PEM)	Certificate bundle (PKCS7)
Middle East (Bahrain)	me-south-1-bundle.pem	me-south-1-bundle.p7b
Middle East (UAE)	me-central-1-bundle.pem	me-central-1-bundle.p7b
South America (São Paulo)	sa-east-1-bundle.pem	sa-east-1-bundle.p7b

AWS GovCloud (US) certificates

To get a certificate bundle that contains both the intermediate and root certificates for the AWS GovCloud (US) Regions, download from https://truststore.pki.us-gov-west-1.rds.amazonaws.com/global/global-bundle.pem.

If your application is on Microsoft Windows and requires a PKCS7 file, you can download the PKCS7 certificate bundle from https://truststore.pki.us-gov-west-1.rds.amazonaws.com/global/global-bundle.p7b.

The bundle contains both the `rds-ca-2019` intermediate and root certificates. The bundle also contains the `rds-ca-rsa2048-g1`, `rds-ca-rsa4096-g1`, and `rds-ca-ecc384-g1` root CA certificates. Your application trust store only needs to register the root CA certificate.

To get a certificate bundle for an AWS GovCloud (US) Region, download from the link for the AWS GovCloud (US) Region in the following table.

AWS GovCloud (US) Region	Certificate bundle (PEM)	Certificate bundle (PKCS7)
AWS GovCloud (US-East)	us-gov-east-1-bundle.pem	us-gov-east-1-bundle.p7b
AWS GovCloud (US-West)	us-gov-west-1-bundle.pem	us-gov-west-1-bundle.p7b

Rotating your SSL/TLS certificate

Amazon RDS Certificate Authority certificates rds-ca-2019 are set to expire in August, 2024. If you use or plan to use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) with certificate verification to connect to your RDS DB instances , consider using one of the new CA certificates rds-ca-rsa2048-g1, rds-ca-rsa4096-g1 or rds-ca-ecc384-g1. If you currently do not use SSL/TLS with certificate verification, you might still have an expired CA certificate and must update them to a new CA certificate if you plan to use SSL/TLS with certificate verification to connect to your RDS databases.

Follow these instructions to complete your updates. Before you update your DB instances to use the new CA certificate, make sure that you update your clients or applications connecting to your RDS databases.

Amazon RDS provides new CA certificates as an AWS security best practice. For information about the new certificates and the supported AWS Regions, see Using SSL/TLS to encrypt a connection to a DB cluster.

> **Note**
>
> Amazon RDS Proxy and Aurora Serverless v1 use certificates from the AWS Certificate Manager (ACM). If you're using RDS Proxy, when you rotate your SSL/TLS certificate, you don't need to update applications that use RDS Proxy connections. For more information, see Using TLS/SSL with RDS Proxy.
> If you're using Aurora Serverless v1, downloading Amazon RDS certificates isn't required. For more information, see Using TLS/SSL with Aurora Serverless v1.

> **Note**
>
> If you're using a Go version 1.15 application with a DB instance that was created or updated to the rds-ca-2019 certificate prior to July 28, 2020, you must update the certificate again. Update the certificate to rds-ca-rsa2048-g1, rds-ca-rsa4096-g1, or rds-ca-ecc384-g1 depending on your engine. Run the `modify-db-instance` command , using the new CA certificate identifier. You can find the CAs that are available for a specific DB engine and DB engine version using the `describe-db-engine-versions` command.
> If you created your database or updated its certificate after July 28, 2020, no action is required. For more information, see Go GitHub issue #39568.

Topics

- Updating your CA certificate by modifying your DB instance
- Updating your CA certificate by applying maintenance
- Automatic server certificate rotation

- Sample script for importing certificates into your trust store

Updating your CA certificate by modifying your DB instance

The following example updates your CA certificate from *rds-ca-2019* to *rds-ca-rsa2048-g1*. You can choose a different certificate. For more information, see Certificate authorities.

Update your application trust store to reduce any down time associated with updating your CA certificate. For more information about restarts associated with CA certificate rotation, see Automatic server certificate rotation.

To update your CA certificate by modifying your DB instance

1. Download the new SSL/TLS certificate as described in Using SSL/TLS to encrypt a connection to a DB cluster.
2. Update your applications to use the new SSL/TLS certificate.

   The methods for updating applications for new SSL/TLS certificates depend on your specific applications. Work with your application developers to update the SSL/TLS certificates for your applications.

   For information about checking for SSL/TLS connections and updating applications for each DB engine, see the following topics:

   - Updating applications to connect to Aurora MySQL DB clusters using new TLS certificates
   - Updating applications to connect to Aurora PostgreSQL DB clusters using new SSL/TLS certificates

   For a sample script that updates a trust store for a Linux operating system, see Sample script for importing certificates into your trust store.

   > **Note**
   >
   > The certificate bundle contains certificates for both the old and new CA, so you can upgrade your application safely and maintain connectivity during the transition period. If you are using the AWS Database Migration Service to migrate a database to a DB cluster, we recommend using the certificate bundle to ensure connectivity during the migration.

3. Modify the DB instance to change the CA from **rds-ca-2019** to **rds-ca-rsa2048-g1**. To check if your database requires a restart to update the CA certificates, use the describe-db-engine-versions command and check the `SupportsCertificateRotationWithoutRestart` flag.

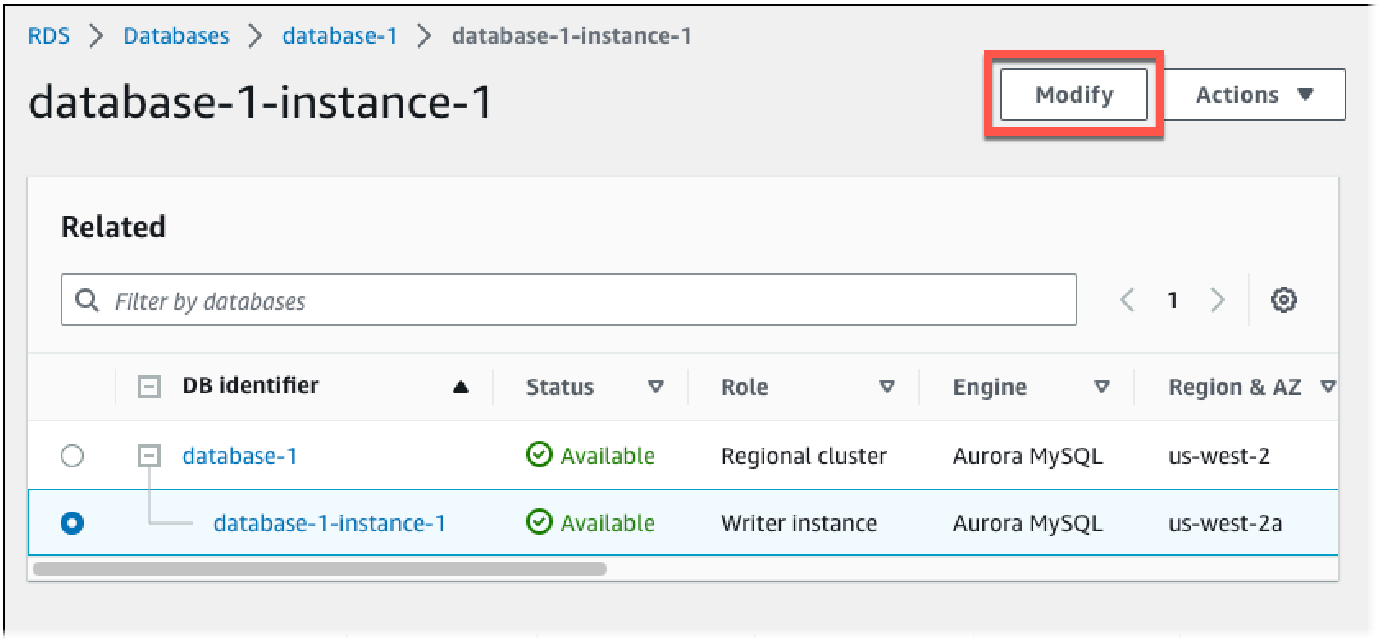
   > **Note**
   >
   > Reboot your Babelfish cluster after modifying to update the CA certificate.

   > **Important**
   >
   > If you are experiencing connectivity issues after certificate expiry, use the apply immediately option by specifying **Apply immediately** in the console or by specifying the `--apply-immediately` option using the AWS CLI. By default, this operation is scheduled to run during your next maintenance window.
   > To set an override for your cluster CA that's different from the default RDS CA, use the modify-certificates CLI command.

You can use the AWS Management Console or the AWS CLI to change the CA certificate from **rds-ca-2019** to **rds-ca-rsa2048-g1** for a DB instance .

Console

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**, and then choose the DB instance that you want to modify.
3. Choose **Modify**.

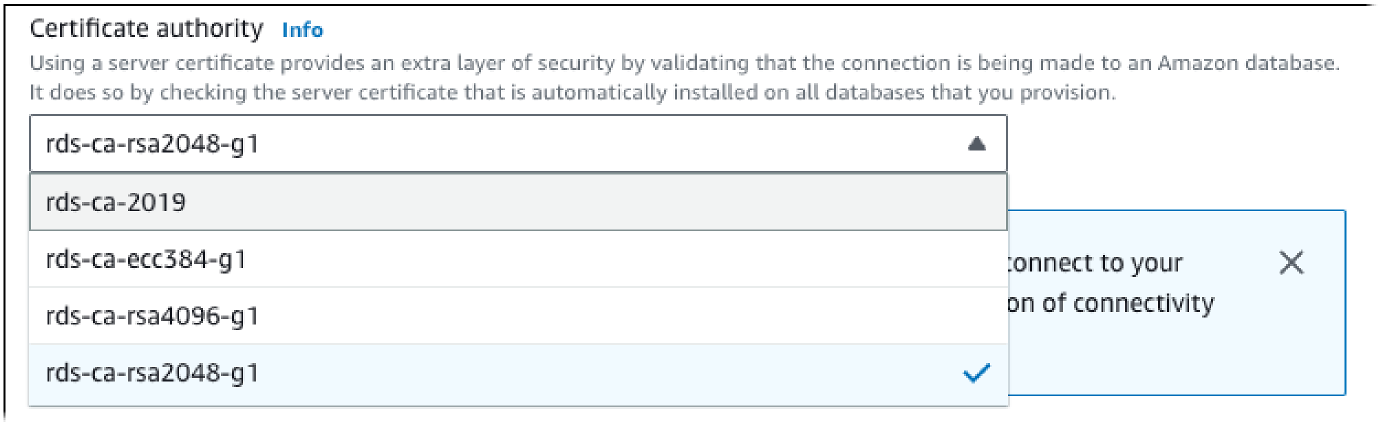


database-1-instance-1

Related

	DB identifier	Status	Role	Engine	Region & AZ
○	database-1	Available	Regional cluster	Aurora MySQL	us-west-2
●	database-1-instance-1	Available	Writer instance	Aurora MySQL	us-west-2a

4. In the **Connectivity** section, choose **rds-ca-rsa2048-g1**.



Certificate authority

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

5. Choose **Continue** and check the summary of modifications.

6. To apply the changes immediately, choose **Apply immediately**.

7. On the confirmation page, review your changes. If they are correct, choose **Modify DB Instance** to save your changes.

> **Important**
>
> When you schedule this operation, make sure that you have updated your client-side trust store beforehand.

Or choose **Back** to edit your changes or **Cancel** to cancel your changes.

AWS CLI

To use the AWS CLI to change the CA from **rds-ca-2019** to **rds-ca-rsa2048-g1** for a DB instance , call the modify-db-instance or modify-db-cluster command. Specify the DB instance identifier and the `--ca-certificate-identifier` option.

Use the `--apply-immediately` parameter to apply the update immediately. By default, this operation is scheduled to run during your next maintenance window.

> **Important**
>
> When you schedule this operation, make sure that you have updated your client-side trust store beforehand.

Example

The following example modifies `mydbinstance` by setting the CA certificate to `rds-ca-rsa2048-g1`.

For Linux, macOS, or Unix:

```
aws rds modify-db-instance \
    --db-instance-identifier mydbinstance \
    --ca-certificate-identifier rds-ca-rsa2048-g1
```

For Windows:

```
aws rds modify-db-instance ^
    --db-instance-identifier mydbinstance ^
    --ca-certificate-identifier rds-ca-rsa2048-g1
```

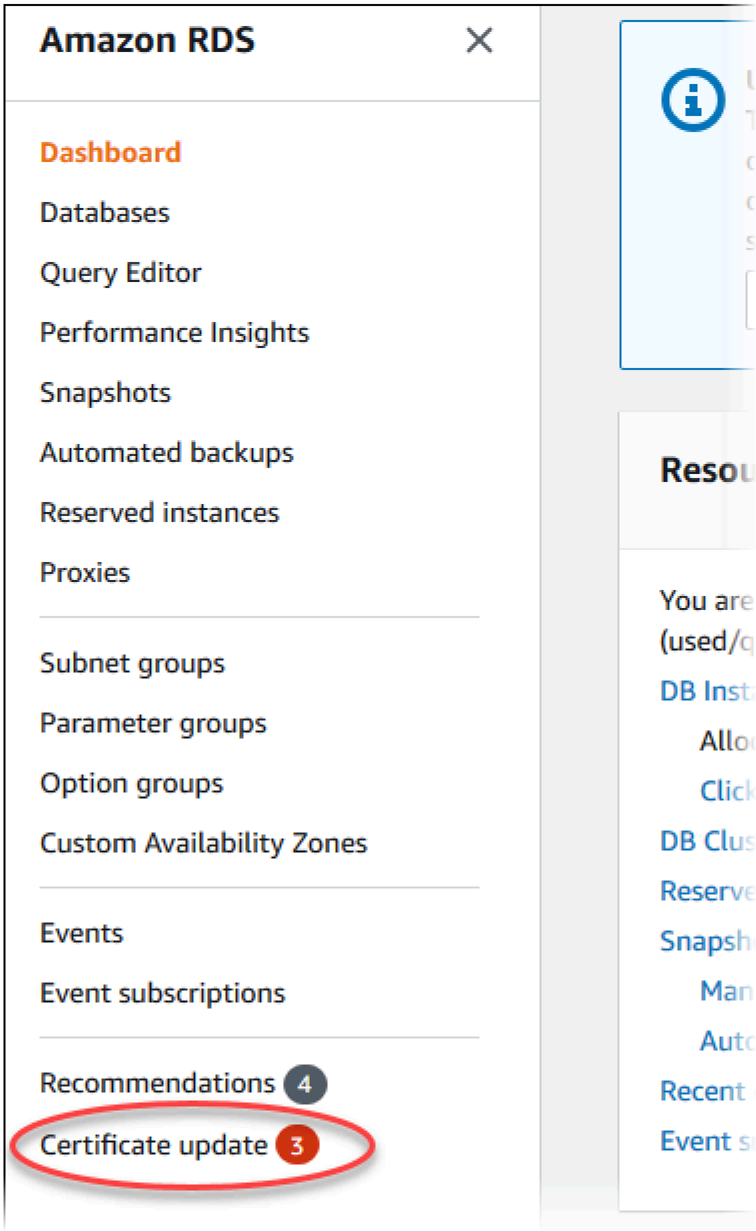
> **Note**
>
> If your instance requires reboot, you can use the modify-db-instance CLI command and specify the `--no-certificate-rotation-restart` option.

Updating your CA certificate by applying maintenance

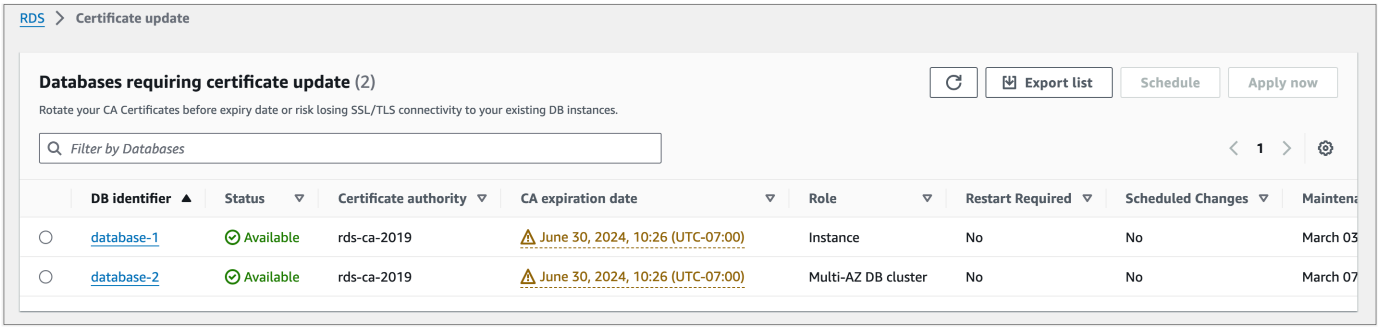
Perform the following steps to update your CA certificate by applying maintenance.

To update your CA certificate by applying maintenance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Certificate update**.



The **Databases requiring certificate update** page appears.



Databases requiring certificate update (2)

Rotate your CA Certificates before expiry date or risk losing SSL/TLS connectivity to your existing DB instances.

	DB identifier ▲	Status	Certificate authority	CA expiration date	Role	Restart Required	Scheduled Changes	Maintena
○	database-1	Available	rds-ca-2019	June 30, 2024, 10:26 (UTC-07:00)	Instance	No	No	March 03
○	database-2	Available	rds-ca-2019	June 30, 2024, 10:26 (UTC-07:00)	Multi-AZ DB cluster	No	No	March 07

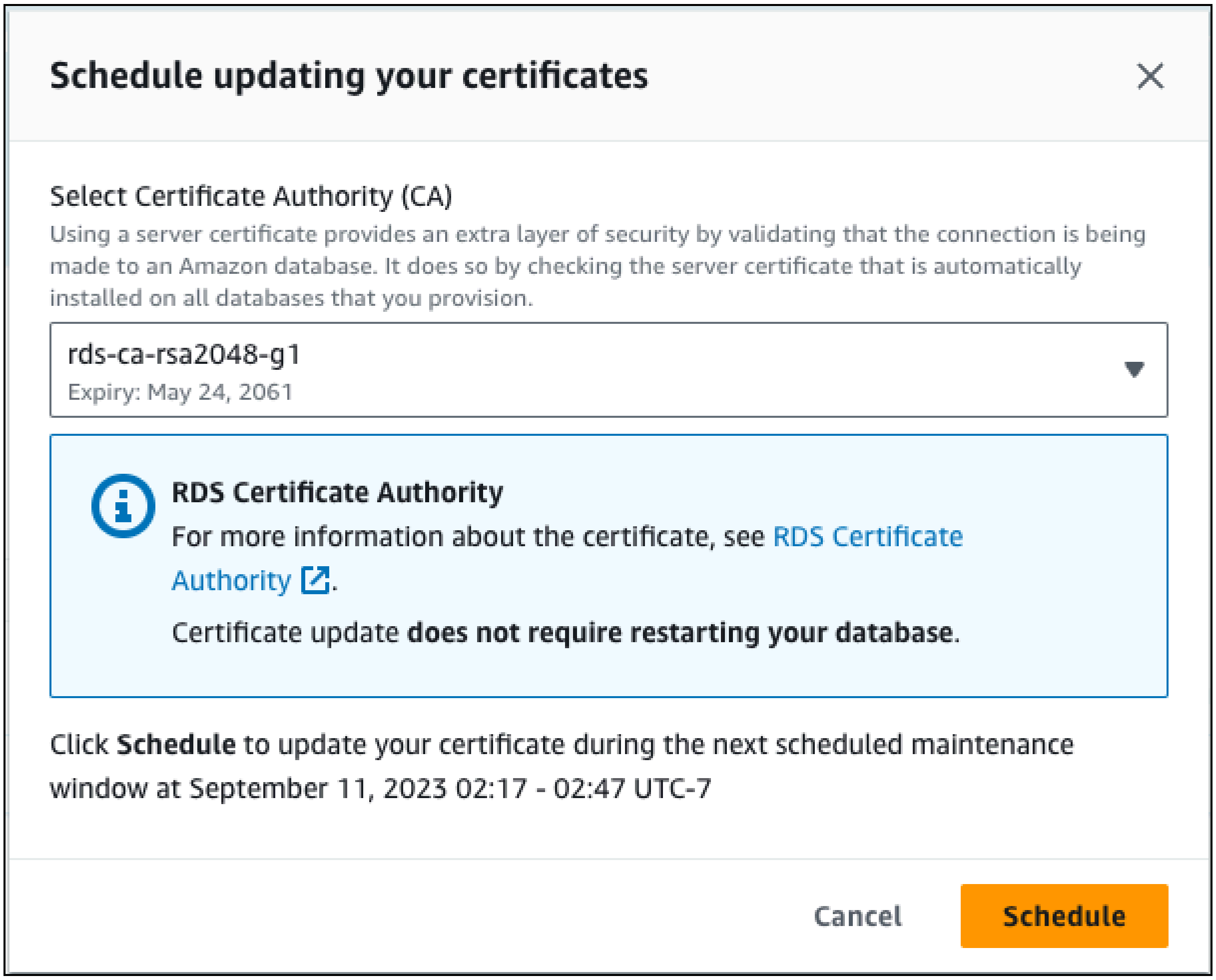
> **Note**
>
> This page only shows the DB instances for the current AWS Region. If you have databases in more than one AWS Region, check this page in each AWS Region to see all DB instances with old SSL/TLS certificates.

3. Choose the DB instance that you want to update.

   You can schedule the certificate rotation for your next maintenance window by choosing **Schedule**. Apply the rotation immediately by choosing **Apply now**.

> **Important**
>
> If you experience connectivity issues after certificate expiry, use the **Apply now** option.

4. a. If you choose **Schedule**, you are prompted to confirm the CA certificate rotation. This prompt also states the scheduled window for your update.



Schedule updating your certificates

Select Certificate Authority (CA)

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-rsa2048-g1
Expiry: May 24, 2061

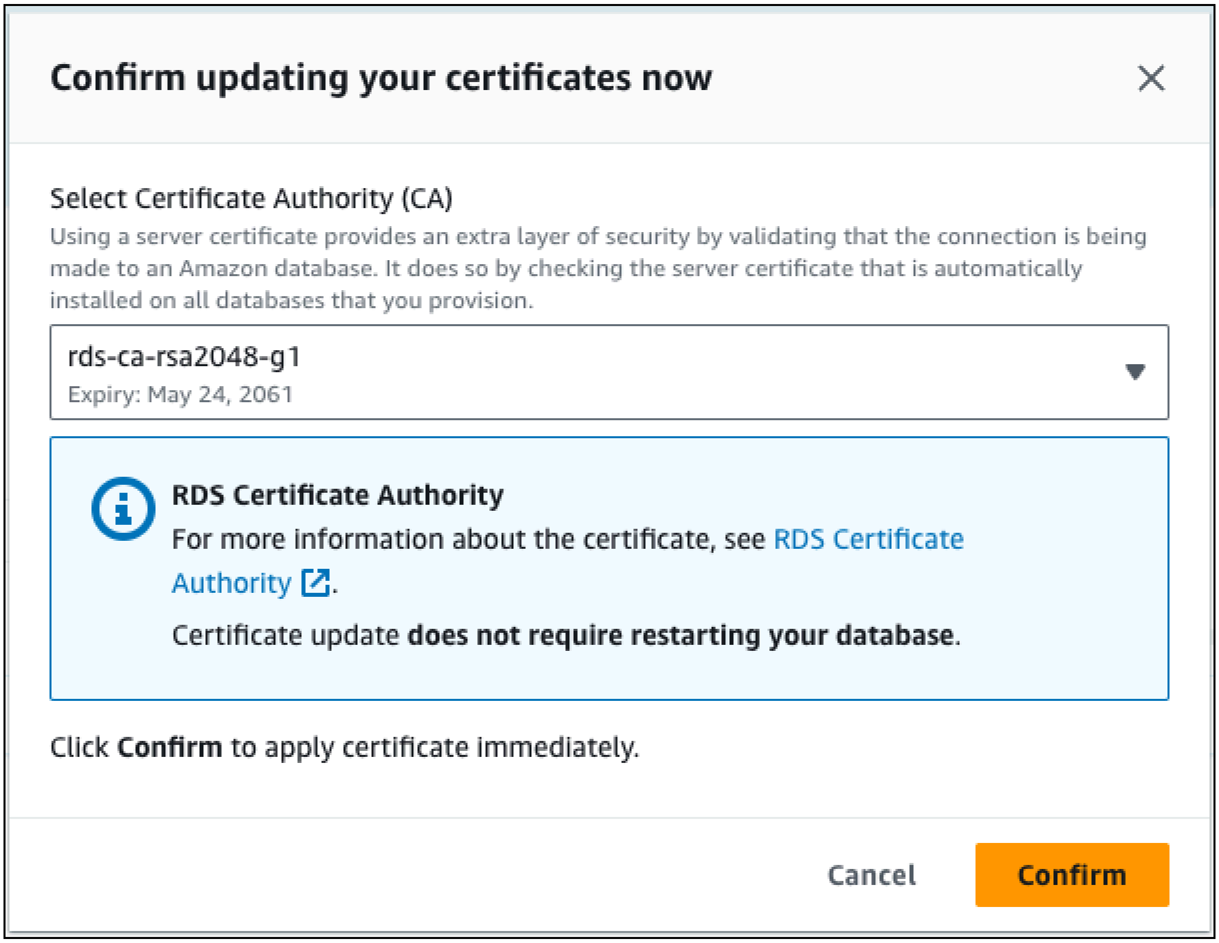
RDS Certificate Authority

For more information about the certificate, see RDS Certificate Authority.

Certificate update **does not require restarting your database**.

Click **Schedule** to update your certificate during the next scheduled maintenance window at September 11, 2023 02:17 - 02:47 UTC-7

b. If you choose **Apply now**, you are prompted to confirm the CA certificate rotation.



Confirm updating your certificates now

Select Certificate Authority (CA)

Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-rsa2048-g1
Expiry: May 24, 2061

RDS Certificate Authority

For more information about the certificate, see RDS Certificate Authority.

Certificate update **does not require restarting your database**.

Click **Confirm** to apply certificate immediately.

> **Important**
>
> Before scheduling the CA certificate rotation on your database, update any client applications that use SSL/TLS and the server certificate to connect. These updates are specific to your DB engine. After you have updated these client applications, you can confirm the CA certificate rotation.

To continue, choose the check box, and then choose **Confirm**.

5. Repeat steps 3 and 4 for each DB instance that you want to update.

Automatic server certificate rotation

If your CA supports automatic server certificate rotation, RDS automatically handles the rotation of the DB server certificate. RDS uses the same root CA for this automatic rotation, so you don't need to download a new CA bundle. See Certificate authorities.

The rotation and validity of your DB server certificate depend on your DB engine:

- If your DB engine supports rotation without restart, RDS automatically rotates the DB server certificate without requiring any action from you. RDS attempts to rotate your DB server certificate in your preferred maintenance window at the DB server certificate half life. The new DB server certificate is valid for 12 months.
- If your DB engine doesn't support rotation without restart, RDS notifies you about a maintenance event at least 6 months before the DB server certificate expires. The new DB server certificate is valid for 36 months.

Use the describe-db-engine-versions command and inspect the `SupportsCertificateRotationWithoutRestart` flag to identify whether the DB engine version supports rotating the certificate without restart. For more information, see Setting the CA for your database.

Sample script for importing certificates into your trust store

The following are sample shell scripts that import the certificate bundle into a trust store.

Each sample shell script uses keytool, which is part of the Java Development Kit (JDK). For information about installing the JDK, see JDK Installation Guide.

Topics

Sample script for importing certificates on Linux

The following is a sample shell script that imports the certificate bundle into a trust store on a Linux operating system.

```
mydir=tmp/certs
```

```
if [ ! -e "${mydir}" ]
then
mkdir -p "${mydir}"
fi

truststore=${mydir}/rds-truststore.jks
storepassword=changeit

curl -sS "https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem" >
 ${mydir}/global-bundle.pem
awk 'split_after == 1 {n++;split_after=0} /-----END CERTIFICATE-----/ {split_after=1}
{print > "rds-ca-" n+1 ".pem"}' < ${mydir}/global-bundle.pem

for CERT in rds-ca-*; do
  alias=$(openssl x509 -noout -text -in $CERT | perl -ne 'next unless /Subject:/;
 s/.*(CN=|CN = )//; print')
  echo "Importing $alias"
  keytool -import -file ${CERT} -alias "${alias}" -storepass ${storepassword} -keystore
 ${truststore} -noprompt
  rm $CERT
done

rm ${mydir}/global-bundle.pem

echo "Trust store content is: "

keytool -list -v -keystore "$truststore" -storepass ${storepassword} | grep Alias | cut
 -d " " -f3- | while read alias
do
   expiry=`keytool -list -v -keystore "$truststore" -storepass ${storepassword} -alias
 "${alias}" | grep Valid | perl -ne 'if(/until: (.*?)\n/) { print "$1\n"; }'`
   echo " Certificate ${alias} expires in '$expiry'"
done
```

Sample script for importing certificates on macOS

The following is a sample shell script that imports the certificate bundle into a trust store on macOS.

```
mydir=tmp/certs
if [ ! -e "${mydir}" ]
```

```
then
mkdir -p "${mydir}"
fi

truststore=${mydir}/rds-truststore.jks
storepassword=changeit

curl -sS "https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem" >
 ${mydir}/global-bundle.pem
split -p "-----BEGIN CERTIFICATE-----" ${mydir}/global-bundle.pem rds-ca-

for CERT in rds-ca-*; do
  alias=$(openssl x509 -noout -text -in $CERT | perl -ne 'next unless /Subject:/;
 s/.*(CN=|CN = )//; print')
  echo "Importing $alias"
  keytool -import -file ${CERT} -alias "${alias}" -storepass ${storepassword} -keystore
 ${truststore} -noprompt
  rm $CERT
done

rm ${mydir}/global-bundle.pem

echo "Trust store content is: "

keytool -list -v -keystore "$truststore" -storepass ${storepassword} | grep Alias | cut
 -d " " -f3- | while read alias
do
   expiry=`keytool -list -v -keystore "$truststore" -storepass ${storepassword} -alias
 "${alias}" | grep Valid | perl -ne 'if(/until: (.*?)\n/) { print "$1\n"; }'`
   echo " Certificate ${alias} expires in '$expiry'"
done
```

Internetwork traffic privacy

Connections are protected both between Amazon Aurora and on-premises applications and between Amazon Aurora and other AWS resources within the same AWS Region.

Traffic between service and on-premises clients and applications

You have two connectivity options between your private network and AWS:

- An AWS Site-to-Site VPN connection. For more information, see What is AWS Site-to-Site VPN?

- An AWS Direct Connect connection. For more information, see What is AWS Direct Connect?

You get access to Amazon Aurora through the network by using AWS-published API operations. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Identity and access management for Amazon Aurora

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon RDS resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in Amazon Aurora.

Service user – If you use the Aurora service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Aurora features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Aurora, see Troubleshooting Amazon Aurora identity and access.

Service administrator – If you're in charge of Aurora resources at your company, you probably have full access to Aurora. It's your job to determine which Aurora features and resources your employees should access. You must then submit requests to your administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Aurora, see How Amazon Aurora works with IAM.

Administrator – If you're an administrator, you might want to learn details about how you can write policies to manage access to Aurora. To view example Aurora identity-based policies that you can use in IAM, see Identity-based policy examples for Amazon Aurora.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signing AWS API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account.

We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

You can authenticate to your DB cluster using IAM database authentication.

IAM database authentication works with Aurora. For more information about authenticating to your DB cluster using IAM, see IAM database authentication.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to a user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary user permissions** – A user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the

principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when an entity (root user, user, or IAM role) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

An administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (permission set or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password.

To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as a permission set or role. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single permission set or role. Managed policies are standalone policies that you can attach to multiple permission sets and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

For information about AWS managed policies that are specific to Amazon Aurora, see AWS managed policies for Amazon RDS.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (permission set or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the permission set or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the permission sets or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Aurora works with IAM

Before you use IAM to manage access to Amazon Aurora, you should understand what IAM features are available to use with Aurora.

IAM features you can use with Amazon Aurora

IAM feature	Amazon Aurora support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes

IAM feature	Amazon Aurora support
ACLs	No
Attribute-based access control (ABAC) (tags in policies)	Yes
Temporary credentials	Yes
Forward access sessions	Yes
Service roles	Yes
Service-linked roles	Yes

To get a high-level view of how Amazon Aurora and other AWS services work with IAM, see AWS services that work with IAM in the *IAM User Guide*.

Topics

- Aurora identity-based policies
- Resource-based policies within Aurora
- Policy actions for Aurora
- Policy resources for Aurora
- Policy condition keys for Aurora
- Access control lists (ACLs) in Aurora
- Attribute-based access control (ABAC) in policies with Aurora tags
- Using temporary credentials with Aurora
- Forward access sessions for Aurora
- Service roles for Aurora
- Service-linked roles for Aurora

Aurora identity-based policies

Supports identity-based policies	Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Aurora

To view examples of Aurora identity-based policies, see Identity-based policy examples for Amazon Aurora.

Resource-based policies within Aurora

Supports resource-based policies	No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Aurora

Supports policy actions	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in Aurora use the following prefix before the action: `rds:`. For example, to grant someone permission to describe DB instances with the Amazon RDS `DescribeDBInstances` API operation, you include the `rds:DescribeDBInstances` action in their policy. Policy statements must include either an `Action` or `NotAction` element. Aurora defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows.

```
"Action": [
      "rds:action1",
      "rds:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action.

```
"Action": "rds:Describe*"
```

To see a list of Aurora actions, see Actions Defined by Amazon RDS in the *Service Authorization Reference*.

Policy resources for Aurora

Supports policy resources	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

The DB instance resource has the following Amazon Resource Name (ARN).

```
arn:${Partition}:rds:${Region}:${Account}:{ResourceType}/${Resource}
```

For more information about the format of ARNs, see Amazon Resource Names (ARNs) and AWS service namespaces.

For example, to specify the `dbtest` DB instance in your statement, use the following ARN.

```
"Resource": "arn:aws:rds:us-west-2:123456789012:db:dbtest"
```

To specify all DB instances that belong to a specific account, use the wildcard (*).

```
"Resource": "arn:aws:rds:us-east-1:123456789012:db:*"
```

Some RDS API operations, such as those for creating resources, can't be performed on a specific resource. In those cases, use the wildcard (*).

```
"Resource": "*"
```

Many Amazon RDS API operations involve multiple resources. For example, `CreateDBInstance` creates a DB instance. You can specify that an user must use a specific security group and parameter group when creating a DB instance. To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
      "resource1",
      "resource2"
```

To see a list of Aurora resource types and their ARNs, see Resources Defined by Amazon RDS in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Amazon RDS.

Policy condition keys for Aurora

Supports service-specific policy condition keys	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Aurora defines its own set of condition keys and also supports using some global condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

All RDS API operations support the `aws:RequestedRegion` condition key.

To see a list of Aurora condition keys, see [Condition Keys for Amazon RDS]() in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions Defined by Amazon RDS]().

Access control lists (ACLs) in Aurora

Supports access control lists (ACLs)	No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) in policies with Aurora tags

Supports attribute-based access control (ABAC) tags in policies	Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element]() of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see What is ABAC? in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

For more information about tagging Aurora resources, see Specifying conditions: Using custom tags. To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Grant permission for actions on a resource with a specific tag with two different values.

Using temporary credentials with Aurora

Supports temporary credentials	Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switching to a role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Forward access sessions for Aurora

Supports forward access sessions	Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Aurora

Supports service roles	Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

> ⚠ **Warning**
>
> Changing the permissions for a service role might break Aurora functionality. Edit service roles only when Aurora provides guidance to do so.

Service-linked roles for Aurora

Supports service-linked roles	Yes

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about using Aurora service-linked roles, see Using service-linked roles for Amazon Aurora.

Identity-based policy examples for Amazon Aurora

By default, permission sets and roles don't have permission to create or modify Aurora resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An administrator must create IAM policies that grant permission sets and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the permission sets or roles that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating policies on the JSON tab in the *IAM User Guide*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Amazon RDS resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.

- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see IAM Access Analyzer policy validation in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Aurora console

To access the Amazon Aurora console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon Aurora resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

To ensure that those entities can still use the Aurora console, also attach the following AWS managed policy to the entities.

```
AmazonRDSReadOnlyAccess
```

For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Allow a user to create DB instances in an AWS account

The following is an example policy that allows the user with the ID 123456789012 to create DB instances for your AWS account. The policy requires that the name of the new DB instance begin with test. The new DB instance must also use the MySQL database engine and the db.t2.micro DB instance class. In addition, the new DB instance must use an option group and a DB parameter group that starts with default, and it must use the default subnet group.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowCreateDBInstanceOnly",
         "Effect": "Allow",
         "Action": [
            "rds:CreateDBInstance"
         ],
         "Resource": [
            "arn:aws:rds:*:123456789012:db:test*",
            "arn:aws:rds:*:123456789012:og:default*",
            "arn:aws:rds:*:123456789012:pg:default*",
            "arn:aws:rds:*:123456789012:subgrp:default"
         ],
         "Condition": {
            "StringEquals": {
               "rds:DatabaseEngine": "mysql",
               "rds:DatabaseClass": "db.t2.micro"
            }
         }
      }
   ]
}
```

The policy includes a single statement that specifies the following permissions for the user:

- The policy allows the user to create a DB instance using the CreateDBInstance API operation (this also applies to the create-db-instance AWS CLI command and the AWS Management Console).
- The Resource element specifies that the user can perform actions on or with resources. You specify resources using an Amazon Resources Name (ARN). This ARN includes the name of the service that the resource belongs to (rds), the AWS Region (* indicates any region in this example), the AWS account number (123456789012 is the account number in this example),

  and the type of resource. For more information about creating ARNs, see [Working with Amazon Resource Names (ARNs) in Amazon RDS]().

  The `Resource` element in the example specifies the following policy constraints on resources for the user:

  - The DB instance identifier for the new DB instance must begin with `test` (for example, `testCustomerData1`, `test-region2-data`).
  - The option group for the new DB instance must begin with `default`.
  - The DB parameter group for the new DB instance must begin with `default`.
  - The subnet group for the new DB instance must be the `default` subnet group.

- The `Condition` element specifies that the DB engine must be MySQL and the DB instance class must be `db.t2.micro`. The `Condition` element specifies the conditions when a policy should take effect. You can add additional permissions or restrictions by using the `Condition` element. For more information about specifying conditions, see [Policy condition keys for Aurora](). This example specifies the `rds:DatabaseEngine` and `rds:DatabaseClass` conditions. For information about the valid condition values for `rds:DatabaseEngine`, see the list under the `Engine` parameter in [CreateDBInstance](). For information about the valid condition values for `rds:DatabaseClass`, see [Supported DB engines for DB instance classes]().

The policy doesn't specify the `Principal` element because in an identity-based policy you don't specify the principal who gets the permission. When you attach policy to a user, the user is the implicit principal. When you attach a permission policy to an IAM role, the principal identified in the role's trust policy gets the permissions.

To see a list of Aurora actions, see [Actions Defined by Amazon RDS]() in the *Service Authorization Reference*.

Permissions required to use the console

For a user to work with the console, that user must have a minimum set of permissions. These permissions allow the user to describe the Amazon Aurora resources for their AWS account and to provide other related information, including Amazon EC2 security and network information.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console doesn't function as intended for users with that IAM policy. To ensure that those users can still use the console, also attach the `AmazonRDSReadOnlyAccess` managed policy to the user, as described in [Managing access using policies]().

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the Amazon RDS API.

The following policy grants full access to all Amazon Aurora resources for the root AWS account:

```
AmazonRDSFullAccess
```

Allow a user to perform any describe action on any RDS resource

The following permissions policy grants permissions to a user to run all of the actions that begin with `Describe`. These actions show information about an RDS resource, such as a DB instance. The wildcard character (*) in the `Resource` element indicates that the actions are allowed for all Amazon Aurora resources owned by the account.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowRDSDescribe",
         "Effect": "Allow",
         "Action": "rds:Describe*",
         "Resource": "*"
      }
   ]
}
```

Allow a user to create a DB instance that uses the specified DB parameter group and subnet group

The following permissions policy grants permissions to allow a user to only create a DB instance that must use the `mydbpg` DB parameter group and the `mydbsubnetgroup` DB subnet group.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": "rds:CreateDBInstance",
         "Resource": [
            "arn:aws:rds:*:*:pg:mydbpg",
            "arn:aws:rds:*:*:subgrp:mydbsubnetgroup"
         ]
      }
   ]
}
```

Grant permission for actions on a resource with a specific tag with two different values

You can use conditions in your identity-based policy to control access to Aurora resources based on tags. The following policy allows permission to perform the `CreateDBSnapshot` API operation on DB instances with either the `stage` tag set to `development` or `test`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAnySnapshotName",
         "Effect":"Allow",
         "Action":[
            "rds:CreateDBSnapshot"
         ],
         "Resource":"arn:aws:rds:*:123456789012:snapshot:*"
      },
      {
         "Sid":"AllowDevTestToCreateSnapshot",
         "Effect":"Allow",
         "Action":[
            "rds:CreateDBSnapshot"
         ],
         "Resource":"arn:aws:rds:*:123456789012:db:*",
         "Condition":{
            "StringEquals":{
                "rds:db-tag/stage":[
                  "development",
                  "test"
               ]
            }
         }
      }
   ]
}
```

The following policy allows permission to perform the `ModifyDBInstance` API operation on DB instances with either the `stage` tag set to `development` or `test`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowChangingParameterOptionSecurityGroups",
         "Effect":"Allow",
         "Action":[
            "rds:ModifyDBInstance"
         ],
         "Resource": [
            "arn:aws:rds:*:123456789012:pg:*",
            "arn:aws:rds:*:123456789012:secgrp:*",
            "arn:aws:rds:*:123456789012:og:*"
         ]
      },
      {
         "Sid":"AllowDevTestToModifyInstance",
         "Effect":"Allow",
         "Action":[
            "rds:ModifyDBInstance"
         ],
         "Resource":"arn:aws:rds:*:123456789012:db:*",
         "Condition":{
            "StringEquals":{
                "rds:db-tag/stage":[
                  "development",
                  "test"
               ]
            }
         }
      }
   ]
}
```

Prevent a user from deleting a DB instance

The following permissions policy grants permissions to prevent a user from deleting a specific DB instance. For example, you might want to deny the ability to delete your production DB instances to any user that is not an administrator.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "DenyDelete1",
         "Effect": "Deny",
         "Action": "rds:DeleteDBInstance",
         "Resource": "arn:aws:rds:us-west-2:123456789012:db:my-mysql-instance"
      }
   ]
}
```

Deny all access to a resource

You can explicitly deny access to a resource. Deny policies take precedence over allow policies. The following policy explicitly denies a user the ability to manage a resource:

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Deny",
         "Action": "rds:*",
         "Resource": "arn:aws:rds:us-east-1:123456789012:db:mydb"
      }
   ]
}
```

Example policies: Using condition keys

Following are examples of how you can use condition keys in Amazon Aurora IAM permissions policies.

Example 1: Grant permission to create a DB instance that uses a specific DB engine and isn't MultiAZ

The following policy uses an RDS condition key and allows a user to create only DB instances that use the MySQL database engine and don't use MultiAZ. The `Condition` element indicates the requirement that the database engine is MySQL.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowMySQLCreate",
         "Effect": "Allow",
         "Action": "rds:CreateDBInstance",
         "Resource": "*",
         "Condition": {
            "StringEquals": {
               "rds:DatabaseEngine": "mysql"
            },
            "Bool": {
               "rds:MultiAz": false
            }
         }
      }
   ]
}
```

Example 2: Explicitly deny permission to create DB instances for certain DB instance classes and create DB instances that use Provisioned IOPS

The following policy explicitly denies permission to create DB instances that use the DB instance classes `r3.8xlarge` and `m4.10xlarge`, which are the largest and most expensive DB instance classes. This policy also prevents users from creating DB instances that use Provisioned IOPS, which incurs an additional cost.

Explicitly denying permission supersedes any other permissions granted. This ensures that identities to not accidentally get permission that you never want to grant.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "DenyLargeCreate",
         "Effect": "Deny",
         "Action": "rds:CreateDBInstance",
         "Resource": "*",
         "Condition": {
            "StringEquals": {
               "rds:DatabaseClass": [
                  "db.r3.8xlarge",
                  "db.m4.10xlarge"
               ]
            }
         }
      },
      {
         "Sid": "DenyPIOPSCreate",
         "Effect": "Deny",
         "Action": "rds:CreateDBInstance",
         "Resource": "*",
         "Condition": {
            "NumericNotEquals": {
               "rds:Piops": "0"
            }
         }
      }
   ]
}
```

Example 3: Limit the set of tag keys and values that can be used to tag a resource

The following policy uses an RDS condition key and allows the addition of a tag with the key `stage` to be added to a resource with the values `test`, `qa`, and `production`.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "rds:AddTagsToResource",
            "rds:RemoveTagsFromResource"
         ],
         "Resource": "*",
         "Condition": {
            "streq": {
               "rds:req-tag/stage": [
                  "test",
                  "qa",
                  "production"
               ]
            }
         }
      }
   ]
}
```

Specifying conditions: Using custom tags

Amazon Aurora supports specifying conditions in an IAM policy using custom tags.

For example, suppose that you add a tag named `environment` to your DB instances with values such as `beta`, `staging`, `production`, and so on. If you do, you can create a policy that restricts certain users to DB instances based on the `environment` tag value.

> **Note**
>
> Custom tag identifiers are case-sensitive.

The following table lists the RDS tag identifiers that you can use in a `Condition` element.

RDS tag identifier	Applies to
`db-tag`	DB instances, including read replicas
`snapshot-tag`	DB snapshots
`ri-tag`	Reserved DB instances
`og-tag`	DB option groups
`pg-tag`	DB parameter groups
`subgrp-tag`	DB subnet groups

RDS tag identifier	Applies to
`es-tag`	Event subscriptions
`cluster-tag`	DB clusters
`cluster-pg-tag`	DB cluster parameter groups
`cluster-snapshot-tag`	DB cluster snapshots

The syntax for a custom tag condition is as follows:

```
"Condition":{"StringEquals":{"rds:rds-tag-identifier/tag-name":
["value"]} }
```

For example, the following `Condition` element applies to DB instances with a tag named `environment` and a tag value of `production`.

```
"Condition":{"StringEquals":{"rds:db-tag/environment": ["production"]} }
```

For information about creating tags, see Tagging Amazon RDS resources.

Important

If you manage access to your RDS resources using tagging, we recommend that you secure access to the tags for your RDS resources. You can manage access to tags by creating policies for the `AddTagsToResource` and `RemoveTagsFromResource` actions. For example, the following policy denies users the ability to add or remove tags for all resources. You can then create policies to allow specific users to add or remove tags.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"DenyTagUpdates",
         "Effect":"Deny",
         "Action":[
            "rds:AddTagsToResource",
            "rds:RemoveTagsFromResource"
         ],
         "Resource":"*"
      }
   ]
}
```

To see a list of Aurora actions, see Actions Defined by Amazon RDS in the *Service Authorization Reference*.

Example policies: Using custom tags

Following are examples of how you can use custom tags in Amazon Aurora IAM permissions policies. For more information about adding tags to an Amazon Aurora resource, see Working with Amazon Resource Names (ARNs) in Amazon RDS.

Note

All examples use the us-west-2 region and contain fictitious account IDs.

Example 1: Grant permission for actions on a resource with a specific tag with two different values

The following policy allows permission to perform the `CreateDBSnapshot` API operation on DB instances with either the `stage` tag set to `development` or `test`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowAnySnapshotName",
         "Effect":"Allow",
         "Action":[
            "rds:CreateDBSnapshot"
         ],
         "Resource":"arn:aws:rds:*:123456789012:snapshot:*"
      },
      {
         "Sid":"AllowDevTestToCreateSnapshot",
         "Effect":"Allow",
         "Action":[
            "rds:CreateDBSnapshot"
         ],
         "Resource":"arn:aws:rds:*:123456789012:db:*",
         "Condition":{
            "StringEquals":{
                "rds:db-tag/stage":[
                  "development",
                  "test"
               ]
            }
         }
      }
   ]
}
```

The following policy allows permission to perform the `ModifyDBInstance` API operation on DB instances with either the `stage` tag set to `development` or `test`.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowChangingParameterOptionSecurityGroups",
         "Effect":"Allow",
         "Action":[
            "rds:ModifyDBInstance"
         ],
         "Resource":" [
            "arn:aws:rds:*:123456789012:pg:*",
            "arn:aws:rds:*:123456789012:secgrp:*",
            "arn:aws:rds:*:123456789012:og:*"
         ]
      },
      {
         "Sid":"AllowDevTestToModifyInstance",
         "Effect":"Allow",
         "Action":[
            "rds:ModifyDBInstance"
         ],
         "Resource":"arn:aws:rds:*:123456789012:db:*",
         "Condition":{
            "StringEquals":{
                "rds:db-tag/stage":[
                  "development",
                  "test"
               ]
            }
         }
      }
   ]
}
```

Example 2: Explicitly deny permission to create a DB instance that uses specified DB parameter groups

The following policy explicitly denies permission to create a DB instance that uses DB parameter groups with specific tag values. You might apply this policy if you require that a specific customer-created DB parameter group always be used when creating DB instances. Policies that use Deny are most often used to restrict access that was granted by a broader policy.

Explicitly denying permission supersedes any other permissions granted. This ensures that identities to not accidentally get permission that you never want to grant.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"DenyProductionCreate",
         "Effect":"Deny",
         "Action":"rds:CreateDBInstance",
         "Resource":"arn:aws:rds:*:123456789012:pg:*",
         "Condition":{
            "StringEquals":{
               "rds:pg-tag/usage":"prod"
            }
         }
      }
   ]
}
```

Example 3: Grant permission for actions on a DB instance with an instance name that is prefixed with a user name

The following policy allows permission to call any API (except to `AddTagsToResource` or `RemoveTagsFromResource`) on a DB instance that has a DB instance name that is prefixed with the user's name and that has a tag called `stage` equal to `devo` or that has no tag called `stage`.

The `Resource` line in the policy identifies a resource by its Amazon Resource Name (ARN). For more information about using ARNs with Amazon Aurora resources, see Working with Amazon Resource Names (ARNs) in Amazon RDS.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowFullDevAccessNoTags",
         "Effect":"Allow",
         "NotAction":[
            "rds:AddTagsToResource",
            "rds:RemoveTagsFromResource"
         ],
         "Resource":"arn:aws:rds:*:123456789012:db:${aws:username}*",
         "Condition":{
            "StringEqualsIfExists":{
               "rds:db-tag/stage":"devo"
            }
         }
      }
   ]
}
```

AWS managed policies for Amazon RDS

To add permissions to permission sets and roles, it's easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (permission sets and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services don't remove permissions from an AWS managed policy, so policy updates don't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the `ReadOnlyAccess` AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

Topics

AWS managed policy: AmazonRDSReadOnlyAccess

This policy allows read-only access to Amazon RDS through the AWS Management Console.

Permissions details

This policy includes the following permissions:

- `rds` – Allows principals to describe Amazon RDS resources and list the tags for Amazon RDS resources.
- `cloudwatch` – Allows principals to get Amazon CloudWatch metric statistics.
- `ec2` – Allows principals to describe Availability Zones and networking resources.
- `logs` – Allows principals to describe CloudWatch Logs log streams of log groups, and get CloudWatch Logs log events.
- `devops-guru` – Allows principals to describe resources that have Amazon DevOps Guru coverage, which is specified either by CloudFormation stack names or resource tags.

For more information about this policy, including the JSON policy document, see AmazonRDSReadOnlyAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSFullAccess

This policy provides full access to Amazon RDS through the AWS Management Console.

Permissions details

This policy includes the following permissions:

- `rds` – Allows principals full access to Amazon RDS.
- `application-autoscaling` – Allows principals describe and manage Application Auto Scaling scaling targets and policies.
- `cloudwatch` – Allows principals get CloudWatch metric statics and manage CloudWatch alarms.
- `ec2` – Allows principals to describe Availability Zones and networking resources.
- `logs` – Allows principals to describe CloudWatch Logs log streams of log groups, and get CloudWatch Logs log events.
- `outposts` – Allows principals to get AWS Outposts instance types.
- `pi` – Allows principals to get Performance Insights metrics.
- `sns` – Allows principals to Amazon Simple Notification Service (Amazon SNS) subscriptions and topics, and to publish Amazon SNS messages.

- `devops-guru` – Allows principals to describe resources that have Amazon DevOps Guru coverage, which is specified either by CloudFormation stack names or resource tags.

For more information about this policy, including the JSON policy document, see AmazonRDSFullAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSDataFullAccess

This policy allows full access to use the Data API and the query editor on Aurora Serverless clusters in a specific AWS account. This policy allows the AWS account to get the value of a secret from AWS Secrets Manager.

You can attach the `AmazonRDSDataFullAccess` policy to your IAM identities.

Permissions details

This policy includes the following permissions:

- `dbqms` – Allows principals to access, create, delete, describe, and update queries. The Database Query Metadata Service (`dbqms`) is an internal-only service. It provides your recent and saved queries for the query editor on the AWS Management Console for multiple AWS services, including Amazon RDS.
- `rds-data` – Allows principals to run SQL statements on Aurora Serverless databases.
- `secretsmanager` – Allows principals to get the value of a secret from AWS Secrets Manager.

For more information about this policy, including the JSON policy document, see AmazonRDSDataFullAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSEnhancedMonitoringRole

This policy provides access to Amazon CloudWatch Logs for Amazon RDS Enhanced Monitoring.

Permissions details

This policy includes the following permissions:

- `logs` – Allows principals to create CloudWatch Logs log groups and retention policies, and to create and describe CloudWatch Logs log streams of log groups. It also allows principals to put and get CloudWatch Logs log events.

For more information about this policy, including the JSON policy document, see AmazonRDSEnhancedMonitoringRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSPerformanceInsightsReadOnly

This policy provides read-only access to Amazon RDS Performance Insights for Amazon RDS DB instances and Amazon Aurora DB clusters.

This policy now includes `Sid` (statement ID) as an identifier for the policy statement.

Permissions details

This policy includes the following permissions:

- `rds` – Allows principals to describe Amazon RDS DB instances and Amazon Aurora DB clusters.
- `pi` – Allows principals make calls to the Amazon RDS Performance Insights API and access Performance Insights metrics.

For more information about this policy, including the JSON policy document, see AmazonRDSPerformanceInsightsReadOnly in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSPerformanceInsightsFullAccess

This policy provides full access to Amazon RDS Performance Insights for Amazon RDS DB instances and Amazon Aurora DB clusters.

This policy now includes `Sid` (statement ID) as an identifier for the policy statement.

Permissions details

This policy includes the following permissions:

- `rds` – Allows principals to describe Amazon RDS DB instances and Amazon Aurora DB clusters.
- `pi` – Allows principals make calls to the Amazon RDS Performance Insights API, and create, view, and delete performance analysis reports.
- `cloudwatch` – Allows principals to list all the Amazon CloudWatch metrics, and get metric data and statistics.

For more information about this policy, including the JSON policy document, see AmazonRDSPerformanceInsightsFullAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSDirectoryServiceAccess

This policy allows Amazon RDS to make calls to the AWS Directory Service.

Permissions details

This policy includes the following permission:

- `ds` – Allows principals to describe AWS Directory Service directories and control authorization to AWS Directory Service directories.

For more information about this policy, including the JSON policy document, see AmazonRDSDirectoryServiceAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AmazonRDSServiceRolePolicy

You can't attach the `AmazonRDSServiceRolePolicy` policy to your IAM entities. This policy is attached to a service-linked role that allows Amazon RDS to perform actions on your behalf. For more information, see Service-linked role permissions for Amazon Aurora.

Amazon RDS updates to AWS managed policies

View details about updates to AWS managed policies for Amazon RDS since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Amazon RDS Document history page.

Change	Description	Date
AWS managed policies for Amazon RDS – Update to existing policy	Amazon RDS added a new permission to the `AmazonRDSCustomServiceRolePolicy` of the `AWSServiceRoleForRDSCustom` service-linked role to allow RDS Custom for SQL Server to modify the underlying database host instance type. RDS also added the `ec2:DescribeInstanceTypes` permission to get instance type information for database host. For more information, see AWS managed policies for Amazon RDS.	April 8, 2024
AWS managed policies for Amazon RDS – New policy	Amazon RDS added a new managed policy named `AmazonRDSCustom InstanceProfileRolePolicy` to allow RDS Custom to perform automation actions and database management tasks through an EC2 instance profile. For more information, see AWS	February 27, 2024

Change	Description	Date
	managed policies for Amazon RDS.	
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added new statement IDs to the `AmazonRDSServiceRolePolicy` of the `AWSServiceRoleForRDS` service-linked role. For more information, see Service-linked role permissions for Amazon Aurora.	January 19, 2024
AWS managed policies for Amazon RDS – Update to existing policies	The `AmazonRDSPerformanceInsightsReadOnly` and `AmazonRDSPerformanceInsightsFullAccess` managed policies now includes `Sid` (statement ID) as an identifier in the policy statement. For more information, see AWS managed policy: AmazonRDSPerformanceInsightsReadOnly and AWS managed policy: AmazonRDSPerformanceInsightsFullAccess	October 23, 2023

Change	Description	Date
AWS managed policies for Amazon RDS – Update to existing policy	Amazon RDS added new permissions to `AmazonRDSFullAccess` managed policy. The permissions allow you to generate, view, and delete the performance analysis report for a time period. For more information about configuring access policies for Performance Insights, see Configuring access policies for Performance Insights	August 17, 2023
AWS managed policies for Amazon RDS – New policy and update to existing policy	Amazon RDS added new permissions to `AmazonRDSPerformanceInsightsReadOnly` managed policy and a new managed policy named `AmazonRDSPerformanceInsightsFullAccess` . These permissions allow you to analyse the Performance Insights for a time period, view the analysis results along with the recommendations, and delete the reports. For more information about configuring access policies for Performance Insights, see Configuring access policies for Performance Insights	August 16, 2023

Change	Description	Date
AWS managed policies for Amazon RDS – Update to an existing policy	Amazon RDS added a new Amazon CloudWatch namespace `ListMetrics` to `AmazonRDSFullAccess` and `AmazonRDSReadOnlyAccess` . This namespace is required for Amazon RDS to list specific resource usage metrics. For more information, see Overview of managing access permissions to your CloudWatch resources in the *Amazon CloudWatch User Guide*.	April 4, 2023

Change	Description	Date
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added new permissions to the `AmazonRDSServiceRolePolicy` of the `AWSServiceRoleForRDS` service-linked role for integration with AWS Secrets Manager. RDS requires integration with Secrets Manager for managing master user passwords in Secrets Manager. The secret uses a reserved naming convention and restricts customer updates. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.	December 22, 2022

Change	Description	Date
AWS managed policies for Amazon RDS – Update to existing policies	Amazon RDS added a new permission to the `AmazonRDSFullAccess` and `AmazonRDSReadOnlyAccess` managed policies to allow you to turn on Amazon DevOps Guru in the RDS console. This permission is required to check whether DevOps Guru is turned on. For more information, see Configuring IAM access policies for DevOps Guru for RDS.	December 19, 2022
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added a new Amazon CloudWatch namespace to `AmazonRDSPreviewServiceRolePolicy` for `PutMetricData` . This namespace is required for Amazon RDS to publish resource usage metrics. For more information, see Using condition keys to limit access to CloudWatch namespaces in the *Amazon CloudWatch User Guide*.	June 7, 2022

Change	Description	Date
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added a new Amazon CloudWatch namespace to `AmazonRDSBetaServiceRolePolicy` for `PutMetricData`. This namespace is required for Amazon RDS to publish resource usage metrics. For more information, see Using condition keys to limit access to CloudWatch namespaces in the *Amazon CloudWatch User Guide*.	June 7, 2022
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added a new Amazon CloudWatch namespace to `AWSServiceRoleForRDS` for `PutMetricData`. This namespace is required for Amazon RDS to publish resource usage metrics. For more information, see Using condition keys to limit access to CloudWatch namespaces in the *Amazon CloudWatch User Guide*.	April 22, 2022

Change	Description	Date
AWS managed policies for Amazon RDS – New policy	Amazon RDS added a new managed policy named `AmazonRDSPerformanceInsightsReadOnly` to allow Amazon RDS to call AWS services on behalf of your DB instances. For more information about configuring access policies for Performance Insights, see Configuring access policies for Performance Insights	March 10, 2022
Service-linked role permissions for Amazon Aurora – Update to an existing policy	Amazon RDS added new Amazon CloudWatch namespaces to `AWSServiceRoleForRDS` for `PutMetricData` . These namespaces are required for Amazon DocumentDB (with MongoDB compatibility) and Amazon Neptune to publish CloudWatch metrics. For more information, see Using condition keys to limit access to CloudWatch namespaces in the *Amazon CloudWatch User Guide*.	March 4, 2022
Amazon RDS started tracking changes	Amazon RDS started tracking changes for its AWS managed policies.	October 26, 2021

Preventing cross-service confused deputy problems

The *confused deputy problem* is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem.

Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way that it shouldn't have permission to access. To prevent this, AWS provides tools that can help you protect your data for all services with service principals that have been given access to resources in your account. For more information, see The confused deputy problem in the *IAM User Guide*.

To limit the permissions that Amazon RDS gives another service for a specific resource, we recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies.

In some cases, the `aws:SourceArn` value doesn't contain the account ID, for example when you use the Amazon Resource Name (ARN) for an Amazon S3 bucket. In these cases, make sure to use both global condition context keys to limit permissions. In some cases, you use both global condition context keys and the `aws:SourceArn` value contains the account ID. In these cases, make sure that the `aws:SourceAccount` value and the account in the `aws:SourceArn` use the same account ID when they're used in the same policy statement. If you want only one resource to be associated with the cross-service access, use `aws:SourceArn`. If you want to allow any resource in the specified AWS account to be associated with the cross-service use, use `aws:SourceAccount`.

Make sure that the value of `aws:SourceArn` is an ARN for an Amazon RDS resource type. For more information, see Working with Amazon Resource Names (ARNs) in Amazon RDS.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. In some cases, you might not know the full ARN of the resource or you might be specifying multiple resources. In these cases, use the `aws:SourceArn` global context condition key with wildcards (*) for the unknown portions of the ARN. An example is `arn:aws:rds:*:123456789012:*`.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Amazon RDS to prevent the confused deputy problem.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "rds.amazonaws.com"
    },
    "Action": "sts:AssumeRole",
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:rds:us-east-1:123456789012:db:mydbinstance"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

For more examples of policies that use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys, see the following sections:

- Granting permissions to publish notifications to an Amazon SNS topic
- Setting up access to an Amazon S3 bucket (PostgreSQL import)
- Setting up access to an Amazon S3 bucket (PostgreSQL export)

IAM database authentication

You can authenticate to your DB cluster using AWS Identity and Access Management (IAM) database authentication. IAM database authentication works with Aurora MySQL, and Aurora PostgreSQL. With this authentication method, you don't need to use a password when you connect to a DB cluster. Instead, you use an authentication token.

An *authentication token* is a unique string of characters that Amazon Aurora generates on request. Authentication tokens are generated using AWS Signature Version 4. Each token has a lifetime of 15 minutes. You don't need to store user credentials in the database, because authentication is managed externally using IAM. You can also still use standard database authentication. The token is only used for authentication and doesn't affect the session after it is established.

IAM database authentication provides the following benefits:

- Network traffic to and from the database is encrypted using Secure Socket Layer (SSL) or Transport Layer Security (TLS). For more information about using SSL/TLS with Amazon Aurora, see Using SSL/TLS to encrypt a connection to a DB cluster.
- You can use IAM to centrally manage access to your database resources, instead of managing access individually on each DB cluster.
- For applications running on Amazon EC2, you can use profile credentials specific to your EC2 instance to access your database instead of a password, for greater security.

In general, consider using IAM database authentication when your applications create fewer than 200 connections per second, and you don't want to manage usernames and passwords directly in your application code.

The Amazon Web Services (AWS) JDBC Driver supports IAM database authentication. For more information, see AWS IAM Authentication Plugin in the Amazon Web Services (AWS) JDBC Driver GitHub repository.

The Amazon Web Services (AWS) Python Driver supports IAM database authentication. For more information, see AWS IAM Authentication Plugin in the Amazon Web Services (AWS) Python Driver GitHub repository.

Topics

- CLI and SDK support
- Limitations for IAM database authentication
- Recommendations for IAM database authentication
- Unsupported AWS global condition context keys
- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication
- Connecting to your DB cluster using IAM authentication

Region and version availability

Feature availability and support varies across specific versions of each Aurora database engine, and across AWS Regions. For more information on version and Region availability with Aurora and IAM database authentication, see Supported Regions and Aurora DB engines for IAM database authentication.

For Aurora MySQL, all supported DB instance classes support IAM database authentication, except for db.t2.small and db.t3.small. For information about the supported DB instance classes, see Supported DB engines for DB instance classes.

CLI and SDK support

IAM database authentication is available for the AWS CLI and for the following language-specific AWS SDKs:

- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP
- AWS SDK for Python (Boto3)
- AWS SDK for Ruby

Limitations for IAM database authentication

When using IAM database authentication, the following limitations apply:

- IAM database authentication throttles connections in the following scenarios:
  - You exceed 20 connections per second using authentication tokens each signed by a different IAM identity.
  - You exceed 200 connections per second using different authentication tokens.

  Connections that use the same authentication token are not throttled. We recommend that you reuse authentication tokens when possible.
- Currently, IAM database authentication doesn't support all global condition context keys.

  For more information about global condition context keys, see AWS global condition context keys in the *IAM User Guide*.
- For PostgreSQL, if the IAM role (`rds_iam`) is added to a user (including the RDS master user), IAM authentication takes precedence over password authentication, so the user must log in as an IAM user.
- For Aurora PostgreSQL, you cannot use IAM authentication to establish a replication connection.
- You cannot use a custom Route 53 DNS record instead of the DB cluster endpoint to generate the authentication token.
- CloudWatch and CloudTrail don't log IAM authentication. These services do not track `generate-db-auth-token` API calls that authorize the IAM role to enable database connection.

Recommendations for IAM database authentication

We recommend the following when using IAM database authentication:

- Use IAM database authentication when your application requires fewer than 200 new IAM database authentication connections per second.

  The database engines that work with Amazon Aurora don't impose any limits on authentication attempts per second. However, when you use IAM database authentication, your application must generate an authentication token. Your application then uses that token to connect to the DB cluster. If you exceed the limit of maximum new connections per second, then the extra overhead of IAM database authentication can cause connection throttling.

Consider using connection pooling in your applications to mitigate constant connection creation. This can reduce the overhead from IAM DB authentication and allow your applications to reuse existing connections. Alternatively, consider using RDS Proxy for these use cases. RDS Proxy has additional costs. See RDS Proxy pricing.

- The size of an IAM database authentication token depends on many things including the number of IAM tags, IAM service policies, ARN lengths, as well as other IAM and database properties. The minimum size of this token is generally about 1 KB but can be larger. Since this token is used as the password in the connection string to the database using IAM authentication, you should ensure that your database driver (e.g., ODBC) and/or any tools do not limit or otherwise truncate this token due to its size. A truncated token will cause the authentication validation done by the database and IAM to fail.
- If you are using temporary credentials when creating an IAM database authentication token, the temporary credentials must still be valid when using the IAM database authentication token to make a connection request.

Unsupported AWS global condition context keys

IAM database authentication does not support the following subset of AWS global condition context keys.

- `aws:Referer`
- `aws:SourceIp`
- `aws:SourceVpc`
- `aws:SourceVpce`
- `aws:UserAgent`
- `aws:VpcSourceIp`

For more information, see AWS global condition context keys in the *IAM User Guide*.

Enabling and disabling IAM database authentication

By default, IAM database authentication is disabled on DB clusters. You can enable or disable IAM database authentication using the AWS Management Console, AWS CLI, or the API.

You can enable IAM database authentication when you perform one of the following actions:

- To create a new DB cluster with IAM database authentication enabled, see Creating an Amazon Aurora DB cluster.
- To modify a DB cluster to enable IAM database authentication, see Modifying an Amazon Aurora DB cluster.
- To restore a DB cluster from a snapshot with IAM database authentication enabled, see Restoring from a DB cluster snapshot.
- To restore a DB cluster to a point in time with IAM database authentication enabled, see Restoring a DB cluster to a specified time.

Console

Each creation or modification workflow has a **Database authentication** section, where you can enable or disable IAM database authentication. In that section, choose **Password and IAM database authentication** to enable IAM database authentication.

To enable or disable IAM database authentication for an existing DB cluster

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Databases**.
3. Choose the DB cluster that you want to modify.

> **Note**
>
> You can only enable IAM authentication if all DB instances in the DB cluster are compatible with IAM. Check the compatibility requirements in Region and version availability.

4. Choose **Modify**.
5. In the **Database authentication** section, choose **Password and IAM database authentication** to enable IAM database authentication. Choose **Password authentication** or **Password and Kerberos authentication** to disable IAM authentication.
6. Choose **Continue**.
7. To apply the changes immediately, choose **Immediately** in the **Scheduling of modifications** section.
8. Choose **Modify cluster**.

AWS CLI

To create a new DB cluster with IAM authentication by using the AWS CLI, use the `create-db-cluster` command. Specify the `--enable-iam-database-authentication` option.

To update an existing DB cluster to have or not have IAM authentication, use the AWS CLI command `modify-db-cluster`. Specify either the `--enable-iam-database-authentication` or `--no-enable-iam-database-authentication` option, as appropriate.

> **Note**
>
> You can only enable IAM authentication if all DB instances in the DB cluster are compatible with IAM. Check the compatibility requirements in Region and version availability.

By default, Aurora performs the modification during the next maintenance window. If you want to override this and enable IAM DB authentication as soon as possible, use the `--apply-immediately` parameter.

If you are restoring a DB cluster, use one of the following AWS CLI commands:

- `restore-db-cluster-to-point-in-time`
- `restore-db-cluster-from-db-snapshot`

The IAM database authentication setting defaults to that of the source snapshot. To change this setting, set the `--enable-iam-database-authentication` or `--no-enable-iam-database-authentication` option, as appropriate.

RDS API

To create a new DB instance with IAM authentication by using the API, use the API operation `CreateDBCluster`. Set the `EnableIAMDatabaseAuthentication` parameter to `true`.

To update an existing DB cluster to have IAM authentication, use the API operation `ModifyDBCluster`. Set the `EnableIAMDatabaseAuthentication` parameter to `true` to enable IAM authentication, or `false` to disable it.

Note

You can only enable IAM authentication if all DB instances in the DB cluster are compatible with IAM. Check the compatibility requirements in Region and version availability.

If you are restoring a DB cluster, use one of the following API operations:

- `RestoreDBClusterFromSnapshot`
- `RestoreDBClusterToPointInTime`

The IAM database authentication setting defaults to that of the source snapshot. To change this setting, set the `EnableIAMDatabaseAuthentication` parameter to `true` to enable IAM authentication, or `false` to disable it.

Creating and using an IAM policy for IAM database access

To allow a user or role to connect to your DB cluster, you must create an IAM policy. After that, you attach the policy to a permissions set or role.

Note

To learn more about IAM policies, see Identity and access management for Amazon Aurora.

The following example policy allows a user to connect to a DB cluster using IAM database authentication.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
             "rds-db:connect"
         ],
         "Resource": [
             "arn:aws:rds-db:us-east-2:1234567890:dbuser:cluster-ABCDEFGHIJKL01234/
db_user"
```

```
            ]
        }
    ]
}
```

Important

A user with administrator permissions can access DB clusters without explicit permissions in an IAM policy. If you want to restrict administrator access to DB clusters, you can create an IAM role with the appropriate, lesser privileged permissions and assign it to the administrator.

Note

Don't confuse the `rds-db:` prefix with other RDS API operation prefixes that begin with `rds:`. You use the `rds-db:` prefix and the `rds-db:connect` action only for IAM database authentication. They aren't valid in any other context.

The example policy includes a single statement with the following elements:

- `Effect` – Specify `Allow` to grant access to the DB cluster. If you don't explicitly allow access, then access is denied by default.
- `Action` – Specify `rds-db:connect` to allow connections to the DB cluster.
- `Resource` – Specify an Amazon Resource Name (ARN) that describes one database account in one DB cluster. The ARN format is as follows.

  ```
  arn:aws:rds-db:region:account-id:dbuser:DbClusterResourceId/db-user-name
  ```

  In this format, replace the following:

  - *`region`* is the AWS Region for the DB cluster. In the example policy, the AWS Region is `us-east-2`.

- *account-id* is the AWS account number for the DB cluster. In the example policy, the account number is 1234567890. The user must be in the same account as the account for the DB cluster.

  To perform cross-account access, create an IAM role with the policy shown above in the account for the DB cluster and allow your other account to assume the role.
- *DbClusterResourceId* is the identifier for the DB cluster. This identifier is unique to an AWS Region and never changes. In the example policy, the identifier is cluster-ABCDEFGHIJKL01234.

  To find a DB cluster resource ID in the AWS Management Console for Amazon Aurora, choose the DB cluster to see its details. Then choose the **Configuration** tab. The **Resource ID** is shown in the **Configuration** section.

  Alternatively, you can use the AWS CLI command to list the identifiers and resource IDs for all of your DB cluster in the current AWS Region, as shown following.

  ```
  aws rds describe-db-clusters --query "DBClusters[*].
  [DBClusterIdentifier,DbClusterResourceId]"
  ```

  > **Note**
  >
  > If you are connecting to a database through RDS Proxy, specify the proxy resource ID, such as prx-ABCDEFGHIJKL01234. For information about using IAM database authentication with RDS Proxy, see Connecting to a proxy using IAM authentication.
- *db-user-name* is the name of the database account to associate with IAM authentication. In the example policy, the database account is db_user.

You can construct other ARNs to support various access patterns. The following policy allows access to two different database accounts in a DB cluster.

```
{
   "Version": "2012-10-17",
   "Statement": [
```

```
       {
          "Effect": "Allow",
          "Action": [
              "rds-db:connect"
          ],
          "Resource": [
              "arn:aws:rds-db:us-east-2:123456789012:dbuser:cluster-ABCDEFGHIJKL01234/
jane_doe",
              "arn:aws:rds-db:us-east-2:123456789012:dbuser:cluster-ABCDEFGHIJKL01234/
mary_roe"
          ]
       }
    ]
}
```

The following policy uses the "*" character to match all DB clusters and database accounts for a particular AWS account and AWS Region.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-db:connect"
            ],
            "Resource": [
                "arn:aws:rds-db:us-east-2:1234567890:dbuser:*/*"
            ]
        }
    ]
}
```

The following policy matches all of the DB clusters for a particular AWS account and AWS Region. However, the policy only grants access to DB clusters that have a `jane_doe` database account.

```
{
   "Version": "2012-10-17",
    "Statement": [
       {
          "Effect": "Allow",
          "Action": [
              "rds-db:connect"
          ],
          "Resource": [
              "arn:aws:rds-db:us-east-2:123456789012:dbuser:*/jane_doe"
          ]
       }
    ]
}
```

The user or role has access to only those databases that the database user does. For example, suppose that your DB cluster has a database named *dev*, and another database named *test*. If the database user `jane_doe` has access only to *dev*, any users or roles that access that DB cluster with the `jane_doe` user also have access only to *dev*. This access restriction is also true for other database objects, such as tables, views, and so on.

An administrator must create IAM policies that grant entities permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the permission sets or roles that require those permissions. For examples of policies, see Identity-based policy examples for Amazon Aurora.

Attaching an IAM policy to a permission set or role

After you create an IAM policy to allow database authentication, you need to attach the policy to a permission set or role. For a tutorial on this topic, see Create and attach your first customer managed policy in the *IAM User Guide*.

As you work through the tutorial, you can use one of the policy examples shown in this section as a starting point and tailor it to your needs. At the end of the tutorial, you have a permission set with an attached policy that can make use of the `rds-db:connect` action.

> **Note**
>
> You can map multiple permission sets or roles to the same database user account. For example, suppose that your IAM policy specified the following resource ARN.

```
arn:aws:rds-db:us-east-2:123456789012:dbuser:cluster-12ABC34DEFG5HIJ6KLMNOP78QR/
jane_doe
```

If you attach the policy to *Jane*, *Bob*, and *Diego*, then each of those users can connect to the specified DB cluster using the `jane_doe` database account.

Creating a database account using IAM authentication

With IAM database authentication, you don't need to assign database passwords to the user accounts you create. If you remove a user that is mapped to a database account, you should also remove the database account with the `DROP USER` statement.

Note

The user name used for IAM authentication must match the case of the user name in the database.

Topics

- Using IAM authentication with Aurora MySQL
- Using IAM authentication with Aurora PostgreSQL

Using IAM authentication with Aurora MySQL

With Aurora MySQL, authentication is handled by `AWSAuthenticationPlugin`—an AWS-provided plugin that works seamlessly with IAM to authenticate your users. Connect to the DB cluster as the master user or a different user who can create users and grant privileges. After connecting, issue the `CREATE USER` statement, as shown in the following example.

```
CREATE USER jane_doe IDENTIFIED WITH AWSAuthenticationPlugin AS 'RDS';
```

The `IDENTIFIED WITH` clause allows Aurora MySQL to use the `AWSAuthenticationPlugin` to authenticate the database account (`jane_doe`). The `AS 'RDS'` clause refers to the authentication method. Make sure the specified database user name is the same as a resource in the IAM policy for

IAM database access. For more information, see Creating and using an IAM policy for IAM database access.

> **Note**
>
> If you see the following message, it means that the AWS-provided plugin is not available for the current DB cluster.
> `ERROR 1524 (HY000): Plugin 'AWSAuthenticationPlugin' is not loaded`
> To troubleshoot this error, verify that you are using a supported configuration and that you have enabled IAM database authentication on your DB cluster. For more information, see Region and version availability and Enabling and disabling IAM database authentication.

After you create an account using `AWSAuthenticationPlugin`, you manage it in the same way as other database accounts. For example, you can modify account privileges with `GRANT` and `REVOKE` statements, or modify various account attributes with the `ALTER USER` statement.

Database network traffic is encrypted using SSL/TLS when using IAM. To allow SSL connections, modify the user account with the following command.

```
ALTER USER 'jane_doe'@'%' REQUIRE SSL;
```

Using IAM authentication with Aurora PostgreSQL

To use IAM authentication with Aurora PostgreSQL, connect to the DB cluster as the master user or a different user who can create users and grant privileges. After connecting, create database users and then grant them the `rds_iam` role as shown in the following example.

```
CREATE USER db_userx;
GRANT rds_iam TO db_userx;
```

Make sure the specified database user name is the same as a resource in the IAM policy for IAM database access. For more information, see Creating and using an IAM policy for IAM database access. You must grant the `rds_iam` role to use IAM authentication. You can use nested memberships or indirect grants of the role as well.

Note that a PostgreSQL database user can use either IAM or Kerberos authentication but not both, so this user can't also have the `rds_ad` role. This also applies to nested memberships. For more information, see Step 7: Create PostgreSQL users for your Kerberos principals .

Connecting to your DB cluster using IAM authentication

With IAM database authentication, you use an authentication token when you connect to your DB cluster. An *authentication token* is a string of characters that you use instead of a password. After you generate an authentication token, it's valid for 15 minutes before it expires. If you try to connect using an expired token, the connection request is denied.

Every authentication token must be accompanied by a valid signature, using AWS signature version 4. (For more information, see Signature Version 4 signing process in the *AWS General Reference.*) The AWS CLI and an AWS SDK, such as the AWS SDK for Java or AWS SDK for Python (Boto3), can automatically sign each token you create.

You can use an authentication token when you connect to Amazon Aurora from another AWS service, such as AWS Lambda. By using a token, you can avoid placing a password in your code. Alternatively, you can use an AWS SDK to programmatically create and programmatically sign an authentication token.

After you have a signed IAM authentication token, you can connect to an Aurora DB cluster. Following, you can find out how to do this using either a command line tool or an AWS SDK, such as the AWS SDK for Java or AWS SDK for Python (Boto3).

For more information, see the following blog posts:

- Use IAM authentication to connect with SQL Workbench/J to Aurora MySQL or Amazon RDS for MySQL
- Using IAM authentication to connect with pgAdmin Amazon Aurora PostgreSQL or Amazon RDS for PostgreSQL

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

Topics

- Connecting to your DB cluster using IAM authentication with the AWS drivers
- Connecting to your DB cluster using IAM authentication from the command line: AWS CLI and mysql client
- Connecting to your DB cluster using IAM authentication from the command line: AWS CLI and psql client
- Connecting to your DB cluster using IAM authentication and the AWS SDK for .NET
- Connecting to your DB cluster using IAM authentication and the AWS SDK for Go
- Connecting to your DB cluster using IAM authentication and the AWS SDK for Java
- Connecting to your DB cluster using IAM authentication and the AWS SDK for Python (Boto3)

Connecting to your DB cluster using IAM authentication with the AWS drivers

The AWS suite of drivers has been designed to provide support for faster switchover and failover times, and authentication with AWS Secrets Manager, AWS Identity and Access Management (IAM), and Federated Identity. The AWS drivers rely on monitoring DB cluster status and being aware of the cluster topology to determine the new writer. This approach reduces switchover and failover times to single-digit seconds, compared to tens of seconds for open-source drivers.

For more information on the AWS drivers, see the corresponding language driver for your Aurora MySQL or Aurora PostgreSQL DB cluster.

Connecting to your DB cluster using IAM authentication from the command line: AWS CLI and mysql client

You can connect from the command line to an Aurora DB cluster with the AWS CLI and `mysql` command line tool as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

Note

For information about connecting to your database using SQL Workbench/J with IAM authentication, see the blog post Use IAM authentication to connect with SQL Workbench/J to Aurora MySQL or Amazon RDS for MySQL.

Topics

- Generating an IAM authentication token
- Connecting to a DB cluster

Generating an IAM authentication token

The following example shows how to get a signed authentication token using the AWS CLI.

```
aws rds generate-db-auth-token \
   --hostname rdsmysql.123456789012.us-west-2.rds.amazonaws.com \
   --port 3306 \
   --region us-west-2 \
   --username jane_doe
```

In the example, the parameters are as follows:

- `--hostname` – The host name of the DB cluster that you want to access
- `--port` – The port number used for connecting to your DB cluster
- `--region` – The AWS Region where the DB cluster is running
- `--username` – The database account that you want to access

The first several characters of the token look like the following.

```
rdsmysql.123456789012.us-west-2.rds.amazonaws.com:3306/?
Action=connect&DBUser=jane_doe&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Expires=900...
```

Note

You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

Connecting to a DB cluster

The general format for connecting is shown following.

```
mysql --host=hostName --port=portNumber --ssl-ca=full_path_to_ssl_certificate --enable-cleartext-plugin --user=userName --password=authToken
```

The parameters are as follows:

- `--host` – The host name of the DB cluster that you want to access
- `--port` – The port number used for connecting to your DB cluster
- `--ssl-ca` – The full path to the SSL certificate file that contains the public key

  For more information, see Using TLS with Aurora MySQL DB clusters.

  To download an SSL certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.
- `--enable-cleartext-plugin` – A value that specifies that `AWSAuthenticationPlugin` must be used for this connection

  If you are using a MariaDB client, the `--enable-cleartext-plugin` option isn't required.
- `--user` – The database account that you want to access
- `--password` – A signed IAM authentication token

The authentication token consists of several hundred characters. It can be unwieldy on the command line. One way to work around this is to save the token to an environment variable, and then use that variable when you connect. The following example shows one way to perform this workaround. In the example, */sample_dir/* is the full path to the SSL certificate file that contains the public key.

```

RDSHOST="mysqlcluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
```

```
TOKEN="$(aws rds generate-db-auth-token --hostname $RDSHOST --port 3306 --region us-west-2 --username jane_doe )"

mysql --host=$RDSHOST --port=3306 --ssl-ca=/sample_dir/global-bundle.pem --enable-cleartext-plugin --user=jane_doe --password=$TOKEN
```

When you connect using `AWSAuthenticationPlugin`, the connection is secured using SSL. To verify this, type the following at the `mysql>` command prompt.

```
show status like 'Ssl%';
```

The following lines in the output show more details.

```
+---------------+-------------+
| Variable_name | Value

                                                                         |
+---------------+-------------+
| ...           | ...
| Ssl_cipher    | AES256-SHA

                                                                         |
| ...           | ...
| Ssl_version   | TLSv1.1

                                                                         |
| ...           | ...
+-----------------------------+
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting to your DB cluster using IAM authentication from the command line: AWS CLI and psql client

You can connect from the command line to an Aurora PostgreSQL DB cluster with the AWS CLI and psql command line tool as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

> **Note**
>
> For information about connecting to your database using pgAdmin with IAM authentication, see the blog post Using IAM authentication to connect with pgAdmin Amazon Aurora PostgreSQL or Amazon RDS for PostgreSQL.

Topics

- Generating an IAM authentication token
- Connecting to an Aurora PostgreSQL cluster

Generating an IAM authentication token

The authentication token consists of several hundred characters so it can be unwieldy on the command line. One way to work around this is to save the token to an environment variable, and then use that variable when you connect. The following example shows how to use the AWS CLI to get a signed authentication token using the `generate-db-auth-token` command, and store it in a `PGPASSWORD` environment variable.

```
export RDSHOST="mypostgres-cluster.cluster-123456789012.us-west-2.rds.amazonaws.com"
export PGPASSWORD="$(aws rds generate-db-auth-token --hostname $RDSHOST --port 5432 --
region us-west-2 --username jane_doe )"
```

In the example, the parameters to the `generate-db-auth-token` command are as follows:

- `--hostname` – The host name of the DB cluster (cluster endpoint) that you want to access
- `--port` – The port number used for connecting to your DB cluster
- `--region` – The AWS Region where the DB cluster is running
- `--username` – The database account that you want to access

The first several characters of the generated token look like the following.

```
mypostgres-cluster.cluster-123456789012.us-west-2.rds.amazonaws.com:5432/?
Action=connect&DBUser=jane_doe&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Expires=900...
```

> **Note**
>
> You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

Connecting to an Aurora PostgreSQL cluster

The general format for using psql to connect is shown following.

```
psql "host=hostName port=portNumber sslmode=verify-full
 sslrootcert=full_path_to_ssl_certificate dbname=DBName user=userName
 password=authToken"
```

The parameters are as follows:

- `host` – The host name of the DB cluster (cluster endpoint) that you want to access
- `port` – The port number used for connecting to your DB cluster
- `sslmode` – The SSL mode to use

  When you use `sslmode=verify-full`, the SSL connection verifies the DB cluster endpoint against the endpoint in the SSL certificate.
- `sslrootcert` – The full path to the SSL certificate file that contains the public key

  For more information, see Securing Aurora PostgreSQL data with SSL/TLS.

  To download an SSL certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.
- `dbname` – The database that you want to access
- `user` – The database account that you want to access
- `password` – A signed IAM authentication token

> **Note**
>
> You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

The following example shows using psql to connect. In the example, psql uses the environment variable RDSHOST for the host and the environment variable PGPASSWORD for the generated token. Also, */sample_dir/* is the full path to the SSL certificate file that contains the public key.

```
export RDSHOST="mypostgres-cluster.cluster-123456789012.us-west-2.rds.amazonaws.com"
export PGPASSWORD="$(aws rds generate-db-auth-token --hostname $RDSHOST --port 5432 --
region us-west-2 --username jane_doe )"

psql "host=$RDSHOST port=5432 sslmode=verify-full sslrootcert=/sample_dir/global-
bundle.pem dbname=DBName user=jane_doe password=$PGPASSWORD"
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting to your DB cluster using IAM authentication and the AWS SDK for .NET

You can connect to an Aurora MySQL or Aurora PostgreSQL DB cluster with the AWS SDK for .NET as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

Examples

The following code examples show how to generate an authentication token, and then use it to connect to a DB cluster.

To run this code example, you need the AWS SDK for .NET, found on the AWS site. The AWSSDK.CORE and the AWSSDK.RDS packages are required. To connect to a DB cluster, use the .NET database connector for the DB engine, such as MySqlConnector for MariaDB or MySQL, or Npgsql for PostgreSQL.

This code connects to an Aurora MySQL DB cluster. Modify the values of the following variables as needed:

- `server` – The endpoint of the DB cluster that you want to access
- `user` – The database account that you want to access
- `database` – The database that you want to access
- `port` – The port number used for connecting to your DB cluster
- `SslMode` – The SSL mode to use

  When you use `SslMode=Required`, the SSL connection verifies the DB cluster endpoint against the endpoint in the SSL certificate.
- `SslCa` – The full path to the SSL certificate for Amazon Aurora

  To download a certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

Note

You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

```
using System;
using System.Data;
using MySql.Data;
using MySql.Data.MySqlClient;
using Amazon;

namespace ubuntu
{
  class Program
  {
    static void Main(string[] args)
    {
      var pwd =
 Amazon.RDS.Util.RDSAuthTokenGenerator.GenerateAuthToken(RegionEndpoint.USEast1,
 "mysqlcluster.cluster-123456789012.us-east-1.rds.amazonaws.com", 3306, "jane_doe");
```

```
      // for debug only Console.Write("{0}\n", pwd);  //this verifies the token is
 generated

      MySqlConnection conn = new
 MySqlConnection($"server=mysqlcluster.cluster-123456789012.us-
east-1.rds.amazonaws.com;user=jane_doe;database=mydB;port=3306;password={pwd};SslMode=Required;
      conn.Open();

      // Define a query
      MySqlCommand sampleCommand = new MySqlCommand("SHOW DATABASES;", conn);

      // Execute a query
      MySqlDataReader mysqlDataRdr = sampleCommand.ExecuteReader();

      // Read all rows and output the first column in each row
      while (mysqlDataRdr.Read())
        Console.WriteLine(mysqlDataRdr[0]);

      mysqlDataRdr.Close();
      // Close connection
      conn.Close();
    }
  }
}
```

This code connects to an Aurora PostgreSQL DB cluster.

Modify the values of the following variables as needed:

- `Server` – The endpoint of the DB cluster that you want to access
- `User ID` – The database account that you want to access
- `Database` – The database that you want to access
- `Port` – The port number used for connecting to your DB cluster
- `SSL Mode` – The SSL mode to use

  When you use `SSL Mode=Required`, the SSL connection verifies the DB cluster endpoint against the endpoint in the SSL certificate.
- `Root Certificate` – The full path to the SSL certificate for Amazon Aurora

  To download a certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

Note

You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

```
using System;
using Npgsql;
using Amazon.RDS.Util;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            var pwd =
 RDSAuthTokenGenerator.GenerateAuthToken("postgresmycluster.cluster-123456789012.us-
east-1.rds.amazonaws.com", 5432, "jane_doe");
// for debug only Console.Write("{0}\n", pwd);  //this verifies the token is generated

            NpgsqlConnection conn = new
 NpgsqlConnection($"Server=postgresmycluster.cluster-123456789012.us-
east-1.rds.amazonaws.com;User Id=jane_doe;Password={pwd};Database=mydb;SSL
 Mode=Require;Root Certificate=full_path_to_ssl_certificate");
            conn.Open();

            // Define a query
                   NpgsqlCommand cmd = new NpgsqlCommand("select count(*) FROM
 pg_user", conn);

            // Execute a query
            NpgsqlDataReader dr = cmd.ExecuteReader();

            // Read all rows and output the first column in each row
            while (dr.Read())
                Console.Write("{0}\n", dr[0]);

            // Close connection
            conn.Close();
        }
    }
}
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting to your DB cluster using IAM authentication and the AWS SDK for Go

You can connect to an Aurora MySQL or Aurora PostgreSQL DB cluster with the AWS SDK for Go as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

Examples

To run these code examples, you need the AWS SDK for Go, found on the AWS site.

Modify the values of the following variables as needed:

- `dbName` – The database that you want to access
- `dbUser` – The database account that you want to access
- `dbHost` – The endpoint of the DB cluster that you want to access

> **Note**
>
> You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

- `dbPort` – The port number used for connecting to your DB cluster
- `region` – The AWS Region where the DB cluster is running

In addition, make sure the imported libraries in the sample code exist on your system.

Important

The examples in this section use the following code to provide credentials that access a database from a local environment:

`creds := credentials.NewEnvCredentials()`

If you are accessing a database from an AWS service, such as Amazon EC2 or Amazon ECS, you can replace the code with the following code:

`sess := session.Must(session.NewSession())`

`creds := sess.Config.Credentials`

If you make this change, make sure you add the following import:

`"github.com/aws/aws-sdk-go/aws/session"`

Topics

- Connecting using IAM authentication and the AWS SDK for Go V2
- Connecting using IAM authentication and the AWS SDK for Go V1.

Connecting using IAM authentication and the AWS SDK for Go V2

You can connect to a DB cluster using IAM authentication and the AWS SDK for Go V2.

The following code examples show how to generate an authentication token, and then use it to connect to a DB cluster.

This code connects to an Aurora MySQL DB cluster.

```
package main

import (
     "context"
     "database/sql"
     "fmt"

     "github.com/aws/aws-sdk-go-v2/config"
     "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
     _ "github.com/go-sql-driver/mysql"
)

func main() {
     var dbName string = "DatabaseName"
     var dbUser string = "DatabaseUser"
     var dbHost string = "mysqlcluster.cluster-123456789012.us-
east-1.rds.amazonaws.com"
     var dbPort int = 3306
     var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
     var region string = "us-east-1"

    cfg, err := config.LoadDefaultConfig(context.TODO())
    if err != nil {
     panic("configuration error: " + err.Error())
    }

    authenticationToken, err := auth.BuildAuthToken(
     context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
    if err != nil {
     panic("failed to create authentication token: " + err.Error())
    }

    dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
        dbUser, authenticationToken, dbEndpoint, dbName,
    )

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        panic(err)
    }

    err = db.Ping()
    if err != nil {
        panic(err)
    }
}
```

This code connects to an Aurora PostgreSQL DB cluster.

```
package main

import (
     "context"
     "database/sql"
     "fmt"
     "github.com/aws/aws-sdk-go-v2/config"
     "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
     _ "github.com/lib/pq"
)

func main() {

     var dbName string = "DatabaseName"
     var dbUser string = "DatabaseUser"
     var dbHost string = "postgresmycluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
     var dbPort int = 5432
     var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
     var region string = "us-east-1"

    cfg, err := config.LoadDefaultConfig(context.TODO())
    if err != nil {
     panic("configuration error: " + err.Error())
    }

    authenticationToken, err := auth.BuildAuthToken(
     context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
    if err != nil {
     panic("failed to create authentication token: " + err.Error())
    }

    dsn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s",
        dbHost, dbPort, dbUser, authenticationToken, dbName,
    )

    db, err := sql.Open("postgres", dsn)
    if err != nil {
        panic(err)
    }

    err = db.Ping()
    if err != nil {
        panic(err)
    }
}
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting using IAM authentication and the AWS SDK for Go V1.

You can connect to a DB cluster using IAM authentication and the AWS SDK for Go V1

The following code examples show how to generate an authentication token, and then use it to connect to a DB cluster.

This code connects to an Aurora MySQL DB cluster.

```
package main

import (
    "database/sql"
    "fmt"
    "log"

    "github.com/aws/aws-sdk-go/aws/credentials"
    "github.com/aws/aws-sdk-go/service/rds/rdsutils"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dbName := "app"
    dbUser := "jane_doe"
    dbHost := "mysqlcluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
    dbPort := 3306
    dbEndpoint := fmt.Sprintf("%s:%d", dbHost, dbPort)
    region := "us-east-1"

    creds := credentials.NewEnvCredentials()
    authToken, err := rdsutils.BuildAuthToken(dbEndpoint, region, dbUser, creds)
    if err != nil {
        panic(err)
    }

    dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
        dbUser, authToken, dbEndpoint, dbName,
    )

    db, err := sql.Open("mysql", dsn)
    if err != nil {
        panic(err)
    }

    err = db.Ping()
    if err != nil {
        panic(err)
    }
}
```

This code connects to an Aurora PostgreSQL DB cluster.

```
package main

import (
 "database/sql"
 "fmt"

 "github.com/aws/aws-sdk-go/aws/credentials"
 "github.com/aws/aws-sdk-go/service/rds/rdsutils"
 _ "github.com/lib/pq"
)

func main() {
    dbName := "app"
    dbUser := "jane_doe"
    dbHost := "postgresmycluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
    dbPort := 5432
    dbEndpoint := fmt.Sprintf("%s:%d", dbHost, dbPort)
    region := "us-east-1"

    creds := credentials.NewEnvCredentials()
    authToken, err := rdsutils.BuildAuthToken(dbEndpoint, region, dbUser, creds)
    if err != nil {
        panic(err)
    }

    dsn := fmt.Sprintf("host=%s port=%d user=%s password=%s dbname=%s",
        dbHost, dbPort, dbUser, authToken, dbName,
    )

    db, err := sql.Open("postgres", dsn)
    if err != nil {
        panic(err)
    }

    err = db.Ping()
    if err != nil {
        panic(err)
    }
}
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting to your DB cluster using IAM authentication and the AWS SDK for Java

You can connect to an Aurora MySQL or Aurora PostgreSQL DB cluster with the AWS SDK for Java as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication
- Set up the AWS SDK for Java

For examples on how to use the SDK for Java 2.x, see Amazon RDS examples using SDK for Java 2.x.

Topics

- Generating an IAM authentication token
- Manually constructing an IAM authentication token
- Connecting to a DB cluster

Generating an IAM authentication token

If you are writing programs using the AWS SDK for Java, you can get a signed authentication token using the `RdsIamAuthTokenGenerator` class. Using this class requires that you provide AWS credentials. To do this, you create an instance of the `DefaultAWSCredentialsProviderChain` class. `DefaultAWSCredentialsProviderChain` uses the first AWS access key and secret key that it finds in the default credential provider chain. For more information about AWS access keys, see Managing access keys for users.

Note

You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

After you create an instance of `RdsIamAuthTokenGenerator`, you can call the `getAuthToken` method to obtain a signed token. Provide the AWS Region, host name, port number, and user name. The following code example illustrates how to do this.

```
package com.amazonaws.codesamples;

import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.services.rds.auth.GetIamAuthTokenRequest;
import com.amazonaws.services.rds.auth.RdsIamAuthTokenGenerator;

public class GenerateRDSAuthToken {

    public static void main(String[] args) {

     String region = "us-west-2";
     String hostname = "rdsmysql.123456789012.us-west-2.rds.amazonaws.com";
     String port = "3306";
     String username = "jane_doe";

     System.out.println(generateAuthToken(region, hostname, port, username));
    }

    static String generateAuthToken(String region, String hostName, String port, String
 username) {

     RdsIamAuthTokenGenerator generator = RdsIamAuthTokenGenerator.builder()
      .credentials(new DefaultAWSCredentialsProviderChain())
      .region(region)
      .build();

     String authToken = generator.getAuthToken(
      GetIamAuthTokenRequest.builder()
      .hostname(hostName)
      .port(Integer.parseInt(port))
      .userName(username)
      .build());

     return authToken;
    }

}
```

Manually constructing an IAM authentication token

In Java, the easiest way to generate an authentication token is to use `RdsIamAuthTokenGenerator`. This class creates an authentication token for you, and then signs it using AWS signature version 4. For more information, see Signature version 4 signing process in the *AWS General Reference.*

However, you can also construct and sign an authentication token manually, as shown in the following code example.

```
package com.amazonaws.codesamples;

import com.amazonaws.SdkClientException;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.auth.SigningAlgorithm;
import com.amazonaws.util.BinaryUtils;
import org.apache.commons.lang3.StringUtils;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;
import java.nio.charset.Charset;
import java.security.MessageDigest;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.SortedMap;
import java.util.TreeMap;

import static com.amazonaws.auth.internal.SignerConstants.AWS4_TERMINATOR;
import static com.amazonaws.util.StringUtils.UTF8;

public class CreateRDSAuthTokenManually {
    public static String httpMethod = "GET";
    public static String action = "connect";
    public static String canonicalURIParameter = "/";
```

```
   public static SortedMap<String, String> canonicalQueryParameters = new TreeMap();
   public static String payload = StringUtils.EMPTY;
   public static String signedHeader = "host";
   public static String algorithm = "AWS4-HMAC-SHA256";
   public static String serviceName = "rds-db";
   public static String requestWithoutSignature;

   public static void main(String[] args) throws Exception {

       String region = "us-west-2";
       String instanceName = "rdsmysql.123456789012.us-west-2.rds.amazonaws.com";
       String port = "3306";
       String username = "jane_doe";

       Date now = new Date();
       String date = new SimpleDateFormat("yyyyMMdd").format(now);
       String dateTimeStamp = new
SimpleDateFormat("yyyyMMdd'T'HHmmss'Z'").format(now);
       DefaultAWSCredentialsProviderChain creds = new
DefaultAWSCredentialsProviderChain();
    String awsAccessKey = creds.getCredentials().getAWSAccessKeyId();
    String awsSecretKey = creds.getCredentials().getAWSSecretKey();
       String expiryMinutes = "900";

       System.out.println("Step 1:  Create a canonical request:");
       String canonicalString = createCanonicalString(username, awsAccessKey, date,
dateTimeStamp, region, expiryMinutes, instanceName, port);
       System.out.println(canonicalString);
       System.out.println();

       System.out.println("Step 2:  Create a string to sign:");
       String stringToSign = createStringToSign(dateTimeStamp, canonicalString,
awsAccessKey, date, region);
       System.out.println(stringToSign);
       System.out.println();

       System.out.println("Step 3:  Calculate the signature:");
       String signature = BinaryUtils.toHex(calculateSignature(stringToSign,
newSigningKey(awsSecretKey, date, region, serviceName)));
       System.out.println(signature);
       System.out.println();

       System.out.println("Step 4:  Add the signing info to the request");
```

```
        System.out.println(appendSignature(signature));
        System.out.println();

    }

    //Step 1: Create a canonical request date should be in format YYYYMMDD and dateTime
 should be in format YYYYMMDDTHHMMSSZ
    public static String createCanonicalString(String user, String accessKey, String
 date, String dateTime, String region, String expiryPeriod, String hostName, String
 port) throws Exception {
        canonicalQueryParameters.put("Action", action);
        canonicalQueryParameters.put("DBUser", user);
        canonicalQueryParameters.put("X-Amz-Algorithm", "AWS4-HMAC-SHA256");
        canonicalQueryParameters.put("X-Amz-Credential", accessKey + "%2F" + date +
 "%2F" + region + "%2F" + serviceName + "%2Faws4_request");
        canonicalQueryParameters.put("X-Amz-Date", dateTime);
        canonicalQueryParameters.put("X-Amz-Expires", expiryPeriod);
        canonicalQueryParameters.put("X-Amz-SignedHeaders", signedHeader);
        String canonicalQueryString = "";
        while(!canonicalQueryParameters.isEmpty()) {
            String currentQueryParameter = canonicalQueryParameters.firstKey();
            String currentQueryParameterValue =
 canonicalQueryParameters.remove(currentQueryParameter);
            canonicalQueryString = canonicalQueryString + currentQueryParameter + "=" +
 currentQueryParameterValue;
            if (!currentQueryParameter.equals("X-Amz-SignedHeaders")) {
                canonicalQueryString += "&";
            }
        }
        String canonicalHeaders = "host:" + hostName + ":" + port + '\n';
        requestWithoutSignature = hostName + ":" + port + "/?" + canonicalQueryString;

        String hashedPayload = BinaryUtils.toHex(hash(payload));
        return httpMethod + '\n' + canonicalURIParameter + '\n' + canonicalQueryString
 + '\n' + canonicalHeaders + '\n' + signedHeader + '\n' + hashedPayload;

    }

    //Step 2: Create a string to sign using sig v4
    public static String createStringToSign(String dateTime, String canonicalRequest,
 String accessKey, String date, String region) throws Exception {
        String credentialScope = date + "/" + region + "/" + serviceName + "/
aws4_request";
```

```
       return algorithm + '\n' + dateTime + '\n' + credentialScope + '\n' +
BinaryUtils.toHex(hash(canonicalRequest));

   }

   //Step 3: Calculate signature
   /**
    * Step 3 of the &AWS; Signature version 4 calculation. It involves deriving
    * the signing key and computing the signature. Refer to
    * http://docs.aws.amazon
    * .com/general/latest/gr/sigv4-calculate-signature.html
    */
   public static byte[] calculateSignature(String stringToSign,
                                           byte[] signingKey) {
       return sign(stringToSign.getBytes(Charset.forName("UTF-8")), signingKey,
               SigningAlgorithm.HmacSHA256);
   }

   public static byte[] sign(byte[] data, byte[] key,
                         SigningAlgorithm algorithm) throws SdkClientException {
       try {
           Mac mac = algorithm.getMac();
           mac.init(new SecretKeySpec(key, algorithm.toString()));
           return mac.doFinal(data);
       } catch (Exception e) {
           throw new SdkClientException(
                   "Unable to calculate a request signature: "
                           + e.getMessage(), e);
       }
   }

   public static byte[] newSigningKey(String secretKey,
                                  String dateStamp, String regionName, String
serviceName) {
       byte[] kSecret = ("AWS4" + secretKey).getBytes(Charset.forName("UTF-8"));
       byte[] kDate = sign(dateStamp, kSecret, SigningAlgorithm.HmacSHA256);
       byte[] kRegion = sign(regionName, kDate, SigningAlgorithm.HmacSHA256);
       byte[] kService = sign(serviceName, kRegion,
               SigningAlgorithm.HmacSHA256);
       return sign(AWS4_TERMINATOR, kService, SigningAlgorithm.HmacSHA256);
   }

   public static byte[] sign(String stringData, byte[] key,
                      SigningAlgorithm algorithm) throws SdkClientException {
```

```
        try {
            byte[] data = stringData.getBytes(UTF8);
            return sign(data, key, algorithm);
        } catch (Exception e) {
            throw new SdkClientException(
                    "Unable to calculate a request signature: "
                            + e.getMessage(), e);
        }
    }

    //Step 4: append the signature
    public static String appendSignature(String signature) {
        return requestWithoutSignature + "&X-Amz-Signature=" + signature;
    }

    public static byte[] hash(String s) throws Exception {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            md.update(s.getBytes(UTF8));
            return md.digest();
        } catch (Exception e) {
            throw new SdkClientException(
                    "Unable to compute hash while signing request: "
                            + e.getMessage(), e);
        }
    }
}
```

Connecting to a DB cluster

The following code example shows how to generate an authentication token, and then use it to connect to a cluster running Aurora MySQL.

To run this code example, you need the AWS SDK for Java, found on the AWS site. In addition, you need the following:

- MySQL Connector/J. This code example was tested with `mysql-connector-java-5.1.33-bin.jar`.
- An intermediate certificate for Amazon Aurora that is specific to an AWS Region. (For more information, see Using SSL/TLS to encrypt a connection to a DB cluster.) At runtime, the class loader looks for the certificate in the same directory as this Java code example, so that the class loader can find it.

- Modify the values of the following variables as needed:
  - RDS_INSTANCE_HOSTNAME – The host name of the DB cluster that you want to access.
  - RDS_INSTANCE_PORT – The port number used for connecting to your PostgreSQL DB cluster.
  - REGION_NAME – The AWS Region where the DB cluster is running.
  - DB_USER – The database account that you want to access.
  - SSL_CERTIFICATE – An SSL certificate for Amazon Aurora that is specific to an AWS Region.

    To download a certificate for your AWS Region, see Using SSL/TLS to encrypt a connection to a DB cluster. Place the SSL certificate in the same directory as this Java program file, so that the class loader can find the certificate at runtime.

This code example obtains AWS credentials from the default credential provider chain.

Note

Specify a password for DEFAULT_KEY_STORE_PASSWORD other than the prompt shown here as a security best practice.

```
package com.amazonaws.samples;

import com.amazonaws.services.rds.auth.RdsIamAuthTokenGenerator;
import com.amazonaws.services.rds.auth.GetIamAuthTokenRequest;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.auth.AWSStaticCredentialsProvider;

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.security.KeyStore;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;
import java.net.URL;

public class IAMDatabaseAuthenticationTester {
    //&AWS; Credentials of the IAM user with policy enabling IAM Database Authenticated
 access to the db by the db user.
    private static final DefaultAWSCredentialsProviderChain creds = new
 DefaultAWSCredentialsProviderChain();
    private static final String AWS_ACCESS_KEY =
 creds.getCredentials().getAWSAccessKeyId();
    private static final String AWS_SECRET_KEY =
 creds.getCredentials().getAWSSecretKey();

    //Configuration parameters for the generation of the IAM Database Authentication
 token
    private static final String RDS_INSTANCE_HOSTNAME = "rdsmysql.123456789012.us-
west-2.rds.amazonaws.com";
    private static final int RDS_INSTANCE_PORT = 3306;
    private static final String REGION_NAME = "us-west-2";
    private static final String DB_USER = "jane_doe";
    private static final String JDBC_URL = "jdbc:mysql://" + RDS_INSTANCE_HOSTNAME +
 ":" + RDS_INSTANCE_PORT;

    private static final String SSL_CERTIFICATE = "rds-ca-2019-us-west-2.pem";

    private static final String KEY_STORE_TYPE = "JKS";
    private static final String KEY_STORE_PROVIDER = "SUN";
    private static final String KEY_STORE_FILE_PREFIX = "sys-connect-via-ssl-test-
cacerts";
    private static final String KEY_STORE_FILE_SUFFIX = ".jks";
    private static final String DEFAULT_KEY_STORE_PASSWORD = "changeit";

    public static void main(String[] args) throws Exception {
        //get the connection
        Connection connection = getDBConnectionUsingIam();

        //verify the connection is successful
        Statement stmt= connection.createStatement();
        ResultSet rs=stmt.executeQuery("SELECT 'Success!' FROM DUAL;");
        while (rs.next()) {
             String id = rs.getString(1);
            System.out.println(id); //Should print "Success!"
        }
        //close the connection
        stmt.close();
        connection.close();

        clearSslProperties();

    }

    /**
     * This method returns a connection to the db instance authenticated using IAM
 Database Authentication
     * @return
     * @throws Exception
     */
    private static Connection getDBConnectionUsingIam() throws Exception {
        setSslProperties();
        return DriverManager.getConnection(JDBC_URL, setMySqlConnectionProperties());
    }

    /**
     * This method sets the mysql connection properties which includes the IAM Database
 Authentication token
     * as the password. It also specifies that SSL verification is required.
     * @return
     */
    private static Properties setMySqlConnectionProperties() {
        Properties mysqlConnectionProperties = new Properties();
        mysqlConnectionProperties.setProperty("verifyServerCertificate","true");
        mysqlConnectionProperties.setProperty("useSSL", "true");
        mysqlConnectionProperties.setProperty("user",DB_USER);
        mysqlConnectionProperties.setProperty("password",generateAuthToken());
        return mysqlConnectionProperties;
    }

    /**
     * This method generates the IAM Auth Token.
     * An example IAM Auth Token would look like follows:
     * btusi123.cmz7kenwo2ye.rds.cn-north-1.amazonaws.com.cn:3306/?
Action=connect&DBUser=iamtestuser&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-
Date=20171003T010726Z&X-Amz-SignedHeaders=host&X-Amz-Expires=899&X-Amz-
Credential=AKIAPFXHGVDI5RNFO4AQ%2F20171003%2Fcn-north-1%2Frds-db%2Faws4_request&X-Amz-
Signature=f9f45ef96c1f770cdad11a53e33ffa4c3730bc03fdee820cfdf1322eed15483b
     * @return
     */
```

```
   private static String generateAuthToken() {
       BasicAWSCredentials awsCredentials = new BasicAWSCredentials(AWS_ACCESS_KEY, 
AWS_SECRET_KEY);

       RdsIamAuthTokenGenerator generator = RdsIamAuthTokenGenerator.builder()
               .credentials(new 
AWSStaticCredentialsProvider(awsCredentials)).region(REGION_NAME).build();
       return generator.getAuthToken(GetIamAuthTokenRequest.builder()
 
.hostname(RDS_INSTANCE_HOSTNAME).port(RDS_INSTANCE_PORT).userName(DB_USER).build());
   }

   /**
    * This method sets the SSL properties which specify the key store file, its type 
and password:
    * @throws Exception
    */
   private static void setSslProperties() throws Exception {
       System.setProperty("javax.net.ssl.trustStore", createKeyStoreFile());
       System.setProperty("javax.net.ssl.trustStoreType", KEY_STORE_TYPE);
       System.setProperty("javax.net.ssl.trustStorePassword", 
DEFAULT_KEY_STORE_PASSWORD);
   }

   /**
    * This method returns the path of the Key Store File needed for the SSL 
verification during the IAM Database Authentication to
    * the db instance.
    * @return
    * @throws Exception
    */
   private static String createKeyStoreFile() throws Exception {
       return createKeyStoreFile(createCertificate()).getPath();
   }

   /**
    *  This method generates the SSL certificate
    * @return
    * @throws Exception
    */
   private static X509Certificate createCertificate() throws Exception {
       CertificateFactory certFactory = CertificateFactory.getInstance("X.509");
       URL url = new File(SSL_CERTIFICATE).toURI().toURL();
       if (url == null) {
            throw new Exception();
        }
        try (InputStream certInputStream = url.openStream()) {
            return (X509Certificate) certFactory.generateCertificate(certInputStream);
        }
    }

    /**
     * This method creates the Key Store File
     * @param rootX509Certificate - the SSL certificate to be stored in the KeyStore
     * @return
     * @throws Exception
     */
    private static File createKeyStoreFile(X509Certificate rootX509Certificate) throws
 Exception {
        File keyStoreFile = File.createTempFile(KEY_STORE_FILE_PREFIX,
 KEY_STORE_FILE_SUFFIX);
        try (FileOutputStream fos = new FileOutputStream(keyStoreFile.getPath())) {
            KeyStore ks = KeyStore.getInstance(KEY_STORE_TYPE, KEY_STORE_PROVIDER);
            ks.load(null);
            ks.setCertificateEntry("rootCaCertificate", rootX509Certificate);
            ks.store(fos, DEFAULT_KEY_STORE_PASSWORD.toCharArray());
        }
        return keyStoreFile;
    }

    /**
     * This method clears the SSL properties.
     * @throws Exception
     */
    private static void clearSslProperties() throws Exception {
           System.clearProperty("javax.net.ssl.trustStore");
           System.clearProperty("javax.net.ssl.trustStoreType");
           System.clearProperty("javax.net.ssl.trustStorePassword");
    }

}
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Connecting to your DB cluster using IAM authentication and the AWS SDK for Python (Boto3)

You can connect to an Aurora MySQL or Aurora PostgreSQL DB cluster with the AWS SDK for Python (Boto3) as described following.

Prerequisites

The following are prerequisites for connecting to your DB cluster using IAM authentication:

- Enabling and disabling IAM database authentication
- Creating and using an IAM policy for IAM database access
- Creating a database account using IAM authentication

In addition, make sure the imported libraries in the sample code exist on your system.

Examples

The code examples use profiles for shared credentials. For information about the specifying credentials, see Credentials in the AWS SDK for Python (Boto3) documentation.

The following code examples show how to generate an authentication token, and then use it to connect to a DB cluster.

To run this code example, you need the AWS SDK for Python (Boto3), found on the AWS site.

Modify the values of the following variables as needed:

- `ENDPOINT` – The endpoint of the DB cluster that you want to access
- `PORT` – The port number used for connecting to your DB cluster
- `USER` – The database account that you want to access
- `REGION` – The AWS Region where the DB cluster is running
- `DBNAME` – The database that you want to access
- `SSLCERTIFICATE` – The full path to the SSL certificate for Amazon Aurora

  For `ssl_ca`, specify an SSL certificate. To download an SSL certificate, see Using SSL/TLS to encrypt a connection to a DB cluster.

> **Note**
>
> You cannot use a custom Route 53 DNS record or an Aurora custom endpoint instead of the DB cluster endpoint to generate the authentication token.

This code connects to an Aurora MySQL DB cluster.

Before running this code, install the PyMySQL driver by following the instructions in the Python Package Index.

```
import pymysql
import sys
import boto3
import os

ENDPOINT="mysqlcluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
PORT="3306"
USER="jane_doe"
REGION="us-east-1"
DBNAME="mydb"
os.environ['LIBMYSQL_ENABLE_CLEARTEXT_PLUGIN'] = '1'

#gets the credentials from .aws/credentials
session = boto3.Session(profile_name='default')
client = session.client('rds')

token = client.generate_db_auth_token(DBHostname=ENDPOINT, Port=PORT, DBUsername=USER,
 Region=REGION)

try:
    conn =  pymysql.connect(host=ENDPOINT, user=USER, passwd=token, port=PORT,
 database=DBNAME, ssl_ca='SSLCERTIFICATE')
    cur = conn.cursor()
    cur.execute("""SELECT now()""")
    query_results = cur.fetchall()
    print(query_results)
except Exception as e:
    print("Database connection failed due to {}".format(e))
```

This code connects to an Aurora PostgreSQL DB cluster.

Before running this code, install psycopg2 by following the instructions in Psycopg documentation.

```
import psycopg2
import sys
import boto3
import os

ENDPOINT="postgresmycluster.cluster-123456789012.us-east-1.rds.amazonaws.com"
PORT="5432"
USER="jane_doe"
REGION="us-east-1"
DBNAME="mydb"

#gets the credentials from .aws/credentials
session = boto3.Session(profile_name='RDSCreds')
client = session.client('rds')

token = client.generate_db_auth_token(DBHostname=ENDPOINT, Port=PORT, DBUsername=USER,
 Region=REGION)

try:
    conn = psycopg2.connect(host=ENDPOINT, port=PORT, database=DBNAME, user=USER,
 password=token, sslrootcert="SSLCERTIFICATE")
    cur = conn.cursor()
    cur.execute("""SELECT now()""")
    query_results = cur.fetchall()
    print(query_results)
except Exception as e:
    print("Database connection failed due to {}".format(e))
```

If you want to connect to a DB cluster through a proxy, see Connecting to a proxy using IAM authentication.

Troubleshooting Amazon Aurora identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Aurora and IAM.

Topics

- [I'm not authorized to perform an action in Aurora]
- [I'm not authorized to perform iam:PassRole]
- [I want to allow people outside of my AWS account to access my Aurora resources]

I'm not authorized to perform an action in Aurora

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials.

The following example error occurs when the `mateojackson` user tries to use the console to view details about a *`widget`* but does not have `rds:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 rds:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-widget`* resource using the `rds:`*`GetWidget`* action.

I'm not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your sign-in credentials. Ask that person to update your policies to allow you to pass a role to Aurora.

Some AWS services allow you to pass an existing role to that service, instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when a user named `marymajor` tries to use the console to perform an action in Aurora. However, the action requires the service to have permissions granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

I want to allow people outside of my AWS account to access my Aurora resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Aurora supports these features, see How Amazon Aurora works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Logging and monitoring in Amazon Aurora

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Aurora and your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your Amazon Aurora resources and responding to potential incidents:

Amazon CloudWatch Alarms

> Using Amazon CloudWatch alarms, you watch a single metric over a time period that you specify. If the metric exceeds a given threshold, a notification is sent to an Amazon SNS topic or AWS Auto Scaling policy. CloudWatch alarms do not invoke actions because they are in a particular state. Rather the state must have changed and been maintained for a specified number of periods.

AWS CloudTrail Logs

CloudTrail provides a record of actions taken by a user, role, or an AWS service in Amazon Aurora. CloudTrail captures all API calls for Amazon Aurora as events, including calls from the console and from code calls to Amazon RDS API operations. Using the information collected by CloudTrail, you can determine the request that was made to Amazon Aurora, the IP address from which the request was made, who made the request, when it was made, and additional details. For more information, see Monitoring Amazon Aurora API calls in AWS CloudTrail.

Enhanced Monitoring

Amazon Aurora provides metrics in real time for the operating system (OS) that your DB cluster runs on. You can view the metrics for your DB cluster using the console, or consume the Enhanced Monitoring JSON output from Amazon CloudWatch Logs in a monitoring system of your choice. For more information, see Monitoring OS metrics with Enhanced Monitoring.

Amazon RDS Performance Insights

Performance Insights expands on existing Amazon Aurora monitoring features to illustrate your database's performance and help you analyze any issues that affect it. With the Performance Insights dashboard, you can visualize the database load and filter the load by waits, SQL statements, hosts, or users. For more information, see Monitoring DB load with Performance Insights on Amazon Aurora.

Database Logs

You can view, download, and watch database logs using the AWS Management Console, AWS CLI, or RDS API. For more information, see Monitoring Amazon Aurora log files.

Amazon Aurora Recommendations

Amazon Aurora provides automated recommendations for database resources. These recommendations provide best practice guidance by analyzing DB cluster configuration, usage, and performance data. For more information, see Viewing and responding to Amazon Aurora recommendations.

Amazon Aurora Event Notification

Amazon Aurora uses the Amazon Simple Notification Service (Amazon SNS) to provide notification when an Amazon Aurora event occurs. These notifications can be in any notification form supported by Amazon SNS for an AWS Region, such as an email, a text message, or a call to an HTTP endpoint. For more information, see Working with Amazon RDS event notification.

AWS Trusted Advisor

Trusted Advisor draws upon best practices learned from serving hundreds of thousands of AWS customers. Trusted Advisor inspects your AWS environment and then makes recommendations when opportunities exist to save money, improve system availability and performance, or help close security gaps. All AWS customers have access to five Trusted Advisor checks. Customers with a Business or Enterprise support plan can view all Trusted Advisor checks.

Trusted Advisor has the following Amazon Aurora-related checks:

- Amazon Aurora Idle DB Instances
- Amazon Aurora Security Group Access Risk
- Amazon Aurora Backups
- Amazon Aurora Multi-AZ
- Aurora DB Instance Accessibility

For more information on these checks, see Trusted Advisor best practices (checks).

Database activity streams

Database activity streams can protect your databases from internal threats by controlling DBA access to the database activity streams. Thus, the collection, transmission, storage, and subsequent processing of the database activity stream is beyond the access of the DBAs that manage the database. Database activity streams can provide safeguards for your database and meet compliance and regulatory requirements. For more information, see Monitoring Amazon Aurora with Database Activity Streams.

For more information about monitoring Aurora see Monitoring metrics in an Amazon Aurora cluster.

Compliance validation for Amazon Aurora

Third-party auditors assess the security and compliance of Amazon Aurora as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS services in scope by compliance program. For general information, see AWS compliance programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading reports in AWS Artifact.

Your compliance responsibility when using Amazon Aurora is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and compliance quick start guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- AWS compliance resources – This collection of workbooks and guides that might apply to your industry and location.
- AWS Config – This AWS service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.

Resilience in Amazon Aurora

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS global infrastructure.

In addition to the AWS global infrastructure, Aurora offers features to help support your data resiliency and backup needs.

Backup and restore

Aurora backs up your cluster volume automatically and retains restore data for the length of the *backup retention period*. Aurora backups are continuous and incremental so you can quickly restore to any point within the backup retention period. No performance impact or interruption of database service occurs as backup data is being written. You can specify a backup retention period, from 1 to 35 days, when you create or modify a DB cluster.

If you want to retain a backup beyond the backup retention period, you can also take a snapshot of the data in your cluster volume. Aurora retains incremental restore data for the entire backup retention period. Thus, you need to create a snapshot only for data that you want to retain beyond the backup retention period. You can create a new DB cluster from the snapshot.

You can recover your data by creating a new Aurora DB cluster from the backup data that Aurora retains, or from a DB cluster snapshot that you have saved. You can quickly create a new copy of a DB cluster from backup data to any point in time during your backup retention period. The continuous and incremental nature of Aurora backups during the backup retention period means you don't need to take frequent snapshots of your data to improve restore times.

For more information, see Backing up and restoring an Amazon Aurora DB cluster.

Replication

Aurora Replicas are independent endpoints in an Aurora DB cluster, best used for scaling read operations and increasing availability. Up to 15 Aurora Replicas can be distributed across the

Availability Zones that a DB cluster spans within an AWS Region. The DB cluster volume is made up of multiple copies of the data for the DB cluster. However, the data in the cluster volume is represented as a single, logical volume to the primary DB instance and to Aurora Replicas in the DB cluster. If the primary DB instance fails, an Aurora Replica is promoted to be the primary DB instance.

Aurora also supports replication options that are specific to Aurora MySQL and Aurora PostgreSQL.

For more information, see Replication with Amazon Aurora.

Failover

Aurora stores copies of the data in a DB cluster across multiple Availability Zones in a single AWS Region. This storage occurs regardless of whether the DB instances in the DB cluster span multiple Availability Zones. When you create Aurora Replicas across Availability Zones, Aurora automatically provisions and maintains them synchronously. The primary DB instance is synchronously replicated across Availability Zones to Aurora Replicas to provide data redundancy, eliminate I/O freezes, and minimize latency spikes during system backups. Running a DB cluster with high availability can enhance availability during planned system maintenance, and help protect your databases against failure and Availability Zone disruption.

For more information, see High availability for Amazon Aurora.

Infrastructure security in Amazon Aurora

As a managed service, Amazon Relational Database Service is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access Amazon RDS through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

In addition, Aurora offers features to help support infrastructure security.

Security groups

Security groups control the access that traffic has in and out of a DB cluster. By default, network access is turned off to a DB cluster. You can specify rules in a security group that allow access from an IP address range, port, or security group. After ingress rules are configured, the same rules apply to all DB clusters that are associated with that security group.

For more information, see Controlling access with security groups.

Public accessibility

When you launch a DB instance inside a virtual private cloud (VPC) based on the Amazon VPC service, you can turn on or off public accessibility for that DB instance. To designate whether the DB instance that you create has a DNS name that resolves to a public IP address, you use the *Public accessibility* parameter. By using this parameter, you can designate whether there is public access to the DB instance. You can modify a DB instance to turn on or off public accessibility by modifying the *Public accessibility* parameter.

For more information, see Hiding a DB cluster in a VPC from the internet.

Note

If your DB instance is in a VPC but isn't publicly accessible, you can also use an AWS Site-to-Site VPN connection or an AWS Direct Connect connection to access it from a private network. For more information, see Internetwork traffic privacy.

Amazon RDS API and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and Amazon RDS API endpoints by creating an *interface VPC endpoint*. Interface endpoints are powered by AWS PrivateLink.

AWS PrivateLink enables you to privately access Amazon RDS API operations without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. DB instances in your VPC don't need public IP addresses to communicate with Amazon RDS API endpoints to launch, modify, or terminate DB instances and DB clusters. Your DB instances also don't need public IP addresses to use any of the available RDS API operations. Traffic between your VPC and Amazon RDS doesn't leave the Amazon network.

Each interface endpoint is represented by one or more elastic network interfaces in your subnets. For more information on elastic network interfaces, see Elastic network interfaces in the *Amazon EC2 User Guide.*

For more information about VPC endpoints, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*. For more information about RDS API operations, see Amazon RDS API Reference.

You don't need an interface VPC endpoint to connect to a DB cluster. For more information, see Scenarios for accessing a DB cluster in a VPC.

Considerations for VPC endpoints

Before you set up an interface VPC endpoint for Amazon RDS API endpoints, ensure that you review Interface endpoint properties and limitations in the *Amazon VPC User Guide*.

All RDS API operations relevant to managing Amazon Aurora resources are available from your VPC using AWS PrivateLink.

VPC endpoint policies are supported for RDS API endpoints. By default, full access to RDS API operations is allowed through the endpoint. For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Availability

Amazon RDS API currently supports VPC endpoints in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (N. California)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- Asia Pacific (Mumbai)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- Canada West (Calgary)
- China (Beijing)
- China (Ningxia)
- Europe (Frankfurt)
- Europe (Zurich)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Europe (Stockholm)
- Europe (Milan)
- Israel (Tel Aviv)
- Middle East (Bahrain)
- South America (São Paulo)
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)

Creating an interface VPC endpoint for Amazon RDS API

You can create a VPC endpoint for the Amazon RDS API using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see [Creating an interface endpoint](#) in the *Amazon VPC User Guide*.

Create a VPC endpoint for Amazon RDS API using the service name `com.amazonaws.region.rds`.

Excluding AWS Regions in China, if you enable private DNS for the endpoint, you can make API requests to Amazon RDS with the VPC endpoint using its default DNS name for the AWS Region, for example `rds.us-east-1.amazonaws.com`. For the China (Beijing) and China (Ningxia) AWS Regions, you can make API requests with the VPC endpoint using `rds-api.cn-north-1.amazonaws.com.cn` and `rds-api.cn-northwest-1.amazonaws.com.cn`, respectively.

For more information, see [Accessing a service through an interface endpoint](#) in the *Amazon VPC User Guide*.

Creating a VPC endpoint policy for Amazon RDS API

You can attach an endpoint policy to your VPC endpoint that controls access to Amazon RDS API. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see [Controlling access to services with VPC endpoints](#) in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for Amazon RDS API actions

The following is an example of an endpoint policy for Amazon RDS API. When attached to an endpoint, this policy grants access to the listed Amazon RDS API actions for all principals on all resources.

```
{
   "Statement":[
      {
         "Principal":"*",
         "Effect":"Allow",
         "Action":[
            "rds:CreateDBInstance",
            "rds:ModifyDBInstance",
            "rds:CreateDBSnapshot"
         ],
         "Resource":"*"
      }
   ]
}
```

Example: VPC endpoint policy that denies all access from a specified AWS account

The following VPC endpoint policy denies AWS account `123456789012` all access to resources using the endpoint. The policy allows all actions from other accounts.

```
{
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Resource": "*",
      "Principal": "*"
    },
    {
      "Action": "*",
      "Effect": "Deny",
      "Resource": "*",
      "Principal": { "AWS": [ "123456789012" ] }
     }
   ]
}
```

Security best practices for Amazon Aurora

Use AWS Identity and Access Management (IAM) accounts to control access to Amazon RDS API operations, especially operations that create, modify, or delete Amazon Aurora resources. Such resources include DB clusters, security groups, and parameter groups. Also use IAM to control actions that perform common administrative actions such as backing up and restoring DB clusters.

- Create an individual user for each person who manages Amazon Aurora resources, including yourself. Don't use AWS root credentials to manage Amazon Aurora resources.
- Grant each user the minimum set of permissions required to perform his or her duties.
- Use IAM groups to effectively manage permissions for multiple users.
- Rotate your IAM credentials regularly.
- Configure AWS Secrets Manager to automatically rotate the secrets for Amazon Aurora. For more information, see Rotating your AWS Secrets Manager secrets in the *AWS Secrets Manager User Guide*. You can also retrieve the credential from AWS Secrets Manager programmatically. For more information, see Retrieving the secret value in the *AWS Secrets Manager User Guide*.

For more information about Amazon Aurora security, see Security in Amazon Aurora. For more information about IAM, see AWS Identity and Access Management. For information on IAM best practices, see IAM best practices.

AWS Security Hub uses security controls to evaluate resource configurations and security standards to help you comply with various compliance frameworks. For more information about using Security Hub to evaluate RDS resources, see Amazon Relational Database Service controls in the AWS Security Hub User Guide.

You can monitor your usage of RDS as it relates to security best practices by using Security Hub. For more information, see What is AWS Security Hub?.

Use the AWS Management Console, the AWS CLI, or the RDS API to change the password for your master user. If you use another tool, such as a SQL client, to change the master user password, it might result in privileges being revoked for the user unintentionally.

Amazon GuardDuty is a continuous security monitoring service that analyzes and processes various data sources, including Amazon RDS login activity. It uses threat intelligence feeds and machine learning to identify unexpected, potentially unauthorized, suspicious login behavior, and malicious activity within your AWS environment.

When Amazon GuardDuty RDS Protection detects a potentially suspicious or anomalous login attempt that indicates a threat to your database, GuardDuty generates a new finding with details about the potentially compromised database. For more information, see Monitoring threats with Amazon GuardDuty RDS Protection.

Controlling access with security groups

VPC security groups control the access that traffic has in and out of a DB cluster. By default, network access is turned off for a DB cluster. You can specify rules in a security group that allow access from an IP address range, port, or security group. After ingress rules are configured, the same rules apply to all DB clusters that are associated with that security group. You can specify up to 20 rules in a security group.

Overview of VPC security groups

Each VPC security group rule makes it possible for a specific source to access a DB cluster in a VPC that is associated with that VPC security group. The source can be a range of addresses (for example, 203.0.113.0/24), or another VPC security group. By specifying a VPC security group as the source, you allow incoming traffic from all instances (typically application servers) that use the source VPC security group. VPC security groups can have rules that govern both inbound and outbound traffic. However, the outbound traffic rules typically don't apply to DB clusters. Outbound traffic rules apply only if the DB cluster acts as a client. You must use the Amazon EC2 API or the **Security Group** option on the VPC console to create VPC security groups.

When you create rules for your VPC security group that allow access to the clusters in your VPC, you must specify a port for each range of addresses that the rule allows access for. For example, if you want to turn on Secure Shell (SSH) access for instances in the VPC, create a rule allowing access to TCP port 22 for the specified range of addresses.

You can configure multiple VPC security groups that allow access to different ports for different instances in your VPC. For example, you can create a VPC security group that allows access to TCP port 80 for web servers in your VPC. You can then create another VPC security group that allows access to TCP port 3306 for Aurora MySQL DB instances in your VPC.

> **Note**
>
> In an Aurora DB cluster, the VPC security group associated with the DB cluster is also associated with all of the DB instances in the DB cluster. If you change the VPC security group for the DB cluster or for a DB instance, the change is applied automatically to all of the DB instances in the DB cluster.

For more information on VPC security groups, see Security groups in the *Amazon Virtual Private Cloud User Guide*.

Note

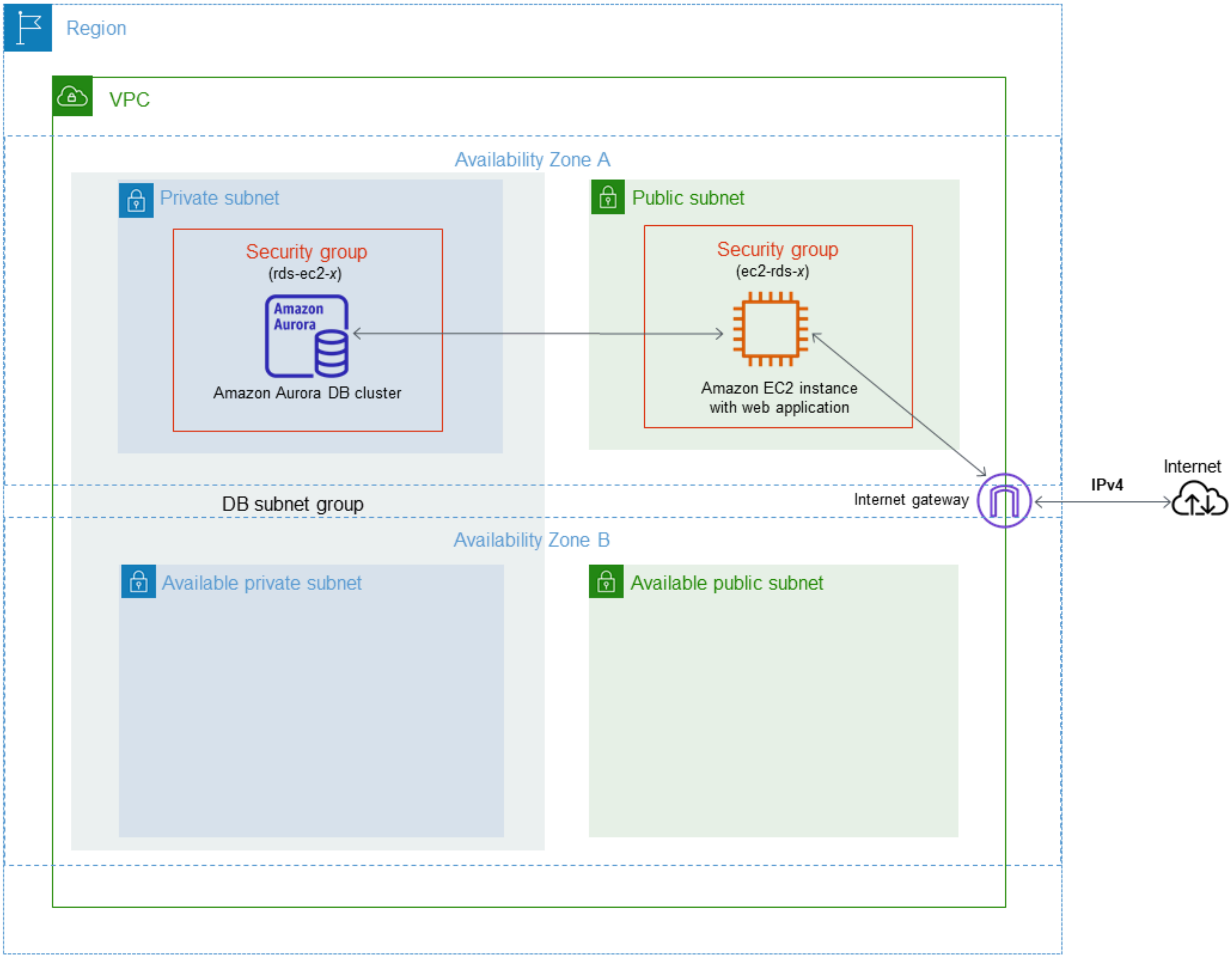
If your DB cluster is in a VPC but isn't publicly accessible, you can also use an AWS Site-to-Site VPN connection or an AWS Direct Connect connection to access it from a private network. For more information, see Internetwork traffic privacy.

Security group scenario

A common use of a DB cluster in a VPC is to share data with an application server running in an Amazon EC2 instance in the same VPC, which is accessed by a client application outside the VPC. For this scenario, you use the RDS and VPC pages on the AWS Management Console or the RDS and EC2 API operations to create the necessary instances and security groups:

1. Create a VPC security group (for example, `sg-0123ec2example`) and define inbound rules that use the IP addresses of the client application as the source. This security group allows your client application to connect to EC2 instances in a VPC that uses this security group.
2. Create an EC2 instance for the application and add the EC2 instance to the VPC security group (`sg-0123ec2example`) that you created in the previous step.
3. Create a second VPC security group (for example, `sg-6789rdsexample`) and create a new rule by specifying the VPC security group that you created in step 1 (`sg-0123ec2example`) as the source.
4. Create a new DB cluster and add the DB cluster to the VPC security group (`sg-6789rdsexample`) that you created in the previous step. When you create the DB cluster, use the same port number as the one specified for the VPC security group (`sg-6789rdsexample`) rule that you created in step 3.

The following diagram shows this scenario.



For detailed instructions about configuring a VPC for this scenario, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only). For more information about using a VPC, see Amazon VPC VPCs and Amazon Aurora.

Creating a VPC security group

You can create a VPC security group for a DB instance by using the VPC console. For information about creating a security group, see Provide access to the DB cluster in the VPC by creating a security group and Security groups in the *Amazon Virtual Private Cloud User Guide*.

Associating a security group with a DB cluster

You can associate a security group with a DB cluster by using **Modify cluster** on the RDS console, the `ModifyDBCluster` Amazon RDS API, or the `modify-db-cluster` AWS CLI command.

The following CLI example associates a specific VPC group and removes DB security groups from the DB cluster

```
aws rds modify-db-cluster --db-cluster-identifier dbName --vpc-security-group-ids sg-ID
```

For information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

Master user account privileges

When you create a new DB cluster, the default master user that you use gets certain privileges for that DB cluster. You can't change the master user name after the DB cluster is created.

⚠ **Important**

We strongly recommend that you do not use the master user directly in your applications. Instead, adhere to the best practice of using a database user created with the minimal privileges required for your application.

ⓘ **Note**

If you accidentally delete the permissions for the master user, you can restore them by modifying the DB cluster and setting a new master user password. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

The following table shows the privileges and database roles the master user gets for each of the database engines.

Database engine	System privilege	Database role
Aurora MySQL	Version 2: ALTER, ALTER ROUTINE, CREATE, CREATE ROUTINE, CREATE TEMPORARY TABLES, CREATE USER, CREATE VIEW, DELETE, DROP, EVENT, EXECUTE, GRANT OPTION, INDEX, INSERT, LOAD FROM S3, LOCK TABLES, PROCESS, REFERENCES, RELOAD, REPLICATION CLIENT, REPLICATION SLAVE, SELECT, SELECT INTO S3, SHOW DATABASES, SHOW VIEW, TRIGGER, UPDATE	—
	Version 3: ALTER, APPLICATION_PASSWORD_ADMIN, ALTER ROUTINE, CONNECTION_ADMIN, CREATE, CREATE ROLE, CREATE ROUTINE, CREATE TEMPORARY TABLES, CREATE USER, CREATE VIEW, DELETE, DROP, DROP ROLE, EVENT, EXECUTE, INDEX, INSERT, LOCK TABLES, PROCESS, REFERENCES, RELOAD, REPLICATION CLIENT, REPLICATION SLAVE, ROLE_ADMIN, SET_USER_ID, SELECT, SHOW DATABASES, SHOW_ROUTINE (Aurora MySQL version 3.04 and higher), SHOW VIEW, TRIGGER, UPDATE, XA_RECOVER_ADMIN	rds_superuser_role For more information about rds_superuser_role, see Role-based privilege model.
Aurora PostgreSQL	LOGIN, NOSUPERUSER, INHERIT, CREATEDB, CREATEROLE, NOREPLICATION, VALID UNTIL 'infinity'	RDS_SUPERUSER For more information about RDS_SUPERUSER, see Understanding PostgreSQL roles and permissions.

Using service-linked roles for Amazon Aurora

Amazon Aurora uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Amazon Aurora. Service-linked roles are predefined by Amazon Aurora and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes using Amazon Aurora easier because you don't have to manually add the necessary permissions. Amazon Aurora defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon Aurora can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your Amazon Aurora resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for Amazon Aurora

Amazon Aurora uses the service-linked role named AWSServiceRoleForRDS to allow Amazon RDS to call AWS services on behalf of your DB clusters.

The AWSServiceRoleForRDS service-linked role trusts the following services to assume the role:

- `rds.amazonaws.com`

This service-linked role has a permissions policy attached to it called `AmazonRDSServiceRolePolicy` that grants it permissions to operate in your account. The role permissions policy allows Amazon Aurora to complete the following actions on the specified resources:

For more information about this policy, including the JSON policy document, see AmazonRDSServiceRolePolicy in the *AWS Managed Policy Reference Guide*.

Note

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. If you encounter the following error message:

Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later.

Make sure you have the following permissions enabled:

```
{
    "Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName":"rds.amazonaws.com"
        }
    }
}
```

For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for Amazon Aurora

You don't need to manually create a service-linked role. When you create a DB cluster, Amazon Aurora creates the service-linked role for you.

Important

If you were using the Amazon Aurora service before December 1, 2017, when it began supporting service-linked roles, then Amazon Aurora created the AWSServiceRoleForRDS role in your account. To learn more, see A new role appeared in my AWS account.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create a DB cluster, Amazon Aurora creates the service-linked role for you again.

Editing a service-linked role for Amazon Aurora

Amazon Aurora does not allow you to edit the AWSServiceRoleForRDS service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for Amazon Aurora

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your DB clusters before you can delete the service-linked role.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the AWSServiceRoleForRDS role.
3. On the **Summary** page for the chosen role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review recent activity for the service-linked role.

> **Note**
>
> If you are unsure whether Amazon Aurora is using the AWSServiceRoleForRDS role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the AWS Regions where the role is being used. If the role is being used, then you must wait for the session to end before you can delete the role. You cannot revoke the session for a service-linked role.

If you want to remove the AWSServiceRoleForRDS role, you must first delete *all* of your DB clusters.

Deleting all of your clusters

Use one of the following procedures to delete a single cluster. Repeat the procedure for each of your clusters.

To delete a cluster (console)

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the **Databases** list, choose the cluster that you want to delete.
3. For **Cluster Actions**, choose **Delete**.
4. Choose **Delete**.

To delete a cluster (CLI)

See `delete-db-cluster` in the *AWS CLI Command Reference*.

To delete a cluster (API)

See `DeleteDBCluster` in the *Amazon RDS API Reference*.

You can use the IAM console, the IAM CLI, or the IAM API to delete the AWSServiceRoleForRDS service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Amazon VPC VPCs and Amazon Aurora

Amazon Virtual Private Cloud (Amazon VPC) makes it possible for you to launch AWS resources, such as Aurora DB clusters, into a virtual private cloud (VPC).

When you use a VPC, you have control over your virtual networking environment. You can choose your own IP address range, create subnets, and configure routing and access control lists. There is no additional cost to run your DB cluster in a VPC.

Accounts have a default VPC. All new DB clusters are created in the default VPC unless you specify otherwise.

Topics

Following, you can find a discussion about VPC functionality relevant to Amazon Aurora DB clusters. For more information about Amazon VPC, see Amazon VPC Getting Started Guide and Amazon VPC User Guide.

Working with a DB cluster in a VPC

Your DB cluster is in a virtual private cloud (VPC). A VPC is a virtual network that is logically isolated from other virtual networks in the AWS Cloud. Amazon VPC makes it possible for you to launch AWS resources, such as an Amazon Aurora DB cluster or Amazon EC2 instance, into a VPC. The VPC can either be a default VPC that comes with your account or one that you create. All VPCs are associated with your AWS account.

Your default VPC has three subnets that you can use to isolate resources inside the VPC. The default VPC also has an internet gateway that can be used to provide access to resources inside the VPC from outside the VPC.

For a list of scenarios involving Amazon Aurora DB clusters in a VPC , see Scenarios for accessing a DB cluster in a VPC.

Topics

- Working with a DB cluster in a VPC
- Working with DB subnet groups
- Shared subnets
- Amazon Aurora IP addressing
- Hiding a DB cluster in a VPC from the internet
- Creating a DB cluster in a VPC

In the following tutorials, you can learn to create a VPC that you can use for a common Amazon Aurora scenario:

- Tutorial: Create a VPC for use with a DB cluster (IPv4 only)
- Tutorial: Create a VPC for use with a DB cluster (dual-stack mode)

Working with a DB cluster in a VPC

Here are some tips on working with a DB cluster in a VPC:

- Your VPC must have at least two subnets. These subnets must be in two different Availability Zones in the AWS Region where you want to deploy your DB cluster. A *subnet* is a segment of a VPC's IP address range that you can specify and that you can use to group DB clusters based on your security and operational needs.
- If you want your DB cluster in the VPC to be publicly accessible, make sure to turn on the VPC attributes *DNS hostnames* and *DNS resolution*.
- Your VPC must have a DB subnet group that you create. You create a DB subnet group by specifying the subnets you created. Amazon Aurora chooses a subnet and an IP address within that subnet to associate with the primary DB instance in your DB cluster. The primary DB instance uses the Availability Zone that contains the subnet.
- Your VPC must have a VPC security group that allows access to the DB cluster.

  For more information, see Scenarios for accessing a DB cluster in a VPC.
- The CIDR blocks in each of your subnets must be large enough to accommodate spare IP addresses for Amazon Aurora to use during maintenance activities, including failover and compute scaling. For example, a range such as 10.0.0.0/24 and 10.0.1.0/24 is typically large enough.

- A VPC can have an *instance tenancy* attribute of either *default* or *dedicated*. All default VPCs have the instance tenancy attribute set to default, and a default VPC can support any DB instance class.

  If you choose to have your DB cluster in a dedicated VPC where the instance tenancy attribute is set to dedicated, the DB instance class of your DB cluster must be one of the approved Amazon EC2 dedicated instance types. For example, the r5.large EC2 dedicated instance corresponds to the db.r5.large DB instance class. For information about instance tenancy in a VPC, see Dedicated instances in the *Amazon Elastic Compute Cloud User Guide*.

  For more information about the instance types that can be in a dedicated instance, see Amazon EC2 dedicated instances on the EC2 pricing page.

> **Note**
>
> When you set the instance tenancy attribute to dedicated for a DB cluster, it doesn't guarantee that the DB cluster will run on a dedicated host.

Working with DB subnet groups

Subnets are segments of a VPC's IP address range that you designate to group your resources based on security and operational needs. A *DB subnet group* is a collection of subnets (typically private) that you create in a VPC and that you then designate for your DB clusters. By using a DB subnet group, you can specify a particular VPC when creating DB clusters using the AWS CLI or RDS API. If you use the console, you can choose the VPC and subnet groups you want to use.

Each DB subnet group should have subnets in at least two Availability Zones in a given AWS Region. When creating a DB cluster in a VPC, you choose a DB subnet group for it. From the DB subnet group, Amazon Aurora chooses a subnet and an IP address within that subnet to associate with the primary DB instance in your DB cluster. The DB uses the Availability Zone that contains the subnet.

The subnets in a DB subnet group are either public or private. The subnets are public or private, depending on the configuration that you set for their network access control lists (network ACLs) and routing tables. For a DB cluster to be publicly accessible, all of the subnets in its DB subnet group must be public. If a subnet that's associated with a publicly accessible DB cluster changes from public to private, it can affect DB cluster availability.

To create a DB subnet group that supports dual-stack mode, make sure that each subnet that you add to the DB subnet group has an Internet Protocol version 6 (IPv6) CIDR block associated with it. For more information, see Amazon Aurora IP addressing and Migrating to IPv6 in the *Amazon VPC User Guide.*

When Amazon Aurora creates a DB cluster in a VPC, it assigns a network interface to your DB cluster by using an IP address from your DB subnet group. However, we strongly recommend that you use the Domain Name System (DNS) name to connect to your DB cluster. We recommend this because the underlying IP address changes during failover.

> **Note**
>
> For each DB cluster that you run in a VPC, make sure to reserve at least one address in each subnet in the DB subnet group for use by Amazon Aurora for recovery actions.

Shared subnets

You can create a DB cluster in a shared VPC.

Some considerations to keep in mind while using shared VPCs:

- You can move a DB cluster from a shared VPC subnet to a non-shared VPC subnet and vice-versa.
- Participants in a shared VPC must create a security group in the VPC to allow them to create a DB cluster.
- Owners and participants in a shared VPC can access the database by using SQL queries. However, only the creator of a resource can make any API calls on the resource.

Amazon Aurora IP addressing

IP addresses enable resources in your VPC to communicate with each other, and with resources over the internet. Amazon Aurora supports both IPv4 and IPv6 addressing protocols. By default, Amazon Aurora and Amazon VPC use the IPv4 addressing protocol. You can't turn off this behavior. When you create a VPC, make sure to specify an IPv4 CIDR block (a range of private IPv4 addresses). You can optionally assign an IPv6 CIDR block to your VPC and subnets, and assign IPv6 addresses from that block to DB clusters in your subnet.

Support for the IPv6 protocol expands the number of supported IP addresses. By using the IPv6 protocol, you ensure that you have sufficient available addresses for the future growth of the internet. New and existing RDS resources can use IPv4 and IPv6 addresses within your VPC. Configuring, securing, and translating network traffic between the two protocols used in different parts of an application can cause operational overhead. You can standardize on the IPv6 protocol for Amazon RDS resources to simplify your network configuration.

Topics

IPv4 addresses

When you create a VPC, you must specify a range of IPv4 addresses for the VPC in the form of a CIDR block, such as `10.0.0.0/16`. A *DB subnet group* defines the range of IP addresses in this CIDR block that a DB cluster can use. These IP addresses can be private or public.

A private IPv4 address is an IP address that's not reachable over the internet. You can use private IPv4 addresses for communication between your DB cluster and other resources, such as Amazon EC2 instances, in the same VPC. Each DB cluster has a private IP address for communication in the VPC.

A public IP address is an IPv4 address that's reachable from the internet. You can use public addresses for communication between your DB cluster and resources on the internet, such as a SQL client. You control whether your DB cluster receives a public IP address.

For a tutorial that shows you how to create a VPC with only private IPv4 addresses that you can use for a common Amazon Aurora scenario, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

IPv6 addresses

You can optionally associate an IPv6 CIDR block with your VPC and subnets, and assign IPv6 addresses from that block to the resources in your VPC. Each IPv6 address is globally unique.

The IPv6 CIDR block for your VPC is automatically assigned from Amazon's pool of IPv6 addresses. You can't choose the range yourself.

When connecting to an IPv6 address, make sure that the following conditions are met:

- The client is configured so that client to database traffic over IPv6 is allowed.
- RDS security groups used by the DB instance are configured correctly so that client to database traffic over IPv6 is allowed.
- The client operating system stack allows traffic on the IPv6 address, and operating system drivers and libraries are configured to choose the correct default DB instance endpoint (either IPv4 or IPv6).

For more information about IPv6, see IP Addressing in the *Amazon VPC User Guide*.

Dual-stack mode

When a DB cluster can communicate over both the IPv4 and IPv6 addressing protocols, it's running in dual-stack mode. So, resources can communicate with the DB cluster over IPv4, IPv6, or both. RDS disables Internet Gateway access for IPv6 endpoints of private dual-stack mode DB instances. RDS does this to ensure that your IPv6 endpoints are private and can only be accessed from within your VPC.

Topics

For a tutorial that shows you how to create a VPC with both IPv4 and IPv6 addresses that you can use for a common Amazon Aurora scenario, see Tutorial: Create a VPC for use with a DB cluster (dual-stack mode).

Dual-stack mode and DB subnet groups

To use dual-stack mode, make sure that each subnet in the DB subnet group that you associate with the DB cluster has an IPv6 CIDR block associated with it. You can create a new DB subnet group or modify an existing DB subnet group to meet this requirement. After a DB cluster is in dual-stack mode, clients can connect to it normally. Make sure that client security firewalls and RDS DB instance security groups are accurately configured to allow traffic over IPv6. To connect, clients use the DB cluster primary instance's endpoint. Client applications can specify which protocol is preferred when connecting to a database. In dual-stack mode, the DB cluster detects the client's preferred network protocol, either IPv4 or IPv6, and uses that protocol for the connection.

If a DB subnet group stops supporting dual-stack mode because of subnet deletion or CIDR disassociation, there's a risk of an incompatible network state for DB instances that are associated with the DB subnet group. Also, you can't use the DB subnet group when you create a new dual-stack mode DB cluster.

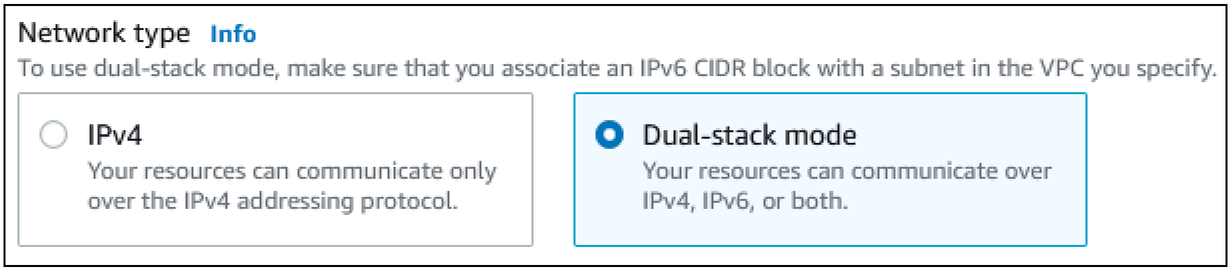
To determine whether a DB subnet group supports dual-stack mode by using the AWS Management Console, view the **Network type** on the details page of the DB subnet group. To determine whether a DB subnet group supports dual-stack mode by using the AWS CLI, run the describe-db-subnet-groups command and view `SupportedNetworkTypes` in the output.

Read replicas are treated as independent DB instances and can have a network type that's different from the primary DB instance. If you change the network type of a read replica's primary DB instance, the read replica isn't affected. When you are restoring a DB instance, you can restore it to any network type that's supported.

Working with dual-stack mode DB instances

When you create or modify a DB cluster, you can specify dual-stack mode to allow your resources to communicate with your DB cluster over IPv4, IPv6, or both.

When you use the AWS Management Console to create or modify a DB instance, you can specify dual-stack mode in the **Network type** section. The following image shows the **Network type** section in the console.



Network type Info

To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

○ IPv4
Your resources can communicate only over the IPv4 addressing protocol.

● Dual-stack mode
Your resources can communicate over IPv4, IPv6, or both.

When you use the AWS CLI to create or modify a DB cluster, set the `--network-type` option to `DUAL` to use dual-stack mode. When you use the RDS API to create or modify a DB cluster, set the `NetworkType` parameter to `DUAL` to use dual-stack mode. When you are modifying the network type of a DB instance, downtime is possible. If dual-stack mode isn't supported by the specified DB engine version or DB subnet group, the `NetworkTypeNotSupported` error is returned.

For more information about creating a DB cluster, see Creating an Amazon Aurora DB cluster. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

To determine whether a DB cluster is in dual-stack mode by using the console, view the **Network type** on the **Connectivity & security** tab for the DB cluster.

Modifying IPv4-only DB clusters to use dual-stack mode

You can modify an IPv4-only DB cluster to use dual-stack mode. To do so, change the network type of the DB cluster. The modification might result in downtime.

It is recommended that you change the network type of your Amazon Aurora DB clusters during a maintenance window. Currently, setting the network type of new instances to dual-stack mode isn't supported. You can set network type manually by using the `modify-db-cluster` command.

Before modifying a DB cluster to use dual-stack mode, make sure that its DB subnet group supports dual-stack mode. If the DB subnet group associated with the DB cluster doesn't support dual-stack mode, specify a different DB subnet group that supports it when you modify the DB cluster. Modifying the DB subnet group of a DB cluster can cause downtime.

If you modify the DB subnet group of a DB cluster before you change the DB cluster to use dual-stack mode, make sure that the DB subnet group is valid for the DB cluster before and after the change.

We recommend that you run the modify-db-cluster API with only the `--network-type` parameter with value `DUAL` to change the network of an Amazon Aurora cluster to dual-stack mode. Adding other parameters along with the `--network-type` parameter in the same API call could result in downtime.

If you can't connect to the DB cluster after the change, make sure that the client and database security firewalls and route tables are accurately configured to allow traffic to the database on the selected network (either IPv4 or IPv6). You might also need to modify operating system parameter, libraries, or drivers to connect using an IPv6 address.

To modify an IPv4-only DB cluster to use dual-stack mode

1. Modify a DB subnet group to support dual-stack mode, or create a DB subnet group that supports dual-stack mode:

    a. Associate an IPv6 CIDR block with your VPC.

    For instructions, see [Add an IPv6 CIDR block to your VPC](#) in the *Amazon VPC User Guide*.

  b. Attach the IPv6 CIDR block to all of the subnets in your the DB subnet group.

     For instructions, see [Add an IPv6 CIDR block to your subnet](#) in the *Amazon VPC User Guide*.

  c. Confirm that the DB subnet group supports dual-stack mode.

     If you are using the AWS Management Console, select the DB subnet group, and make sure that the **Supported network types** value is **Dual, IPv4**.

     If you are using the AWS CLI, run the [describe-db-subnet-groups](#) command, and make sure that the `SupportedNetworkType` value for the DB instance is `Dual, IPv4`.

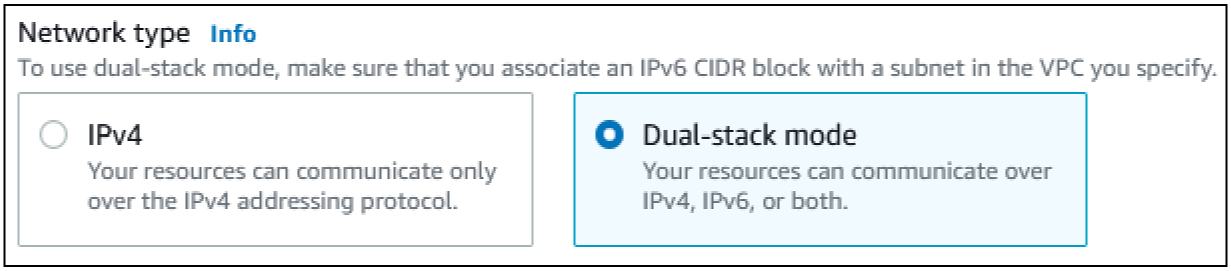
2. Modify the security group associated with the DB cluster to allow IPv6 connections to the database, or create a new security group that allows IPv6 connections.

   For instructions, see [Security group rules](#) in the *Amazon VPC User Guide*.

3. Modify the DB cluster to support dual-stack mode. To do so, set the **Network type** to **Dual-stack mode**.

   If you are using the console, make sure that the following settings are correct:

   - **Network type – Dual-stack mode**



Network type Info

To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

○ IPv4
Your resources can communicate only over the IPv4 addressing protocol.

● Dual-stack mode
Your resources can communicate over IPv4, IPv6, or both.

   - **DB subnet group –** The DB subnet group that you configured in a previous step
   - **Security group –** The security that you configured in a previous step

   If you are using the AWS CLI, make sure that the following settings are correct:

   - `--network-type` – `dual`
   - `--db-subnet-group-name` – The DB subnet group that you configured in a previous step

- `--vpc-security-group-ids` – The VPC security group that you configured in a previous step

For example:

```
aws rds modify-db-cluster --db-cluster-identifier my-cluster --network-type "DUAL"
```

4. Confirm that the DB cluster supports dual-stack mode.

   If you are using the console, choose the **Configuration** tab for the DB cluster. On that tab, make sure that the **Network type** value is **Dual-stack mode**.

   If you are using the AWS CLI, run the describe-db-clusters command, and make sure that the `NetworkType` value for the DB cluster is `dual`.

   Run the `dig` command on the writer DB instance endpoint to identify the IPv6 address associated with it.

   ```
   dig db-instance-endpoint AAAA
   ```

   Use the writer DB instance endpoint, not the IPv6 address, to connect to the DB cluster.

Availability of dual-stack network DB clusters

The following DB engine versions support dual-stack network DB clusters, except in the Asia Pacific (Hyderabad), Asia Pacific (Melbourne), Canada West (Calgary), Europe (Spain), Europe (Zurich), Israel (Tel Aviv), and Middle East (UAE) Regions:

- Aurora MySQL versions:
  - 3.02 and higher 3 versions
  - 2.09.1 and higher 2 versions

  For more information about Aurora MySQL versions, see the *Release Notes for Aurora MySQL*.
- Aurora PostgreSQL versions:
  - 14.3 and higher 14 versions
  - 13.7 and higher 13 versions

For more information about Aurora PostgreSQL versions, see the *Release Notes for Aurora PostgreSQL*.

Limitations for dual-stack network DB clusters

The following limitations apply to dual-stack network DB clusters:

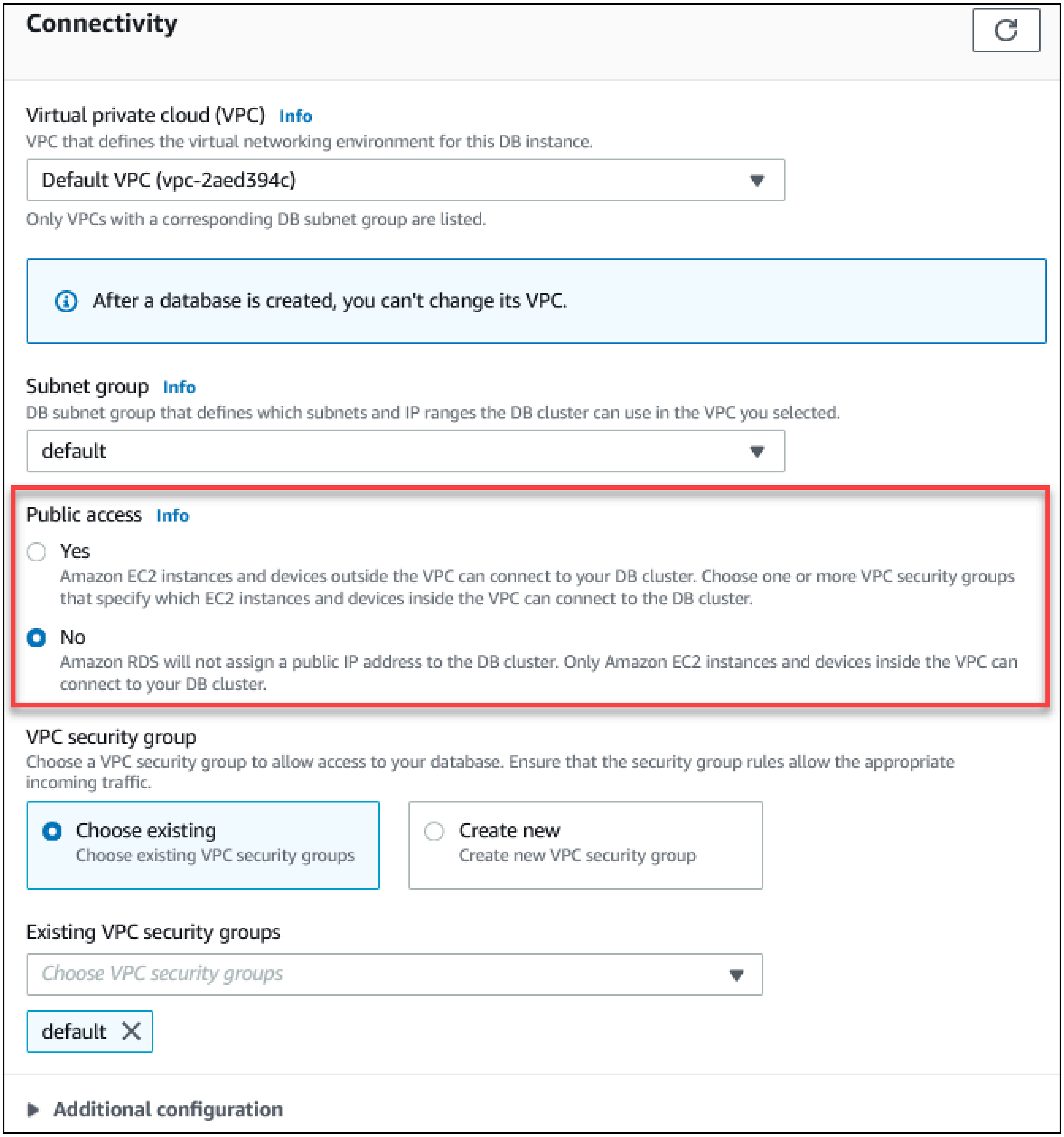
- DB clusters can't use the IPv6 protocol exclusively. They can use IPv4 exclusively, or they can use the IPv4 and IPv6 protocol (dual-stack mode).
- Amazon RDS doesn't support native IPv6 subnets.
- DB clusters that use dual-stack mode must be private. They can't be publicly accessible.
- Dual-stack mode doesn't support the db.r3 DB instance classes.
- You can't use RDS Proxy with dual-stack mode DB clusters.

Hiding a DB cluster in a VPC from the internet

One common Amazon Aurora scenario is to have a VPC in which you have an EC2 instance with a public-facing web application and a DB cluster with a database that isn't publicly accessible. For example, you can create a VPC that has a public subnet and a private subnet. Amazon EC2 instances that function as web servers can be deployed in the public subnet. The DB clusters are deployed in the private subnet. In such a deployment, only the web servers have access to the DB clusters. For an illustration of this scenario, see A DB cluster in a VPC accessed by an EC2 instance in the same VPC.

When you launch a DB cluster inside a VPC, the DB cluster has a private IP address for traffic inside the VPC. This private IP address isn't publicly accessible. You can use the **Public access** option to designate whether the DB cluster also has a public IP address in addition to the private IP address. If the DB cluster is designated as publicly accessible, its DNS endpoint resolves to the private IP address from within the VPC. It resolves to the public IP address from outside of the VPC. Access to the DB cluster is ultimately controlled by the security group it uses. That public access is not permitted if the security group assigned to the DB cluster doesn't include inbound rules that permit it. In addition, for a DB cluster to be publicly accessible, the subnets in its DB subnet group must have an internet gateway. For more information, see Can't connect to Amazon RDS DB instance

You can modify a DB cluster to turn on or off public accessibility by modifying the **Public access** option. The following illustration shows the **Public access** option in the **Additional connectivity configuration** section. To set the option, open the **Additional connectivity configuration** section in the **Connectivity** section.



Connectivity

Virtual private cloud (VPC) Info

VPC that defines the virtual networking environment for this DB instance.

Default VPC (vpc-2aed394c) ▼

Only VPCs with a corresponding DB subnet group are listed.

ⓘ After a database is created, you can't change its VPC.

Subnet group Info

DB subnet group that defines which subnets and IP ranges the DB cluster can use in the VPC you selected.

default ▼

Public access Info

○ Yes

Amazon EC2 instances and devices outside the VPC can connect to your DB cluster. Choose one or more VPC security groups that specify which EC2 instances and devices inside the VPC can connect to the DB cluster.

● No

Amazon RDS will not assign a public IP address to the DB cluster. Only Amazon EC2 instances and devices inside the VPC can connect to your DB cluster.

VPC security group

Choose a VPC security group to allow access to your database. Ensure that the security group rules allow the appropriate incoming traffic.

● Choose existing

Choose existing VPC security groups

○ Create new

Create new VPC security group

Existing VPC security groups

Choose VPC security groups ▼

default ✕

▶ Additional configuration

For information about modifying a DB instance to set the **Public access** option, see Modifying a DB instance in a DB cluster.

Creating a DB cluster in a VPC

The following procedures help you create a DB cluster in a VPC. To use the default VPC, you can begin with step 2, and use the VPC and DB subnet group have already been created for you. If you want to create an additional VPC, you can create a new VPC.

> **Note**
>
> If you want your DB cluster in the VPC to be publicly accessible, you must update the DNS information for the VPC by enabling the VPC attributes *DNS hostnames* and *DNS resolution*. For information about updating the DNS information for a VPC instance, see Updating DNS support for your VPC.

Follow these steps to create a DB instance in a VPC:

- Step 1: Create a VPC
- Step 2: Create a DB subnet group
- Step 3: Create a VPC security group
- Step 4: Create a DB instance in the VPC

Step 1: Create a VPC

Create a VPC with subnets in at least two Availability Zones. You use these subnets when you create a DB subnet group. If you have a default VPC, a subnet is automatically created for you in each Availability Zone in the AWS Region.

For more information, see Create a VPC with private and public subnets, or see Create a VPC in the *Amazon VPC User Guide*.

Step 2: Create a DB subnet group

A DB subnet group is a collection of subnets (typically private) that you create for a VPC and that you then designate for your DB clusters. A DB subnet group allows you to specify a particular VPC when you create DB clusters using the AWS CLI or RDS API. If you use the console, you can just choose the VPC and subnets you want to use. Each DB subnet group must have at least one subnet in at least two Availability Zones in the AWS Region. As a best practice, each DB subnet group should have at least one subnet for every Availability Zone in the AWS Region.

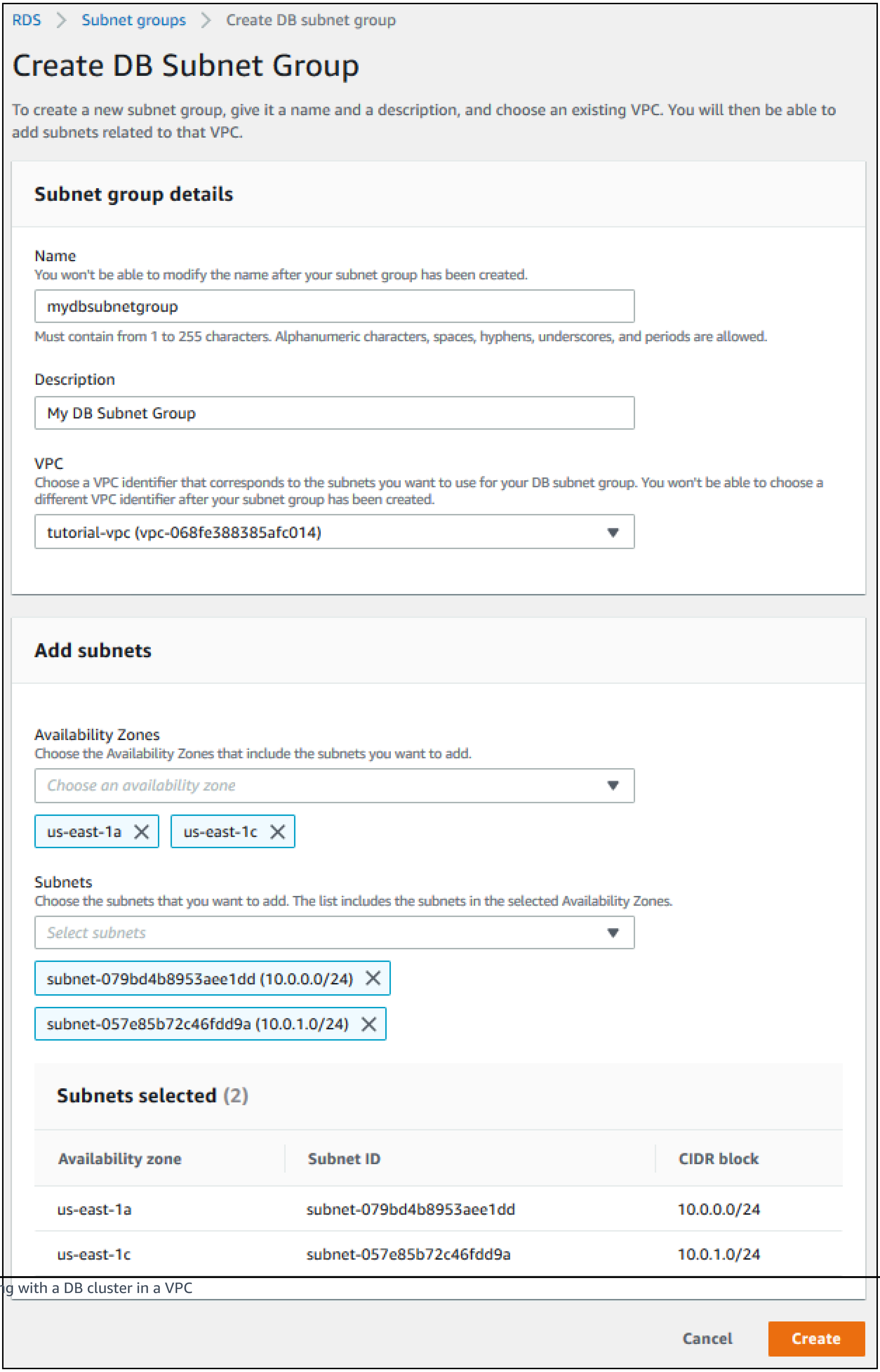
For a DB cluster to be publicly accessible, the subnets in the DB subnet group must have an internet gateway. For more information about internet gateways for subnets, see Connect to the internet using an internet gateway in the *Amazon VPC User Guide*.

When you create a DB cluster in a VPC, you can choose a DB subnet group. Amazon Aurora chooses a subnet and an IP address within that subnet to associate with your DB cluster. If no DB subnet groups exist, Amazon Aurora creates a default subnet group when you create a DB cluster. Amazon Aurora creates and associates an Elastic Network Interface to your DB cluster with that IP address. The DB cluster uses the Availability Zone that contains the subnet.

In this step, you create a DB subnet group and add the subnets that you created for your VPC.

To create a DB subnet group

1. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the navigation pane, choose **Subnet groups**.
3. Choose **Create DB Subnet Group**.
4. For **Name**, type the name of your DB subnet group.
5. For **Description**, type a description for your DB subnet group.
6. For **VPC**, choose the default VPC or the VPC that you created.
7. In the **Add subnets** section, choose the Availability Zones that include the subnets from **Availability Zones**, and then choose the subnets from **Subnets**.



Create DB Subnet Group

To create a new subnet group, give it a name and a description, and choose an existing VPC. You will then be able to add subnets related to that VPC.

Subnet group details

Name
You won't be able to modify the name after your subnet group has been created.
mydbsubnetgroup
Must contain from 1 to 255 characters. Alphanumeric characters, spaces, hyphens, underscores, and periods are allowed.

Description
My DB Subnet Group

VPC
Choose a VPC identifier that corresponds to the subnets you want to use for your DB subnet group. You won't be able to choose a different VPC identifier after your subnet group has been created.
tutorial-vpc (vpc-068fe388385afc014) ▼

Add subnets

Availability Zones
Choose the Availability Zones that include the subnets you want to add.
Choose an availability zone ▼
us-east-1a ✕ us-east-1c ✕

Subnets
Choose the subnets that you want to add. The list includes the subnets in the selected Availability Zones.
Select subnets ▼
subnet-079bd4b8953aee1dd (10.0.0.0/24) ✕
subnet-057e85b72c46fdd9a (10.0.1.0/24) ✕

Subnets selected (2)

Availability zone	Subnet ID	CIDR block
us-east-1a	subnet-079bd4b8953aee1dd	10.0.0.0/24
us-east-1c	subnet-057e85b72c46fdd9a	10.0.1.0/24

8. Choose **Create**.

   Your new DB subnet group appears in the DB subnet groups list on the RDS console. You can choose the DB subnet group to see details, including all of the subnets associated with the group, in the details pane at the bottom of the window.

Step 3: Create a VPC security group

Before you create your DB cluster, you can create a VPC security group to associate with your DB cluster. If you don't create a VPC security group, you can use the default security group when you create a DB cluster. For instructions on how to create a security group for your DB cluster, see Create a VPC security group for a private DB cluster, or see Control traffic to resources using security groups in the *Amazon VPC User Guide*.

Step 4: Create a DB instance in the VPC

In this step, you create a DB cluster and use the VPC name, the DB subnet group, and the VPC security group you created in the previous steps.

> **Note**
>
> If you want your DB cluster in the VPC to be publicly accessible, you must enable the VPC attributes *DNS hostnames* and *DNS resolution*. For more information, see DNS attributes for your VPC in the *Amazon VPC User Guide*.

For details on how to create a DB cluster, see Creating an Amazon Aurora DB cluster.

When prompted in the **Connectivity** section, enter the VPC name, the DB subnet group, and the VPC security group.

> **Note**
>
> Updating VPCs isn't currently supported for Aurora DB clusters.

Scenarios for accessing a DB cluster in a VPC

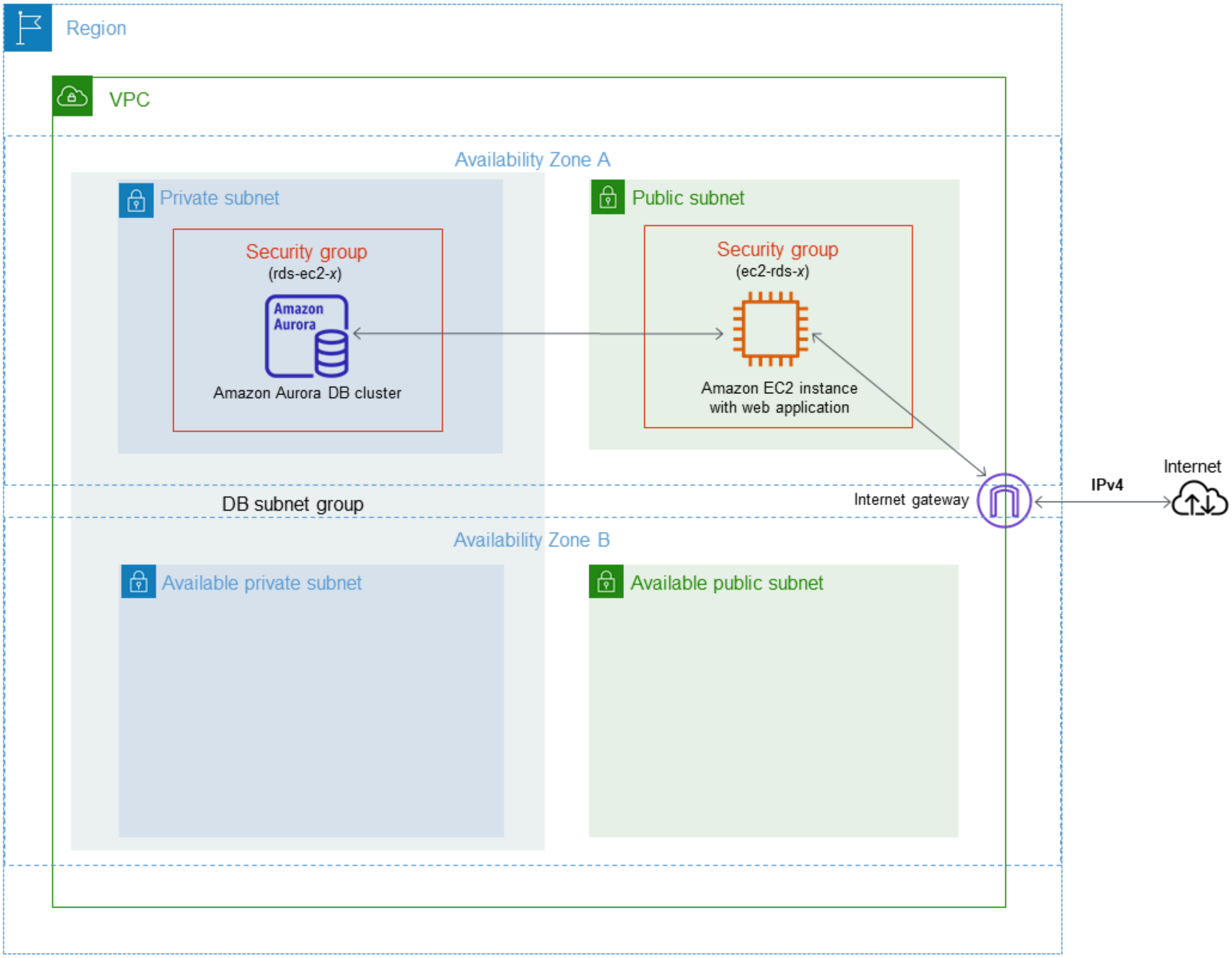
Amazon Aurora supports the following scenarios for accessing a DB cluster in a VPC:

- An EC2 instance in the same VPC
- An EC2 instance in a different VPC
- A client application through the internet
- A private network

A DB cluster in a VPC accessed by an EC2 instance in the same VPC

A common use of a DB cluster in a VPC is to share data with an application server that is running in an EC2 instance in the same VPC.

The following diagram shows this scenario.



The simplest way to manage access between EC2 instances and DB clusters in the same VPC is to do the following:

- Create a VPC security group for your DB clusters to be in. This security group can be used to restrict access to the DB clusters. For example, you can create a custom rule for this security group. This might allow TCP access using the port that you assigned to the DB cluster when you created it and an IP address you use to access the DB cluster for development or other purposes.
- Create a VPC security group for your EC2 instances (web servers and clients) to be in. This security group can, if needed, allow access to the EC2 instance from the internet by using the VPC's routing table. For example, you can set rules on this security group to allow TCP access to the EC2 instance over port 22.
- Create custom rules in the security group for your DB clusters that allow connections from the security group you created for your EC2 instances. These rules might allow any member of the security group to access the DB clusters.

There is an additional public and private subnet in a separate Availability Zone. An RDS DB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easy to switch to a Multi-AZ DB instance deployment in the future.

For a tutorial that shows you how to create a VPC with both public and private subnets for this scenario, see Tutorial: Create a VPC for use with a DB cluster (IPv4 only).

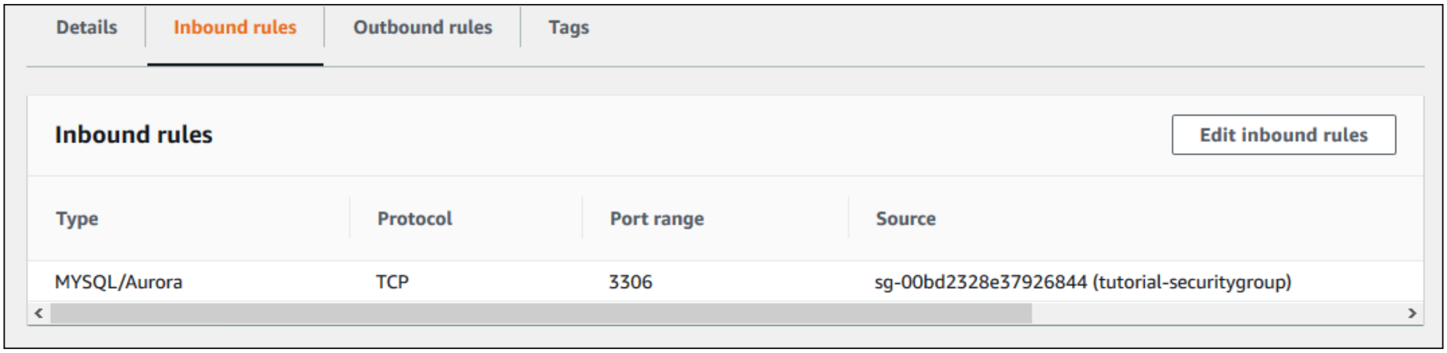
> **Tip**
>
> You can set up network connectivity between an Amazon EC2 instance and a DB cluster automatically when you create the DB cluster. For more information, see Configure automatic network connectivity with an EC2 instance .

To create a rule in a VPC security group that allows connections from another security group, do the following:

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. In the navigation pane, choose **Security groups**.

3. Choose or create a security group for which you want to allow access to members of another security group. In the preceding scenario, this is the security group that you use for your DB clusters. Choose the **Inbound rules** tab, and then choose **Edit inbound rules**.
4. On the **Edit inbound rules** page, choose **Add rule**.
5. For **Type**, choose the entry that corresponds to the port you used when you created your DB cluster, such as **MYSQL/Aurora**.
6. In the **Source** box, start typing the ID of the security group, which lists the matching security groups. Choose the security group with members that you want to have access to the resources protected by this security group. In the scenario preceding, this is the security group that you use for your EC2 instance.
7. If required, repeat the steps for the TCP protocol by creating a rule with **All TCP** as the **Type** and your security group in the **Source** box. If you intend to use the UDP protocol, create a rule with **All UDP** as the **Type** and your security group in **Source**.
8. Choose **Save rules**.

The following screen shows an inbound rule with a security group for its source.



Inbound rules

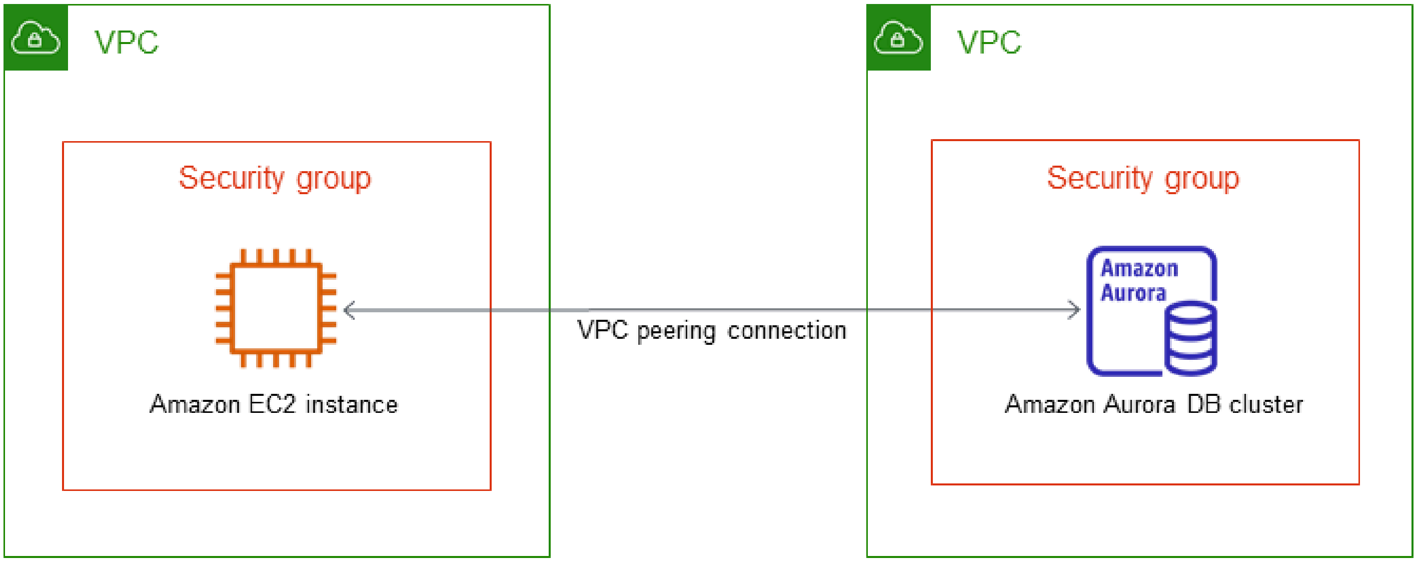
Type	Protocol	Port range	Source
MYSQL/Aurora	TCP	3306	sg-00bd2328e37926844 (tutorial-securitygroup)

For more information about connecting to the DB cluster from your EC2 instance, see Connecting to an Amazon Aurora DB cluster.

A DB cluster in a VPC accessed by an EC2 instance in a different VPC

When your DB clusters is in a different VPC from the EC2 instance you are using to access it, you can use VPC peering to access the DB cluster.

The following diagram shows this scenario.

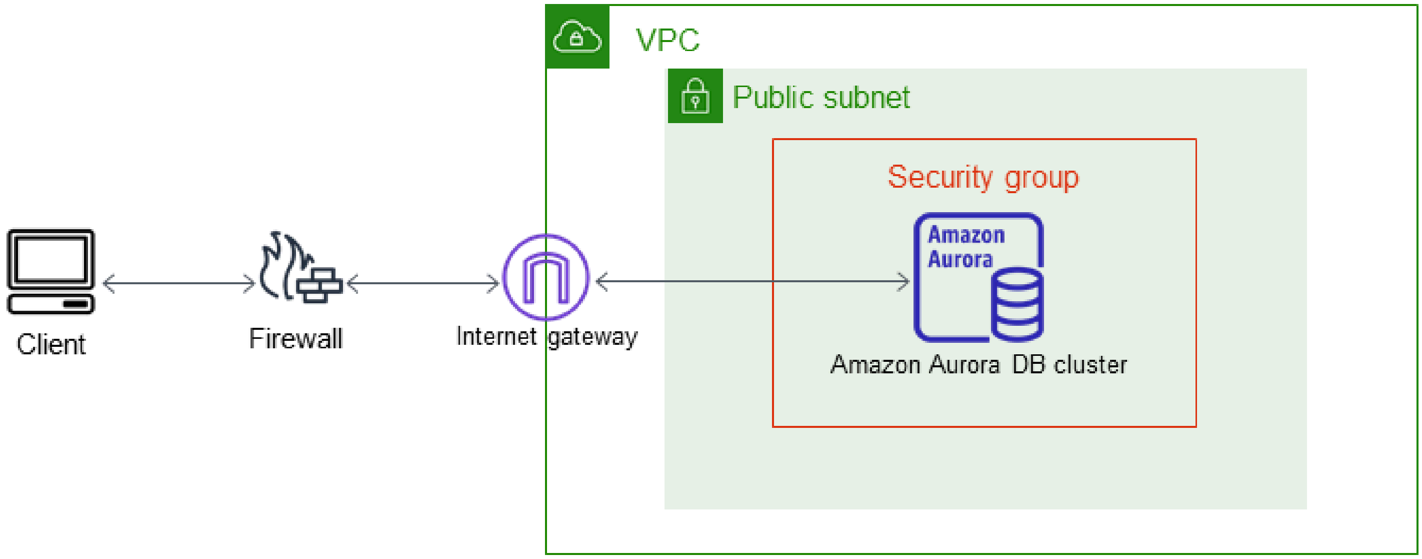


A VPC peering connection is a networking connection between two VPCs that enables you to route traffic between them using private IP addresses. Resources in either VPC can communicate with each other as if they are within the same network. You can create a VPC peering connection between your own VPCs, with a VPC in another AWS account, or with a VPC in a different AWS Region. To learn more about VPC peering, see VPC peering in the *Amazon Virtual Private Cloud User Guide*.

A DB cluster in a VPC accessed by a client application through the internet

To access a DB clusters in a VPC from a client application through the internet, you configure a VPC with a single public subnet, and an internet gateway to enable communication over the internet.

The following diagram shows this scenario.



We recommend the following configuration:

- A VPC of size /16 (for example CIDR: 10.0.0.0/16). This size provides 65,536 private IP addresses.
- A subnet of size /24 (for example CIDR: 10.0.0.0/24). This size provides 256 private IP addresses.
- An Amazon Aurora DB cluster that is associated with the VPC and the subnet. Amazon RDS assigns an IP address within the subnet to your DB cluster.
- An internet gateway which connects the VPC to the internet and to other AWS products.
- A security group associated with the DB cluster. The security group's inbound rules allow your client application to access to your DB cluster.

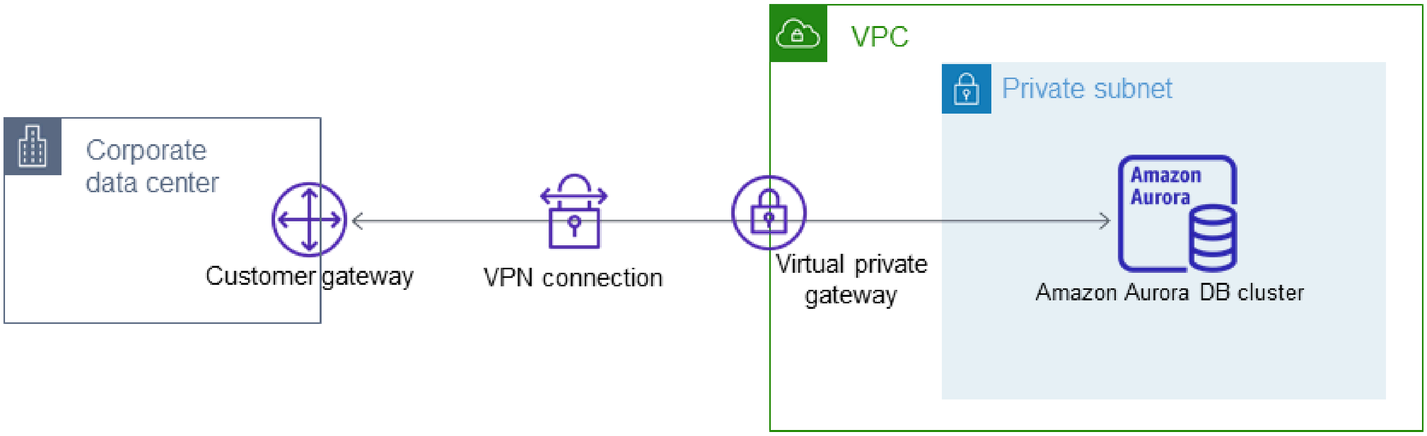
For information about creating a DB clusters in a VPC, see Creating a DB cluster in a VPC.

A DB cluster in a VPC accessed by a private network

If your DB cluster isn't publicly accessible, you have the following options for accessing it from a private network:

- An AWS Site-to-Site VPN connection. For more information, see What is AWS Site-to-Site VPN?
- An AWS Direct Connect connection. For more information, see What is AWS Direct Connect?
- An AWS Client VPN connection. For more information, see What is AWS Client VPN?

The following diagram shows a scenario with an AWS Site-to-Site VPN connection.

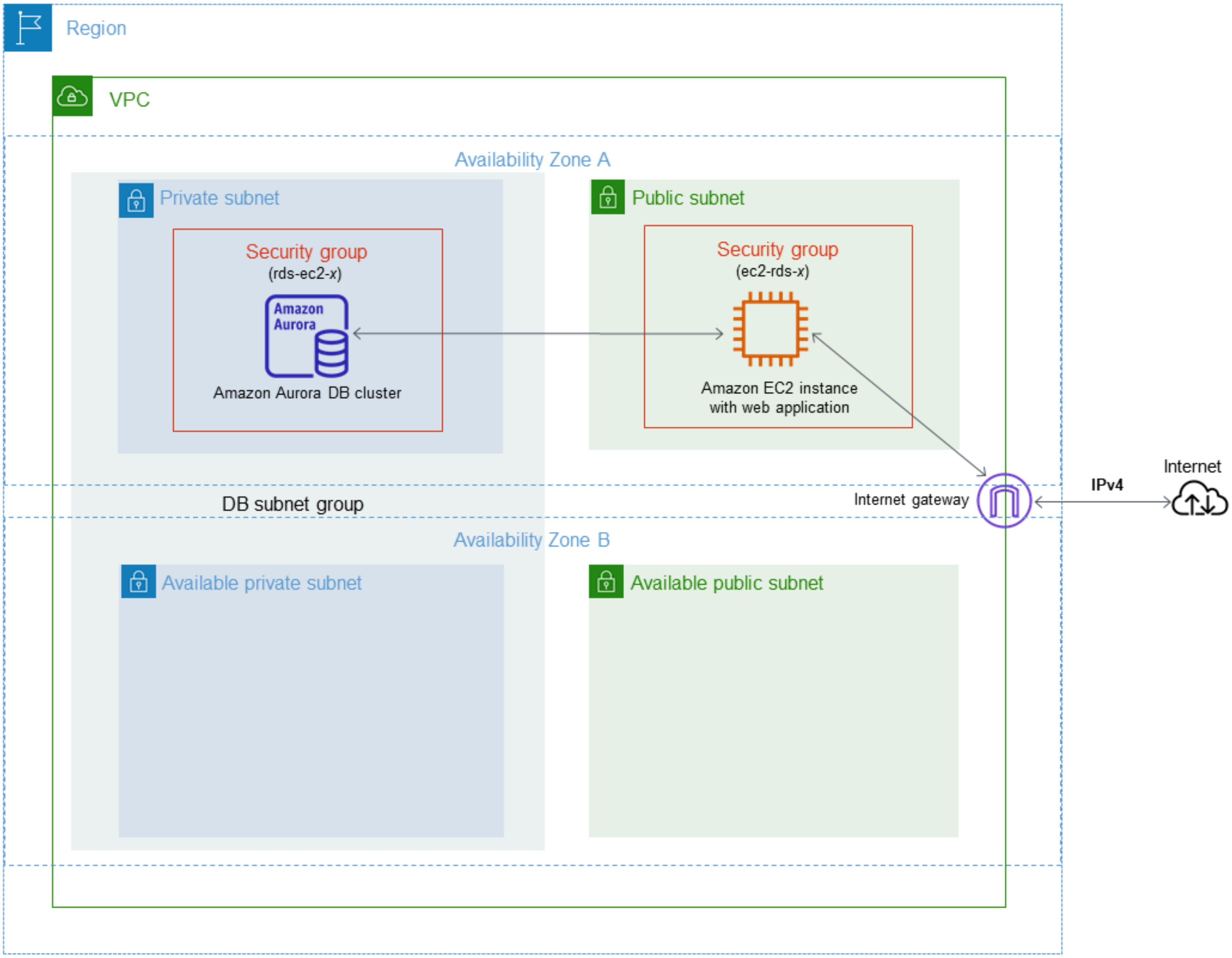


For more information, see Internetwork traffic privacy.

Tutorial: Create a VPC for use with a DB cluster (IPv4 only)

A common scenario includes a DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. This VPC shares data with a web server that is running in the same VPC. In this tutorial, you create the VPC for this scenario.

The following diagram shows this scenario. For information about other scenarios, see Scenarios for accessing a DB cluster in a VPC.



Your DB cluster needs to be available only to your web server, and not to the public internet. Thus, you create a VPC with both public and private subnets. The web server is hosted in the public subnet, so that it can reach the public internet. The DB cluster is hosted in a private subnet. The web server can connect to the DB cluster because it is hosted within the same VPC. But the DB cluster isn't available to the public internet, providing greater security.

This tutorial configures an additional public and private subnet in a separate Availability Zone. These subnets aren't used by the tutorial. An RDS DB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easier to configure more than one Aurora DB instance.

This tutorial describes configuring a VPC for Amazon Aurora DB clusters. For a tutorial that shows you how to create a web server for this VPC scenario, see Tutorial: Create a web server and an Amazon Aurora DB cluster. For more information about Amazon VPC, see Amazon VPC Getting Started Guide and Amazon VPC User Guide.

> **Tip**
>
> You can set up network connectivity between an Amazon EC2 instance and a DB cluster automatically when you create the DB cluster. The network configuration is similar to the one described in this tutorial. For more information, see Configure automatic network connectivity with an EC2 instance.

Create a VPC with private and public subnets

Use the following procedure to create a VPC with both public and private subnets.

To create a VPC and subnets

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the top-right corner of the AWS Management Console, choose the Region to create your VPC in. This example uses the US West (Oregon) Region.
3. In the upper-left corner, choose **VPC dashboard**. To begin creating a VPC, choose **Create VPC**.
4. For **Resources to create** under **VPC settings**, choose **VPC and more**.
5. For the **VPC settings**, set these values:

    - **Name tag auto-generation – `tutorial`**
    - **IPv4 CIDR block – `10.0.0.0/16`**
    - **IPv6 CIDR block – No IPv6 CIDR block**
    - **Tenancy – Default**
    - **Number of Availability Zones (AZs) – 2**
    - **Customize AZs –** Keep the default values.

- **Number of public subnet – 2**
- **Number of private subnets – 2**
- **Customize subnets CIDR blocks** – Keep the default values.
- **NAT gateways ($) – None**
- **VPC endpoints – None**
- **DNS options** – Keep the default values.

6. Choose **Create VPC**.

Create a VPC security group for a public web server

Next, you create a security group for public access. To connect to public EC2 instances in your VPC, you add inbound rules to your VPC security group. These allow traffic to connect from the internet.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:
   - **Security group name:** **`tutorial-securitygroup`**
   - **Description:** **`Tutorial Security Group`**
   - **VPC:** Choose the VPC that you created earlier, for example: **vpc-*`identifier`* (tutorial-vpc)**
4. Add inbound rules to the security group.

   a. Determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com. An example of an IP address is `203.0.113.25/32`.

      In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, find the range of IP addresses used by client computers.

> ⚠ **Warning**
>
> If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your instances using SSH.

b. In the **Inbound rules** section, choose **Add rule**.

c. Set the following values for your new inbound rule to allow SSH access to your Amazon EC2 instance. If you do this, you can connect to your Amazon EC2 instance to install the web server and other utilities. You also connect to your EC2 instance to upload content for your web server.

- **Type: SSH**
- **Source:** The IP address or range from Step a, for example: **`203.0.113.25/32`**.

d. Choose **Add rule**.

e. Set the following values for your new inbound rule to allow HTTP access to your web server:

- **Type: HTTP**
- **Source: `0.0.0.0/0`**

5. Choose **Create security group** to create the security group.

   Note the security group ID because you need it later in this tutorial.

Create a VPC security group for a private DB cluster

To keep your DB cluster private, create a second security group for private access. To connect to private DB clustersin your VPC, you add inbound rules to your VPC security group that allow traffic from your web server only.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.

3. On the **Create security group** page, set these values:

   - **Security group name: `tutorial-db-securitygroup`**
   - **Description: `Tutorial DB Instance Security Group`**
   - **VPC:** Choose the VPC that you created earlier, for example: **vpc-*`identifier`* (tutorial-vpc)**

4. Add inbound rules to the security group.

   a. In the **Inbound rules** section, choose **Add rule**.

   b. Set the following values for your new inbound rule to allow MySQL traffic on port 3306 from your Amazon EC2 instance. If you do this, you can connect from your web server to your DB cluster. By doing so, you can store and retrieve data from your web application to your database.

      - **Type: `MySQL/Aurora`**
      - **Source:** The identifier of the **tutorial-securitygroup** security group that you created previously in this tutorial, for example: **sg-9edd5cfb**.

5. Choose **Create security group** to create the security group.

Create a DB subnet group

A *DB subnet group* is a collection of subnets that you create in a VPC and that you then designate for your DB clusters. A DB subnet group makes it possible for you to specify a particular VPC when creating DB clusters.

To create a DB subnet group

1. Identify the private subnets for your database in the VPC.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **Subnets**.

   c. Note the subnet IDs of the subnets named **tutorial-subnet-private1-us-west-2a** and **tutorial-subnet-private2-us-west-2b**.

      You need the subnet IDs when you create your DB subnet group.

2. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   Make sure that you connect to the Amazon RDS console, not to the Amazon VPC console.

3. In the navigation pane, choose **Subnet groups**.
4. Choose **Create DB subnet group**.
5. On the **Create DB subnet group** page, set these values in **Subnet group details**:

   - **Name: `tutorial-db-subnet-group`**
   - **Description: `Tutorial DB Subnet Group`**
   - **VPC: tutorial-vpc (vpc-*`identifier`*)**
6. In the **Add subnets** section, choose the **Availability Zones** and **Subnets**.

   For this tutorial, choose **us-west-2a** and **us-west-2b** for the **Availability Zones**. For **Subnets**, choose the private subnets you identified in the previous step.
7. Choose **Create**.

   Your new DB subnet group appears in the DB subnet groups list on the RDS console. You can choose the DB subnet group to see details in the details pane at the bottom of the window. These details include all of the subnets associated with the group.

> **Note**
>
> If you created this VPC to complete Tutorial: Create a web server and an Amazon Aurora DB cluster, create the DB cluster by following the instructions in Create an Amazon Aurora DB cluster.

Deleting the VPC

After you create the VPC and other resources for this tutorial, you can delete them if they are no longer needed.

> **Note**
>
> If you added resources in the VPC that you created for this tutorial, you might need to delete these before you can delete the VPC. For example, these resources might include Amazon EC2 instances or Amazon RDS DB clusters. For more information, see Delete your VPC in the *Amazon VPC User Guide*.

To delete a VPC and related resources

1. Delete the DB subnet group.

   a. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   b. In the navigation pane, choose **Subnet groups**.

   c. Select the DB subnet group you want to delete, such as **tutorial-db-subnet-group**.

   d. Choose **Delete**, and then choose **Delete** in the confirmation window.

2. Note the VPC ID.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **VPCs**.

   c. In the list, identify the VPC that you created, such as **tutorial-vpc**.

   d. Note the **VPC ID** of the VPC that you created. You need the VPC ID in later steps.

3. Delete the security groups.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **Security Groups**.

   c. Select the security group for the Amazon RDS DB instance, such as **tutorial-db-securitygroup**.

   d. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation page.

   e. On the **Security Groups** page, select the security group for the Amazon EC2 instance, such as **tutorial-securitygroup**.

   f. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation page.

4. Delete the VPC.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **VPCs**.

   c. Select the VPC you want to delete, such as **tutorial-vpc**.

   d. For **Actions**, choose **Delete VPC**.

   The confirmation page shows other resources that are associated with the VPC that will also be deleted, including the subnets associated with it.

e. On the confirmation page, enter **`delete`**, and then choose **Delete**.

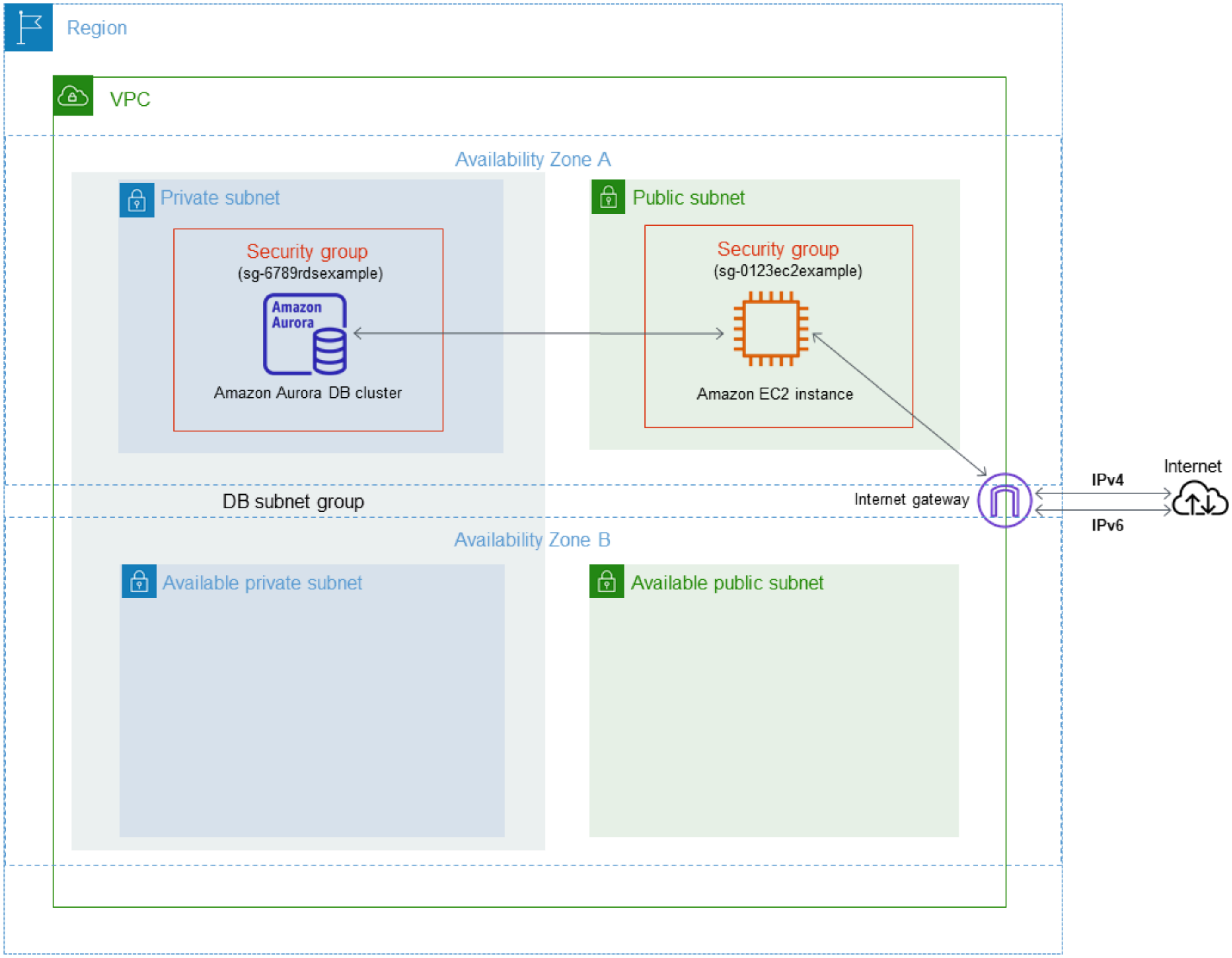
Tutorial: Create a VPC for use with a DB cluster (dual-stack mode)

A common scenario includes a DB cluster in a virtual private cloud (VPC) based on the Amazon VPC service. This VPC shares data with a public Amazon EC2 instance that is running in the same VPC.

In this tutorial, you create the VPC for this scenario that works with a database running in dual-stack mode. Dual-stack mode to enable connection over the IPv6 addressing protocol. For more information about IP addressing, see Amazon Aurora IP addressing.

Dual-stack network clusters are supported in most regions. For more information see Availability of dual-stack network DB clusters. To see the limitations of dual-stack mode, see Limitations for dual-stack network DB clusters.

The following diagram shows this scenario.



For information about other scenarios, see Scenarios for accessing a DB cluster in a VPC.

Your DB cluster needs to be available only to your Amazon EC2 instance, and not to the public internet. Thus, you create a VPC with both public and private subnets. The Amazon EC2 instance is hosted in the public subnet, so that it can reach the public internet. The DB cluster is hosted in a private subnet. The Amazon EC2 instance can connect to the DB cluster because it's hosted within the same VPC. However, the DB cluster is not available to the public internet, providing greater security.

This tutorial configures an additional public and private subnet in a separate Availability Zone. These subnets aren't used by the tutorial. An RDS DB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easy to configure more than one Aurora DB instance.

To create a DB cluster that uses dual-stack mode, specify **Dual-stack mode** for the **Network type** setting. You can also modify a DB cluster with the same setting. For more information about creating a DB cluster, see Creating an Amazon Aurora DB cluster. For more information about modifying a DB cluster, see Modifying an Amazon Aurora DB cluster.

This tutorial describes configuring a VPC for Amazon Aurora DB clusters. For more information about Amazon VPC, see Amazon VPC User Guide.

Create a VPC with private and public subnets

Use the following procedure to create a VPC with both public and private subnets.

To create a VPC and subnets

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the upper-right corner of the AWS Management Console, choose the Region to create your VPC in. This example uses the US East (Ohio) Region.
3. In the upper-left corner, choose **VPC dashboard**. To begin creating a VPC, choose **Create VPC**.
4. For **Resources to create** under **VPC settings**, choose **VPC and more**.
5. For the remaining **VPC settings**, set these values:

   - **Name tag auto-generation – `tutorial-dual-stack`**
   - **IPv4 CIDR block – `10.0.0.0/16`**
   - **IPv6 CIDR block – Amazon-provided IPv6 CIDR block**
   - **Tenancy – Default**
   - **Number of Availability Zones (AZs) – 2**
   - **Customize AZs –** Keep the default values.
   - **Number of public subnet – 2**
   - **Number of private subnets – 2**
   - **Customize subnets CIDR blocks –** Keep the default values.
   - **NAT gateways ($) – None**
   - **Egress only internet gateway – No**
   - **VPC endpoints – None**
   - **DNS options –** Keep the default values.

> **Note**
>
> Amazon RDS requires at least two subnets in two different Availability Zones to support Multi-AZ DB instance deployments. This tutorial creates a Single-AZ deployment, but the requirement makes it easy to convert to a Multi-AZ DB instance deployment in the future.

6. Choose **Create VPC**.

Create a VPC security group for a public Amazon EC2 instance

Next, you create a security group for public access. To connect to public EC2 instances in your VPC, add inbound rules to your VPC security group that allow traffic to connect from the internet.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:
   - **Security group name: `tutorial-dual-stack-securitygroup`**
   - **Description: `Tutorial Dual-Stack Security Group`**
   - **VPC:** Choose the VPC that you created earlier, for example: **vpc-*`identifier`* (tutorial-dual-stack-vpc)**
4. Add inbound rules to the security group.

   a. Determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH).

      An example of an Internet Protocol version 4 (IPv4) address is `203.0.113.25/32`. An example of an Internet Protocol version 6 (IPv6) address range is `2001:db8:1234:1a00::/64`.

      In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, find the range of IP addresses used by client computers.

> ⚠ **Warning**
>
> If you use `0.0.0.0/0` for IPv4 or `::0` for IPv6, you make it possible for all IP addresses to access your public instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your instances.

   b. In the **Inbound rules** section, choose **Add rule**.

   c. Set the following values for your new inbound rule to allow Secure Shell (SSH) access to your Amazon EC2 instance. If you do this, you can connect to your EC2 instance to install SQL clients and other applications. Specify an IP address so you can access your EC2 instance:

   - **Type: SSH**
   - **Source:** The IP address or range from step a. An example of an IPv4 IP address is **`203.0.113.25/32`**. An example of an IPv6 IP address is **`2001:DB8::/32`**.

5. Choose **Create security group** to create the security group.

   Note the security group ID because you need it later in this tutorial.

Create a VPC security group for a private DB cluster

To keep your DB cluster private, create a second security group for private access. To connect to private DB clusters in your VPC, add inbound rules to your VPC security group. These allow traffic from your Amazon EC2 instance only.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:

   - **Security group name: `tutorial-dual-stack-db-securitygroup`**
   - **Description: `Tutorial Dual-Stack DB Instance Security Group`**

- **VPC:** Choose the VPC that you created earlier, for example: **vpc-*`identifier`* (tutorial-dual-stack-vpc)**

4. Add inbound rules to the security group:

   a. In the **Inbound rules** section, choose **Add rule**.

   b. Set the following values for your new inbound rule to allow MySQL traffic on port 3306 from your Amazon EC2 instance. If you do, you can connect from your EC2 instance to your DB cluster. Doing this means that you can send data from your EC2 instance to your database.

      - **Type: MySQL/Aurora**
      - **Source:** The identifier of the **tutorial-dual-stack-securitygroup** security group that you created previously in this tutorial, for example **sg-9edd5cfb**.

5. To create the security group, choose **Create security group**.

Create a DB subnet group

A *DB subnet group* is a collection of subnets that you create in a VPC and that you then designate for your DB clusters. By using a DB subnet group, you can specify a particular VPC when creating DB clusters. To create a DB subnet group that is `DUAL` compatible, all subnets must be `DUAL` compatible. To be `DUAL` compatible, a subnet must have an IPv6 CIDR associated with it.

To create a DB subnet group

1. Identify the private subnets for your database in the VPC.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **Subnets**.

   c. Note the subnet IDs of the subnets named **tutorial-dual-stack-subnet-private1-us-west-2a** and **tutorial-dual-stack-subnet-private2-us-west-2b**.

      You will need the subnet IDs when you create your DB subnet group.

2. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.

   Make sure that you connect to the Amazon RDS console, not to the Amazon VPC console.

3. In the navigation pane, choose **Subnet groups**.

4. Choose **Create DB subnet group**.

5. On the **Create DB subnet group** page, set these values in **Subnet group details**:

   - **Name: `tutorial-dual-stack-db-subnet-group`**
   - **Description: `Tutorial Dual-Stack DB Subnet Group`**
   - **VPC: tutorial-dual-stack-vpc (vpc-*`identifier`*)**

6. In the **Add subnets** section, choose values for the **Availability Zones** and **Subnets** options.

   For this tutorial, choose **us-east-2a** and **us-east-2b** for the **Availability Zones**. For **Subnets**, choose the private subnets you identified in the previous step.

7. Choose **Create**.

Your new DB subnet group appears in the DB subnet groups list on the RDS console. You can choose the DB subnet group to see its details. These include the supported addressing protocols and all of the subnets associated with the group and the network type supported by the DB subnet group.

Create an Amazon EC2 instance in dual-stack mode

To create an Amazon EC2 instance, follow the instructions in Launch an instance using the new launch instance wizard in the *Amazon EC2 User Guide*.

On the **Configure Instance Details** page, set these values and keep the other values as their defaults:

- **Network** – Choose an existing VPC with both public and private subnets, such as **tutorial-dual-stack-vpc** (vpc-*`identifier`*) created in Create a VPC with private and public subnets.
- **Subnet** – Choose an existing public subnet, such as **subnet-*`identifier`* | tutorial-dual-stack-subnet-public1-us-east-2a | us-east-2a** created in Create a VPC security group for a public Amazon EC2 instance.
- **Auto-assign Public IP** – Choose **Enable**.
- **Auto-assign IPv6 IP** – Choose **Enable**.
- **Firewall (security groups)** – Choose **Select an existing security group**.
- **Common security groups** – Choose an existing security group, such as the `tutorial-securitygroup` created in Create a VPC security group for a public Amazon EC2 instance. Make sure that the security group that you choose includes inbound rules for Secure Shell (SSH) and HTTP access.

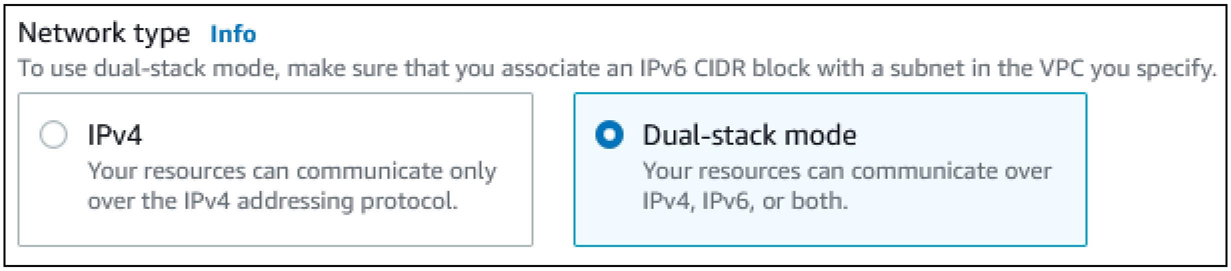
Create a DB cluster in dual-stack mode

In this step, you create a DB cluster that runs in dual-stack mode.

To create a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
2. In the upper-right corner of the console, choose the AWS Region where you want to create the DB cluster. This example uses the US East (Ohio) Region.
3. In the navigation pane, choose **Databases**.
4. Choose **Create database**.
5. On the **Create database** page, make sure that the **Standard create** option is chosen, and then choose the Aurora MySQL DB engine type.
6. In the **Connectivity** section, set these values:

   - **Network type** – Choose **Dual-stack mode**.

     Network type Info

     To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

     ○ IPv4
     Your resources can communicate only over the IPv4 addressing protocol.

     ● Dual-stack mode
     Your resources can communicate over IPv4, IPv6, or both.

   - **Virtual private cloud (VPC)** – Choose an existing VPC with both public and private subnets, such as **tutorial-dual-stack-vpc** (vpc-*identifier*) created in [Create a VPC with private and public subnets](#).

     The VPC must have subnets in different Availability Zones.
   - **DB subnet group** – Choose a DB subnet group for the VPC, such as **tutorial-dual-stack-db-subnet-group** created in [Create a DB subnet group](#).
   - **Public access** – Choose **No**.
   - **VPC security group (firewall)** – Select **Choose existing**.
   - **Existing VPC security groups** – Choose an existing VPC security group that is configured for private access, such as **tutorial-dual-stack-db-securitygroup** created in [Create a VPC security group for a private DB cluster](#).

Remove other security groups, such as the default security group, by choosing the **X** associated with each.

- **Availability Zone** – Choose **us-west-2a**.

  To avoid cross-AZ traffic, make sure the DB instance and the EC2 instance are in the same Availability Zone.

7. For the remaining sections, specify your DB cluster settings. For information about each setting, see Settings for Aurora DB clusters.

Connect to your Amazon EC2 instance and DB cluster

After you create your Amazon EC2 instance and DB cluster in dual-stack mode, you can connect to each one using the IPv6 protocol. To connect to an Amazon EC2 instance using the IPv6 protocol, follow the instructions in Connect to your Linux instance in the *Amazon EC2 User Guide*.

To connect to your Aurora MySQL DB cluster from the Amazon EC2 instance, follow the instructions in Connect to an Aurora MySQL DB cluster.

Deleting the VPC

After you create the VPC and other resources for this tutorial, you can delete them if they are no longer needed.

If you added resources in the VPC that you created for this tutorial, you might need to delete these before you can delete the VPC. Examples of resources are Amazon EC2 instances or DB clusters. For more information, see Delete your VPC in the *Amazon VPC User Guide*.

To delete a VPC and related resources

1. Delete the DB subnet group:

   a. Open the Amazon RDS console at https://console.aws.amazon.com/rds/.
   b. In the navigation pane, choose **Subnet groups**.
   c. Select the DB subnet group to delete, such as **tutorial-db-subnet-group**.
   d. Choose **Delete**, and then choose **Delete** in the confirmation window.

2. Note the VPC ID:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **VPCs**.

   c. In the list, identify the VPC you created, such as **tutorial-dual-stack-vpc**.

   d. Note the **VPC ID** value of the VPC that you created. You need this VPC ID in subsequent steps.

3. Delete the security groups:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **Security Groups**.

   c. Select the security group for the Amazon RDS DB instance, such as **tutorial-dual-stack-db-securitygroup**.

   d. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation page.

   e. On the **Security Groups** page, select the security group for the Amazon EC2 instance, such as **tutorial-dual-stack-securitygroup**.

   f. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation page.

4. Delete the NAT gateway:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **NAT Gateways**.

   c. Select the NAT gateway of the VPC that you created. Use the VPC ID to identify the correct NAT gateway.

   d. For **Actions**, choose **Delete NAT gateway**.

   e. On the confirmation page, enter **`delete`**, and then choose **Delete**.

5. Delete the VPC:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.

   b. Choose **VPC Dashboard**, and then choose **VPCs**.

   c. Select the VPC that you want to delete, such as **tutorial-dual-stack-vpc**.

   d. For **Actions**, choose **Delete VPC**.

   The confirmation page shows other resources that are associated with the VPC that will also be deleted, including the subnets associated with it.

   e. On the confirmation page, enter **`delete`**, and then choose **Delete**.

6. Release the Elastic IP addresses:

   a. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.

   b. Choose **EC2 Dashboard**, and then choose **Elastic IPs**.

   c. Select the Elastic IP address that you want to release.

   d. For **Actions**, choose **Release Elastic IP addresses**.

   e. On the confirmation page, choose **Release**.

Quotas and constraints for Amazon Aurora

Following, you can find a description of the resource quotas and naming constraints for Amazon Aurora.

Topics

- Quotas in Amazon Aurora
- Naming constraints in Amazon Aurora
- Amazon Aurora size limits

Quotas in Amazon Aurora

Each AWS account has quotas, for each AWS Region, on the number of Amazon Aurora resources that can be created. After a quota for a resource has been reached, additional calls to create that resource fail with an exception.

The following table lists the resources and their quotas per AWS Region.

Name	Default	Adjus e	Description
Authorizations per DB security group	Each supported Region: 20	No	Number of security group authorizations per DB security group
Custom engine versions	Each supported Region: 40	Yes	The maximum number of custom engine versions allowed in this account in the current Region
DB cluster parameter groups	Each supported Region: 50	No	The maximum number of DB cluster parameter groups
DB clusters	Each supported Region: 40	Yes	The maximum number of Aurora clusters allowed

Name	Default	Adjustable	Description
			in this account in the current Region
DB instances	Each supported Region: 40	Yes	The maximum number of DB instances allowed in this account in the current Region
DB subnet groups	Each supported Region: 50	Yes	The maximum number of DB subnet groups
Data API HTTP request body size	Each supported Region: 4 Megabytes	No	The maximum size allowed for the HTTP request body.
Data API maximum concurrent cluster-secret pairs	Each supported Region: 30	No	The maximum number of unique pairs of Aurora Serverless v1 DB clusters and secrets in concurrent Data API requests for this account in the current AWS Region.
Data API maximum concurrent requests	Each supported Region: 500	No	The maximum number of Data API requests to an Aurora Serverless v1 DB cluster that use the same secret and can be processed at the same time. Additional requests are queued and processed as in-process requests complete.

Name	Default	Adjustable	Description
Data API maximum result set size	Each supported Region: 1 Megabytes	No	The maximum size of the database result set that can be returned by the Data API.
Data API maximum size of JSON response string	Each supported Region: 10 Megabytes	No	The maximum size of the simplified JSON response string returned by the RDS Data API.
Data API requests per second	Each supported Region: 1,000 per second	No	The maximum number of requests to the Data API per second allowed for this account in the current AWS Region. This quota only applies to Amazon Aurora Serverless v1 clusters.
Event subscriptions	Each supported Region: 20	Yes	The maximum number of event subscriptions
IAM roles per DB cluster	Each supported Region: 5	Yes	The maximum number of IAM roles associated with a DB cluster
IAM roles per DB instance	Each supported Region: 5	Yes	The maximum number of IAM roles associated with a DB instance
Manual DB cluster snapshots	Each supported Region: 100	Yes	The maximum number of manual DB cluster snapshots

Name	Default	Adjuse	Description
Manual DB instance snapshots	Each supported Region: 100	Yes	The maximum number of manual DB instance snapshots
Option groups	Each supported Region: 20	Yes	The maximum number of option groups
Parameter groups	Each supported Region: 50	Yes	The maximum number of parameter groups
Proxies	Each supported Region: 20	Yes	The maximum number of proxies allowed in this account in the current AWS Region
Read replicas per primary	Each supported Region: 15	Yes	The maximum number of read replicas per primary DB instance. This quota cant be adjusted for Amazon Aurora.
Reserved DB instances	Each supported Region: 40	Yes	The maximum number of reserved DB instances allowed in this account in the current AWS Region
Rules per security group	Each supported Region: 20	No	The maximum number of rules per DB security group
Security groups	Each supported Region: 25	Yes	The maximum number of DB security groups
Security groups (VPC)	Each supported Region: 5	No	The maximum number of DB security groups per Amazon VPC

Name	Default	Adjustable	Description
Subnets per DB subnet group	Each supported Region: 20	No	The maximum number of subnets per DB subnet group
Tags per resource	Each supported Region: 50	No	The maximum number of tags per Amazon RDS resource
Total storage for all DB instances	Each supported Region: 100,000 Gigabytes	Yes	The maximum total storage (in GB) on EBS volumes for all Amazon RDS DB instances added together. This quota does not apply to Amazon Aurora, which has a maximum cluster volume of 128 TiB for each DB cluster.

Note

By default, you can have up to a total of 40 DB instances. RDS DB instances, Aurora DB instances, Amazon Neptune instances, and Amazon DocumentDB instances apply to this quota.
If your application requires more DB instances, you can request additional DB instances by opening the Service Quotas console. In the navigation pane, choose **AWS services**. Choose **Amazon Relational Database Service (Amazon RDS)**, choose a quota, and follow the directions to request a quota increase. For more information, see Requesting a quota increase in the *Service Quotas User Guide*.
Backups managed by AWS Backup are considered manual DB clustersnapshots, but don't count toward the manual cluster snapshot quota. For information about AWS Backup, see the *AWS Backup Developer Guide*.

If you use any RDS API operations and exceed the default quota for the number of calls per second, the Amazon RDS API issues an error like the following one.

ClientError: An error occurred (ThrottlingException) when calling the *API_name* operation: Rate exceeded.

Here, reduce the number of calls per second. The quota is meant to cover most use cases. If higher quotas are needed, you can request a quota increase by using one of the following options:

- From the console, open the Service Quotas console.
- From the AWS CLI, use the request-service-quota-increase AWS CLI command.

For more information, see the *Service Quotas User Guide*.

Naming constraints in Amazon Aurora

The following table describes naming constraints in Amazon Aurora.

Resource or item	Constraints
DB cluster identifier	Identifiers have these naming constraints: • Must contain 1–63 alphanumeric characters or hyphens. • First character must be a letter. • Can't end with a hyphen or contain two consecutive hyphens. • Must be unique for all DB instances per AWS account, per AWS Region.
Initial database name	Database name constraints differ between Aurora MySQL and PostgreSQL. For more information, see the available settings when creating each DB cluster.
Master user name	Master user name constraints differ for each database engine. For more information, see the available settings when creating each DB cluster.

Resource or item	Constraints
Master password	The password for the database master user can include any printable ASCII character except `/`, `'`, `"`, `@`, or a space. For Oracle, & is an additional character limitation. The password has the following number of printable ASCII characters depending on the DB engine: • Aurora MySQL: 8–41 • Aurora PostgreSQL: 8–99
DB parameter group name	These names have these constraints: • Must contain 1–255 alphanumeric characters. • First character must be a letter. • Hyphens are allowed, but the name cannot end with a hyphen or contain two consecutive hyphens.
DB subnet group name	These names have these constraints: • Must contain 1–255 characters. • Alphanumeric characters, spaces, hyphens, underscores, and periods are allowed.

Amazon Aurora size limits

Storage size limits

An Aurora cluster volume can grow to a maximum size of 128 tebibytes (TiB) for the following engine versions:

- All available Aurora MySQL version 3 versions; Aurora MySQL version 2, versions 2.09 and higher
- All available Aurora PostgreSQL versions

For lower engine versions, the maximum size of an Aurora cluster volume is 64 TiB. For more information, see How Aurora storage automatically resizes.

To monitor the remaining storage space, you can use the `AuroraVolumeBytesLeftTotal` metric. For more information, see Cluster-level metrics for Amazon Aurora.

SQL table size limits

For an Aurora MySQL DB cluster, the maximum table size is 64 tebibytes (TiB). For an Aurora PostgreSQL DB cluster, the maximum table size is 32 tebibytes (TiB). We recommend that you follow table design best practices, such as partitioning of large tables.

Table space ID limits

The maximum table space ID for Aurora MySQL is 2147483647. If you frequently create and drop tables, make sure to be aware of your table space IDs and plan to use logical dumps. For more information, see Logical migration from MySQL to Amazon Aurora MySQL by using mysqldump.

Troubleshooting for Amazon Aurora

Use the following sections to help troubleshoot problems you have with DB instances in Amazon RDS and Amazon Aurora.

Topics

For information about debugging problems using the Amazon RDS API, see Troubleshooting applications on Aurora.

Can't connect to Amazon RDS DB instance

When you can't connect to a DB instance, the following are common causes:

- **Inbound rules** – The access rules enforced by your local firewall and the IP addresses authorized to access your DB instance might not match. The problem is most likely the inbound rules in your security group.

  By default, DB instances don't allow access. Access is granted through a security group associated with the VPC that allows traffic into and out of the DB instance. If necessary, add inbound and outbound rules for your particular situation to the security group. You can specify an IP address, a range of IP addresses, or another VPC security group.

  > **Note**
  >
  > When adding a new inbound rule, you can choose **My IP** for **Source** to allow access to the DB instance from the IP address detected in your browser.

For more information about setting up security groups, see Provide access to the DB cluster in the VPC by creating a security group.

> **Note**
>
> Client connections from IP addresses within the range 169.254.0.0/16 aren't permitted. This is the Automatic Private IP Addressing Range (APIPA), which is used for local-link addressing.

- **Public accessibility** – To connect to your DB instance from outside of the VPC, such as by using a client application, the instance must have a public IP address assigned to it.

  To make the instance publicly accessible, modify it and choose **Yes** under **Public accessibility**. For more information, see Hiding a DB cluster in a VPC from the internet.
- **Port** – The port that you specified when you created the DB instance can't be used to send or receive communications due to your local firewall restrictions. To determine if your network allows the specified port to be used for inbound and outbound communication, check with your network administrator.
- **Availability** – For a newly created DB instance, the DB instance has a status of `creating` until the DB instance is ready to use. When the state changes to `available`, you can connect to the DB instance. Depending on the size of your DB instance, it can take up to 20 minutes before an instance is available.
- **Internet gateway** – For a DB instance to be publicly accessible, the subnets in its DB subnet group must have an internet gateway.

  To configure an internet gateway for a subnet

  1. Sign in to the AWS Management Console and open the Amazon RDS console at https://console.aws.amazon.com/rds/.
  2. In the navigation pane, choose **Databases**, and then choose the name of the DB instance.
  3. In the **Connectivity & security** tab, write down the values of the VPC ID under **VPC** and the subnet ID under **Subnets**.
  4. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
  5. In the navigation pane, choose **Internet Gateways**. Verify that there is an internet gateway attached to your VPC. Otherwise, choose **Create Internet Gateway** to create an internet

gateway. Select the internet gateway, and then choose **Attach to VPC** and follow the directions to attach it to your VPC.

6. In the navigation pane, choose **Subnets**, and then select your subnet.
7. On the **Route Table** tab, verify that there is a route with `0.0.0.0/0` as the destination and the internet gateway for your VPC as the target.

   If you're connecting to your instance using its IPv6 address, verify that there is a route for all IPv6 traffic (`::/0`) that points to the internet gateway. Otherwise, do the following:

   a. Choose the ID of the route table (rtb-*xxxxxxxx*) to navigate to the route table.
   b. On the **Routes** tab, choose **Edit routes**. Choose **Add route**, use `0.0.0.0/0` as the destination and the internet gateway as the target.

      For IPv6, choose **Add route**, use `::/0` as the destination and the internet gateway as the target.
   c. Choose **Save routes**.

   Also, if you are trying to connect to IPv6 endpoint, make sure that client IPv6 address range is authorized to connect to the DB instance.

For more information, see Working with a DB cluster in a VPC.

Testing a connection to a DB instance

You can test your connection to a DB instance using common Linux or Microsoft Windows tools.

From a Linux or Unix terminal, you can test the connection by entering the following. Replace *DB-instance-endpoint* with the endpoint and *port* with the port of your DB instance.

```
nc -zv DB-instance-endpoint port
```

For example, the following shows a sample command and the return value.

```
nc -zv postgresql1.c6c8mn7fake0.us-west-2.rds.amazonaws.com 8299

  Connection to postgresql1.c6c8mn7fake0.us-west-2.rds.amazonaws.com 8299 port [tcp/
vvr-data] succeeded!
```

Windows users can use Telnet to test the connection to a DB instance. Telnet actions aren't supported other than for testing the connection. If a connection is successful, the action returns no message. If a connection isn't successful, you receive an error message such as the following.

```
C:\>telnet sg-postgresql1.c6c8mntfake0.us-west-2.rds.amazonaws.com 819

  Connecting To sg-postgresql1.c6c8mntfake0.us-west-2.rds.amazonaws.com...Could not
 open
  connection to the host, on port 819: Connect failed
```

If Telnet actions return success, your security group is properly configured.

> **Note**
>
> Amazon RDS doesn't accept internet control message protocol (ICMP) traffic, including ping.

Troubleshooting connection authentication

In some cases, you can connect to your DB instance but you get authentication errors. In these cases, you might want to reset the master user password for the DB instance. You can do this by modifying the RDS instance.

Amazon RDS security issues

To avoid security issues, never use your master AWS user name and password for a user account. Best practice is to use your master AWS account to create users and assign those to DB user accounts. You can also use your master account to create other user accounts, if necessary.

For information about creating users, see Creating an IAM user in your AWS account. For information about creating users in AWS IAM Identity Center, see Manage identities in IAM Identity Center.

Error message "failed to retrieve account attributes, certain console functions may be impaired."

You can get this error for several reasons. It might be because your account is missing permissions, or your account hasn't been properly set up. If your account is new, you might not have waited for the account to be ready. If this is an existing account, you might lack permissions in your access policies to perform certain actions such as creating a DB instance. To fix the issue, your administrator needs to provide the necessary roles to your account. For more information, see the IAM documentation.

Resetting the DB instance owner password

If you get locked out of your DB cluster, you can log in as the master user. Then you can reset the credentials for other administrative users or roles. If you can't log in as the master user, the AWS account owner can reset the master user password. For details of which administrative accounts or roles you might need to reset, see Master user account privileges.

You can change the DB instance password by using the Amazon RDS console, the AWS CLI command modify-db-instance, or by using the ModifyDBInstance API operation. For more information about modifying a DB instance in a DB cluster, see Modifying a DB instance in a DB cluster.

Amazon RDS DB instance outage or reboot

A DB instance outage can occur when a DB instance is rebooted. It can also occur when the DB instance is put into a state that prevents access to it, and when the database is restarted. A reboot can occur when you manually reboot your DB instance. A reboot can also occur when you change a DB instance setting that requires a reboot before it can take effect.

A DB instance reboot occurs when you change a setting that requires a reboot, or when you manually cause a reboot. A reboot can occur immediately if you change a setting and request that the change take effect immediately. Or it can occur during the DB instance's maintenance window.

A DB instance reboot occurs immediately when one of the following occurs:

- You change the backup retention period for a DB instance from 0 to a nonzero value or from a nonzero value to 0. You then set **Apply Immediately** to `true`.
- You change the DB instance class, and **Apply Immediately** is set to `true`.

A DB instance reboot occurs during the maintenance window when one of the following occurs:

- You change the backup retention period for a DB instance from 0 to a nonzero value or from a nonzero value to 0, and **Apply Immediately** is set to `false`.

- You change the DB instance class, and **Apply Immediately** is set to `false`.

When you change a static parameter in a DB parameter group, the change doesn't take effect until the DB instance associated with the parameter group is rebooted. The change requires a manual reboot. The DB instance isn't automatically rebooted during the maintenance window.

Amazon RDS DB parameter changes not taking effect

In some cases, you might change a parameter in a DB parameter group but don't see the changes take effect. If so, you likely need to reboot the DB instance associated with the DB parameter group. When you change a dynamic parameter, the change takes effect immediately. When you change a static parameter, the change doesn't take effect until you reboot the DB instance associated with the parameter group.

You can reboot a DB instance using the RDS console. Or you can explicitly call the `RebootDBInstance` API operation. You can reboot without failover if the DB instance is in a Multi-AZ deployment. The requirement to reboot the associated DB instance after a static parameter change helps mitigate the risk of a parameter misconfiguration affecting an API call. An example of this is calling `ModifyDBInstance` to change the DB instance class. For more information, see Modifying parameters in a DB parameter group.

Freeable memory issues in Amazon Aurora

Freeable memory is the total random access memory (RAM) on a DB instance that can be made available to the database engine. It's the sum of the free operating-system (OS) memory and the available buffer and page cache memory. The database engine uses most of the memory on the host, but OS processes also use some RAM. Memory currently allocated to the database engine or used by OS processes isn't included in freeable memory. When the database engine is running out of memory, the DB instance can use the temporary space that is normally used for buffering and caching. As previously mentioned, this temporary space is included in freeable memory.

You use the `FreeableMemory` metric in Amazon CloudWatch to monitor the freeable memory. For more information, see Overview of monitoring metrics in Amazon Aurora.

If your DB instance consistently runs low on freeable memory or uses swap space, consider scaling up to a larger DB instance class. For more information, see Aurora DB instance classes.

You can also change the memory settings. For example, on Aurora MySQL , you might adjust the size of the `innodb_buffer_pool_size` parameter. This parameter is set by default to 75 percent of physical memory. For more MySQL troubleshooting tips, see How can I troubleshoot low freeable memory in an Amazon RDS for MySQL database?

For Aurora Serverless v2, `FreeableMemory` represents the amount of unused memory that's available when the Aurora Serverless v2 DB instance is scaled to its maximum capacity. You might have the instance scaled down to relatively low capacity, but it still reports a high value for `FreeableMemory`, because the instance can scale up. That memory isn't available right now, but you can get it if you need it.

For every Aurora capacity unit (ACU) that the current capacity is below the maximum capacity, `FreeableMemory` increases by approximately 2 GiB. Thus, this metric doesn't approach zero until the DB instance is scaled up as high as it can.

If this metric approaches a value of `0`, the DB instance has scaled up as much as it can. It's nearing the limit of its available memory. Consider increasing the maximum ACU setting for the cluster. If this metric approaches a value of `0` on a reader DB instance, consider adding additional reader DB instances to the cluster. That way, the read-only part of the workload can be spread across more DB instances, reducing the memory usage on each reader DB instance. For more information, see Important Amazon CloudWatch metrics for Aurora Serverless v2.

For Aurora Serverless v1, you can change the capacity range to use more ACUs. For more information, see Modifying an Aurora Serverless v1 DB cluster.

Amazon Aurora MySQL replication issues

Some MySQL replication issues also apply to Aurora MySQL. You can diagnose and correct these.

Topics

Diagnosing and resolving lag between read replicas

After you create a MySQL read replica and the replica is available, Amazon RDS first replicates the changes made to the source DB instance from the time the read replica create operation started.

During this phase, the replication lag time for the read replica is greater than 0. You can monitor this lag time in Amazon CloudWatch by viewing the Amazon RDS `AuroraBinlogReplicaLag` metric.

The `AuroraBinlogReplicaLag` metric reports the value of the `Seconds_Behind_Master` field of the MySQL `SHOW REPLICA STATUS` command. For more information, see SHOW REPLICA STATUS Statement in the MySQL documentation.

When the `AuroraBinlogReplicaLag` metric reaches 0, the replica has caught up to the source DB instance. If the `AuroraBinlogReplicaLag` metric returns -1, replication might not be active. To troubleshoot a replication error, see Diagnosing and resolving a MySQL read replication failure. A `AuroraBinlogReplicaLag` value of -1 can also mean that the `Seconds_Behind_Master` value can't be determined or is `NULL`.

Note

Previous versions of Aurora MySQL used `SHOW SLAVE STATUS` instead of `SHOW REPLICA STATUS`. If you are using Aurora MySQL version 1 or 2, then use `SHOW SLAVE STATUS`. Use `SHOW REPLICA STATUS` for Aurora MySQL version 3 and higher.

The `AuroraBinlogReplicaLag` metric returns -1 during a network outage or when a patch is applied during the maintenance window. In this case, wait for network connectivity to be restored or for the maintenance window to end before you check the `AuroraBinlogReplicaLag` metric again.

The MySQL read replication technology is asynchronous. Thus, you can expect occasional increases for the `BinLogDiskUsage` metric on the source DB instance and for the `AuroraBinlogReplicaLag` metric on the read replica. For example, consider a situation where a high volume of write operations to the source DB instance occur in parallel. At the same time, write operations to the read replica are serialized using a single I/O thread. Such a situation can lead to a lag between the source instance and read replica.

For more information about read replicas and MySQL, see Replication implementation details in the MySQL documentation.

You can reduce the lag between updates to a source DB instance and the subsequent updates to the read replica by doing the following:

- Set the DB instance class of the read replica to have a storage size comparable to that of the source DB instance.
- Make sure that parameter settings in the DB parameter groups used by the source DB instance and the read replica are compatible. For more information and an example, see the discussion of the `max_allowed_packet` parameter in the next section.
- Disable the query cache. For tables that are modified often, using the query cache can increase replica lag because the cache is locked and refreshed often. If this is the case, you might see less replica lag if you disable the query cache. You can disable the query cache by setting the `query_cache_type parameter` to 0 in the DB parameter group for the DB instance. For more information on the query cache, see Query cache configuration.
- Warm the buffer pool on the read replica for InnoDB for MySQL. For example, suppose that you have a small set of tables that are being updated often and you're using the InnoDB or XtraDB table schema. In this case, dump those tables on the read replica. Doing this causes the database engine to scan through the rows of those tables from the disk and then cache them in the buffer pool. This approach can reduce replica lag. The following shows an example.

  For Linux, macOS, or Unix:

  ```
  PROMPT> mysqldump \
      -h <endpoint> \
      --port=<port> \
      -u=<username> \
      -p <password> \
      database_name table1 table2 > /dev/null
  ```

  For Windows:

  ```
  PROMPT> mysqldump ^
      -h <endpoint> ^
      --port=<port> ^
      -u=<username> ^
      -p <password> ^
      database_name table1 table2 > /dev/null
  ```

Diagnosing and resolving a MySQL read replication failure

Amazon RDS monitors the replication status of your read replicas. RDS updates the **Replication State** field of the read replica instance to `Error` if replication stops for any reason. You can review the details of the associated error thrown by the MySQL engines by viewing the **Replication Error** field. Events that indicate the status of the read replica are also generated, including RDS-EVENT-0045, RDS-EVENT-0046, and RDS-EVENT-0057. For more information about events and subscribing to events, see Working with Amazon RDS event notification. If a MySQL error message is returned, check the error in the MySQL error message documentation.

Common situations that can cause replication errors include the following:

- The value for the `max_allowed_packet` parameter for a read replica is less than the `max_allowed_packet` parameter for the source DB instance.

  The `max_allowed_packet` parameter is a custom parameter that you can set in a DB parameter group. The `max_allowed_packet` parameter is used to specify the maximum size of data manipulation language (DML) that can be run on the database. In some cases, the `max_allowed_packet` value for the source DB instance might be larger than the `max_allowed_packet` value for the read replica. If so, the replication process can throw an error and stop replication. The most common error is `packet bigger than 'max_allowed_packet' bytes`. You can fix the error by having the source and read replica use DB parameter groups with the same `max_allowed_packet` parameter values.
- Writing to tables on a read replica. If you're creating indexes on a read replica, you need to have the `read_only` parameter set to *0* to create the indexes. If you're writing to tables on the read replica, it can break replication.
- Using a nontransactional storage engine such as MyISAM. Read replicas require a transactional storage engine. Replication is only supported for the following storage engines: InnoDB for MySQL or MariaDB.

  You can convert a MyISAM table to InnoDB with the following command:

  ```
  alter table <schema>.<table_name> engine=innodb;
  ```
- Using unsafe nondeterministic queries such as `SYSDATE()`. For more information, see Determination of safe and unsafe statements in binary logging in the MySQL documentation.

The following steps can help resolve your replication error:

- If you encounter a logical error and you can safely skip the error, follow the steps described in Skipping the current replication error. Your Aurora MySQL DB instance must be running a version that includes the `mysql_rds_skip_repl_error` procedure. For more information, see mysql_rds_skip_repl_error.
- If you encounter a binary log (binlog) position issue, you can change the replica replay position. You do so with the `mysql.rds_next_master_log` command for Aurora MySQL version 1 and 2. You do so with the `mysql.rds_next_source_log` command for Aurora MySQL version 3 and higher. Your Aurora MySQL DB instance must be running a version that supports this command to change the replica replay position. For version information, see mysql_rds_next_master_log.
- If you encounter a temporary performance issue due to high DML load, you can set the `innodb_flush_log_at_trx_commit` parameter to 2 in the DB parameter group on the read replica. Doing this can help the read replica catch up, though it temporarily reduces atomicity, consistency, isolation, and durability (ACID).
- You can delete the read replica and create an instance using the same DB instance identifier. This way, the endpoint remains the same as that of your old read replica.

If a replication error is fixed, the **Replication State** changes to **replicating**. For more information, see Troubleshooting a MySQL read replica problem.

Replication stopped error

When you call the `mysql.rds_skip_repl_error` command, you might receive an error message stating that replication is down or disabled.

This error message appears because replication is stopped and can't be restarted.

If you need to skip a large number of errors, the replication lag can increase beyond the default retention period for binary log files. In this case, you might encounter a fatal error due to binary log files being purged before they have been replayed on the replica. This purge causes replication to stop, and you can no longer call the `mysql.rds_skip_repl_error` command to skip replication errors.

You can mitigate this issue by increasing the number of hours that binary log files are retained on your replication source. After you have increased the binlog retention time, you can restart replication and call the `mysql.rds_skip_repl_error` command as needed.

To set the binlog retention time, use the mysql_rds_set_configuration procedure. Specify a configuration parameter of 'binlog retention hours' along with the number of hours to retain binlog files on the DB cluster, up to 2160 (90 days). The default for Aurora MySQL is 24 (1 day). The following example sets the retention period for binlog files to 48 hours.

```
CALL mysql.rds_set_configuration('binlog retention hours', 48);
```

Amazon RDS API reference

In addition to the AWS Management Console and the AWS Command Line Interface (AWS CLI), Amazon RDS also provides an API. You can use the API to automate tasks for managing your DB instances and other objects in Amazon RDS.

- For an alphabetical list of API operations, see Actions.
- For an alphabetical list of data types, see Data types.
- For a list of common query parameters, see Common parameters.
- For descriptions of the error codes, see Common errors.

For more information about the AWS CLI, see AWS Command Line Interface reference for Amazon RDS.

Topics

- Using the Query API
- Troubleshooting applications on Aurora

Using the Query API

The following sections briefly discuss the parameters and request authentication used with the Query API.

For general information about how the Query API works, see Query requests in the *Amazon EC2 API Reference.*

Query parameters

HTTP Query-based requests are HTTP requests that use the HTTP verb GET or POST and a Query parameter named `Action`.

Each Query request must include some common parameters to handle authentication and selection of an action.

Some operations take lists of parameters. These lists are specified using the `param.`*n* notation. Values of *n* are integers starting from 1.

For information about Amazon RDS Regions and endpoints, go to Amazon Relational Database Service (RDS) in the Regions and Endpoints section of the *Amazon Web Services General Reference*.

Query request authentication

You can only send Query requests over HTTPS, and you must include a signature in every Query request. You must use either AWS signature version 4 or signature version 2. For more information, see Signature Version 4 signing process and Signature version 2 signing process.

Troubleshooting applications on Aurora

Amazon RDS provides specific and descriptive errors to help you troubleshoot problems while interacting with the Amazon RDS API.

Topics

- Retrieving errors
- Troubleshooting tips

For information about troubleshooting for Amazon RDS DB instances, see Troubleshooting for Amazon Aurora.

Retrieving errors

Typically, you want your application to check whether a request generated an error before you spend any time processing results. The easiest way to find out if an error occurred is to look for an `Error` node in the response from the Amazon RDS API.

XPath syntax provides a simple way to search for the presence of an `Error` node. It also provides a relatively easy way to retrieve the error code and message. The following code snippet uses Perl and the XML::XPath module to determine if an error occurred during a request. If an error occurred, the code prints the first error code and message in the response.

```
use XML::XPath;
    my $xp = XML::XPath->new(xml =>$response);
    if ( $xp->find("//Error") )
    {print "There was an error processing your request:\n", " Error code: ",
    $xp->findvalue("//Error[1]/Code"), "\n", " ",
    $xp->findvalue("//Error[1]/Message"), "\n\n"; }
```

Troubleshooting tips

We recommend the following processes to diagnose and resolve problems with the Amazon RDS API:

- Verify that Amazon RDS is operating normally in the AWS Region that you're targeting by checking http://status.aws.amazon.com.
- Check the structure of your request.

  Each Amazon RDS operation has a reference page in the *Amazon RDS API Reference*. Double-check that you are using parameters correctly. For ideas about what might be wrong, look at the sample requests or user scenarios to see if those examples do similar operations.
- Check AWS re:Post.

  Amazon RDS has a development community where you can search for solutions to problems others have experienced along the way. To view the topics, go to AWS re:Post.

Document history

Current API version: 2014-10-31

The following table describes important changes to the *Amazon Aurora User Guide*. For notification about updates to this documentation, you can subscribe to an RSS feed. For information about Amazon Relational Database Service (Amazon RDS), see the *Amazon Relational Database Service User Guide*.

> **Note**
>
> Before August 31, 2018, Amazon Aurora was documented in the *Amazon Relational Database Service User Guide*. For earlier Aurora document history, see Document history in the *Amazon Relational Database Service User Guide*.

You can filter new Amazon Aurora features on the What's New with Database? page. For **Products**, choose **Amazon Aurora**. Then search using keywords such as **`global database`** or **`Serverless`**.

Change	Description	Date
AWS Python Driver generally available	The Amazon Web Services (AWS) Python Driver is designed as an advanced Python wrapper. This wrapper is complementary to and extends the functionality of the open-source Psycopg driver. For more information, see Connecting to Aurora DB clusters with the AWS drivers.	May 23, 2024
Zero-ETL integrations available in China Regions	Zero-ETL integrations are now available in the AWS Regions China (Beijing) and China (Ningxia). For more informati	May 21, 2024

	on, see Zero-ETL integrations with Amazon Redshift.	
RDS Proxy is available in more Regions	RDS Proxy is now available in the Asia Pacific (Hyderabad), Asia Pacific (Melbourne), Middle East (UAE), Israel (Tel Aviv), Canada West (Calgary), and Europe (Zurich) regions. For more information about RDS Proxy, see Using Amazon RDS Proxy.	May 21, 2024
Amazon RDS Extended Support	Creating or restoring an Aurora MySQL version 2 or 3, or Aurora PostgreSQL version 11 database now automatically enrolls that database into Amazon RDS Extended Support so your existing applications continue to work as they are. You can opt out of RDS Extended Support to avoid charges after the Aurora end of standard support date for your database engine. For more information, see Using Amazon RDS Extended Support.	March 21, 2024

Data filtering for zero-ETL integrations	Amazon RDS supports data filtering at the database and table level for zero-ETL integrations with Amazon Redshift. For more information, see Data filtering for Aurora zero-ETL integrations with Amazon Redshift.	March 20, 2024
Aurora MySQL integrations with Amazon Bedrock	You can now integrate Amazon Aurora MySQL databases with Amazon Bedrock to power generative AI applications. For more information, see Using Amazon Aurora machine learning with Aurora MySQL.	March 8, 2024
New AWS managed policy	Amazon RDS added a new managed policy named `AmazonRDSCustomInstanceProfileRolePolicy` to allow RDS Custom to perform automation actions and database management tasks through an EC2 instance profile. For more information, see Amazon RDS updates to AWS managed policies.	February 27, 2024

Amazon RDS support for AWS Secrets Manager in the Israel (Tel Aviv) Region	Amazon RDS supports Secrets Manager in the Israel (Tel Aviv) Region. For more information, see Password management with Amazon RDS and AWS Secrets Manager.	February 21, 2024
Amazon RDS Extended Support	Amazon RDS now automatically enables Amazon RDS Extended Support when Aurora MySQL and Aurora PostgreSQL major engine versions in your DB clusters and global clusters reach the Aurora end of standard support date. For more information, see Using Amazon RDS Extended Support.	February 15, 2024
Aurora PostgreSQL 16.1 supports Babelfish for Aurora PostgreSQL 4.0.0	Aurora PostgreSQL 16.1 supports Babelfish 4.0.0. For a list of new features, see 16.1. For a list of features supported in each Babelfish release, see Supported functionality in Babelfish by version. For usage information, see Working with Babelfish for Aurora PostgreSQL.	January 31, 2024

Update to default CA Certificate	The default CA certificate is set to `rds-ca-rsa2048-g1`. For more information, see Using SSL/TLS to encrypt a connection to a DB cluster.	January 26, 2024
RDS Proxy is available in the Europe (Spain) Region	RDS Proxy is now available in the Europe (Spain) region. For more information about RDS Proxy, see Using Amazon RDS Proxy.	January 8, 2024
RDS Data API with Aurora PostgreSQL Serverless v2 and provisioned	You can now use RDS Data API with Aurora PostgreSQL Serverless v2 and provisioned DB clusters. With RDS Data API, you can access your Aurora clusters through a secure HTTP endpoint and run SQL statements without using database drivers or managing connections. For more information, see Using RDS Data API.	December 21, 2023
Aurora PostgreSQL integrations with Amazon Bedrock	You can now integrate Amazon Aurora PostgreSQL databases with Amazon Bedrock to power generative AI applications. For more information, see Using Amazon Aurora machine learning with Aurora PostgreSQL.	December 21, 2023

Amazon Aurora is available in the Canada West (Calgary) Region	Amazon Aurora is now available in the Canada West (Calgary) Region. For more information, see Regions and Availability Zones.	December 20, 2023
Amazon RDS supports viewing and responding to recommendations	Amazon Aurora recommendations now includes threshold based proactive and machine learning based reactive recommendations. For more information, see Viewing and responding to Amazon Aurora recommendations.	December 19, 2023
Aurora PostgreSQL zero-ETL integrations with Amazon Redshift (preview)	You can now create zero-ETL integrations with Amazon Redshift using an Aurora PostgreSQL source DB cluster. For the preview release, you must create all integrations in the Amazon RDS Database Preview Environment, in the US East (Ohio) (us-east-2) AWS Region. For more information, see Working with Aurora zero-ETL integrations with Amazon Redshift.	November 28, 2023
Amazon Aurora PostgreSQL supports global database write forwarding	You can now enable write forwarding on secondary clusters in an Aurora PostgreSQL-based global database. For more information, see Using write forwarding in an Aurora PostgreSQL global database.	November 9, 2023

Aurora PostgreSQL support for Optimized Reads	You can achieve faster query processing for Aurora PostgreSQL with Aurora Optimized Reads. For more information, see Improving query performance for Aurora PostgreSQL with Aurora Optimized Reads.	November 8, 2023
Amazon RDS exports Performance Insights metrics to Amazon CloudWatch	Performance Insights lets you export the preconfigured or custom metrics dashboards to Amazon CloudWatch. The exported metrics dashboards are available to view in the CloudWatch console. You can also export a selected Performance Insights metric widget and view the metrics data in the CloudWatch console. For more information, see Exporting Performance Insights metrics to CloudWatch.	November 8, 2023
Aurora MySQL zero-ETL integrations with Amazon Redshift general availability	Zero-ETL integrations with Amazon Redshift are now generally available for Aurora MySQL. For more information, see Working with Aurora zero-ETL integrations with Amazon Redshift.	November 7, 2023

Aurora PostgreSQL support for RDS Blue/Green Deployments	You can now create a blue/green deployment from an Aurora PostgreSQL DB cluster. For more information, see Using Amazon RDS Blue/Gree n Deployments for database updates.	October 26, 2023
Aurora MySQL supports server-side encryption with AWS KMS keys (SSE-KMS)	In Aurora MySQL version 3.05 and higher, you can use SSE-KMS, including AWS managed keys and customer managed keys, for server-side encryption of data that you load from or save to Amazon S3. For more information, see Loading data into an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket and Saving data from an Amazon Aurora MySQL DB cluster from text files in an Amazon S3 bucket.	October 25, 2023
Aurora MySQL optimizations reduce database restart time	In Aurora MySQL version 3.05 and higher, we've introduce d optimizations that reduce the database restart time. These optimizations provide up to 65% less downtime than without optimizations, and fewer disruptions to your database workloads, after a restart. For more information, see Optimizations to reduce database restart time.	October 25, 2023

Update to AWS managed policies	The `AmazonRDSPerformanceInsightsReadOnly` and `AmazonRDSPerformanceInsightsFullAccess` managed policies now includes `Sid` (statement ID) as an identifier in the policy statement. For more information, see Amazon RDS updates to AWS managed policies.	October 23, 2023
Amazon RDS publishes Performance Insights counter metrics to Amazon CloudWatch	The **DB_PERF_INSIGHTS** metric math function in the CloudWatch console allows you to query Amazon RDS for Performance Insights counter metrics. For more information, see Creating CloudWatch alarms to monitor Amazon Aurora.	September 20, 2023
Amazon Aurora supports point-in-time recovery (PITR) with AWS Backup	You can now manage Aurora automated (continuous) backups in AWS Backup and restore to specified times from them. For more information, see Restoring a DB cluster to a specified time using AWS Backup.	September 7, 2023

Amazon RDS Extended Support	Amazon Aurora announces the upcoming ability to continue running Aurora MySQL and Aurora PostgreSQL major engine versions in your DB instances past the Aurora end of standard support date. For more information, see Using Amazon RDS Extended Support.	September 1, 2023
Amazon Aurora MySQL extends support for Percona XtraBackup	You can now perform physical migrations of MySQL 8.0 databases to Aurora MySQL version 3 DB clusters. For more information, see Physical migration from MySQL by using Percona XtraBackup and Amazon S3.	August 24, 2023

Aurora global database supports global database failover	Aurora global database now supports managed global failover, allowing you to more easily recover from a true Regional disaster or complete service-level outage. To learn more about this feature, see Performing managed failovers for Aurora global databases. The feature previously called "managed planned failover" is now called "switchover." For information about switchovers, see Performing switchovers for Amazon Aurora global databases.	August 21, 2023
Update to AWS managed policy permissions	The `AmazonRDSFullAccess` managed policy has new permissions that allows you to generate, view, and delete the performance analysis report for a time period. For more information, see Amazon RDS updates to AWS managed policies.	August 17, 2023

Update to AWS managed policy permissions	The addition of new permissions to `AmazonRDSPerformanceInsightsReadOnly` managed policy and addition of new managed policy `AmazonRDSPerformanceInsightsFullAccess` allows you generate a DB load analysis report for a time period. For more information, see Amazon RDS updates to AWS managed policies.	August 16, 2023
Amazon RDS supports DB load time period analysis with Performance Insights	Performance Insights allows you to create performance analysis reports for a specific period of time. The report provides the insights identified and recommendations to resolve the performance issues. For more information, see Analyzing DB load for a period of time.	August 16, 2023
Amazon Aurora supports retaining automated backups for DB clusters	You can now retain automated backups for deleted Aurora clusters and restore them to a specified point in time. For more information, see Retaining automated backups.	August 4, 2023

Amazon Aurora is available in the Israel (Tel Aviv) Region	Amazon Aurora is now available in the Israel (Tel Aviv) Region. For more information, see Regions and Availability Zones.	August 1, 2023
Amazon Aurora MySQL supports local (in-cluster) write forwarding	You can now forward write operations from a reader DB instance to a writer DB instance within an Aurora MySQL DB cluster. For more information, see Using write forwarding in an Amazon Aurora MySQL DB cluster.	July 31, 2023
Amazon Aurora supports Aurora Serverless v2 in an additional AWS Region	You can now create Aurora Serverless v2 DB clusters in the Asia Pacific (Melbourne) AWS Region. For information about Aurora Serverless v2, see Using Aurora Serverless v2.	June 28, 2023
Amazon Aurora introduces zero-ETL integrations with Amazon Redshift (preview)	Zero-ETL integrations provide a fully managed solution for making transactional data available in Amazon Redshift within seconds of it being written to an Aurora MySQL DB cluster. For more information, see Working with Aurora zero-ETL integrations with Amazon Redshift.	June 28, 2023

Amazon RDS provides combined Performance Insights and CloudWatch metrics view in the Performance Insights dashboard	Amazon RDS now provides a consolidated view of Performance Insights and CloudWatch metrics in the Performance Insights dashboard. For more information, see Viewing combined metrics in the Amazon RDS console.	May 24, 2023
Amazon Aurora supports the db.r7g instance classes	You can now use the db.r7g instance classes to create Aurora DB clusters. For more information, see Aurora DB instance classes.	May 11, 2023
Amazon Aurora supports a new DB cluster storage configuration	With Aurora I/O-Optimized, you pay only for the usage and storage of your DB clusters, with no additional charges for read and write I/O operations. For more information, see Storage configurations for Amazon Aurora DB clusters.	May 11, 2023
Amazon Aurora supports Aurora Serverless v2 in additional AWS Regions	You can now create Aurora Serverless v2 DB clusters in the following AWS Regions: Asia Pacific (Hyderabad), Europe (Spain) Europe (Zurich), and Middle East (UAE). For information about Aurora Serverless v2, see Using Aurora Serverless v2.	May 4, 2023

Aurora Serverless v1 supports conversion to provisioned	You can convert an Aurora Serverless v1 DB cluster directly to a provisioned DB cluster. For more information, see Converting an Aurora Serverless v1 DB cluster to provisioned.	April 27, 2023
Aurora Serverless v1 supports Amazon Aurora PostgreSQL version 13	You can now create Aurora Serverless v1 DB clusters that run Aurora PostgreSQL version 13. For more information, see Aurora Serverless v1.	April 27, 2023
Amazon Aurora support for AWS Secrets Manager in the China Regions	Amazon Aurora supports Secrets Manager in the China (Beijing) and China (Ningxia) Regions. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.	April 20, 2023
Amazon Aurora supports publishing events with tags to topic subscribers	Amazon Aurora event notifications sent to Amazon Simple Notification Service (Amazon SNS) or Amazon EventBridge now contain event tags in the message body. These tags provide the resource data that was affected by the service event. For more information, see Amazon RDS event notification tags and attributes.	April 17, 2023

Update to IAM service-linked role permissions	The `AmazonRDSFullAccess` and `AmazonRDSReadOnlyAccess` policies now grants additional permissions to allow the display of Amazon DevOps Guru findings in the RDS console. For more information, see Amazon RDS updates to AWS managed policies.	March 30, 2023
Amazon Aurora supports global databases in the Asia Pacific (Melbourne) Region	You can now create Aurora global databases in the Asia Pacific (Melbourne) Region. For information about Aurora global databases, see Using Amazon Aurora global databases.	March 22, 2023
Update to AWS managed policy permissions	The `AmazonRDSFullAccess` and `AmazonRDSReadOnlyAccess` policies now grants additional permissions to Amazon CloudWatch. For more information, see Amazon RDS updates to AWS managed policies.	March 16, 2023
RDS Proxy is available in the China Regions	RDS Proxy is now available in the China (Beijing) and China (Ningxia) regions. For more information about RDS Proxy, see Using Amazon RDS Proxy.	March 15, 2023

Amazon Aurora supports Aurora Serverless v2 in the China Regions	Aurora Serverless v2 is now available in the China (Beijing) and China (Ningxia) Regions. For more information, see Aurora Serverless v2.	March 15, 2023
RDS Proxy is available in the Asia Pacific (Jakarta) Region	RDS Proxy is now available in the Asia Pacific (Jakarta) Region. For more information about RDS Proxy, see Using Amazon RDS Proxy.	March 8, 2023
Amazon Aurora MySQL supports Kerberos authentication	You can now use Kerberos authentication to authenticate users when they connect to your Aurora MySQL DB clusters. For more information, see Using Kerberos authentication for Aurora MySQL.	March 8, 2023
Amazon Aurora supports global databases in additional AWS Regions	You can now create Aurora global databases in the following Regions: Africa (Cape Town), Asia Pacific (Hong Kong), Asia Pacific (Hyderabad), Asia Pacific (Jakarta), Europe (Milan), Europe (Spain) Europe (Zurich), Middle East (Bahrain), and Middle East (UAE). For information about Aurora global databases, see Using Amazon Aurora global databases.	March 6, 2023

Amazon Aurora supports copying DB cluster snapshots in additional AWS Regions	You can now copy DB cluster snapshots in the following Regions: Africa (Cape Town), Asia Pacific (Hong Kong), Asia Pacific (Hyderabad), Asia Pacific (Jakarta), Asia Pacific (Melbourne), Europe (Milan), Europe (Spain) Europe (Zurich), Middle East (Bahrain), and Middle East (UAE). For information about copying DB cluster snapshots , see Copying a DB cluster snapshot.	March 6, 2023
Amazon DevOps Guru for RDS supports proactive insights	Amazon DevOps Guru for RDS publishes proactive insights with recommendations to help you address issues in your Aurora databases before they are predicted to happen. For more information, see How DevOps Guru for RDS works.	February 28, 2023
Amazon Aurora MySQL version 1 is deprecated	Aurora MySQL version 1 (compatible with MySQL 5.6) has been deprecated. For more information, see How long Amazon Aurora major versions remain available.	February 28, 2023

Aurora Serverless v1 supports setting the DB cluster maintenance window	You can now set the maintenance window for Aurora Serverless v1 DB clusters. For more information, see Adjusting the preferred DB cluster maintenance window.	February 27, 2023
Amazon Aurora supports Database Activity Streams in the Asia Pacific (Hyderabad), Europe (Spain), and Middle East (UAE) Regions.	For more information, see Database Activity Streams.	January 27, 2023
Amazon Aurora is available in the Asia Pacific (Melbourne) Region	Amazon Aurora is now available in the Asia Pacific (Melbourne) Region. For more information, see Regions and Availability Zones.	January 23, 2023
Specify certificate authority (CA) during DB cluster creation	You can now specify which CA to use for a DB cluster's server certificate during DB cluster creation. For more information, see Certificate authorities.	January 5, 2023

Aurora MySQL 3.* support for backtracking	Aurora MySQL 3.* versions now offer a quick way to recover from user errors, such as dropping the wrong table or deleting the wrong row. Backtrack allows you to move your database to a prior point in time without needing to restore from a backup, and it completes within seconds, even for large databases. For details, see Backtracking an Aurora DB cluster.	January 4, 2023
Use Amazon RDS Blue/Green Deployments available in additional AWS Regions	The Blue/Green Deployments feature is now available in the China (Beijing) and China (Ningxia) Regions. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.	December 22, 2022
Update to IAM service-linked role permissions	The AmazonRDSServiceRolePolicy policy now grants additional permissions to AWS Secrets Manager. For more information, see Amazon RDS updates to AWS managed policies.	December 22, 2022

Amazon Aurora integrates with AWS Secrets Manager for password management	Aurora can manage the master user password for a DB cluster in Secrets Manager. For more information, see Password management with Amazon Aurora and AWS Secrets Manager.	December 22, 2022
Amazon Aurora supports Aurora Serverless v2 in additional AWS Regions	Aurora Serverless v2 is now available in the Africa (Cape Town) and Europe (Milan) Regions. For more information, see Aurora Serverless v2.	December 21, 2022
Aurora PostgreSQL supports RDS Proxy with PostgreSQL 14	You can now create an RDS Proxy with an Aurora PostgreSQL 14 DB cluster. For more information about RDS Proxy, see Using Amazon RDS Proxy.	December 13, 2022
Amazon Aurora alerts you to recent anomalies detected by Amazon DevOps Guru	The database details page of the console alerts you both to current and anomalies that occurred in the past 24 hours. For more information, see How DevOps Guru for RDS works.	December 13, 2022
Amazon RDS Proxy supports global databases	You can now use RDS Proxy with Aurora global databases . For more information, see Using RDS Proxy with Aurora global databases.	December 7, 2022

Aurora PostgreSQL DB clusters support Trusted Language Extensions for PostgreSQL	Trusted Language Extensions for PostgreSQL is an open source development kit that allows you to build high performance PostgreSQL extensions and safely run them on your Aurora PostgreSQL DB cluster. For more information, see Working with Trusted Language Extensions for PostgreSQL.	November 30, 2022
Amazon GuardDuty RDS Protection monitors for access threats	When you turn on GuardDuty RDS Protection, GuardDuty consumes RDS login events from your Aurora databases, monitors these events, and profiles them for potential insider threats or external actors. When GuardDuty RDS Protection detects a potential threat, GuardDuty generates a new finding with details about the potentially compromised database. For more information, see Monitoring threats with GuardDuty RDS Protection.	November 30, 2022

Use Amazon RDS Blue/Green Deployments for database updates	You can make changes to a DB cluster in a staging environment and test the changes without affecting your production DB cluster. When you are ready, you can promote the staging environment to be the new production environment, with minimal downtime. For more information, see Using Amazon RDS Blue/Green Deployments for database updates.	November 27, 2022
Amazon Aurora is available in the Asia Pacific (Hyderabad) Region	Amazon Aurora is now available in the Asia Pacific (Hyderabad) Region. For more information, see Regions and Availability Zones.	November 22, 2022
Amazon Aurora is available in the Europe (Spain) Region	Amazon Aurora is now available in the Europe (Spain) Region. For more information, see Regions and Availability Zones.	November 16, 2022
Amazon Aurora is available in the Europe (Zurich) Region	Amazon Aurora is now available in the Europe (Zurich) Region. For more information, see Regions and Availability Zones.	November 9, 2022

Amazon Aurora supports exporting data to Amazon S3 from DB clusters	You can now export Aurora cluster data directly to S3, without having to create a snapshot first. For more information, see Exporting DB cluster data to Amazon S3.	October 27, 2022
Amazon Aurora MySQL supports faster exports to Amazon S3	You can now see up to 10x faster performance for exporting DB cluster snapshot data to S3 for MySQL 5.7- and 8.0-compatible Aurora MySQL clusters. For more information, see Exporting DB cluster snapshot data to Amazon S3.	October 20, 2022
Amazon Aurora supports automatically setting up connectivity between an Aurora DB cluster and an EC2 instance	You can use the AWS Management Console to set up connectivity between an existing Aurora DB cluster and an EC2 instance. For more information, see Connecting an EC2 instance and an Aurora DB cluster automatically.	October 14, 2022
AWS JDBC Driver for PostgreSQL generally available	The AWS JDBC Driver for PostgreSQL is a client driver designed for Aurora PostgreSQL. The AWS JDBC Driver for PostgreSQL is now generally available. For more information, see Connecting with the AWS JDBC Driver for PostgreSQL.	October 6, 2022

Amazon Aurora supports in-place upgrade for MySQL 5.7-compatible Aurora MySQL	You can perform an in-place upgrade to change an existing MySQL 5.7-compatible Aurora MySQL cluster into a MySQL 8.0-compatible Aurora MySQL cluster. For more information, see Upgrading from Aurora MySQL 2.x to 3.x.	September 26, 2022
Performance Insights shows the top 25 SQL queries	In the Performance Insights dashboard, the **Top SQL** tab shows the 25 SQL queries that are contributing the most to DB load. For more information, see Overview of the Top SQL tab.	September 13, 2022
Aurora MySQL supports a new DB instance class	You can now use the db.r6i DB instance class for Aurora MySQL DB clusters. For more information, see DB instance classes.	September 13, 2022
Amazon Aurora is available in the Middle East (UAE) Region	Amazon Aurora is now available in the Middle East (UAE) Region. For more information, see Regions and Availability Zones.	August 30, 2022

Amazon Aurora supports automatically setting up connectivity with an EC2 instance	When you create an Aurora DB cluster, you can use the AWS Management Console to set up connectivity between an Amazon Elastic Compute Cloud instance and the new DB cluster. For more information, see Configure automatic network connectivity with an EC2 instance.	August 22, 2022
Amazon Aurora supports dual-stack mode	DB clusters can now run in dual-stack mode. In dual-stack mode, resources can communicate with the DB cluster over IPv4, IPv6, or both. For more information, see Amazon Aurora IP addressing.	August 17, 2022
Amazon Aurora supports in-place upgrade for PostgreSQL-compatible Aurora Serverless v1	You can perform an in-place upgrade for a PostgreSQL 10-compatible Aurora Serverless v1 cluster to change an existing cluster into a PostgreSQL 11-compatible Aurora Serverless v1 cluster. For the in-place upgrade procedure, see Modifying an Aurora Serverless v1 DB cluster.	August 8, 2022

Performance Insights supports the Asia Pacific (Jakarta) Region	Formerly, you couldn't use Performance Insights in the Asia Pacific (Jakarta) Region. This restriction has been removed. For more informati on, see AWS Region support for Performance Insights.	July 21, 2022
Amazon Aurora supports a new DB instance class	You can now use the db.r6i DB instance class for Aurora PostgreSQL DB clusters. For more information, see DB instance classes.	July 14, 2022
RDS Performance Insights supports additional retention periods	Previously, Performance Insights offered only two retention periods: 7 days (default) or 2 years (731 days). Now, if you need to retain your performance data for longer than 7 days, you can specify from 1–24 months. For more informati on, see Pricing and data retention for Performance Insights.	July 1, 2022
Amazon Aurora supports in-place upgrade for MySQL-compatible Aurora Serverless v1	You can perform an in-place upgrade for a MySQL 5.6-compatible Aurora Serverles s v1 cluster to change an existing cluster into a MySQL 5.7-compatible Aurora Serverless v1 cluster. For the in-place upgrade procedure , see Modifying an Aurora Serverless v1 DB cluster.	June 16, 2022

Aurora supports turning on Amazon DevOps Guru in the RDS console	You can turn on DevOps Guru coverage for your Aurora databases from within the RDS console. For more information, see Setting up DevOps Guru for RDS.	June 9, 2022
Amazon Aurora supports publishing events to encrypted Amazon SNS topics	Amazon Aurora can now publish events to Amazon Simple Notification Service (Amazon SNS) topics that have server-side encryption (SSE) enabled, for additional protection of events that carry sensitive data. For more information, see Subscribing to Amazon RDS event notification.	June 1, 2022
Amazon RDS publishes usage metrics to Amazon CloudWatch	The `AWS/Usage` namespace in Amazon CloudWatch includes account-level usage metrics for your Amazon RDS service quotas. For more information, see Amazon CloudWatch usage metrics for Amazon Aurora.	April 28, 2022

Data API result sets in JSON format	An optional parameter for the `ExecuteStatement` function causes the query result set to be returned as a string in JSON format. The JSON result set is simple and convenient to transform into a data structure in your application's language. For more information, see Processing query results in JSON format.	April 27, 2022
Amazon Aurora Serverless v2 is now generally available	Amazon Aurora Serverless v2 is generally available for all users. For more information, see Using Aurora Serverless v2.	April 21, 2022
Aurora MySQL supports configurable cipher suites	With Aurora MySQL, you can now use configurable cipher suites to control the connection encryption that your database server accepts. For more information, see Configuring cipher suites for connections to Aurora MySQL DB clusters.	April 15, 2022
Aurora PostgreSQL supports RDS Proxy with PostgreSQL 13	You can now create an RDS Proxy with an Aurora PostgreSQL 13 DB cluster. For more information about RDS Proxy, see Using Amazon RDS Proxy.	April 4, 2022

Release Notes for Aurora PostgreSQL	There is now a separate guide for the Amazon Aurora PostgreSQL release notes. For more information, see *Release Notes for Aurora PostgreSQL*.	March 22, 2022
Release Notes for Aurora MySQL	There is now a separate guide for the Amazon Aurora MySQL release notes. For more information, see *Release Notes for Aurora MySQL*.	March 22, 2022
Aurora PostgreSQL supports multi-major version upgrades	You can now perform version upgrades of Aurora PostgreSQL DB clusters across multiple major versions. For more information, see How to perform a major version upgrade.	March 4, 2022
Aurora PostgreSQL supports configurable cipher suites	With Aurora PostgreSQL versions 11.8 and higher, you can now use configurable cipher suites to control the connection encryption that your database server accepts. For information about using configurable cipher suites with Aurora PostgreSQL, see Configuring cipher suites for connections to Aurora PostgreSQL DB clusters.	March 4, 2022

AWS JDBC Driver for MySQL generally available	The AWS JDBC Driver for MySQL is a client driver designed for the high availability of Aurora MySQL. The AWS JDBC Driver for MySQL is now generally available. For more information, see Connecting with the Amazon Web Services JDBC Driver for MySQL.	March 2, 2022
Aurora PostgreSQL 13.5 supports Babelfish for Aurora PostgreSQL 1.1.0	Aurora PostgreSQL 13.5 supports Babelfish 1.1.0. For a list of new features, see 13.5. For a list of features supported in each Babelfish release, see Supported functionality in Babelfish by version. For usage information, see Working with Babelfish for Aurora PostgreSQL.	February 28, 2022
Amazon Aurora supports Database Activity Streams in the Asia Pacific (Jakarta) Region	For more information, see Support for AWS Regions for database activity streams.	February 16, 2022

Performance Insights supports new API operations	Performance Insights now supports the following API operations: `GetResourceMetadata` , `ListAvailableResourceDimensions` , and `ListAvailableResourceMetrics` . For more information, see Retrieving metrics with the Performance Insights API in this manual and the *Amazon RDS Performance Insights API Reference*.	January 12, 2022
Amazon RDS Proxy supports events	RDS Proxy now generates events that you can subscribe to and view in CloudWatch Events or configure to send to Amazon EventBridge. For more information, see Working with RDS Proxy events.	January 11, 2022
RDS Proxy available in additional AWS Regions	RDS Proxy is now available in the following Regions: Africa (Cape Town), Asia Pacific (Hong Kong), Asia Pacific (Osaka), Europe (Milan), Europe (Paris), Europe (Stockholm), Middle East (Bahrain), and South America (São Paulo). For more information about RDS Proxy, see Using Amazon RDS Proxy.	January 5, 2022

Amazon Aurora is available in the Asia Pacific (Jakarta) Region	Amazon Aurora is now available in the Asia Pacific (Jakarta) Region. For more information, see Regions and Availability Zones.	December 13, 2021
DevOps Guru for Amazon RDS provides detailed insights and recommendations for Amazon Aurora	DevOps Guru for RDS mines Performance Insights for performance-related data. Using this data, the service analyzes the performance of your Amazon Aurora DB instances and can help you resolve performance issues. To learn more, see Analyzing performance anomalies with DevOps Guru for RDS in this guide and see Overview of DevOps Guru for RDS in the *Amazon DevOps Guru User Guide*.	December 1, 2021
Aurora PostgreSQL supports RDS Proxy with PostgreSQL 12	You can now create an RDS Proxy with an Aurora PostgreSQL 12 database cluster. For more information about RDS Proxy, see Using Amazon RDS Proxy.	November 22, 2021
Aurora supports AWS Graviton2 instance classes for Database Activity Streams	You can use database activity streams with the db.r6g instance class for Aurora MySQL and Aurora PostgreSQ L. For more information, see Supported DB instance classes.	November 3, 2021

Amazon Aurora support for cross-account AWS KMS keys	You can use a KMS key from a different AWS account for encryption when exporting DB snapshots to Amazon S3. For more information, see Exporting DB snapshot data to Amazon S3.	November 3, 2021
Amazon Aurora supports Babelfish for Aurora PostgreSQL	Babelfish for Aurora PostgreSQL extends your Amazon Aurora PostgreSQL-Compatible Edition database with the ability to accept database connections from Microsoft SQL Server clients. For more information, see Working with Babelfish for Aurora PostgreSQL.	October 28, 2021
Aurora Serverless v1 can require SSL for connections	The Aurora cluster parameters `force_ssl` for PostgreSQL and `require_secure_transport` for MySQL are supported now for Aurora Serverless v1. For more information, see Using TLS/SSL with Aurora Serverless v1.	October 26, 2021
Amazon Aurora supports Performance Insights in additional AWS Regions	Performance Insights is available in the Middle East (Bahrain), Africa (Cape Town), Europe (Milan), and Asia Pacific (Osaka) Regions. For more information, see AWS Region support for Performance Insights.	October 5, 2021

Configurable autoscaling timeout for Aurora Serverless v1	You can choose how long Aurora Serverless v1 waits to find an autoscaling point. If no autoscaling point is found during that period, Aurora Serverless v1 cancels the scaling event or forces the capacity change, depending on the timeout action that you selected. For more information, see Autoscaling for Aurora Serverless v1.	September 10, 2021
Aurora supports X2g and T4g instance classes	Both Aurora MySQL and Aurora PostgreSQL can now use X2g and T4g instance classes. The instance classes that you can use depend on the version of Aurora MySQL or Aurora PostgreSQ L. For information about supported instance types, see DB instance classes.	September 10, 2021
Amazon RDS supports RDS Proxy in a shared VPC	You can now create an RDS Proxy in a shared virtual private cloud (VPC). For more information about RDS Proxy, see "Managing Connections with Amazon RDS Proxy" in the Amazon RDS User Guide or the Aurora User Guide.	August 6, 2021

Aurora version policy page	The *Amazon Aurora User Guide* now includes a section with general information about Aurora versions and associated policies. For details, see Amazon Aurora versions.	July 14, 2021
Exclude Data API events from an AWS CloudTrail trail	You can exclude Data API events from a CloudTrail trail. For more information, see Excluding Data API events from an AWS CloudTrail trail.	July 2, 2021
Amazon Aurora PostgreSQL-Compatible Edition supports additional extensions	Newly supported extensions include pg_bigm, pg_cron, pg_partman, and pg_proctab. For more information, see Extension versions for Amazon Aurora PostgreSQL-Compatible Edition.	June 17, 2021
Cloning for Aurora Serverless clusters	You can now create cloned clusters that are Aurora Serverless. For information about cloning, see Cloning a volume for an Aurora DB cluster.	June 16, 2021
Aurora global databases available in China (Beijing) and China (Ningxia) Regions	You can now create Aurora global databases in the China (Beijing) and China (Ningxia) Regions. For information about Aurora global databases, see Working with Amazon Aurora global databases.	May 19, 2021

FIPS 140-2 support for Data API	The Data API supports the Federal Information Processing Standard Publication 140-2 (FIPS 140-2) for SSL/TLS connections. For more information, see Data API availability.	May 14, 2021
AWS JDBC Driver for PostgreSQL (preview)	The AWS JDBC Driver for PostgreSQL, now available in preview, is a client driver designed for the high availability of Aurora PostgreSQL. For more information, see Connecting with the Amazon Web Services JDBC Driver for PostgreSQL (preview).	April 27, 2021
The Data API available in additional AWS Regions	The Data API is now available in the Asia Pacific (Seoul) and Canada (Central) Regions. For more information, see Availability of the Data API.	April 9, 2021
Amazon Aurora supports the Graviton2 DB instance classes	You can now use the Graviton2 DB instance classes db.r6g.x to create DB clusters running MySQL or PostgreSQL. For more information, see DB instance class types.	March 12, 2021

RDS Proxy endpoint enhancements	You can create additional endpoints associated with each RDS proxy. Creating an endpoint in a different VPC enables cross-VPC access for the proxy. Proxies for Aurora MySQL clusters can also have read-only endpoints. These reader endpoints connect to reader DB instances in the clusters and can improve read scalability and availability for query-intensive applications. For more information about RDS Proxy, see "Managing Connections with Amazon RDS Proxy" in the Amazon RDS User Guide or the Aurora user guide.	March 8, 2021
Amazon Aurora is available in the Asia Pacific (Osaka) Region	Amazon Aurora is now available in the Asia Pacific (Osaka) Region. For more information, see Regions and Availability Zones.	March 1, 2021
Aurora PostgreSQL supports enabling both IAM and Kerberos authentication on the same DB cluster	Aurora PostgreSQL now supports enabling both IAM authentication and Kerberos authentication on the same DB cluster. For more information, see Database authentication with Amazon Aurora.	February 24, 2021

Aurora global database now supports managed planned failover	Aurora global database now supports managed planned failover, allowing you to more easily change the primary AWS Region of your Aurora global database. You can use managed planned failover with healthy Aurora global databases only. To learn more, see Disaster recovery and Amazon Aurora global databases. For reference information, see `FailoverGlobalCluster` in the *Amazon RDS API Reference*.	February 11, 2021
Data API for Aurora Serverless now supports more data types	With the Data API for Aurora Serverless, you can now use `UUID` and `JSON` data types as input to your database. Also with the Data API for Aurora Serverless, you can now have a `LONG` type value returned from your database as a `STRING` value. To learn more, see Calling the Data API. For reference information about supported data types, see `SqlParameter` in the *Amazon RDS Data Service API Reference*.	February 2, 2021

Aurora PostgreSQL supports major version upgrades to PostgreSQL 12	With Aurora PostgreSQL, you can now upgrade the DB engine to major version 12. For more information, see Upgrading the PostgreSQL DB engine for Aurora PostgreSQL.	January 28, 2021
Aurora MySQL supports in-place upgrade	You can upgrade your Aurora MySQL 1.x cluster to Aurora MySQL 2.x, preserving the DB instances, endpoints, and so on of the original cluster. This in-place upgrade technique avoids the inconvenience of setting up a whole new cluster by restoring a snapshot. It also avoids the overhead of copying all your table data into a new cluster. For more information, see Upgrading the major version of an Aurora MySQL DB cluster from 1.x to 2.x.	January 11, 2021
AWS JDBC Driver for MySQL (preview)	The AWS JDBC Driver for MySQL, now available in preview, is a client driver designed for the high availability of Aurora MySQL. For more information, see Connecting with the Amazon Web Services JDBC Driver for MySQL (preview).	January 7, 2021

Aurora supports database activity streams on secondary clusters of a global database	You can start a database a database activity stream on a primary or secondary cluster of Aurora PostgreSQL or Aurora MySQL. For supported engine versions, see Limitations of Aurora global databases.	December 22, 2020
Multi-master clusters with 4 DB instances	The maximum number of DB instances in an Aurora MySQL multi-master cluster is now four. Formerly, the maximum was two DB instances. For more information, see Working with Aurora Multi-Master Clusters.	December 17, 2020
Aurora PostgreSQL supports AWS Lambda functions	You can now invoke AWS Lambda function for your Aurora PostgreSQL DB clusters. For more information, see Invoking a Lambda function from an Aurora PostgreSQL DB cluster.	December 11, 2020
Amazon Aurora supports the Graviton2 DB instance classes in preview	You can now use the Graviton2 DB instance classes db.r6g.x in preview to create DB clusters running MySQL or PostgreSQL. For more information, see DB instance class types.	December 11, 2020

Amazon Aurora Serverless v2 is now available in preview.	Amazon Aurora Serverless v2 is available in preview. To work with Amazon Aurora Serverless v2, apply for access. For more information, see the Aurora Serverless v2 page.	December 1, 2020

Aurora PostgreSQL is now available for Aurora Serverless in more AWS Regions.	Aurora PostgreSQL is now available for Aurora Serverless in more AWS Regions. You can now choose to run Aurora PostgreSQL Serverless v1 in the same AWS Regions that offer Aurora MySQL Serverless v1. Additional AWS Regions with Aurora Serverless support include US West (N. California), Asia Pacific (Singapore) Asia Pacific (Sydney) Asia Pacific (Seoul) Asia Pacific (Mumbai) Canada (Central) Europe (London) and Europe (Paris). For a list of all Regions and supported Aurora DB engines for Aurora Serverless, see Supported Regions and Aurora DB engines for Aurora Serverless v1. Amazon RDS Data API for Aurora Serverless is also now available in these same AWS Regions. For a list of all Regions with support for the Data API for Aurora Serverless, see Data API with Aurora MySQL Serverless v1	November 24, 2020

Amazon RDS Performance Insights introduces new dimensions	You can group database load according to the dimension groups for database, application (PostgreSQL), and session type (PostgreSQL). Amazon RDS also supports the dimensions db.name, db.application.name (PostgreSQL), and db.session_type.name (PostgreSQL). For more information, see Top load table.	November 24, 2020
Aurora Serverless supports Aurora PostgreSQL version 10.12	Aurora PostgreSQL for Aurora Serverless has been upgraded to Aurora PostgreSQL version 10.12 throughout the AWS Regions where Aurora PostgreSQL for Aurora Serverless is supported. For more information, see Supported Regions and Aurora DB engines for Aurora Serverless v1.	November 4, 2020
The Data API now supports tag-based authorization	The Data API supports tag-based authorization. If you've labeled your RDS cluster resources with tags, you can use these tags in your policy statements to control access through the Data API. For more information, see Authorizing access to the Data API.	October 27, 2020

Amazon Aurora extends support for exporting snapshots to Amazon S3	You can now export DB snapshot data to Amazon S3 in all commercial AWS Regions. For more information, see Exporting DB snapshot data to Amazon S3.	October 22, 2020
Aurora global database supports cloning	You can now create clones of the primary and secondary DB clusters of your Aurora global databases. You can do so by using the AWS Management Console and choosing the **Create clone** menu option. You can also use the AWS CLI and run the `restore-db-cluster-to-point-in-time` command with the `--restore-type copy-on-write` option. Using the AWS Management Console or the AWS CLI, you can also clone DB clusters from your Aurora global databases across AWS accounts. For more information about cloning, see Cloning an Aurora DB cluster volume.	October 19, 2020

Amazon Aurora supports dynamic resizing for the cluster volume	Starting with Aurora MySQL 1.23 and 2.09, and Aurora PostgreSQL 3.3.0 and Aurora PostgreSQL 2.6.0, Aurora reduces the size of the cluster volume after you remove data through operations such as `DROP TABLE`. To take advantage of this enhancement, upgrade to one of the appropriate versions depending on the database engine that your cluster uses. For information about this feature and how to check used and available storage space for an Aurora cluster, see Managing Performance and Scaling for Aurora DB Clusters.	October 13, 2020
Amazon Aurora supports volume sizes up to 128 TiB	New and existing Aurora cluster volumes can now grow to a maximum size of 128 tebibytes (TiB). For more information, see How Aurora storage grows.	September 22, 2020
Aurora PostgreSQL supports the db.r5 and db.t3 DB instance classes in the China (Ningxia) Region	You can now create Aurora PostgreSQL DB clusters in the China (Ningxia) Region that use the db.r5 and db.t3 DB instance classes. For more information, see DB instance classes.	September 3, 2020

Aurora parallel query enhancements	Starting with Aurora MySQL 2.09 and 1.23, you can take advantage of enhancements to the parallel query feature. Creating a parallel query cluster no longer requires a special engine mode. You can now turn parallel query on and off using the `aurora_parallel_query` configuration option for any provisioned cluster that's running a compatible Aurora MySQL version. You can upgrade an existing cluster to a compatible Aurora MySQL version and use parallel query, instead of creating a new cluster and importing data into it. You can use Performance Insights for parallel query clusters. You can stop and start parallel query clusters. You can create Aurora parallel query clusters that are compatible with MySQL 5.7. Parallel query works for tables that use the `DYNAMIC` row format. Parallel query clusters can use AWS Identity and Access Management (IAM) authentication. Reader DB instances in parallel query clusters can take advantage of the `READ COMMITTED` isolation	September 2, 2020

	level. You can also now create parallel query clusters in additional AWS Regions. For more information about the parallel query feature and these enhancements, see Working with parallel query for Aurora MySQL.	
Changes to Aurora MySQL parameter binlog_rows_query_log_events	You can now change the value of the Aurora MySQL configuration parameter `binlog_rows_query_log_events` . Formerly, this parameter wasn't modifiable.	August 26, 2020

Support for automatic minor version upgrades for Aurora MySQL	With Aurora MySQL, the setting **Enable auto minor version upgrade** now takes effect when you specify it for an Aurora MySQL DB cluster. When you enable auto minor version upgrade, Aurora automatically upgrades to new minor versions as they are released. The automatic upgrades occur during the maintenance window for the database. For Aurora MySQL, this feature applies only to Aurora MySQL version 2, which is compatible with MySQL 5.7. Initially, the automatic upgrade procedure brings Aurora MySQL DB clusters to version 2.07.2. For more information about how this feature works with Aurora MySQL, see Database Upgrades and Patches for Amazon Aurora MySQL.	August 3, 2020
Aurora PostgreSQL supports major version upgrades to PostgreSQL version 11	With Aurora PostgreSQL, you can now upgrade the DB engine to major version 11. For more information, see Upgrading the PostgreSQL DB engine for Aurora PostgreSQL.	July 28, 2020

Amazon Aurora supports AWS PrivateLink	Amazon Aurora now supports creating Amazon VPC endpoints for Amazon RDS API calls to keep traffic between applications and Aurora in the AWS network. For more information, see Amazon Aurora and interface VPC endpoints (AWS PrivateLink).	July 9, 2020
RDS Proxy generally available	RDS Proxy is now generally available. You can use RDS Proxy with RDS for MySQL, Aurora MySQL, RDS for PostgreSQL, and Aurora PostgreSQL for production workloads. For more information about RDS Proxy, see "Managing Connections with Amazon RDS Proxy" in the Amazon RDS User Guide or the Aurora user guide.	June 30, 2020

Aurora global database write forwarding	You can now enable write capability on secondary clusters in a global database. With write forwarding, you issue DML statements on a secondary cluster, Aurora forwards the write to the primary cluster, and the updated data is replicated to all the secondary clusters. For more information, see Write forwarding for secondary AWS Regions with an Aurora global database.	June 18, 2020
Aurora supports integration with AWS Backup	You can use AWS Backup to manage backups of Aurora DB clusters. For more information, see Overview of backing up and restoring an Aurora DB cluster.	June 10, 2020
Aurora PostgreSQL supports db.t3.large DB instance classes	You can now create Aurora PostgreSQL DB clusters that use the db.t3.large DB instance classes. For more information, see DB instance classes.	June 5, 2020

Aurora global database supports PostgreSQL version 11.7 and managed recovery point objective (RPO)	You can now create Aurora global databases for the PostgreSQL database engine version 11.7. You can also manage how a PostgreSQL global database recovers from a failure using a recovery point objective (RPO). For more information, see Cross-Region Disaster Recovery for Aurora global databases.	June 4, 2020
Aurora MySQL supports database monitoring with database activity streams	Aurora MySQL now includes database activity streams, which provide a near-real-time data stream of the database activity in your relational database. For more information, see Using database activity streams.	June 2, 2020
The query editor available in additional AWS Regions	The query editor for Aurora Serverless is now available in additional AWS Regions. For more information, see Availability of the query editor.	May 28, 2020
The Data API available in additional AWS Regions	The Data API is now available in additional AWS Regions. For more information, see Availability of the Data API.	May 28, 2020

RDS Proxy available in Canada (Central) Region	You can now use the RDS Proxy preview in the Canada (Central) Region. For more information about RDS Proxy, see Managing connections with Amazon RDS proxy (preview).	May 28, 2020
Aurora global database and cross-Region read replicas	For an Aurora global database, you can now create an Aurora MySQL cross-Region read replica from the primary cluster in the same region as a secondary cluster. For more information about Aurora Global Database and cross-Region read replicas, see Working with Amazon Aurora global database and Replicating Amazon Aurora MySQL DB.	May 18, 2020
RDS Proxy available in more AWS Regions	You can now use the RDS Proxy preview in the US West (N. California) Region, the Europe (London) Region, the Europe (Frankfurt) Region, the Asia Pacific (Seoul) Region, the Asia Pacific (Mumbai) Region, the Asia Pacific (Singapore) Region, and the Asia Pacific (Sydney) Region. For more information about RDS Proxy, see Managing connections with Amazon RDS proxy (preview).	May 13, 2020

Aurora PostgreSQL-Compatible Edition supports on-premises or self-hosted Microsoft active directory	You can now use an on-premises or self-hosted Active Directory for Kerberos authentication of users when they connect to your Aurora PostgreSQL DB clusters. For more information, see Using Kerberos authentication with Aurora PostgreSQL.	May 7, 2020
Aurora MySQL multi-master clusters available in more AWS Regions	You can now create Aurora multi-master clusters in the Asia Pacific (Seoul) Region, the Asia Pacific (Tokyo) Region, the Asia Pacific (Mumbai) Region, and the Europe (Frankfurt) Region. For more information about multi-master clusters, see Working with Aurora multi-master clusters.	May 7, 2020
Performance Insights supports analyzing statistics of running Aurora MySQL queries	You can now analyze statistics of running queries with Performance Insights for Aurora MySQL DB instances. For more information, see Analyzing statistics of running queries.	May 5, 2020

Java client library for Data API generally available	The Java client library for the Data API is now generally available. You can download and use a Java client library for Data API. It enables you to map your client-side classes to requests and responses of the Data API. For more information, see Using the Java client library for Data API.	April 30, 2020
Amazon Aurora is available in the Europe (Milan) Region	Amazon Aurora is now available in the Europe (Milan) Region. For more information, see Regions and Availability Zones.	April 28, 2020
Amazon Aurora is available in the Europe (Milan) Region	Amazon Aurora is now available in the Europe (Milan) Region. For more information, see Regions and Availability Zones.	April 27, 2020
Amazon Aurora is available in the Africa (Cape Town) Region	Amazon Aurora is now available in the Africa (Cape Town) Region. For more information, see Regions and Availability Zones.	April 22, 2020

Aurora PostgreSQL now supports db.r5.16xlarge and db.r5.8xlarge DB instance classes	You can now create Aurora PostgreSQL DB clusters running PostgreSQL that use the db.r5.16xlarge and db.r5.8xlarge DB instance classes. For more information, see Hardware specifications for DB instance classes for Aurora.	April 8, 2020
Amazon RDS proxy for PostgreSQL	Amazon RDS Proxy is now available for PostgreSQL. You can use RDS Proxy to reduce the overhead of connection management on your cluster and also the chance of "too many connections" errors. The RDS Proxy is currently in public preview for PostgreSQL. For more information, see Managing connections with Amazon RDS proxy (preview).	April 8, 2020
Aurora global databases now support Aurora PostgreSQL	You can now create Aurora global databases for the PostgreSQL database engine. An Aurora global database spans multiple AWS Regions, enabling low latency global reads and disaster recovery from region-wide outages. For more information, see Working with Amazon Aurora global database.	March 10, 2020

Support for major version upgrades for Aurora PostgreSQL	With Aurora PostgreSQL, you can now upgrade the DB engine to a major version. By doing so, you can skip ahead to a newer major version when you upgrade select PostgreSQL engine versions. For more information, see Upgrading the PostgreSQL DB engine for Aurora PostgreSQL.	March 4, 2020
Aurora PostgreSQL supports Kerberos authentication	You can now use Kerberos authentication to authenticate users when they connect to your Aurora PostgreSQL DB clusters. For more information, see Using Kerberos authentication with Aurora PostgreSQL.	February 28, 2020
The Data API supports AWS PrivateLink	The Data API now supports creating Amazon VPC endpoints for Data API calls to keep traffic between applications and the Data API in the AWS network. For more information, see Creating an Amazon VPC endpoint (AWS PrivateLink) for the Data API.	February 6, 2020

Aurora machine learning support in Aurora PostgreSQL	The aws_ml Aurora PostgreSQL extension provides functions you use in your database queries to call Amazon Comprehend for sentiment analysis and SageMaker to run your own machine learning models. For more information, see Using machine learning (ML) capabilities with Aurora.	February 5, 2020
Aurora PostgreSQL supports exporting data to Amazon S3	You can query data from an Aurora PostgreSQL DB cluster and export it directly into files stored in an Amazon S3 b ucket. For more information, see Exporting data from an Aurora PostgreSQL DB cluster to Amazon S3.	February 5, 2020
Support for exporting DB snapshot data to Amazon S3	Amazon Aurora supports exporting DB snapshot data to Amazon S3 for MySQL and PostgreSQL. For more information, see Exporting DB snapshot data to Amazon S3.	January 9, 2020

Aurora MySQL release notes in document history	This section now includes history entries for Aurora MySQL-Compatible Edition release notes for versions released after August 31, 2018. For the full release notes for a specific version, choose the link in the first column of the history entry.	December 13, 2019
Amazon RDS proxy	You can reduce the overhead of connection management on your cluster, and reduce the chance of "too many connections" errors, by using the Amazon RDS Proxy. You associate each proxy with an RDS DB instance or Aurora DB cluster. Then you use the proxy endpoint in the connection string for your application. The Amazon RDS Proxy is currently in a public preview state. It supports the Aurora MySQL database engine. For more information, see Managing connections with Amazon RDS proxy (preview).	December 3, 2019

Data API for Aurora Serverless v1 supports data type mapping hints	You can now use a hint to instruct the Data API for Aurora Serverless v1 to send a `String` value to the database as a different type. For more information, see Calling the data API.	November 26, 2019
Data API for Aurora Serverless v1 supports a Java client library (preview)	You can download and use a Java client library for Data API. It enables you to map your client-side classes to requests and responses of the Data API. For more information, see Using the Java client library for Data API.	November 26, 2019
Aurora PostgreSQL is FedRAMP HIGH eligible	Aurora PostgreSQL is FedRAMP HIGH eligible. For details about AWS and compliance efforts, see AWS services in scope by compliance program.	November 26, 2019

READ COMMITTED isolation level enabled for Amazon Aurora MySQL replicas	You can now enable the `READ COMMITTED` isolation level on Aurora MySQL Replicas. Doing so requires enabling the `aurora_read_replica_read_committed_isolation_enabled` configuration setting at the session level. Using the `READ COMMITTED` isolation level for long-running queries on OLTP clusters can help address issues with history list length. Before enabling this setting, be sure to understand how the isolation behavior on Aurora Replicas differs from the usual MySQL implementation of `READ COMMITTED`. For more information, see Aurora MySQL isolation levels.	November 25, 2019
Performance Insights supports analyzing statistics of running Aurora PostgreSQL queries	You can now analyze statistics of running queries with Performance Insights for Aurora PostgreSQL DB instances. For more information, see Analyzing statistics of running queries.	November 25, 2019

More clusters in an Aurora global database	You can now add multiple secondary regions to an Aurora global database. You can take advantage of low latency global reads and disaster recovery across a wider geographic area. For more information about Aurora global databases, see Working with Amazon Aurora global databases.	November 25, 2019
Aurora machine learning support in Aurora MySQL	In Aurora MySQL 2.07 and higher, you can call Amazon Comprehend for sentiment analysis and SageMaker for a wide variety of machine learning algorithms. You use the results directly in your database application by embedding calls to stored functions in your queries. For more information, see Using machine learning (ML) capabilities with Aurora.	November 25, 2019

Aurora global database no longer requires engine mode setting	You no longer need to specify `--engine-mode=global` when creating a cluster that is intended to be part of an Aurora global database. All Aurora clusters that meet the compatibility requirements are eligible to be part of a global database. For example, the cluster currently must use Aurora MySQL version 1 with MySQL 5.6 compatibility. For information about Aurora global databases, see Working with Amazon Aurora global databases.	November 25, 2019
Aurora global database is available for Aurora MySQL version 2	Starting in Aurora MySQL 2.07, you can create an Aurora global database with MySQL 5.7 compatibility. You don't need to specify the `global` engine mode for the primary or secondary clusters. You can add any new provisioned cluster with Aurora MySQL 2.07 or higher to an Aurora Global Database. For information about Aurora Global Database, see Working with Amazon Aurora global database.	November 25, 2019

Aurora MySQL hot row contention optimization available without lab mode	The hot row contention optimization is now generally available for Aurora MySQL and does not require the Aurora lab mode setting to be ON. This feature substantially improves throughput for workloads with many transactions contending for rows on the same page. The improvement involves changing the lock release algorithm used by Aurora MySQL.	November 19, 2019
Aurora MySQL hash joins available without lab mode	The hash join feature is now generally available for Aurora MySQL and does not require the Aurora lab mode setting to be ON. This feature can improve query performance when you need to join a large amount of data by using an equijoin. For more information about using this feature, see Working with hash joins in Aurora MySQL.	November 19, 2019
Aurora MySQL 2.* support for more db.r5 instance classes	Aurora MySQL clusters now support the instance types db.r5.8xlarge, db.r5.16xlarge, and db.r5.24xlarge. For more information about instance types for Aurora MySQL clusters, see Choosing the DB instance class.	November 19, 2019

Aurora MySQL 2.* support for backtracking	Aurora MySQL 2.* versions now offer a quick way to recover from user errors, such as dropping the wrong table or deleting the wrong row. Backtrack allows you to move your database to a prior point in time without needing to restore from a backup, and it completes within seconds, even for large databases. For details, see Backtracking an Aurora DB cluster.	November 19, 2019
Billing tag support for Aurora	You can now use tags to keep track of cost allocation for resources such as Aurora clusters, DB instances within Aurora clusters, I/O, backups, snapshots, and so on. You can see costs associated with each tag using AWS Cost Explorer. For more information about using tags with Aurora, see Tagging Amazon RDS resources. For general information about tags and ways to use them for cost analysis, see Using cost allocation tags and User-defined cost allocation tags.	October 23, 2019

Data API for Aurora PostgreSQL	Aurora PostgreSQL now supports using the Data API with Amazon Aurora Serverless v1 DB clusters. For more information, see Using the Data API for Aurora Serverless v1.	September 23, 2019
Aurora PostgreSQL supports uploading database logs to CloudWatch logs	You can configure your Aurora PostgreSQL DB cluster to publish log data to a log group in Amazon CloudWatch Logs. With CloudWatch Logs, you can perform real-time analysis of the log data, and use CloudWatch to create alarms and view metrics. You can use CloudWatch Logs to store your log records in highly durable storage. For more information, see Publishing Aurora PostgreSQL logs to Amazon CloudWatch Logs.	August 9, 2019
Multi-master clusters for Aurora MySQL	You can set up Aurora MySQL multi-master clusters. In these clusters, each DB instance has read/write capability. For more information, see Working with Aurora multi-master clusters.	August 8, 2019

Aurora PostgreSQL supports Aurora Serverless v1	You can now use Amazon Aurora Serverless v1 with Aurora PostgreSQL. An Aurora Serverless DB cluster automatically starts up, shuts down, and scales up or down its compute capacity based on your application's needs. For more information, see Using Amazon Aurora Serverless v1.	July 9, 2019
Cross-account cloning for Aurora MySQL	You can now clone the cluster volume for an Aurora MySQL DB cluster between AWS accounts. You authorize the sharing through AWS Resource Access Manager (AWS RAM). The cloned cluster volume uses a copy-on-write mechanism, which only requires additional storage for new or changed data. For more information about cloning for Aurora, see Cloning databases in an Aurora DB cluster.	July 2, 2019
Aurora PostgreSQL supports db.t3 DB instance classes	You can now create Aurora PostgreSQL DB clusters that use the db.t3 DB instance classes. For more information, see DB instance class.	June 20, 2019

Support for importing data from Amazon S3 for Aurora PostgreSQL	You can now import data from an Amazon S3 file into a table in an Aurora PostgreSQL DB cluster. For more information, see Importing Amazon S3 data into an Aurora PostgreSQL DB cluster.	June 19, 2019
Aurora PostgreSQL now provides fast failover recovery with cluster cache management	Aurora PostgreSQL now provides cluster cache management to ensure fast recovery of the primary DB instance in the event of a failover. For more information, see Fast recovery after failover with cluster cache management.	June 11, 2019
Data API for Aurora Serverless v1 generally available	You can access Aurora Serverless v1 clusters with web services-based applications using the Data API. For more information, see Using the Data API for Aurora Serverless v1.	May 30, 2019
Aurora PostgreSQL supports database monitoring with database activity streams	Aurora PostgreSQL now includes database activity streams, which provide a near-real-time data stream of the database activity in your relational database. For more information, see Using database activity streams.	May 30, 2019

Amazon Aurora recommendations	Amazon Aurora now provides automated recommendations for Aurora resources. For more information, see Using Amazon Aurora recommendations.	May 22, 2019
Performance Insights support for Aurora global database	You can now use Performance Insights with Aurora Global Database. For information about Performance Insights for Aurora, see Using Amazon RDS performance insights. For information about Aurora global databases, see Working with Aurora global database.	May 13, 2019
Performance Insights is available for Aurora MySQL 5.7	Amazon RDS Performance Insights is now available for Aurora MySQL 2.x versions, which are compatible with MySQL 5.7. For more information, see Using Amazon RDS performance insights.	May 3, 2019
Aurora global databases available in more AWS Regions	You can now create Aurora global databases in most AWS Regions where Aurora is available. For information about Aurora global databases, see Working with Amazon Aurora global databases.	April 30, 2019

Minimum capacity of 1 for Aurora Serverless v1	The minimum capacity setting you can use for an Aurora Serverless v1 cluster is 1. Formerly, the minimum was 2. For information about specifying Aurora Serverless capacity values, see Setting the capacity of an Aurora Serverless v1 DB cluster.	April 29, 2019
Aurora Serverless v1 timeout action	You can now specify the action to take when an Aurora Serverless v1 capacity change times out. For more information, see Timeout action for capacity changes.	April 29, 2019
Per-second billing	Amazon RDS is now billed in 1-second increments in all AWS Regions except AWS GovCloud (US) for on-demand instances. For more information, see DB instance billing for Aurora.	April 25, 2019
Sharing Aurora Serverless v1 snapshots across AWS Regions	With Aurora Serverless v1, snapshots are always encrypted. If you encrypt the snapshot with your own AWS KMS key, you can now copy or share the snapshot across AWS Regions. For information about snapshots of Aurora Serverless v1 DB clusters, see Aurora Serverless v1 and snapshots.	April 17, 2019

Restore MySQL 5.7 backups from Amazon S3	You can now create a backup of your MySQL version 5.7 database, store it on Amazon S3, and then restore the backup file onto a new Aurora MySQL DB cluster. For more information, see Migrating data from an external MySQL database to an Aurora MySQL DB cluster.	April 17, 2019
Sharing Aurora Serverless v1 snapshots across regions	With Aurora Serverless v1, snapshots are always encrypted. If you encrypt the snapshot with your own AWS KMS key, you can now copy or share the snapshot across regions. For information about snapshots of Aurora Serverless v1 DB clusters, see Aurora Serverless and snapshots.	April 16, 2019
Aurora proof-of-concept tutorial	You can learn how to perform a proof of concept to try your application and workload with Aurora. For the full tutorial, see Performing an Aurora proof of concept.	April 16, 2019
Aurora Serverless v1 supports restoring from an Amazon S3 backup	You can now import backups from Amazon S3 to an Aurora Serverless cluster. For details about that procedure, see Migrating data from MySQL by using an Amazon S3 bucket.	April 16, 2019

New modifiable parameters for Aurora Serverless v1	You can now modify the following DB parameters for an Aurora Serverless v1 cluster: `innodb_file_format`, `innodb_file_per_table`, `innodb_large_prefix`, `innodb_lock_wait_timeout`, `innodb_monitor_disable`, `innodb_monitor_enable`, `innodb_monitor_reset`, `innodb_monitor_reset_all`, `innodb_print_all_deadlocks`, `log_warnings`, `net_read_timeout`, `net_retry_count`, `net_write_timeout`, `sql_mode`, and `tx_isolation`. For more information about configuration parameters for Aurora Serverless v1 clusters, see Aurora Serverless v1 and parameter groups.	April 4, 2019
Aurora PostgreSQL supports db.r5 DB instance classes	You can now create Aurora PostgreSQL DB clusters that use the db.r5 DB instance classes. For more information, see DB instance class.	April 4, 2019

Aurora PostgreSQL logical replication	You can now use PostgreSQL logical replication to replicate parts of a database for an Aurora PostgreSQL DB cluster. For more information, see Using PostgreSQL logical replication.	March 28, 2019
GTID support for Aurora MySQL 2.04	You can now use replication with the global transaction ID (GTID) feature of MySQL 5.7. This feature simplifies performing binary log (binlog) replication between Aurora MySQL and an external MySQL 5.7-compatible database. The replication can use the Aurora MySQL cluster as the source or the destination. This feature is available for Aurora MySQL 2.04 and higher. For more information about GTID-based replication and Aurora MySQL, see Using GTID-based replication for Aurora MySQL.	March 25, 2019

Uploading Aurora Serverless v1 logs to Amazon CloudWatch	You can now have Aurora upload database logs to CloudWatch for an Aurora Serverless v1 cluster. For more information, see Viewing Aurora Serverless DB clusters. As part of this enhancement, you can now define values for instance-level parameters in a DB cluster parameter group, and those values apply to all DB instances in the cluster unless you override them in the DB parameter group. For more information, see Working with DB parameter groups and DB cluster parameter groups.	February 25, 2019
Aurora MySQL supports db.t3 DB instance classes	You can now create Aurora MySQL DB clusters that use the db.t3 DB instance classes. For more information, see DB instance class.	February 25, 2019
Aurora MySQL supports db.r5 DB instance classes	You can now create Aurora MySQL DB clusters that use the db.r5 DB instance classes. For more information, see DB instance class.	February 25, 2019

Performance Insights counters for Aurora MySQL	You can now add performance counters to your Performance Insights charts for Aurora MySQL DB instances. For more information, see Performance Insights dashboard components.	February 19, 2019
Amazon RDS Performance Insights supports viewing more SQL text for Aurora MySQL	Amazon RDS Performance Insights now supports viewing more SQL text in the Performance Insights dashboard for Aurora MySQL DB instances. For more information, see Viewing more SQL text in the Performance Insights dashboard.	February 6, 2019
Amazon RDS Performance Insights supports viewing more SQL text for Aurora PostgreSQL	Amazon RDS Performance Insights now supports viewing more SQL text in the Performance Insights dashboard for Aurora PostgreSQL DB instances. For more information, see Viewing more SQL text in the Performance Insights dashboard.	January 24, 2019

Aurora backup billing	You can use the Amazon CloudWatch metrics `TotalBackupStorageBilled` , `SnapshotStorageUsed` , and `BackupRetentionPeriodStorageUsed` to monitor the space usage of your Aurora backups. For more information about how to use CloudWatch metrics, see Overview of monitoring. For more information about how to manage storage for backup data, see Understanding Aurora backup storage usage.	January 3, 2019
Performance Insights counters	You can now add performance counters to your Performance Insights charts. For more information, see Performance Insights dashboard components.	December 6, 2018
Aurora global database	You can now create Aurora global databases. An Aurora global database spans multiple AWS Regions, enabling low latency global reads and disaster recovery from region-wide outages. For more information, see Working with Amazon Aurora global database.	November 28, 2018

Query plan management in Aurora PostgreSQL	Aurora PostgreSQL now provides query plan management that you can use to manage PostgreSQL query execution plans. For more information, see Managing query execution plans for Aurora PostgreSQL.	November 20, 2018
Query editor for Aurora Serverless v1 (beta)	You can run SQL statements in the Amazon RDS console on Aurora Serverless v1 clusters. For more information, see Using the query editor for Aurora Serverless v1.	November 20, 2018
Data API for Aurora Serverless v1 (beta)	You can access Aurora Serverless v1 clusters with web services-based applications using the Data API. For more information, see Using the Data API for Aurora Serverless.	November 20, 2018
TLS support for Aurora Serverless v1	Aurora Serverless v1 clusters support TLS/SSL encryption. For more information, see TLS/SSL for Aurora Serverless.	November 19, 2018

Custom endpoints	You can now create endpoints that are associated with an arbitrary set of DB instances. This feature helps with load balancing and high availability for Aurora clusters where some DB instances have different capacity or configuration than others. You can use custom endpoints instead of connecting to a specific DB instance through its instance endpoint. For more information, see Amazon Aurora connection management.	November 12, 2018
IAM authentication support in Aurora PostgreSQL	Aurora PostgreSQL now supports IAM authentication. For more information see IAM database authentication.	November 8, 2018
Custom parameter groups for restore and point in time recovery	You can now specify a custom parameter group when you restore a snapshot or perform a point in time recovery operation. For more information, see Restoring from a DB cluster snapshot and Restoring a DB cluster to a specified time.	October 15, 2018
Deletion protection for Aurora DB clusters	When you enable deletion protection for a DB cluster, the database cannot be deleted by any user. For more information, see Deleting a DB cluster.	September 26, 2018

Stop/Start feature Aurora	You can now stop or start an entire Aurora cluster with a single operation. For more information, see Stopping and starting an Aurora cluster.	September 24, 2018
Parallel query feature for Aurora MySQL	Aurora MySQL now offers an option to parallelize I/O work for queries across the Aurora storage infrastructure. This feature speeds up data-intensive analytic queries, which are often the most time-consuming operations in a workload. For more information, see Working with parallel query for Aurora MySQL.	September 20, 2018
New guide	This is the first release of the *Amazon Aurora User Guide.*	August 31, 2018

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.