aws

User Guide

Amazon Aurora DSQL

Amazon Aurora DSQL: User Guide

Table of Contents

Amazon Aurora DSQL is provided as a Preview service. To learn more, see Betas and Previews in the AWS Service Terms.

What is Amazon Aurora DSQL?

Amazon Aurora DSQL is a serverless, distributed SQL database with virtually unlimited scale, high availability, and zero infrastructure management. Aurora DSQL provides active-active high availability that enables 99.99% single-Region and 99.999% multi-Region availability.

You can use Aurora DSQL to automatically manage system infrastructure and scale your database based on the needs of your workload. With Aurora DSQL, you don't have to worry about maintenance downtime related to provisioning, patching, or infrastructure upgrades.

Aurora DSQL is optimized for transactional workloads that benefit from ACID transactions and a relational data model. Because it's serverless and optimized for transactional workloads, Aurora DSQL is ideal for application patterns of microservice, serverless, and event-driven architectures. Aurora DSQL is PostgreSQL compatible, so you can use familiar drivers, object-relational mappings (ORMs), frameworks, and SQL features.

Aurora DSQL automatically scales compute, I/O, and storage, so it can efficiently adapt to your workload needs. The active-active serverless design automates failure recovery, so you don't have to worry about traditional database failover. This means that your applications benefit from Multi-AZ and multi-Region availability, and you don't have to be concerned about eventual consistency or missing data related to failovers. Aurora DSQL helps you to build and maintain applications that are always available at any scale.

For information about the core components in Aurora DSQL and to get started with the service, see the following:

- the section called “Core components”
- the section called “PostgreSQL compatibility”
- *Getting started*
- *Accessing Aurora DSQL*
- *Working with Aurora DSQL*

Understanding core components in Amazon Aurora DSQL

To learn about the core components in Aurora DSQL, review the following:

Distributed architecture

Aurora DSQL is based on a novel distributed database that is composed of four multi-tenant components:

- Relay and connectivity
- Compute and databases
- Transaction log, concurrency control, and isolation
- User storage

A control plane coordinates all of these. Each of these components provide redundancy across three Availability Zones (AZs), with automatic cluster scaling and self-healing in case of component failures. To learn more about how this architecture supports high availability, see the section called “Resilience”.

Aurora DSQL clusters

Single-Region clusters synchronously replicate data, remove replication lag, and prevent database failovers, and make sure that data is consistent across multiple Availability Zones (AZ) or Regions. If there are infrastructure failures, Aurora DSQL automatically routes requests to healthy infrastructure without manual intervention. Aurora DSQL provides *atomicity, consistency, isolation, and durability (ACID) transactions* with strong consistency, snapshot isolation, atomicity, and cross-AZ and cross-Region durability.

Multi-Region linked clusters provide the same resilience and connectivity as single-Region clusters. However, they improve availability by offering two Regional endpoints, one in each linked cluster Region. Both endpoints of a linked cluster present a single logical database. They are available for concurrent read and write operations, and provide strong data consistency. This means that you can build applications that run in multiple Regions at the same time for performance and resilience—and know that readers will always see the same data.

> **Note**
>
> During preview, you can interact with clusters in us-east-1 – US East (N. Virginia) and us-east-2 – US East (Ohio).

PostgreSQL databases

The distributed database layer (compute) in Aurora DSQL is based on a current major version of PostgreSQL. You can connect to Aurora DSQL with familiar PostgreSQL drivers and tools. Aurora DSQL is currently compatible with PostgreSQL version 16 and supports a subset of PostgreSQL features, expressions, and data types. For more information about the supported SQL features, see the section called “PostgreSQL compatibility”.

Getting started with Aurora DSQL

Use the following steps to get started with Aurora DSQL.

Topics

Prerequisites

- Your IAM identity must have permissions to sign in to the AWS Management Console.
- Your IAM identity must have access to perform any action on any resource in your AWS account, or you must be able to get access to the following IAM policy action: `dsql:*`.

> **Note**
>
> This guide assumes a Unix-like environment with Python v3.8+ and psql v14+. AWS CloudShell provides Python v3.8+ and psql v14+ with no extra setup. You can also use the AWS CLI in a different environment, but you must manually set up Python v3.8+ and psql v14+. If you prefer a GUI, you can Access Aurora DSQL with DBeaver or Access Aurora DSQL with JetBrains DataGrip.

Create a cluster and connect with IAM authentication

To create a new cluster in Aurora DSQL

1. Sign in to the AWS Management Console and open the Aurora DSQL console at https://console.aws.amazon.com/dsql.
2. Choose **Create cluster**. Configure any of the settings that you want, such as deletion protection or tags.
3. Choose **Create cluster**.

To connect to the cluster with an authentication token

1. Choose the cluster that you want to connect to. Choose **Connect**.
2. Copy the endpoint.
3. Use the following command to use psql to start a connection to your cluster. Replace *your_cluster_endpoint* with the cluster endpoint.

   ```
   PGSSLMODE=require \
    psql --dbname postgres \
   --username admin \
   --host your_cluster_endpoint
   ```

   You should see a prompt to provide a password. Generate an authentication token and use it as your password.
4. Make sure that you chose **Connect as admin**.
5. Copy the generated authentication token and paste it into the prompt to connect to Aurora DSQL from your SQL client.
6. Press **Enter**. You should see a PostgreSQL prompt.

   ```
   postgres=>
   ```

   If you get an access denied error, make sure that your IAM identity has the `dsql:DbConnectAdmin` permission. If you have the permission and continue to get access deny errors, see Troubleshoot IAM and How can I troubleshoot access denied or unauthorized operation errors with an IAM policy?.

Run SQL commands in Aurora DSQL

The following steps provide some SQL commands that you can run in Aurora DSQL.

1. Start by creating a schema named example.

   ```
   CREATE SCHEMA example;
   ```
2. Create an invoice table that uses an automatically generated UUID as the primary key.

```
CREATE TABLE example.invoice(id UUID PRIMARY KEY DEFAULT gen_random_uuid(), created
 timestamp, purchaser int, amount float);
```

3. Create a secondary index that uses the empty table.

```
CREATE INDEX invoice_created_idx on example.invoice(created);
```

4. Create a department table.

```
CREATE TABLE example.department(id INT PRIMARY KEY UNIQUE, name text, email text);
```

5. Use `psql \copy` to load in some data. Download the data files named `department-insert-multirow.sql` and `invoice.csv` from the aws-samples/aurora-dsql-samples repository on GitHub.
6. Use the command `psql \include` to load the files. These operations create tables and insert sample data.

```
\include samples/department-insert-multirow.sql
```

```
\copy example.invoice(created, purchaser, amount) from samples/invoice.csv csv
```

7. You can then list departments that are sorted by their total sales.

```
SELECT name, sum(amount) AS sum_amount
FROM example.department LEFT JOIN example.invoice ON
 department.id=invoice.purchaser
GROUP BY name
HAVING sum(amount) > 0
ORDER BY sum_amount DESC;
```

Example output:

```
           name           |     sum_amount
--------------------------+--------------------
 Example Department Three |  54061.67752854594
 Example Department Seven |  53869.65965365204
 Example Department Eight |  52199.73742066634
 Example Department One   | 52034.078869900826
 Example Department Six   |  50886.15556256385
```

```
 Example Department Two   |  50589.98422247931
 Example Department Five  | 49549.852635496005
 Example Department Four  |  49266.15578027619
(8 rows)
```

Create a multi-Region linked cluster

These steps guide you through how to create a multi-Region linked cluster. They also demonstrate cross-Region write replication and consistent reads from both Regional endpoints.

To create a new cluster and connect in multiple Regions

1. From the **Aurora DSQL Clusters** page, choose **Create cluster**.
2. Choose **Add linked Regions** and choose a Region for your linked cluster Region. The linked cluster Region is a separate Region to create another cluster in. Aurora DSQL replicates all writes to this cluster as well, so you can read and write from any linked cluster.
3. Choose a witness Region. The witness Region receives all data that is written to linked clusters, but you can't write to it. The witness Region stores a limited window of encrypted transaction logs. Aurora DSQL uses these capabilities to provide multi-Region durability and availability.

   > **Note**
   >
   > Witness Regions don't host client endpoints and don't provide user data access. A limited window of the encrypted transaction log is maintained in witness Regions. This facilitates recovery and supports transactional quorum in the event of Region unavailability. During preview, you can only choose us-west-2 as the witness Region.

4. Choose **Create**.
5. While Aurora DSQL is creating your cluster, open two instances of AWS CloudShell in different Regions. Make sure that you have the preview toolkit in both environments. Open one in the environment in us-east-1 and another one in us-east-2.

   > **Note**
   >
   > During preview, creating linked clusters takes additional time.

6. Connect to your cluster in us-east-2.

```
export PGSSLMODE=require \
 psql --dbname postgres \
--username admin \
--host replace_with_your_cluster_endpoint_in_us-east-2
```

To write in one Region and read from a second Region

1. In your us-east-2 CloudShell environment, go through the steps in the section called "SQL commands".

   Example transactions

   Example

   ```
   CREATE SCHEMA example;
   CREATE TABLE example.invoice(id UUID PRIMARY KEY DEFAULT gen_random_uuid(), created
    timestamp, purchaser int, amount float);
   CREATE INDEX invoice_created_idx on example.invoice(created);
   CREATE TABLE example.department(id INT PRIMARY KEY UNIQUE, name text, email text);
   ```

2. Use PSQL meta commands to load data. For more information, see the section called "SQL commands".

   ```
   \copy example.invoice(created, purchaser, amount) from samples/invoice.csv csv
   \include samples/department-insert-multirow.sql
   ```

3. In your us-east-1 CloudShell environment, query the data that you inserted from a different Region:

   Example

   ```
   SELECT name, sum(amount) AS sum_amount
   FROM example.department
   LEFT JOIN example.invoice ON department.id=invoice.purchaser
   GROUP BY name
   HAVING sum(amount) > 0
   ORDER BY sum_amount DESC;
   ```

Understanding authentication and authorization for Aurora DSQL

Aurora DSQL uses IAM roles and policies for cluster authorization. You associate IAM roles with PostgreSQL database roles for database authorization. This approach combines benefits from IAM with PostgreSQL privileges. Aurora DSQL uses these features to provide a comprehensive authorization and access policy for your cluster, database, and data.

Manage your clusters using IAM

Cluster management (creating, reading, updating, deleting, tagging) requires you to be authenticated and authorized with IAM to perform those actions.

- Authorization – to manage clusters, you must grant authorization using IAM actions for Aurora DSQL. For example, your IAM identity must have permissions to the IAM action `dsql:CreateCluster` to create a cluster. For more information, see the section called "Using IAM policy actions to manage clusters in Aurora DSQL".
- Authentication – use IAM to manage clusters. You can do so with the AWS Management Console, AWS CLI, or the AWS SDK.

Connect to your cluster using IAM

To connect to your cluster, you must authenticate with IAM and be authorized to connect to the cluster. When you connect, instead of providing a credential, you use a temporary authentication token.

- Authorization – grant the following IAM policy actions to the IAM identity you're using to establish the connection to your cluster's endpoint.
  - Use `dsql:DbConnectAdmin` if you're using the admin role. Aurora DSQL creates and manages this role for you.
  - Use `dsql:DbConnect` if you're using a custom database role. You create and manage this role in your database.
- Authentication – generate an authentication token using an IAM identity with authorization to connect. Provide it as the password when you connect to your database. To learn more, see Generating an authentication token.

- Authorization expiring – after you establish a connection, the role is authorized until the maximum of one hour for the connection. To learn more, see Understanding connections in Aurora DSQL.

Interact with your database using PostgreSQL database roles and IAM roles

To manage database-level authorization, you must use PostgreSQL database roles. Aurora DSQL has two types of roles: an admin role and a custom role. You manage database-level authorization by granting PostgreSQL permissions to your PostgreSQL database roles.

Aurora DSQL creates the admin role for you, and you can't modify it. Connecting as an admin lets you perform actions such as creating new database roles to associate with IAM roles. You must associate your custom database roles with your IAM roles to let the IAM role connect to your database.

- Authorization – use the admin role to connect to your cluster. Use PostgreSQL to set up custom database roles and grant permissions. To learn more, see PostgreSQL database roles and PostgreSQL privileges
- Authentication – use the admin role to connect to your cluster. Use the `AWS IAM GRANT` command to associate the custom database role with the IAM identity with authorization to connect to the cluster. To learn more, see the section called “Authorize custom database roles to connect to a cluster”.

Revoking authorization using IAM and PostgreSQL

Revoking admin authorization to connect to clusters

To revoke authorization to connect to your cluster with the admin role, you must revoke the IAM identity's access to `dsql:DbConnectAdmin`.

After revoking connection authorization from the IAM identity, Aurora DSQL rejects all new connection attempts from that IAM identity. Any active connections using the IAM identity might stay authorized for the connection’s duration. You can find connection duration in Quotas and limits. To learn more about connections, see the section called “Connections”.

Revoking custom role authorization to connect to clusters

To revoke access to non-admin database roles, you must revoke the IAM identity's access to `dsql:DbConnect`.

You can also remove the association between the database role and IAM by using the `AWS IAM REVOKE` operation. To learn more about revoking access from database roles, please see the section called "Revoke database authorization from an IAM role".

Revoking database-level privileges

You can't manage permissions of the admin role. To learn how to manage permissions for custom database roles, see PostgreSQL privileges. Modifications to privileges take effect on the next transaction after Aurora DSQL successfully commits the modification transaction.

Using IAM policy actions to manage clusters in Aurora DSQL

The IAM policy action you use depends on the role you use to connect to your cluster and any other IAM actions you need. For example, if your role only needs to be able to get cluster information, then you might limit their permissions to only the `GetCluster` and `ListClusters` permissions.

The following example policy shows all of the available IAM policy actions for managing clusters. For connecting to clusters, see Using IAM policy actions to connect to clusters.

```
{
"Version" : "2012-10-17",
  "Statement" : [
    {
"Effect" : "Allow",
      "Action" : [
        "dsql:CreateCluster",
        "dsql:GetCluster",
        "dsql:UpdateCluster",
        "dsql:DeleteCluster",
        "dsql:ListClusters",
        "dsql:CreateMultiRegionClusters",
        "dsql:DeleteMultiRegionClusters",
        "dsql:TagResource",
        "dsql:ListTagsForResource",
        "dsql:UntagResource"
      ],
      "Resource" : "*"
    }
```

```
  ]
}
```

Using IAM policy actions to connect to clusters

When connecting to your cluster with the default database role of admin you must use an IAM identity with authorization to perform following IAM policy action.

```
"dsql:DbConnectAdmin"
```

When connecting to your cluster with a custom role, you must first associate IAM role with the database role. The IAM identity you use to connect to your cluster must have authorization to perform the following IAM policy action.

```
"dsql:DbConnect"
```

To learn more about custom roles, see the section called “Using database roles with IAM roles”.

Amazon Resource Name (ARN) format for Aurora DSQL resources

The ARN format for a cluster in Aurora DSQL is `cluster/<clusterID>`.

For example, if your cluster ID was `foo0bar1baz2quux3quuux4`, the ARN of the cluster would be `arn:aws:dsql:us-east-1:123456789012:cluster/foo0bar1baz2quux3quuux4`.

Generating an authentication token in Amazon Aurora DSQL

To connect to Amazon Aurora DSQL with your preferred SQL client, you must generate an authentication token that you use as the password. By default, these tokens automatically expire in one hour if you use the AWS console to create it. If you use the AWS CLI or SDKs to create the token, the default is 15 minutes. The maximum is 604,800 seconds, which is one week. To connect to Aurora DSQL from your client again, you can use the same token if it hasn't expired, or you can generate a new one.

To get started with generating a token, first create an IAM policy and a cluster in Aurora DSQL, and then use the console, AWS CLI, or the AWS SDKs to generate a token.

At a minimum, you must have the following IAM permissions listed in https://docs.aws.amazon.com/aurora-dsql/latest/userguide/authentication-authorization.html#authentication-authorization-iam-role-connect, depending on what database role you want to use to connect.

Topics

Use the AWS console to generate a token in Aurora DSQL

The following steps describe how to use the AWS console to generate an authentication token.

1. After you create a cluster, choose the cluster ID of the cluster for which you want to generate an authentication token. Choose **Connect**.
2. In the modal, choose whether you want to connect as `admin` or with a custom database role.
3. Copy the generated authentication token and use it to connect to Aurora DSQL from your SQL client.

To learn more about custom database roles and IAM in Aurora DSQL, see *Authentication and authorization*.

Use AWS CloudShell to generate a token in Aurora DSQL

Before you can generate an authentication token, you must have completed following prerequisites.

- Created a Aurora DSQL cluster.
- Added permission to run the Amazon S3 operation `get-object` to retrieve objects from an AWS account outside of your organization.

The following steps describe how to use AWS CloudShell to generate an authentication token.

1. Sign in to the AWS Management Console.
2. After you sign in, navigate to Aurora DSQL's home page
3. At the bottom left of the AWS console, choose AWS CloudShell.
4. Follow Installing or updating to the latest verison of the AWS CLI to intall the AWS CLI.

```
sudo ./aws/install --update
```

5. Run the following command that generates an authentication token. Remember to replace the `region` and `cluster_endpoint` parameters with your Region and the endpoint of your own cluster. If you run into issues, see Troubleshoot IAM and How can I troubleshoot access denied or unauthorized operation errors with an IAM policy?.

```
aws dsql generate-db-connect-admin-auth-token \
--expires_in 3600 \
--region us-east-1 \
--hostname cluster_endpoint
```

> **Note**
>
> Use `generate-db-connect-auth-token` if you are _not_ connecting as `admin` user.

6. Use the following command to use psql to start a connection to your cluster.

```
PGSSLMODE=require \
 psql --dbname postgres \
--username admin \
--host cluster_endpoint
```

7. You should see a prompt to provide a password. Copy the token that you generated, and make sure you don't include any additional spaces or characters. Paste it into the following prompt from psql.

```
Password for user admin:
```

8. Press **Enter**. You should see a PostgreSQL prompt.

```
postgres=>
```

If you get an access denied error, make sure that your IAM identity has the `dsql:DbConnectAdmin` permission. If you have the permission and continue to get access deny errors, see Troubleshoot IAM and How can I troubleshoot access denied or unauthorized operation errors with an IAM policy?.

To learn more about custom database roles and IAM in Aurora DSQL, see *Authentication and authorization*.

Use the AWS CLI to generate a token in Aurora DSQL

Once your cluster is `ACTIVE`, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as `admin` user, you use the `generate-db-connect-admin-auth-token`
- If you are connecting with a custom database role, you use the `generate-db-connect-auth-token`

The following example uses the following attributes to generate an authentication token for the admin role.

- your_cluster_endpoint – endpoint of the cluster. Follows the format *`your_cluster_identifier`*.dsql.*`AWS_REGION`*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1

Linux and macOS

```
aws dsql generate-db-connect-admin-auth-token \
--region us-east-1 \
--expires_in 3600 \
--hostname <your_cluster_endpoint>
```

Windows

```
aws dsql generate-db-connect-admin-auth-token ^
--region=us-east-1 ^
--expires_in=3600 ^
--hostname=<your_cluster_endpoint>
```

Use the Python SDK to generate a token in Aurora DSQL

Once your cluster is ACTIVE, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as admin user, you use the generate_connect_admin_auth_token
- If you are connecting with a custom database role, you use the generate_connect_auth_token

The following example uses the following attributes to generate an authentication token for the admin role.

- your_cluster_endpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1

```
def generate_token(your_cluster_endpoint, region):
    client = boto3.client("dsql", region_name=region)
    # use `generate_db_connect_auth_token` if you are _not_ connecting as admin
 instead.
    token = client.generate_connect_admin_auth_token(your_cluster_endpoint, region)
    print(token)
    return token
```

Use the C++ SDK to generate a token in Aurora DSQL

Once your cluster is ACTIVE, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as `admin` user, you use the `GenerateDBConnectAdminAuthToken`
- If you are connecting with a custom database role, you use the `GenerateDBConnectAuthToken`

The following example uses the following attributes to generate an authentication token for the admin role.

- yourClusterEndpoint – endpoint of the cluster. Follows the format *`your_cluster_identifier`*.dsql.*`AWS_REGION`*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;

std::string generateToken(String yourClusterEndpoint, String region) {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQLClient client{clientConfig};
    std::string token = "";

    // If you aren not using admin role to connect, use GenerateDBConnectAuthToken
 instead
    const auto presignedString =
 client.GenerateDBConnectAdminAuthToken(yourClusterEndpoint, region);
    if (presignedString.IsSuccess()) {
        token = presignedString.GetResult();
    } else {
        std::cerr << "Token generation failed." << std::endl;
    }

    std::cout << token << std::endl;

    Aws::ShutdownAPI(options);
    return token;
}
```

Use the JavaScript SDK to generate a token in Aurora DSQL

Once your cluster is `ACTIVE`, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as `admin` user, you use the `getDbConnectAdminAuthToken`
- If you are connecting with a custom database role, you use the `getDbConnectAuthToken`

The following example uses the following attributes to generate an authentication token for the admin role.

- yourClusterEndpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1

```
import { DsqlSigner } from "@aws-sdk/dsql-signer";

async function generateToken(yourClusterEndpoint, region) {
  const signer = new DsqlSigner({
    hostname: endpoint,
    region,
  });
  try {
    // Use `getDbConnectAuthToken` if you are _not_ logging in as `admin` user
    const token = await signer.getDbConnectAdminAuthToken(yourClusterEndpoint, region);
    console.log(token);
    return token;
  } catch (error) {
      console.error("Failed to generate token: ", error);
      throw error;
  }
}
```

Use the Java SDK to generate a token in Aurora DSQL

Once your cluster is `ACTIVE`, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as `admin` user, you use the `generateDbConnectAdminAuthToken`
- If you are connecting with a custom database role, you use the `generateDbConnectAuthToken`

The following example uses the following attributes to generate an authentication token for the admin role.

- yourClusterEndpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws
- region – The AWS Region eg: us-east-1 or us-east-2

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.services.dsql.DsqlUtilities;
import software.amazon.awssdk.regions.Region;

public class GenerateAuthToken {
    public static String generateToken(String yourClusterEndpoint, Region region) {
        DsqlUtilities utilities = DsqlUtilities.builder()
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Use `generateDbConnectAuthToken` if you are _not_ logging in as `admin`
 user
        String token = utilities.generateDbConnectAdminAuthToken(builder -> {
            builder.hostname(yourClusterEndpoint)
                    .region(region);
        });

        System.out.println(token);
        return token;
    }
}
```

Use the Rust SDK to generate a token in Aurora DSQL

Once your cluster is ACTIVE, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as admin user, you use the db_connect_admin_auth_token
- If you are connecting with a custom database role, you use the db_connect_auth_token

The following example uses the following attributes to generate an authentication token for the admin role.

- yourClusterEndpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1

```
use aws_config::{BehaviorVersion, Region};
use aws_sdk_dsql::auth_token::{AuthTokenGenerator, Config};

async fn generateToken(String yourClusterEndpoint, String region) ->
 anyhow::Result<String< {
    let sdk_config = aws_config::load_defaults(BehaviorVersion::latest()).await;
    let signer = AuthTokenGenerator::new(
        Config::builder()
            .hostname(&yourClusterEndpoint)
            .region(Region::new(region))
            .build()
            .unwrap(),
    );

    // Use `db_connect_auth_token` if you are _not_ logging in as `admin` user
    let token = signer.db_connect_admin_auth_token(&sdk_config).await.unwrap();
    print(token);
    return token;
}
```

Use the Ruby SDK to generate a token in Aurora DSQL

Once your cluster is `ACTIVE`, you can generate an authentication token.

There are two ways to generate the token

- If you are connecting as `admin` user, you use the `generate_db_connect_admin_auth_token`
- If you are connecting with a custom database role, you use the `generate_db_connect_auth_token`

The following example uses the following attributes to generate an authentication token for the admin role.

- your_cluster_endpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws

- region – The AWS Region, such as us-east-2 or us-east-1

```
require 'aws-sdk-dsql'

def generate_token(your_cluster_endpoint, region)
  credentials = Aws::SharedCredentials.new()

  begin
      token_generator = Aws::DSQL::AuthTokenGenerator.new({
          :credentials => credentials
      })

      # The token expiration time is optional, and the default value 900 seconds
      # if you are not using admin role, use generate_db_connect_auth_token instead
      token = token_generator.generate_db_connect_admin_auth_token({
          :endpoint => your_cluster_endpoint,
          :region => region
      })
  rescue => error
    puts error.full_message
  end
end
```

Use the .NET to generate a token in Aurora DSQL

> **Note**
>
> .NET SDK does not provide the API to generate the token. The following code sample shows how to generate the authentication token for .NET.

Once your cluster is ACTIVE, you can generate an authentication token.

The following example uses the following attributes to generate an authentication token for the admin role.

- your_cluster_endpoint – endpoint of the cluster. Follows the format *your_cluster_identifier*.dsql.*AWS_REGION*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1
- action – needs to be specified based on the postgres user

- If you are connecting as `admin` user, you use the `DbConnectAdmin` action
- If you are connecting with a custom database role, you use the `DbConnect` action

```
using Amazon.Runtime;
using Amazon.Runtime.Internal;
using Amazon.Runtime.Internal.Auth;
using Amazon.Runtime.Internal.Util;

public static class TokenGenerator
{
    public static string GenerateDbConnectAdminAuthToken(string? your_cluster_endpoint,
 Amazon.RegionEndpoint region, string? action)
    {
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();

        string accessKey = awsCredentials.GetCredentials().AccessKey;
        string secretKey = awsCredentials.GetCredentials().SecretKey;
        string token = awsCredentials.GetCredentials().Token;

        const string DsqlServiceName = "dsql";
        const string HTTPGet = "GET";
        const string HTTPS = "https";
        const string URISchemeDelimiter = "://";
        const string ActionKey = "Action";

        action = action?.Trim();
        if (string.IsNullOrEmpty(action))
            throw new ArgumentException("Action must not be null or empty.");
        string ActionValue = action;
        const string XAmzSecurityToken = "X-Amz-Security-Token";

        ImmutableCredentials immutableCredentials = new ImmutableCredentials(accessKey,
 secretKey, token) ?? throw new ArgumentNullException("immutableCredentials");
        ArgumentNullException.ThrowIfNull(region);

        your_cluster_endpoint = your_cluster_endpoint?.Trim();
        if (string.IsNullOrEmpty(your_cluster_endpoint))
            throw new ArgumentException("Cluster endpoint must not be null or empty.");

        GenerateDsqlAuthTokenRequest authTokenRequest = new
 GenerateDsqlAuthTokenRequest();
        IRequest request = new DefaultRequest(authTokenRequest, DsqlServiceName)
        {
            UseQueryString = true,
            HttpMethod = HTTPGet
        };
        request.Parameters.Add(ActionKey, ActionValue);
        request.Endpoint = new UriBuilder(HTTPS, your_cluster_endpoint).Uri;
```



```
        if (immutableCredentials.UseToken)
        {
            request.Parameters[XAmzSecurityToken] = immutableCredentials.Token;
```

Use the Golang to generate a token in Aurora DSQL

> **Note**
>
> Golang SDK does not provide the API to generate the token. The following code sample shows how to generate the authentication token for Golang.

Once your cluster is `ACTIVE`, you can generate an authentication token.

The following example uses the following attributes to generate an authentication token for the admin role.

- yourClusterEndpoint – endpoint of the cluster. Follows the format *`your_cluster_identifier`*.dsql.*`AWS_REGION`*.on.aws
- region – The AWS Region, such as us-east-2 or us-east-1
- action – needs to be specified based on the postgres user
  - If you are connecting as `admin` user, you use the `DbConnectAdmin` action
  - If you are connecting with a custom database role, you use the `DbConnect` action

```
func GenerateDbConnectAdminAuthToken(yourClusterEndpoint string, region string, action
 string) (string, error) {
 // Fetch credentials
 sess, err := session.NewSession()
 if err != nil {
  return "", err
 }

 creds, err := sess.Config.Credentials.Get()
 if err != nil {
  return "", err
 }
 staticCredentials := credentials.NewStaticCredentials(
  creds.AccessKeyID,
  creds.SecretAccessKey,
  creds.SessionToken,
 )

 // The scheme is arbitrary and is only needed because validation of the URL requires
 one.
 endpoint := "https://" + yourClusterEndpoint
 req, err := http.NewRequest("GET", endpoint, nil)
 if err != nil {
  return "", err
 }
 values := req.URL.Query()
 values.Set("Action", action)
 req.URL.RawQuery = values.Encode()

 signer := v4.Signer{
  Credentials: staticCredentials,
 }
 _, err = signer.Presign(req, nil, "dsql", region, 15*time.Minute, time.Now())
 if err != nil {
  return "", err
 }

 url := req.URL.String()[len("https://"):]

 return url, nil
}
```

Using database roles with IAM roles

The following sections describe how to use database roles from PostgreSQL with IAM roles in Aurora DSQL.

Authorize custom database roles to connect to a cluster

Create an IAM role and grant connection authorization with the IAM policy action: `dsql:DbConnect`

The IAM policy must also grant permission to access the cluster resource(s). Use a wildcard `*` or How to restrict access to cluster ARNs.

Authorize database roles to use SQL in a database

You must use an IAM role with authorization to connect to the cluster.

1. Connect as a admin

   Use the `admin` database role with an IAM identity that is authorized for IAM action `dsql:DbConnectAdmin` to connect to your cluster.
2. Create a new database role

```
CREATE ROLE example WITH LOGIN;
```

3. Associate the database role with the AWS IAM role ARN

```
AWS IAM GRANT example TO 'arn:aws:iam::111122223333:role/example';
```

4. Grant database-level permissions to the database role

   Use GRANT to provide authorization within the database.

```
GRANT USAGE ON myschema TO example;
```

```
GRANT SELECT, INSERT, UPDATE ON ALL TABLES IN SCHEMA myschema TO example;
```

For more information, see PostgreSQL GRANT and PostgreSQL Privileges.

Revoke database authorization from an IAM role

To revoke database authorization, use the `AWS IAM REVOKE` operation.

```
AWS IAM REVOKE example FROM 'arn:aws:iam::111122223333:role/example';
```

To learn more about revoking authorization, see Revoking authorization using IAM and PostgreSQL.

Accessing Aurora DSQL

You can access Aurora DSQL through the following methods.

Topics

Using the PostgreSQL protocol with Aurora DSQL

The following table shows how Aurora DSQL supports the PostgreSQL protocol.

PostgreSQL	Aurora DSQL	Notes
Role (aka User or Group)	Database Role	Aurora DSQL creates a role for you named `admin`. If you create custom database roles, you must use the admin role to associate them with IAM roles for authenticating when connecting to your cluster. TODO For more informati on, see Configure custom database roles.
Host (aka hostname or hostspec)	Cluster Endpoint	Aurora DSQL single region clusters provide a single managed endpoint and will automatically redirect traffic in case there's any unavailab ility within the region.
Port	N/A - use default 5432	PostgreSQL default

PostgreSQL	Aurora DSQL	Notes
Database (dbname)	use `postgres`	Aurora DSQL creates this for you upon Cluster creation
SSL Mode	SSL is always enabled server-side	In Aurora DSQL, Aurora DSQL supports the `require` SSL Mode. Connections without SSL will be rejected by Aurora DSQL.
Password	Authentication Token	Aurora DSQL requires temporary authentication tokens instead of long-lived passwords. To learn more, see the section called “Generate an authentication token”.

You do not need to store credentials in the database because authentication is managed using IAM. An authentication token is a unique string of characters that is generated dynamically. Authentication tokens are generated using AWS Signature Version 4. The token is only used for authentication and doesn't affect the connection after it is established. If you try to re-connect using an expired token, the connection request is denied. For more information, see the section called “Generate an authentication token”.

Access Aurora DSQL through the AWS Management Console

Follow the steps below to access Aurora DSQL through the AWS Management Console.

1. Sign in to the AWS Management Console and open the Aurora DSQL at https://console.aws.amazon.com/dsql
2. To connect to your cluster, open AWS CloudShell and connect with psql.

Using SQL clients with Aurora DSQL

Aurora DSQL uses the PostgreSQL protocol, so you can use your preferred interactive client by providing a signed IAM authentication token as the password when connecting to your cluster.

In Aurora DSQL, extra steps are required to generate an authentication token and provide to the SQL client to connect to your cluster.

Topics

Access Aurora DSQL with psql (PostgreSQL interactive terminal)

For more information about psql, see https://www.postgresql.org/docs/current/app-psql.htm. To download the PostgreSQL-provided installers, see PostgreSQL Downloads.

If you already have the AWS CLI installed, use the following example to connect to your cluster. You can also use AWS CloudShell which comes with psql pre-installed, or, you can install psql directly.

```
# Aurora DSQL requires a valid IAM token as the password when connecting.
# Aurora DSQL provides tools for this and here we're using Python.
export PGPASSWORD=$(aws dsql generate-db-connect-admin-auth-token \
--region us-east-1 \
--expires-in 3600 \
--hostname your_cluster_endpoint)

# Aurora DSQL requires SSL and will reject your connection without it.
export PGSSLMODE=require

# Connect with psql which will automatically use the values set in PGPASSWORD and
 PGSSLMODE.
# Quiet mode will suppress unnecessary warnings and chatty responses. Still outputs
 errors.
psql --quiet \
--username admin \
--dbname postgres \
--host your_cluster_endpoint
```

Access Aurora DSQL with DBeaver

See the following steps to learn how to use the DBeaver Community edition to connect to your cluster.

Set up a new connection

1. Choose **New Database Connection**.
2. In the **New Database Connection** window, choose PostgreSQL.
3. In the **Connection settings/Main** tab, choose **Connect by: Host** and enter the following information.

   - **Host** - Use your cluster endpoint.

     Database - Enter `postgres`

     Authentication - Choose `Database Native`

     Username - Enter `admin`

     Password - Generate an authentication token. Copy the generated token and use it as your password.
4. Ignore any warnings and paste your authentication token into the **DBeaver Password** field.

   > **Note**
   >
   > You must set SSL mode in the client connections. Aurora DSQL supports `SSLMODE=require`. Aurora DSQL enforces SSL communication on the server side and rejects non-SSL connections.
5. You should be connected to your cluster and can start running SQL statements.

> **Important**
>
> The administrative features provided by DBeaver for the PostgreSQL databases (such as **Session Manager** and **Lock Manager**) do not apply to a database, due to its unique architecture. While accessible, these screens will not provide reliable information on the database health or status.

Authentication credentials expiry

Established sessions will remain authenticated for a maximum of 1 hour or until an explicit disconnect or a client-side timeout takes place. If new connections need to be established, a valid Authentication token must be provided in the **Password** field of the **Connection settings**. Trying to open a new session (for example, to list new tables, or a new SQL console) will force a new authentication attempt. If the authentication token configured in the **Connection settings** is no longer valid, that new session will fail and all the previously opened sessions will get invalidated at that point in time too. Have this in mind when choosing the duration of your IAM authentication token with the `expires-in` option.

Access Aurora DSQL with JetBrains DataGrip

Set up a new connection

1. Choose **New Data Source** and choose PostgreSQL.
2. In the Data Sources/General tab, enter the following information:
   - **Host** - Use your cluster endpoint.

     Port - Aurora DSQL uses the PostgreSQL default: `5432`

     Database - Aurora DSQL uses the PostgreSQL default of `postgres`

     Authentication - Choose `User & Password` .

     Username - Enter `admin`.

     Password - Generate a token and paste it into this field.

     URL - Don't modify this field. It will be auto-populated based on the other fields.
3. **Password** - Provide this by generating an authentication token. Copy the resulting output of the token generator and paste it into the password field.

> **Note**
>
> You must set SSL mode in the client connections. Aurora DSQL supports `PGSSLMODE=require`. Aurora DSQL enforces SSL communication on the server side and will reject non-SSL connections.

4. You should be connected to your cluster and can start running SQL statements:

> **Important**
>
> Some of the views provided by DataGrip for the PostgreSQL databases (such as Sessions) don't apply to a database because of its unique architecture. While accessible, these screens don't provide reliable information on the actual sessions connected to the database.

Authentication credentials expiration

Established sessions will remain authenticated for a maximum of 1 hour or until an explicit disconnect or a client-side timeout takes place. If new connections need to be established, a new Authentication token must be generated and provided in the **Password** field of the **Data Source Properties**. Trying to open a new session (for example, to list new tables, or a new SQL console) will force a new authentication attempt. If the authentication token configured in the **Connection** settings is no longer valid, that new session will fail and all the previously opened sessions will become invalid.

Understanding programmatic access to Amazon Aurora DSQL

Aurora DSQL provides you with the following tools to manage your Aurora DSQL resources programmatically.

AWS Command Line Interface (AWS CLI)

You can create and manage your resources by using the AWS CLI in a command-line shell. The AWS CLI provides direct access to the APIs for AWS services, such as Aurora DSQL. For syntax and examples for the commands for Aurora DSQL, see dsql in the *AWS CLI Command Reference*.

AWS software development kits (SDKs)

AWS provides SDKs for many popular technologies and programming languages. They make it easier for you to call AWS services from within your applications in that language or technology. For more information about these SDKs, see Tools for developing and managing applications on AWS.

Aurora DSQL API

This API is another programming interface for Aurora DSQL. When using this API, you must format every HTTPS request correctly and add a valid digital signature to every request. For more information, see *API reference*.

AWS CloudFormation

During Preview, Aurora DSQL doesn't support AWS CloudFormation.

Working with Amazon Aurora DSQL

The following sections describe various ways of working with Aurora DSQL.

Topics

Understanding PostgreSQL compatibility

Aurora DSQL is PostgreSQL compatible, which means that it provides identical behavior for most supported features, identical query results for all SQL features, and supports many popular PostgreSQL drivers and tools with minor configuration changes. Supported SQL expressions return identical data in query results, including sort order, scale and precision for numeric operations, and equivalence for string operations. With a few documented exceptions, such as synchronous replication, no-lock concurrency control, and asynchronous DDL execution, Aurora DSQL behaves comparably to PostgreSQL.

Aurora DSQL supports core relational features like ACID transactions, secondary indexes, joins, insert, and updates. See Supported SQL expressions for an overview of supported SQL features.

Aurora DSQL doesn't support all PostgreSQL features. For more information, see Unsupported PostgreSQL features.

Aurora DSQL uses standard PostgreSQL drivers and supports common PostgreSQL-compatible tools with some configuration changes. To see a list of supported tools, see Utilities, tools, and

sample code. To see code examples and other developer-related topics, see Programming with Aurora DSQL.

Supported data types in Aurora DSQL

See the following table to learn about the supported core data types in Aurora DSQL

Name	Aliases	Description	Aurora DSQL Specific Limits	Storage Size
smallint	int2	signed two-byte integer. 32768 to +3276		2 bytes
integer	int, int4	signed four-byte integer. -2147483648 to +2147483647		4 bytes
bigint	int8	signed eight-byte integer. -92233720 36854775808 to +92233720 36854775807		8 bytes
boolean	bool	logical Boolean (true/false)		1 byte
character [(n)]	char [(n)]	fixed-length character string	4096 bytes [1] [2]	variable up to 4100 bytes
character varying [(n)]	varchar [(n)]	variable-length character string	65535 bytes [1] [2]	variable up to 65539 bytes
bpchar [(n)]		if fixed-length, alias for char	4096 bytes [1] [2]	variable up to 4100 bytes

Name	Aliases	Description	Aurora DSQL Specific Limits	Storage Size
		if variable length, alias for varchar where trailing spaces are semantically insignificant		
text		variable-length character string	1 MiB [1] [2]	variable up to 1MB limit
bytea		binary data ("byte array")	1 MiB [1] [2]	variable up to 1MB limit
real	float4	single precision floating-point number (4 bytes). 6 decimal digits precision		4 bytes
double precision	float8	double precision floating-point number (8 bytes). 15 decimal digits precision		8 bytes
numeric [(p, s)]	decimal [(p, s)], dec[(p,s)]	exact numeric of selectable precision. Up to 38 digits before the decimal point up to 37 digits after the decimal point	Max precision = 38 Max scale = 37 [2]	8 bytes + 2 bytes per percision digit. Max size 27 bytes

Name	Aliases	Description	Aurora DSQL Specific Limits	Storage Size
date		calendar date (year, month, day)		4 bytes
time [(p)] [without time zone]	timestamp	time of day (no time zone)		8 bytes
time [(p)] with time zone	timetz	time of day, including time zone		12 bytes
timestamp [(p)] [without time zone]		date and time (no time zone)		8 bytes
timestamp [(p)] with time zone	timestamptz	date and time, including time zone		8 bytes
interval [fields] [(p)]		time span		16 bytes
UUID		universally unique identifier (v4)		16 bytes

[1] – If you use this data type in a primary key or key column, the maximum size is limited to 255 bytes.

[2] – If you don't explicitly specify a size when you run `CREATE TABLE` or `ALTER TABLE ADD COLUMN`, then Aurora DSQL enforces the defaults. Aurora DSQL applies limits when you run `INSERT` or `UPDATE`.

The following are data types and their implicit limits.

Name	Implicit Limit
character [(n)]	4096 bytes
bpchar [(n)]	4096 bytes
character varying [(n)]	65535 bytes
text	1MB
bytea	1 MB
numeric [(p, s)]	numeric (18,0)

Date time precision

The following are the supported date time data types in Aurora DSQL and their precision values.

Name	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	4713 BC	294276 AD	1 microsecond
timestamp [(p)] with time zone	4713 BC	294276 AD	1 microsecond
date	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	0	1	1 microsecond
time [(p)] with time zone	00:00:00+1559	24:00:00-1559	1 microsecond
interval [fields] [(p)]	-178000000 years	178000000 years	1 microsecond

Types supported during query runtime

Aurora DSQL supports all documented storage types when you apply type transformations, such as CAST and AS, when you run queries. The following types are supported only during query runtime.

- Array types of all of the above types.

```
postgres=> select string_to_array('1,2', ',');
 string_to_array
-----------------
 {1,2}
(1 row)
```

- inet – IPv4, IPv6 host addresses, and their subnets. For more information, see inet in the PostgreSQL documentation.

Supported PostgreSQL features in Aurora DSQL

The table below characterizes general PostgreSQL expression support for the Aurora DSQL. This list is not exhaustive.

> **Warning**
>
> In Aurora DSQL, you might find that SQL expressions beyond those characterized below are working. Be aware that there may be changes to behavior or support for such expressions.

Category	Primary Clause	Supported Clauses
SELECT	FROM	see SELECT rows
SELECT	GROUP BY	ALL, DISTINCT
SELECT	ORDER BY	ASC, DESC, NULLS ..
SELECT	LIMIT	
SELECT	DISTINCT	

Category	Primary Clause	Supported Clauses
SELECT	HAVING	
SELECT	WITH	
SELECT	USING	
SELECT	INNER JOIN	ON
SELECT	OUTER JOIN	LEFT, RIGHT, FULL, ON
SELECT	CROSS JOIN	ON
SELECT	OVER	RANK (), PARTITION BY
SELECT	UNION	UNION ALL
SELECT	INTERSECT	INTERSECT ALL
SELECT	EXCEPT	EXCEPT ALL
SELECT	FOR UPDATE	
INSERT	INTO	VALUES
INSERT	INTO	SELECT, WITH
UPDATE	SET	WHERE, WHERE (SELECT)
UPDATE	SET	FROM, WITH
CREATE	TABLE	PRIMARY KEY
CREATE	INDEX	Can run on empty tables only.
CREATE	INDEX ASYNC	ON, NULLS FIRST, NULLS LAST
DROP	TABLE	
DROP	INDEX	

Category	Primary Clause	Supported Clauses
DELETE	FROM	USING, WHERE
GRANT	[permission]	ON, TO
REVOKE	[permission]	ON, FROM, CASCADE, RESTRICT
CREATE	ROLE, WITH	
CREATE	FUNCTION	LANGUAGE SQL
EXPLAIN	-	-
BEGIN	[WORK | TRANSACTION] [READ ONLY | READ WRITE	
COMMIT		
ANALYZE		relation name only
CREATE	DOMAIN	
ALTER	TABLE	ADD COLUMN (no default)

Supported subsets of PostgreSQL commands in Aurora DSQL

Aurora DSQL doesn't support all of the syntax in supported PostgreSQL commands. For example, `CREATE TABLE` in PostgreSQL has a large number of clauses and parameters that Aurora DSQL doesn't support. This section describes the syntax of PostgreSQL syntax that Aurora DSQL does support for these commands.

CREATE TABLE

```
CREATE TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
```

```
] )

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression )|
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] index_parameters |
  PRIMARY KEY index_parameters |

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |

and like_option is:

{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | GENERATED | IDENTITY |
 INDEXES | STATISTICS | ALL }

index_parameters in UNIQUE, and PRIMARY KEY constraints are:
[ INCLUDE ( column_name [, ... ] ) ]
```

ALTER TABLE

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema

where action is one of:
```

```
    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type
    OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

Unsupported PostgreSQL features in Aurora DSQL

See the following to learn more about which PostgreSQL features are unsupported in Aurora DSQL

Unsupported objects

- Databases - Aurora DSQL supports only one database per cluster at this time.
- Views
- Temporary Tables
- Triggers
- Types
- Tablespaces
- UDFs / Functions other than functions using language = SQL
- Sequences

Unsupported constraints

- Foreign keys
- Exclusion constraints

Unsupported operations

- ALTER SYSTEM
- TRUNCATE
- VACUUM
- SAVEPOINT

Unsupported extensions

Aurora DSQL doesn't support PostgreSQL extensions at this time. The following notable extensions are unsupported.

- PL/pgSQL
- PostGIS
- PGVector
- PGAudit
- Postgres_FDW
- PGCron
- pg_stat_statements

Unsupported SQL expressions

Category	Primary Clause	Unsupported Clauses
CREATE	VIEW	
CREATE	INDEX [ASYNC]	ASC DESC
CREATE	INDEX	if table has data
TRUNCATE		
ALTER	SYSTEM	All alter systemis blocked
CREATE	TABLE	COLLATE, AS SELECT, INHERITS, PARTITION
CREATE	FUNCTION	LANGUAGE plpgsql (any language besides sql)
CREATE	TEMPORARY	TABLES
CREATE	EXTENSION	
CREATE	SEQUENCE	
CREATE	MATERIALIZED	VIEW
CREATE	TABLESPACE	

Category	Primary Clause	Unsupported Clauses
CREATE	TRIGGER	
CREATE	TYPE	
CREATE	DATABASE	

Limitations

- CREATE DATABASE: Aurora DSQL supports a single database postgres which is UTF-8 and collation = C only. You can't modify the system timezone and it's set to UTC
- SET TRANSACTION [ISOLATION LEVEL]: Aurora DSQL isolation level is equivalent to PostgreSQL Repeatable Read. You can't change this isolation level.
- A transaction can't contain mixed DDL and DML operations
- A transaction can contain at most 1 DDL statement
- A transaction cannot modify more than 10,000 rows, and this limit is modified by secondary index entries. For example, consider a table with five columns, where the primary key is the first column, and the fifth column has a secondary index. Given an UPDATE that will change a single row targeting all five columns, the number of rows modified would be two. One for the Primary Key and one for the row in secondary index object. If this same UPDATE affected only the columns without a secondary index, the number of rows modified would be one. This limit applies to all DML statements (INSERT, UPDATE, DELETE).
- A connection cannot exceed 1 hour.
- AutoVacuuming to keep statistics up to date. Vacuum is not required in Aurora DSQL.

Understanding connections in Aurora DSQL

To connect to Aurora DSQL, use a standard Postgres driver configured with TLS. To connect, you specify a Postgres role as the user, a password, and an authentication token. Aurora DSQL provides libraries for you to generate authentication tokens in most AWS supported languages. Once you're connected, you can use your session to run transaction for up to 1 hour with a transaction timeout of 5 minutes each. If you start a transaction after the 55th minute, Aurora DSQL still runs the transaction until you reach the limit of 5 minutes before it closes the session.

Aurora DSQL authenticates each session with a state, such as prepared statements or an active query. A **connection** is a TLS-wrapped TCP connection that might get rejected if Aurora DSQL can't turn it into a session for any reason. Each session maps to exactly one connection. With a connection, a client can't have a session, and a connection can only have one session in Aurora DSQL.

To make sure that a user with revoked Postgres credentials can't connect to a cluster on an existing session, we authenticate the user against Aurora DSQL's IAM trust tables at the beginning of each transaction.

Connection limits

By default, you can create up to 1000 connections per cluster at 10 connections per second with a burst of 100. For example, if one connection is one token in a token bucket, you begin with 100 available tokens in the bucket. If you create 100 tokens, you have zero remaining tokens and have to wait for a second before you can create more connections. The refill rate is 10 tokens per second. To increase these limits, contact AWS support.

Understanding concurrency control in Aurora DSQL

Aurora DSQL is PostgreSQL compatible. Repeatable read operations in PostgreSQL are the same as ACID transactions with snapshot isolation in Aurora DSQL. Unlike PostgreSQL, Aurora DSQL uses a lock-free concurrency control mechanism. This means that a slow transaction can't slow other transactions, and transactional deadlocks can't happen. This approach is often better than optimistic concurrency control.

Optimistic concurrency control (OCC) is evaluated at transaction commit time. This is different than lock-based concurrency control, which first establishes locks on changed rows or tables to make sure that conflicts don't occur when Aurora DSQL processes commits. With an optimistic control scheme, Aurora DSQL assumes that application are designed to minimize conflict, so locking objects is often unnecessary.

If conflicts happen in OCC, such as multiple concurrent transactions updating the same row, Aurora DSQL processes the transaction with the earliest start time. All other conflicting transactions return a PostgreSQL serialization error. This error indicates to a client that you should have abort and retry logic. This is similar to abort and retry logic that would be applied with a standard PostgreSQL lock timeout or deadlock situation. However, this abort and retry logic is exercised more frequently in an OCC-based scheme. The ideal design pattern should be to enable transaction retry as a first recourse whenever possible. This is known as idempotency.

When you consider workload performance, you should still think about common relational database regardless of which concurrency control scheme you use. First, avoid high contention on single keys or small key ranges/hot keys. This means that you should design your schema in a way that spreads out update operations over your cluster key range. This can be as simple as choosing a random primary key for your tables and avoiding patterns that increase contention on single keys as business growth increases the demand to update your database.

Understanding data definition language (DDL) in Aurora DSQL

Aurora DSQL features a Multi-AZ distributed and shared-nothing database layer built on top of multi-tenant compute and storage fleets. Because there isn't a single primary database node or leader, the database catalog is distributed, and schema changes are managed as distributed transactions. As such, there are a few ways in which DDL behaves differently in Aurora DSQL than PostgreSQL.

- Aurora DSQL throws a concurrency control violation error if you run one transaction while another transaction updates a resource. Consider the following example.
  - Create table `foobar` in session 1.
  - After Aurora DSQL creates the table `foobar`, you run the statement `SELECT * from foobar` in session 2. Aurora DSQL returns with the error `SQL Error [40001]: ERROR: schema has been updated by another transaction, please retry: (OC001).`

    > **Note**
    >
    > During preview, there is a known issue that increases the scope of this concurrency control error to all objects within the same schema/namespace.

- Transactions in Aurora DSQL can contain only one DDL statement and can't have both DDL and DML statements. For example, you can't create a table and insert data into the same table within the same transaction.

  For example, Aurora DSQL supports the following statements.

```
BEGIN
CREATE TABLE FOO (ID_col integer);
COMMIT;
```

```
BEGIN
INSERT into FOO VALUES (1);
COMMIT;
```

Aurora DSQL doesn't support the following.

```
BEGIN
CREATE TABLE FOO (ID_col integer);
INSERT into FOO VALUES (1);
COMMIT;
```

- Finally, Aurora DSQL runs DDL statements asynchronously. This means that changes to large tables, such as adding an index or modifying a column, can run without downtime or performance impact. For more information about Aurora DSQL's asynchronous job manager, see the section called “Async indexes”.

Primary keys in Aurora DSQL

In Aurora DSQL, defining a primary key for your table is similar to the CLUSTER operation in PostgreSQL or a clustered index in other database systems. Aurora DSQL applies an INCLUDE statement that references all columns, which creates a table organized by an index. This structure makes it so that any lookup against an Aurora DSQL primary key can access all column values associated with the key, and the data is always ordered according to the primary key. Unlike the CLUSTER operation, Aurora DSQL always maintains the order of this index-organized table.

Aurora DSQL uses this main concept to organize distributed data management. Aurora DSQL uses the primary key to construct a cluster-wide unique key that's assigned to each row in each table or index. Aurora DSQL uses this key to automatically partition storage. This partition key plays a central role in Aurora DSQL automatic scaling and concurrency control mechanisms.

Consider the following when you choose a primary key.

- We recommend that you define a primary key when you create a table in Aurora DSQL. While Aurora DSQL assigns a synthetic hidden ID if you don't define a primary key, this might not support join operations or fast indexed lookups for larger tables.
- Once you create a table, you can't change the primary key, and you can't add a new primary key later.

- For tables with high write volumes, avoid using monotonically increasing integers as primary keys, which can lead to weaker performance. Randomness in primary keys ensures even distribution of new writes across storage partitions. Instead, using monotonically increasing integers as primary keys can lead to all new inserts being directed to a single partition, which creates a bottleneck.
- If your table doesn't change very often or is read-only, you can use an ascending key, even if it is a dense key. Doing so is fine because there you don't need a high level of performance for loading data into the key.
- Generally speaking, if doing a full scan of the table doesn't meet your performance needs, choose a primary key that represents your most common join and lookup key when you query the table.
- The maximum combined size of a column that you can use in a primary key is 1 kibibyte. For more information, see Database limits in Aurora DSQL .
- The maximum number of columns that you can include in a primary key or a secondary index is 8. For more information. see Database limits in Aurora DSQL.

Creating async indexes in Aurora DSQL

The `CREATE INDEX ASYNC` command lets you create an index on a column of a specified table. `CREATE INDEX ASYNC` is an asynchronous DDL operation, so running this command doesn't block your other transactions, and Aurora DSQL immediately returns a `job_id` to you. You can see the status of an asynchronous job at any time with the `sys.jobs` system view.

Aurora DSQL also supports the procedures `sys.wait_for_job(job_id)` and `sys.cancel_job(job_id)`. `sys.wait_for_job` lets you block the session until the specified job completes or fails. This procedure returns a Boolean. `sys.cancel_job` lets you cancel an asynchronous job that is in progress.

When Aurora DSQL finishes an asynchronous index task, it also updates the system catalog to show that the index is now active. If any other transactions reference the objects in the same namespace at this time, you might see a concurrency error.

> **Note**
>
> During Preview, asynchronous task completion might result in concurrency control errors for all in-progress transactions that reference the same namespace.

Syntax

See the following to learn about the parameters for `CREATE INDEX ASYNC`.

```
CREATE [ UNIQUE ] INDEX ASYNC [ IF NOT EXISTS ] name ON table_name
     ( { column_name } [ NULLS { FIRST | LAST } ] )
     [ INCLUDE ( column_name [, ...] ) ]
     [ NULLS [ NOT ] DISTINCT ]
```

Parameters

UNIQUE

Indicates to Aurora DSQL to check for duplicate values in the table when it creates the index and each time you add data. If you specify this parameter, insert and update operations that would result in duplicate entries will generate an error.

During Preview, you can only create unique indexes on empty tables. If you try to create a unique index on a non-empty table, the operation fails. You can see the related error and other information in the details column of the `sys.jobs` view.

IF NOT EXISTS

Indicates that Aurora DSQL shouldn't throw an exception if an index with the same name already exists. If an index with the same name already exists, Aurora DSQL doesn't create the new index. However, the index you're trying to create could have a very different structure than the index that already exists. If you specify this parameter, index name is required.

name

The name of the index to create. You can't include the name of your schema in this parameter. Aurora DSQL always creates the index in the same schema as its parent table. The name of the index must be distinct from the name of any other object, such as table or index, in the schema. If you don't specify a name, Aurora DSQL automatically generates a name based on the name of the parent table and the name of the indexed column. For example, if you run `CREATE INDEX ASYNC on table1 (col1, col2);`, Aurora DSQL automatically names the index as `table1_col1_col2_idx`.

NULLS FIRST | LAST

Specifies the order of how to sort null columns and non-null columns. FIRST indicates that Aurora DSQL should sort null columns before non-null columns. LAST indicates that Aurora DSQL should sort null columns after non-null columns.

INCLUDE

The INCLUDE clause specifies a list of columns to include in the index as non-key columns. You can't use a non-key column in an index scan search qualification, and Aurora DSQL ignores the column in terms of uniqueness for an index.

NULLS DISTINCT | NULLS NOT DISTINCT

Specifies whether Aurora DSQL should consider null values as distinct/not equals in a unique index. Default is DISTINCT, which indicates that null values are distinct, so a unique index can contain multiple null values in a column. NOT DISTINCT indicates that null values aren't distinct, so an index can't contain multiple null values in a column.

Examples

The following example demonstrates how to create a schema, a table, and then an index.

```
CREATE SCHEMA test;
```

```
CREATE TABLE test.departments (name varchar(255) primary key not null,
     manager varchar(255),
     size varchar(4));
```

Add some data into the table.

```
INSERT INTO test.departments('Human Resources', 'John Doe', '10')
```

Then create the index.

```
CREATE INDEX ASYNC test_index on test.departments(name, manager, size);
```

The CREATE INDEX command returns a job_id.

```
job_id
--------------------------
jh2gbtx4mzhgfkbimtgwn5j45y
```

With this `job_id`, you can use the procedures `sys.wait_for_job` or `sys.cancel_job` to block the session from other transactions until Aurora DSQL completes the job or cancel the job.

When you receive the `job_id`, then Aurora DSQL has started to create the new index on a new job. You can use the procedure `sys.wait_for_job(job_id)` to block other work on the session until the job finishes, is canceled, or if the session times out. To cancel an active async index creation job, use the procedure `sys.cancel_job(job_id)`.

```
select relname as index_name, indisvalid as is_valid, pg_get_indexdef(indexrelid) as
 index_definition
from pg_index, pg_class
where pg_class.oid = indexrelid and indrelid = 'test.departments'::regclass;
```

```
    index_name    | is_valid |
 index_definition
------------------+----------
+-------------------------------------------------------------------------------------------------
 department_pkey  |     t    | CREATE UNIQUE INDEX department_pkey ON test.departments
 USING remote_btree_index (title) INCLUDE (name, manager, size)
 test_index1      |     t    | CREATE INDEX test_index1 ON test.departments USING
 remote_btree_index (name, manager, size)
```

To check the creation status of your index, query the `sys.jobs` system view.

```
SELECT * from sys.jobs
```

Aurora DSQL returns a response similar to the following.

```
           job_id           |   status   | details
----------------------------+------------+---------
 vs3kcl3rt5ddpk3a6xcq57cmcy | completed  |
 yzke2pz3xnhsvol4a3jkmotehq | cancelled  |
 ihbyw2aoirfnrdfoc4ojnlamoq | processing |
```

The status column can be one of the following values:

- submitted – The task is submitted, but Aurora DSQL hasn't started to process it yet.
- processing – Aurora DSQL is processing the task.
- failed – the task failed. See the details column for more information. If Aurora DSQL failed to build the index, Aurora DSQL doesn't automatically remove the index definition. You must manually remove the index with the `DROP INDEX` command.
- completed – Aurora DSQL finished the task.
- cancelled – The task is canceled.

Usage notes

When using CREATE INDEX ASYNC, consider the following:

- Running the `CREATE INDEX ASYNC` command doesn't introduce any locks to your applications and doesn't affect the base table that Aurora DSQL uses to create the index.
- During schema migration operations, the `sys.wait_for_job(job_id)` procedure is especially helpful because you can ensure that subsequent DDL and DML operations all target the newly created index.
- If you cancel a task, Aurora DSQL automatically updates the corresponding entry in the `sys.jobs` system view. As Aurora DSQL runs the task, it also checks the `sys.jobs` view to see if the task has been updated to canceled. If it is, Aurora DSQL stops the task. If you encounter an error that Aurora DSQL is updating your schema with another transaction, try to cancel again. After you cancel a task to create an async index, we recommend that you also drop the index.
- While Aurora DSQL creates your index, it has an initial status of `INVALID`. The `indisvalid` flag for the index returns `FALSE` or `f`, which indicates that the index isn't valid. If the flag returns `TRUE` or `t`, the index is ready.
- If Aurora DSQL fails to build an async index, that index stays `INVALID`. This invalid index takes up storage space and can receive updates and inserts from other queries. We recommend that you drop all invalid indexes and recreate them.
- Every time Aurora DSQL runs a new asynchronous task, it checks the `sys.jobs` view and deletes tasks that have the `completed`, `failed`, or `cancelled` statuses for more than 30 minutes. Doing so means `sys.jobs` primarily shows only in-progress tasks and doesn’t contain information about old tasks.

Using system tables and commands in Aurora DSQL

See the following esctions to learn about the supported system tables and catalogs in Aurora DSQL.

System tables and queries in Aurora DSQL

Aurora DSQL is compatible with PostgreSQL, so many system catalog tables and views from PostgreSQL also exist in Aurora DSQL.

Important Postgres catalog tables and views

The following table describes the most common tables and views you might use in Aurora DSQL.

Name	Description
pg_namespace	Information on all schemas
pg_tables	Information on the all tables
pg_attribute	Information on all attributes
pg_view	Information on (pre-)defined views
pg_class	Describes all tables, column, indices, and similar objects
pg_stats	A view on the planner statistics
pg_user	Information on users
pg_roles	Information on users and groups
pg_indexes	Lists all indexes
pg_constraint	Lists constraints on tables

sys.jobs and sys.iam_pg_role_mappings

Aside from these tables and views, Aurora DSQL also adds the views `sys.jobs` and `sys.iam_pg_role_mappings` for your use cases.

`sys.jobs` provides status information about asynchronous jobs. For example, after you create an async index, Aurora DSQL returns a `job_uuid`. You can use this `job_uuid` with `sys.jobs` to look up the status of the job.

```
select * from sys.jobs where job_id = 'example_job_uuid';
```

```
        job_id        |   status   | details
----------------------+------------+---------
 example_job_uuid | processing |
(1 row)
```

The view `sys.iam_pg_role_mappings` provides information about the permissions granted to IAM users. For example, suppose that `DQSLDBConnect` is an IAM role to give access of Aurora DSQL to non-admins, and that there's a user named `testuser` that is granted the `DQSLDBConnect` role and corresponding permissions. You can then query the `sys.iam_pg_role_mappings` view to see which users are granted which permissions.

```
select * from sys.iam_pg_role_mappings;
```

Querying table sizes

To get the approximate count of how many rows are in a table, run the following command.

```
select reltuples from pg_class where relname = '<table_name>';
```

```
  reltuples
--------------
 9.993836e+08
```

If you want the size of bytes of a table, run the following command. Note that 32768 is an internal parameter that you must include in the query.

```
select pg_size_pretty(relpages * 32768::bigint) as relbytes from pg_class where relname
 = '<example_table_name>';
```

Supported and unsupported catalog tables and views in Aurora DSQL

See below for the complete list of which tables and views are supported and unsupported in Aurora DSQL.

System catalog tables

Name	Applicable to Aurora DSQL
pg_aggregate	No
pg_am	Yes
pg_amop	No
pg_amproc	No
pg_attrdef	Yes
pg_attribute	Yes
pg_authid	Yes
pg_auth_members	Yes
pg_cast	Yes
pg_class	Yes
pg_collation	Yes
pg_constraint	Yes
pg_conversion	No
pg_database	No
pg_db_role_setting	Yes
pg_default_acl	Yes
pg_depend	Yes
pg_description	Yes
pg_enum	No

Name	Applicable to Aurora DSQL
pg_event_trigger	No
pg_extension	No
pg_foreign_data_wrapper	No
pg_foreign_server	No
pg_foreign_table	No
pg_index	Yes
pg_inherits	Yes
pg_init_privs	No
pg_language	No
pg_largeobject	Yes
pg_largeobject_metadata	Yes
pg_namespace	Yes
pg_opclass	No
pg_operator	Yes
pg_opfamily	No
pg_parameter_acl	Yes
pg_partitioned_table	Yes
pg_policy	No
pg_proc	No
pg_publication	No

Name	Applicable to Aurora DSQL
pg_publication_namespace	No
pg_publication_rel	No
pg_range	Yes
pg_replication_origin	No
pg_rewrite	No
pg_seclabel	No
pg_sequence	No
pg_shdepend	Yes
pg_shdescription	Yes
pg_shseclabel	No
pg_statistic	Yes
pg_statistic_ext	No
pg_statistic_ext_data	No
pg_subscription	No
pg_subscription_rel	No
pg_tablespace	Yes
pg_transform	No
pg_trigger	No
pg_ts_config	Yes
pg_ts_config_map	Yes

Name	Applicable to Aurora DSQL
pg_ts_dict	Yes
pg_ts_parser	Yes
pg_ts_template	Yes
pg_type	Yes
pg_user_mapping	No

System views

Name	Applicable to Aurora DSQL
pg_available_extensions	No
pg_available_extension_versions	No
pg_backend_memory_contexts	Yes
pg_config	No
pg_cursors	No
pg_file_settings	No
pg_group	Yes
pg_hba_file_rules	No
pg_ident_file_mappings	No
pg_indexes	Yes
pg_locks	No
pg_matviews	No

Name	Applicable to Aurora DSQL
pg_policies	No
pg_prepared_statements	No
pg_prepared_xacts	No
pg_publication_tables	No
pg_replication_origin_status	No
pg_replication_slots	No
pg_roles	Yes
pg_rules	No
pg_seclabels	No
pg_sequences	No
pg_settings	Yes
pg_shadow	Yes
pg_shmem_allocations	Yes
pg_stats	Yes
pg_stats_ext	No
pg_stats_ext_exprs	No
pg_tables	Yes
pg_timezone_abbrevs	Yes
pg_timezone_names	Yes
pg_user	Yes

Name	Applicable to Aurora DSQL
pg_user_mappings	No
pg_views	Yes
pg_stat_activity	No
pg_stat_replication	No
pg_stat_replication_slots	No
pg_stat_wal_receiver	No
pg_stat_recovery_prefetch	No
pg_stat_subscription	No
pg_stat_subscription_stats	No
pg_stat_ssl	Yes
pg_stat_gssapi	No
pg_stat_archiver	No
pg_stat_io	No
pg_stat_bgwriter	No
pg_stat_wal	No
pg_stat_database	No
pg_stat_database_conflicts	No
pg_stat_all_tables	No
pg_stat_all_indexes	No
pg_statio_all_tables	No

Name	Applicable to Aurora DSQL
pg_statio_all_indexes	No
pg_statio_all_sequences	No
pg_stat_slru	No
pg_statio_user_tables	No
pg_statio_user_sequences	No
pg_stat_user_functions	No
pg_stat_user_indexes	No
pg_stat_progress_analyze	No
pg_stat_progress_basebackup	No
pg_stat_progress_cluster	No
pg_stat_progress_create_index	No
pg_stat_progress_vacuum	No
pg_stat_sys_indexes	No
pg_stat_sys_tables	No
pg_stat_xact_all_tables	No
pg_stat_xact_sys_tables	No
pg_stat_xact_user_functions	No
pg_stat_xact_user_tables	No
pg_statio_sys_indexes	No
pg_statio_sys_sequences	No

Name	Applicable to Aurora DSQL
pg_statio_sys_tables	No
pg_statio_user_indexes	No

Analyze

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the `pg_stats` system view. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries. In Aurora DSQL, you can't run the ANALYZE command within an explicit transaction. ANALYZE isn't subject to the database transaction timeout limit.

Programming with Aurora DSQL

Aside from using the AWS console, you can also use the AWS software development kits (SDK) and AWS CLI to interact with Aurora DSQL. For more information about the programmatic interfaces for Aurora DSQL, see the section called "Programmatic access".

Topics

Manage clusters in Aurora DSQL with the AWS SDKs

See the following sections to learn how to manage your clusters in Aurora DSQL with the AWS SDKs.

Topics

Create a cluster in Aurora DSQL in the AWS SDKs

See the following information to learn how to create a cluster in Aurora DSQL.

Python

To create a cluster in a single AWS Region, use the following example.

```
import boto3

def create_cluster(client, tags, deletion_protection):
    try:
        response = client.create_cluster(tags=tags,
 deletionProtectionEnabled=deletion_protection)
        return response
    except:
        print("Unable to create cluster")
        raise

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    tag = {"Name": "FooBar"}
    deletion_protection = True
    response = create_cluster(client, tags=tag,
 deletion_protection=deletion_protection)
    print("Cluster id: " + response['identifier'])

if __name__ == "__main__":
    main()
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
import boto3

def create_multi_region_clusters(client, linkedRegionList, witnessRegion,
 clusterProperties):
    try:
        response = client.create_multi_region_clusters(
            linkedRegionList=linkedRegionList,
            witnessRegion=witnessRegion,
            clusterProperties=clusterProperties,
        )
        return response
    except:
        print("Unable to create multi-region cluster")
        raise

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    linkedRegionList = ["us-east-1", "us-east-2"]
    witnessRegion = "us-west-2"
    clusterProperties = {
        "us-east-1": {"tags": {"Name": "Foo"}},
        "us-east-2": {"tags": {"Name": "Bar"}}
    }
    response = create_multi_region_clusters(client, linkedRegionList, witnessRegion,
 clusterProperties)
    print("Linked Cluster Arns:", response['linkedClusterArns'])

if __name__ == "__main__":
    main()
```

C++

The following example lets you create a cluster in a single AWS Region.

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/CreateClusterRequest.h>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

String createCluster(DSQLClient& client, bool deletionProtectionEnabled, const
 std::map<Aws::String, Aws::String>& tags){
    CreateClusterRequest request;
    request.SetDeletionProtectionEnabled(deletionProtectionEnabled);
    request.SetTags(tags);
    CreateClusterOutcome outcome = client.CreateCluster(request);

    const auto& clusterResult = outcome.GetResult().GetIdentifier();
    if (outcome.IsSuccess()) {
        std::cout << "Cluster Identifier: " << clusterResult << std::endl;
    } else {
        std::cerr << "Create operation failed: " << outcome.GetError().GetMessage()
 << std::endl;
    }
    return clusterResult;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);

    DSQLClientConfiguration clientConfig;
    clientConfig.region = "us-east-1";

    DSQLClient client(clientConfig);
    bool deletionProtectionEnabled = true;
    std::map<Aws::String, Aws::String> tags = {
        { "Name", "FooBar" }
    };
    createCluster(client, deletionProtectionEnabled, tags);
    Aws::ShutdownAPI(options);
    return 0;
}
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
#include <aws/core/client/DefaultRetryStrategy.h>
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/CreateMultiRegionClustersRequest.h>
#include <aws/dsql/model/LinkedClusterProperties.h>

#include <iostream>
#include <vector>
#include <map>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

std::vector<Aws::String> createMultiRegionCluster(DSQLClient& client, const
 std::vector<Aws::String>& linkedRegionList, const Aws::String& witnessRegion, const
 Aws::Map<Aws::String, LinkedClusterProperties>& clusterProperties) {
    CreateMultiRegionClustersRequest request;
    request.SetLinkedRegionList(linkedRegionList);
    request.SetWitnessRegion(witnessRegion);
    request.SetClusterProperties(clusterProperties);

    CreateMultiRegionClustersOutcome outcome =
 client.CreateMultiRegionClusters(request);

    if (outcome.IsSuccess()) {
        const auto& clusterArns = outcome.GetResult().GetLinkedClusterArns();
        return clusterArns;
    } else {
        std::cerr << "Create operation failed: " << outcome.GetError().GetMessage()
 << std::endl;
        return {};
    }
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;

    clientConfig.region = "us-east-1";
    clientConfig.retryStrategy =
 Aws::MakeShared<Aws::Client::DefaultRetryStrategy>("RetryStrategy", 10);
    DSQLClient client(clientConfig);

    std::vector<Aws::String> linkedRegionList = { "us-east-1", "us-east-2" };
    Aws::String witnessRegion = "us-west-2";

    LinkedClusterProperties usEast1Properties;
```

JavaScript

To create a cluster in a single AWS Region, use the following example.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { CreateClusterCommand } from "@aws-sdk/client-dsql";

async function createCluster(client, tags, deletionProtectionEnabled) {
    const createClusterCommand = new CreateClusterCommand({
        deletionProtectionEnabled: deletionProtectionEnabled,
        tags,
    });
    try {
        const response = await client.send(createClusterCommand);
        return response;
    } catch (error) {
        console.error("Failed to create cluster: ", error.message);
    }
}

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({ region });
    const tags = { Name: "FooBar"};
    const deletionProtectionEnabled = true;

    const response = await createCluster(client, tags, deletionProtectionEnabled);
    console.log("Cluster Id:", response.identifier);
}

main();
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { CreateMultiRegionClustersCommand } from "@aws-sdk/client-dsql";

async function createMultiRegionCluster(client, linkedRegionList, witnessRegion,
 clusterProperties) {
    const createMultiRegionClustersCommand = new CreateMultiRegionClustersCommand({
        linkedRegionList: linkedRegionList,
        witnessRegion: witnessRegion,
        clusterProperties: clusterProperties
    });
    try {
        const response = await client.send(createMultiRegionClustersCommand);
        return response;
    } catch (error) {
        console.error("Failed to create multi-region cluster: ", error.message);
    }
}

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({
        region
    });
    const linkedRegionList = ["us-east-1", "us-east-2"];
    const witnessRegion = "us-west-2";
    const clusterProperties = {
        "us-east-1": { tags: { "Name": "Foo" } },
        "us-east-2": { tags: { "Name": "Bar" } }
    };

    const response = await createMultiRegionCluster(client, linkedRegionList,
 witnessRegion, clusterProperties);
    console.log("Linked Cluster ARNs: ", response.linkedClusterArns);
}

main();
```

Java

Use the following example to create a cluster in a single AWS Region.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryMode;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.StandardRetryStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.ClusterStatus;
import software.amazon.awssdk.services.dsql.model.CreateClusterRequest;
import software.amazon.awssdk.services.dsql.model.CreateClusterResponse;

import java.net.URI;
import java.util.HashMap;
import java.util.Map;

public class CreateCluster {
    public static void main(String[] args) throws Exception {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        boolean deletionProtectionEnabled = true;
        Map<String, String> tags = new HashMap<>();
        tags.put("Name", "FooBar");

        String identifier = createCluster(region, client, deletionProtectionEnabled,
 tags);
        System.out.println("Cluster Id: " + identifier);
    }
    public static String createCluster(Region region, DsqlClient client, boolean
 deletionProtectionEnabled, Map<String, String> tags) throws Exception {
        CreateClusterRequest createClusterRequest = CreateClusterRequest
                .builder()
                .deletionProtectionEnabled(deletionProtectionEnabled)
                .tags(tags)
                .build();
        CreateClusterResponse res = client.createCluster(createClusterRequest);
        if (res.status() == ClusterStatus.CREATING) {
            return res.identifier();
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryMode;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.StandardRetryStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.CreateMultiRegionClustersRequest;
import software.amazon.awssdk.services.dsql.model.CreateMultiRegionClustersResponse;
import software.amazon.awssdk.services.dsql.model.LinkedClusterProperties;

import java.net.URI;
import java.util.Arrays;
import java.util.List;
import java.util.HashMap;
import java.util.Map;

public class CreateMultiRegionCluster {
    public static void main(String[] args) throws Exception {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        List<String> linkedRegionList = Arrays.asList(region.toString(), "us-
east-2");
        String witnessRegion = "us-west-2";
        Map<String, LinkedClusterProperties> clusterProperties = new HashMap<String,
 LinkedClusterProperties>() {{
            put("us-east-1", LinkedClusterProperties.builder()
                    .tags(new HashMap<String, String>() {{
                        put("Name", "Foo");
                    }})
                    .build());
            put("us-east-2", LinkedClusterProperties.builder()
                    .tags(new HashMap<String, String>() {{
                        put("Name", "Bar");
                    }})
                    .build());
        }};
```

Rust

Use the following example to create a cluster in a single AWS Region.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};
use std::collections::HashMap;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Create a cluster without delete protection and a name
pub async fn create_cluster(region: &'static str) -> (String, String) {
    let client = dsql_client(region).await;
    let tags = HashMap::from([
        (String::from("Name"), String::from("FooBar"))
    ]);

    let create_cluster_output = client
        .create_cluster()
        .set_tags(Some(tags))
        .deletion_protection_enabled(true)
        .send()
        .await
        .unwrap();

    // Response contains cluster identifier, its ARN, status etc.
    let identifier = create_cluster_output.identifier().to_owned();
    let arn = create_cluster_output.arn().to_owned();
    assert_eq!(create_cluster_output.status().as_str(), "CREATING");
    assert!(create_cluster_output.deletion_protection_enabled());
    (identifier, arn)
}

#[tokio::main(flavor = "current_thread")]
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};
use aws_sdk_dsql::types::LinkedClusterProperties;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

/// Create a multi-region cluster
pub async fn create_multi_region_cluster(region: &'static str) -> Vec<String> {
    let client = dsql_client(region).await;
    let us_east_1_props = LinkedClusterProperties::builder()
        .deletion_protection_enabled(false)
        .tags("Name", "Foo")
        .tags("Usecase", "testing-mr-use1")
        .build();

    let us_east_2_props = LinkedClusterProperties::builder()
        .deletion_protection_enabled(false)
        .tags(String::from("Name"), String::from("Bar"))
        .tags(String::from("Usecase"), String::from("testing-mr-use2"))
        .build();

    let create_mr_cluster_output = client
        .create_multi_region_clusters()
        .linked_region_list("us-east-1")
        .linked_region_list("us-east-2")
        .witness_region("us-west-2")
        .cluster_properties("us-east-1", us_east_1_props)
        .cluster_properties("us-east-2", us_east_2_props)
        .send()
        .await
```

Ruby

Use the following example to create a cluster in a single AWS Region.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def create_cluster(region)
  begin
    # Create client with default configuration and credentials
    client = Aws::DSQL::Client.new(region: region)

    response = client.create_cluster(
      deletion_protection_enabled: true,
      tags: {
        "Name" => "example_cluster_ruby"
        }
    )

    # Extract and verify response data
    identifier = response.identifier
    arn = response.arn
    puts arn
    raise "Unexpected status when creating cluster: #{response.status}" unless
 response.status == 'CREATING'
    raise "Deletion protection not enabled" unless
 response.deletion_protection_enabled

    [identifier, arn]
  rescue Aws::Errors::ServiceError => e
    raise "Failed to create cluster: #{e.message}"
  end
end
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def create_multi_region_cluster(region)
  us_east_1_props = {
    deletion_protection_enabled: false,
    tags: {
      'Name' => 'Foo',
      'Usecase' => 'testing-mr-use1'
    }
  }

  us_east_2_props = {
    deletion_protection_enabled: false,
    tags: {
      'Name' => 'Bar',
      'Usecase' => 'testing-mr-use2'
    }
  }

  begin
    # Create client with default configuration and credentials
    client = Aws::DSQL::Client.new(region: region)
    response = client.create_multi_region_clusters(
      linked_region_list: ['us-east-1', 'us-east-2'],
      witness_region: 'us-west-2',
      cluster_properties: {
        'us-east-1' => us_east_1_props,
        'us-east-2' => us_east_2_props
      }
    )

    # Extract cluster ARNs from the response
    arns = response.linked_cluster_arns
    raise "Expected 2 cluster ARNs, got #{arns.length}" unless arns.length == 2

    arns
  rescue Aws::Errors::ServiceError => e
    raise "Failed to create multi-region clusters: #{e.message}"
  end
end
```

.NET

Use the following example to create a cluster in a single AWS Region.

```
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class SingleRegionClusterCreation {
    public static async Task<CreateClusterResponse> Create(RegionEndpoint region)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);

        // Create a single region cluster
        CreateClusterRequest createClusterRequest = new()
        {
            DeletionProtectionEnabled = true
        };

        CreateClusterResponse createClusterResponse = await
 client.CreateClusterAsync(createClusterRequest);

        Console.WriteLine(createClusterResponse.Identifier);
        Console.WriteLine(createClusterResponse.Status);

        return createClusterResponse;
    }
}
```

To create a multi-Region cluster, use the following example. Creating a multi-Region cluster might take some time.

```
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class MultiRegionClusterCreation {
    public static async Task<CreateMultiRegionClustersResponse>
 Create(RegionEndpoint region)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);


        // Create multi region cluster
        LinkedClusterProperties USEast1Props = new() {
            DeletionProtectionEnabled = false,
            Tags = new Dictionary<string, string>
            {
                { "Name", "Foo" },
                { "Usecase", "testing-mr-use1" }
            }
        };

        LinkedClusterProperties USEast2Props = new() {
            DeletionProtectionEnabled = false,
            Tags = new Dictionary<string, string>
            {
                { "Name", "Bar" },
                { "Usecase", "testing-mr-use2" }
            }
        };

        CreateMultiRegionClustersRequest createMultiRegionClustersRequest = new()
        {
            LinkedRegionList = new List<string> { "us-east-1", "us-east-2" },
            WitnessRegion = "us-west-2",
            ClusterProperties = new Dictionary<string, LinkedClusterProperties>
            {
                { "us-east-1", USEast1Props },
                { "us-east-2", USEast2Props }
            }
        };
```

Get a cluster in Aurora DSQL with the AWS SDKs

See the following information to learn how to return information a a cluster in Aurora DSQL.

Python

To get information about a single or a multi-Region cluster, use the following example.

```
import boto3

def get_cluster(cluster_id, client):
    try:
        return client.get_cluster(identifier=cluster_id)
    except:
        print("Unable to get cluster")
        raise

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    cluster_id = "foo0bar1baz2quux3quuux4"
    response = get_cluster(cluster_id, client)
    print("Cluster Status: " + response['status'])


if __name__ == "__main__":
    main()
```

C++

Use the following example to get information about a single or a multi-Region cluster.

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/GetClusterRequest.h>
#include <aws/dsql/model/ClusterStatus.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

ClusterStatus getCluster(const String& clusterId, DSQLClient& client) {
    GetClusterRequest request;
    request.SetIdentifier(clusterId);
    GetClusterOutcome outcome = client.GetCluster(request);
    ClusterStatus status = ClusterStatus::NOT_SET;

    if (outcome.IsSuccess()) {
        const auto& cluster = outcome.GetResult();
        status = cluster.GetStatus();
    } else {
        std::cerr << "Get operation failed: " << outcome.GetError().GetMessage() <<
 std::endl;
    }
    std::cout << "Cluster Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(status) << std::endl;
    return status;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;

    clientConfig.region = "us-east-1";

    DSQLClient client(clientConfig);
    String clusterId = "foo0bar1baz2quux3quuux4";

    getCluster(clusterId, client);
    Aws::ShutdownAPI(options);
    return 0;
}
```

JavaScript

To get information about a single or multi-Region cluster, use the following example.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { GetClusterCommand } from "@aws-sdk/client-dsql";

async function getCluster(clusterId, client) {
    const getClusterCommand = new GetClusterCommand({
      identifier: clusterId,
    });

    try {
      return await client.send(getClusterCommand);
    } catch (error) {
      if (error.name === "ResourceNotFoundException") {
        console.log("Cluster ID not found or deleted");
      } else {
        console.error("Unable to poll cluster status:", error.message);
      }
      throw error;
    }
  }

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({ region });

    const clusterId = "foo0bar1baz2quux3quuux4";

    const response = await getCluster(clusterId, client);
    console.log("Cluster Status:", response.status);

}

main()
```

Java

The following example lets you get information about a single or multi-Region cluster.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryMode;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.StandardRetryStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.GetClusterRequest;
import software.amazon.awssdk.services.dsql.model.GetClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

import java.net.URI;

public class GetCluster {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        String cluster_id = "foo0bar1baz2quux3quuux4";

        GetClusterResponse response = getCluster(cluster_id, client);
        System.out.println("cluster status: " + response.status());
    }

    public static GetClusterResponse getCluster(String cluster_id, DsqlClient
 client) {
        GetClusterRequest getClusterRequest = GetClusterRequest.builder()
                .identifier(cluster_id)
                .build();
        try {
            return client.getCluster(getClusterRequest);
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("Cluster id is not found / deleted");
            throw rnfe;
        } catch (Exception e) {
            System.out.println(("Unable to poll cluster status: " +
 e.getMessage()));
            throw e;
```

Rust

The following example lets you get information about a single or multi-Region cluster.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};
use aws_sdk_dsql::operation::get_cluster::GetClusterOutput;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

// Get a ClusterResource from DSQL cluster identifier
pub async fn get_cluster(
    region: &'static str,
    identifier: String,
) -> GetClusterOutput {
    let client = dsql_client(region).await;
    client
        .get_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap()
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";

    get_cluster(region, "<your cluster id>".to_owned()).await;

    Ok(())
}
```

Ruby

The following example lets you get information about a single or multi-Region cluster.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def get_cluster(region, identifier)
  begin
    # Create client with default configuration and credentials
    client = Aws::DSQL::Client.new(region: region)
    client.get_cluster(
        identifier: identifier
    )
  rescue Aws::Errors::ServiceError => e
    raise "Failed to get cluster details: #{e.message}"
  end
end
```

.NET

The following example lets you get information about a single or multi-Region cluster.

```
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class GetCluster {
    public static async Task<GetClusterResponse> Get(RegionEndpoint region, string
 clusterId)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);

        // Get cluster details
        GetClusterRequest getClusterRequest = new()
        {
            Identifier = clusterId
        };

        // Assert that operation is successful
        GetClusterResponse getClusterResponse = await
 client.GetClusterAsync(getClusterRequest);
        Console.WriteLine(getClusterResponse.Status);

        return getClusterResponse;
    }
}
```

Update a cluster in Aurora DSQL with the AWS SDKs

See the following information to learn how to update a cluster in Aurora DSQL. Updating a cluster can take a minute or two. We recommend that you wait some time and then run get cluster to get the status of the cluster.

Python

To update a single or multi-Region cluster, use the following example.

```
import boto3

def update_cluster(cluster_id, deletionProtectionEnabled, client):
    try:
        return client.update_cluster(identifier=cluster_id,
 deletionProtectionEnabled=deletionProtectionEnabled)
    except:
        print("Unable to update cluster")
        raise

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    cluster_id = "foo0bar1baz2quux3quuux4"
    deletionProtectionEnabled = True
    response = update_cluster(cluster_id, deletionProtectionEnabled, client)
    print("Deletion Protection Updating to: " + str(deletionProtectionEnabled) +  ",
 Cluster Status: " + response['status'])

if __name__ == "__main__":
    main()
```

C++

Use the following example to update a single or multi-Region cluster.

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/UpdateClusterRequest.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

ClusterStatus updateCluster(const String& clusterId, bool deletionProtection,
 DSQLClient& client) {
    UpdateClusterRequest request;
    request.SetIdentifier(clusterId);
    request.SetDeletionProtectionEnabled(deletionProtection);
    UpdateClusterOutcome outcome = client.UpdateCluster(request);
    ClusterStatus status = ClusterStatus::NOT_SET;

    if (outcome.IsSuccess()) {
        const auto& cluster = outcome.GetResult();
        status = cluster.GetStatus();
    } else {
        std::cerr << "Update operation failed: " << outcome.GetError().GetMessage()
 << std::endl;
    }

    std::cout << "Cluster Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(status) << std::endl;
    return status;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;

    clientConfig.region = "us-east-1";

    DSQLClient client(clientConfig);

    String clusterId = "foo0bar1baz2quux3quuux4";
    bool deletionProtection = true;

    updateCluster(clusterId, deletionProtection, client);
    Aws::ShutdownAPI(options);

    return 0;
}
```

JavaScript

To update a single or multi-Region cluster, use the following example.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { UpdateClusterCommand } from "@aws-sdk/client-dsql";

async function updateCluster(clusterId, deletionProtectionEnabled, client) {
    const updateClusterCommand = new UpdateClusterCommand({
      identifier: clusterId,
      deletionProtectionEnabled: deletionProtectionEnabled
    });

    try {
        return await client.send(updateClusterCommand);
    } catch (error) {
        console.error("Unable to update cluster", error.message);
        throw error;
    }
  }

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({ region });

    const clusterId = "foo0bar1baz2quux3quuux4";
    const deletionProtectionEnabled = true;

    const response = await updateCluster(clusterId, deletionProtectionEnabled,
 client);
    console.log("Updating deletion protection: " + deletionProtectionEnabled + "-
 Cluster Status: " + response.status);

}

main();
```

Java

Use the following example to update a single or a multi-Region cluster.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryMode;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.StandardRetryStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.UpdateClusterRequest;
import software.amazon.awssdk.services.dsql.model.UpdateClusterResponse;

import java.net.URI;

public class UpdateCluster {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        String cluster_id = "foo0bar1baz2quux3quuux4";
        Boolean deletionProtectionEnabled = false;

        UpdateClusterResponse response = updateCluster(cluster_id,
 deletionProtectionEnabled, client);
        System.out.println("Deletion Protection updating to: " +
 deletionProtectionEnabled.toString() + ", Status: " + response.status());
    }

    public static UpdateClusterResponse updateCluster(String cluster_id, boolean
 deletionProtectionEnabled, DsqlClient client){
        UpdateClusterRequest updateClusterRequest = UpdateClusterRequest.builder()
                .identifier(cluster_id)
                .deletionProtectionEnabled(deletionProtectionEnabled)
                .build();
        try {
            return client.updateCluster(updateClusterRequest);
        } catch (Exception e) {
            System.out.println(("Unable to update deletion protection: " +
 e.getMessage()));
            throw e;
```

Rust

Use the following example to update a single or a multi-Region cluster.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};
use aws_sdk_dsql::operation::update_cluster::UpdateClusterOutput;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

// Update a DSQL cluster and set delete protection to false. Also add new tags.
pub async fn update_cluster(region: &'static str, identifier: String) ->
 UpdateClusterOutput {
    let client = dsql_client(region).await;
    // Update delete protection
    let update_response = client
        .update_cluster()
        .identifier(identifier)
        .deletion_protection_enabled(false)
        .send()
        .await
        .unwrap();

    // Add new tags
    client
        .tag_resource()
        .resource_arn(update_response.arn().to_owned())
        .tags(String::from("Function"), String::from("Billing"))
        .tags(String::from("Environment"), String::from("Production"))
        .send()
        .await
        .unwrap();

    update_response
```

Ruby

Use the following example to update a single or a multi-Region cluster.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def update_cluster(region, identifier)
    begin
      # Create client with default configuration and credentials
      client = Aws::DSQL::Client.new(region: region)

      update_response = client.update_cluster(
          identifier: identifier,
          deletion_protection_enabled: false
      )

      client.tag_resource(
          resource_arn: update_response.arn,
          tags: {
              "Function" => "Billing",
              "Environment" => "Production"
          }
      )
      raise "Unexpected status when updating cluster: #{update_response.status}"
 unless update_response.status == 'UPDATING'
      update_response
    rescue Aws::Errors::ServiceError => e
      raise "Failed to update cluster details: #{e.message}"
    end
end
```

.NET

Use the following example to update a single or a multi-Region cluster.

```
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class UpdateCluster {
    public static async Task Update(RegionEndpoint region, string clusterId)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);

        // Update cluster details by setting delete protection to false
        UpdateClusterRequest updateClusterRequest = new UpdateClusterRequest()
        {
            Identifier = clusterId,
            DeletionProtectionEnabled = false
        };

        await client.UpdateClusterAsync(updateClusterRequest);
    }
}
```

Delete cluster in Aurora DSQL with AWS SDKs

See the following information to learn how to delete a cluster in Aurora DSQL.

Python

To delete a cluster in a single AWS Region, use the following example.

```
import boto3

def delete_cluster(cluster_id, client):
    try:
        return client.delete_cluster(identifier=cluster_id)
    except:
        print("Unable to delete cluster " + cluster_id)
        raise

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    cluster_id = "foo0bar1baz2quux3quuux4"
    response = delete_cluster(cluster_id, client)
    print("Deleting cluster with ID: " + cluster_id +  ", Cluster Status: " +
 response['status'])

if __name__ == "__main__":
    main()
```

To delete a multi-Region cluster, use the following example.

```
import boto3

def delete_multi_region_clusters(linkedClusterArns, client):
    client.delete_multi_region_clusters(linkedClusterArns=linkedClusterArns)

def main():
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)
    linkedClusterArns = [
        "arn:aws:dsql:us-east-1:111111999999::cluster/foo0bar1baz2quux3quuux4",
        "arn:aws:dsql:us-east-2:111111999999::cluster/bar0foo1baz2quux3quuux4"
    ]
    delete_multi_region_clusters(linkedClusterArns, client)
    print("Deleting clusters with ARNs:", linkedClusterArns)

if __name__ == "__main__":
    main()
```

C++

The following example lets you delete a cluster in a single AWS Region.

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/DeleteClusterRequest.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

ClusterStatus deleteCluster(const String& clusterId, DSQLClient& client) {
    DeleteClusterRequest request;
    request.SetIdentifier(clusterId);

    DeleteClusterOutcome outcome = client.DeleteCluster(request);
    ClusterStatus status = ClusterStatus::NOT_SET;

    if (outcome.IsSuccess()) {
        const auto& cluster = outcome.GetResult();
        status = cluster.GetStatus();
    } else {
        std::cerr << "Delete operation failed: " << outcome.GetError().GetMessage()
 << std::endl;
    }
    std::cout << "Cluster Status: " <<
 ClusterStatusMapper::GetNameForClusterStatus(status) << std::endl;
    return status;
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;

    clientConfig.region = "us-east-1";

    DSQLClient client(clientConfig);
    String clusterId = "foo0bar1baz2quux3quuux4";

    deleteCluster(clusterId, client);
    Aws::ShutdownAPI(options);
    return 0;
}
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <aws/dsql/model/DeleteMultiRegionClustersRequest.h>

#include <iostream>
#include <vector>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

std::vector<Aws::String> deleteMultiRegionClusters(const std::vector<Aws::String>&
 linkedClusterArns, DSQLClient& client) {
    DeleteMultiRegionClustersRequest request;
    request.SetLinkedClusterArns(linkedClusterArns);

    DeleteMultiRegionClustersOutcome outcome =
 client.DeleteMultiRegionClusters(request);

    if (outcome.IsSuccess()) {
        std::cout << "Successfully deleted clusters." << std::endl;
        return linkedClusterArns;
    } else {
        std::cerr << "Delete operation failed: " << outcome.GetError().GetMessage()
 << std::endl;
        return {};
    }
}

int main() {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;

    clientConfig.region = "us-east-1";

    DSQLClient client(clientConfig);

    std::vector<Aws::String> linkedClusterArns = {
        "arn:aws:dsql:us-east-1:111111999999::cluster/foo0bar1baz2quux3quuux4",
        "arn:aws:dsql:us-east-2:111111999999::cluster/bar0foo1baz2quux3quuux4"
    };

    std::vector<Aws::String> deletedArns =
 deleteMultiRegionClusters(linkedClusterArns, client);
```



```
    if (!deletedArns.empty()) {
        std::cout << "Deleted Cluster ARNs: " << std::endl;
        for (const auto& arn : deletedArns) {
```

JavaScript

To delete a cluster in a single AWS Region, use the following example.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { DeleteClusterCommand } from "@aws-sdk/client-dsql";

async function deleteCluster(clusterId, client) {
    const deleteClusterCommand = new DeleteClusterCommand({
      identifier: clusterId,
    });

    try {
      const response = await client.send(deleteClusterCommand);
      return response;
    } catch (error) {
      if (error.name === "ResourceNotFoundException") {
        console.log("Cluster ID not found or already deleted");
      } else {
        console.error("Unable to delete cluster: ", error.message);
      }
      throw error;
    }
  }

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({ region });

    const clusterId = "foo0bar1baz2quux3quuux4";

    const response = await deleteCluster(clusterId, client);
    console.log("Deleting Cluster with Id:", clusterId, "- Cluster Status:",
 response.status);

}

main();
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
import { DSQLClient } from "@aws-sdk/client-dsql";
import { DeleteMultiRegionClustersCommand } from "@aws-sdk/client-dsql";

async function deleteMultiRegionClusters(linkedClusterArns, client) {
    const deleteMultiRegionClustersCommand = new DeleteMultiRegionClustersCommand({
        linkedClusterArns: linkedClusterArns,
    });
    try {
        const response = await client.send(deleteMultiRegionClustersCommand);
        return response;
    } catch (error) {
        if (error.name === "ResourceNotFoundException") {
            console.log("Some or all Cluster ARNs not found or already deleted");
        } else {
            console.error("Unable to delete multi-region clusters: ",
 error.message);
        }
        throw error;
    }
}

async function main() {
    const region = "us-east-1";
    const client = new DSQLClient({ region });
    const linkedClusterArns = [
        "arn:aws:dsql:us-east-1:111111999999::cluster/foo0bar1baz2quux3quuux4",
        "arn:aws:dsql:us-east-2:111111999999::cluster/bar0foo1baz2quux3quuux4"
    ];

    const response = await deleteMultiRegionClusters(linkedClusterArns, client);
    console.log("Deleting Clusters with ARNs:", linkedClusterArns);
}

main();
```

Java

To delete a cluster in a single AWS Region, use the following example.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryMode;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.retries.StandardRetryStrategy;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteClusterRequest;
import software.amazon.awssdk.services.dsql.model.DeleteClusterResponse;
import software.amazon.awssdk.services.dsql.model.ResourceNotFoundException;

import java.net.URI;

public class DeleteCluster {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        String cluster_id = "foo0bar1baz2quux3quuux4";

        DeleteClusterResponse response = deleteCluster(cluster_id, client);
        System.out.println("Deleting Cluster with ID: " + cluster_id + ", Status: "
 + response.status());
    }

    public static DeleteClusterResponse deleteCluster(String cluster_id, DsqlClient
 client) {
        DeleteClusterRequest deleteClusterRequest = DeleteClusterRequest.builder()
                .identifier(cluster_id)
                .build();
        try {
            return client.deleteCluster(deleteClusterRequest);
        } catch (ResourceNotFoundException rnfe) {
            System.out.println("Cluster id is not found / deleted");
            throw rnfe;
        } catch (Exception e) {
            System.out.println("Unable to poll cluster status: " + e.getMessage());
            throw e;
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration;
import software.amazon.awssdk.core.retry.RetryPolicy;
import software.amazon.awssdk.http.urlconnection.UrlConnectionHttpClient;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlClient;
import software.amazon.awssdk.services.dsql.model.DeleteMultiRegionClustersRequest;
import software.amazon.awssdk.services.dsql.model.DeleteMultiRegionClustersResponse;

import java.net.URI;
import java.util.Arrays;
import java.util.List;

public class DeleteMultiRegionClusters {
    public static void main(String[] args) {
        Region region = Region.US_EAST_1;

        ClientOverrideConfiguration clientOverrideConfiguration =
 ClientOverrideConfiguration.builder()
                .retryStrategy(StandardRetryStrategy.builder().build())
                .build();

        DsqlClient client = DsqlClient.builder()
                .httpClient(UrlConnectionHttpClient.create())
                .overrideConfiguration(clientOverrideConfiguration)
                .region(region)
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        List<String> linkedClusterArns = Arrays.asList(
                "arn:aws:dsql:us-east-1:111111999999::cluster/
foo0bar1baz2quux3quuux4",
                "arn:aws:dsql:us-east-2:111111999999::cluster/
bar0foo1baz2quux3quuux4"
        );

        deleteMultiRegionClusters(linkedClusterArns, client);
        System.out.println("Deleting Clusters with ARNs: " + linkedClusterArns);
    }
    public static void deleteMultiRegionClusters(List<String> linkedClusterArns,
 DsqlClient client) {
        DeleteMultiRegionClustersRequest deleteMultiRegionClustersRequest =
 DeleteMultiRegionClustersRequest.builder()
                .linkedClusterArns(linkedClusterArns)
                .build();
```



```
        try {
            client.deleteMultiRegionClusters(deleteMultiRegionClustersRequest);
        } catch (Exception e) {
```

Rust

To delete a cluster in a single AWS Region, use the following example.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

// Delete a DSQL cluster
pub async fn delete_cluster(region: &'static str, identifier: String) {
    let client = dsql_client(region).await;
    let delete_response = client
        .delete_cluster()
        .identifier(identifier)
        .send()
        .await
        .unwrap();
    assert_eq!(delete_response.status().as_str(), "DELETING");
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";
    delete_cluster(region, "<cluster to be deleted>".to_owned()).await;
    Ok(())
}
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
use aws_config::load_defaults;
use aws_sdk_dsql::{config::{BehaviorVersion, Region}, Client, Config};
use aws_sdk_dsql::operation::RequestId;

/// Create a client. We will use this later for performing operations on the
 cluster.
async fn dsql_client(region: &'static str) -> Client {
    // Load default SDK configuration
    let sdk_defaults = load_defaults(BehaviorVersion::latest()).await;

    // You can set your own credentials by following this guide
    // https://docs.aws.amazon.com/sdk-for-rust/latest/dg/credproviders.html
    let credentials = sdk_defaults
        .credentials_provider()
        .unwrap();

    let config = Config::builder()
        .behavior_version(BehaviorVersion::latest())
        .credentials_provider(credentials)
        .region(Region::new(region))
        .build();

    Client::from_conf(config)
}

// Delete a Multi region DSQL cluster
pub async fn delete_multi_region_cluster(region: &'static str, arns: Vec<String>) {
    let client = dsql_client(region).await;
    let delete_response = client
        .delete_multi_region_clusters()
        .set_linked_cluster_arns(Some(arns))
        .send()
        .await
        .unwrap();
    assert!(delete_response.request_id().is_some());
}

#[tokio::main(flavor = "current_thread")]
pub async fn main() -> anyhow::Result<()> {
    let region = "us-east-1";
    let arns = vec![
        "<cluster arn from us-east-1>".to_owned(),
        "<cluster arn from us-east-2>".to_owned()
    ];
    delete_multi_region_cluster(region, arns).await;
    Ok(())
}
```

Ruby

To delete a cluster in a single AWS Region, use the following example.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def delete_cluster(region, identifier)
  begin
    # Create client with default configuration and credentials
    client = Aws::DSQL::Client.new(region: region)

    delete_response = client.delete_cluster(
        identifier: identifier
    )
    raise "Unexpected status when deleting cluster: #{delete_response.status}"
 unless delete_response.status == 'DELETING'
    delete_response
  rescue Aws::Errors::ServiceError => e
    raise "Failed to delete cluster: #{e.message}"
  end
end
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
require 'aws-sdk-core'
require 'aws-sdk-dsql'

def delete_multi_region_cluster(region, arns)
  begin
    # Create client with default configuration and credentials
    client = Aws::DSQL::Client.new(region: region)
    client.delete_multi_region_clusters(
      linked_cluster_arns: arns
    )
  rescue Aws::Errors::ServiceError => e
    raise "Failed to delete multi-region cluster: #{e.message}"
  end
end
```

.NET

To delete a cluster in a single AWS Region, use the following example.

```
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class SingleRegionClusterDeletion {
    public static async Task<DeleteClusterResponse> Delete(RegionEndpoint region,
 string clusterId)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);

        // Delete a single region cluster
        DeleteClusterRequest deleteClusterRequest = new()
        {
            Identifier = clusterId
        };
        DeleteClusterResponse deleteClusterResponse = await
 client.DeleteClusterAsync(deleteClusterRequest);
        Console.WriteLine(deleteClusterResponse.Status);

        return deleteClusterResponse;
    }
}
```

To delete a multi-Region cluster, use the following example. Deleting a multi-Region cluster might take some time.

```
using Amazon;
using Amazon.DSQL;
using Amazon.DSQL.Model;
using Amazon.Runtime;

class MultiRegionClusterDeletion {
    public static async Task Delete(RegionEndpoint region, List<string> arns)
    {
        // Create the sdk client
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();
        AmazonDSQLConfig clientConfig = new()
        {
            AuthenticationServiceName = "dsql",
            RegionEndpoint = region
        };
        AmazonDSQLClient client = new(awsCredentials, clientConfig);

        // Delete a multi region clusters
        DeleteMultiRegionClustersRequest deleteMultiRegionClustersRequest = new()
        {
            LinkedClusterArns = arns
        };
        DeleteMultiRegionClustersResponse deleteMultiRegionClustersResponse =
            await
 client.DeleteMultiRegionClustersAsync(deleteMultiRegionClustersRequest);

 Console.WriteLine(deleteMultiRegionClustersResponse.ResponseMetadata.RequestId);
    }
}
```

Manage clusters in Aurora DSQL with the AWS CLI

See the following sections to learn how to manage your clusters with the AWS CLI.

CreateCluster

To create a cluster, use the `create-cluster` command.

> **Note**
>
> Cluster creation happens asynchronously. Call the `GetCluster` API until the status is `ACTIVE`. You can connect to a cluster once it becomes `ACTIVE`.

Sample command

```
aws dsql create-cluster --region us-east-1
```

> **Note**
>
> If you want to disable deletion protection upon creation, include the `--no-deletion-protection-enabled` flag.

Sample response

```
{
 "identifier": "foo0bar1baz2quux3quuux4",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
 "status": "CREATING",
 "creationTime": "2024-05-25T16:56:49.784000-07:00",
 "deletionProtectionEnabled": true
}
```

GetCluster

To describe an cluster, use the `get-cluster` command.

Sample command

```
aws dsql get-cluster \
--region us-east-1 \
--identifier <your_cluster_id>
```

Sample response

```
{
```

```
 "identifier": "foo0bar1baz2quux3quuux4",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
 "status": "ACTIVE",
 "creationTime": "2024-05-24T09:15:32.708000-07:00",
 "deletionProtectionEnabled": false
}
```

UpdateCluster

To update an existing cluster, use the `update-cluster` command.

> **Note**
>
> Updates happen asynchronously. Call the `GetCluster` API until the status is `ACTIVE` and you'll observe the changes.

Sample command

```
aws dsql update-cluster \
--region us-east-1 \
--no-deletion-protection-enabled \
--identifier your_cluster_id
```

Sample response

```
{
 "identifier": "foo0bar1baz2quux3quuux4",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
 "status": "UPDATING",
 "creationTime": "2024-05-24T09:15:32.708000-07:00",
 "deletionProtectionEnabled": true
}
```

DeleteCluster

To delete an existing cluster, use the `delete-cluster` command.

> **Note**
>
> You can only delete a cluster which has deletion protection disabled. Deletion protection is enabled by default when creating new clusters.

Sample command

```
aws dsql delete-cluster \
--region us-east-1 \
--identifier your_cluster_id
```

Sample response

```
{
 "identifier": "foo0bar1baz2quux3quuux4",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
 "status": "DELETING",
 "creationTime": "2024-05-24T09:16:43.778000-07:00",
 "deletionProtectionEnabled": false
}
```

ListClusters

To get the a of clusters, use the `list-clusters` command.

Sample command

```
aws dsql list-clusters --region us-east-1
```

Sample response

```
{
 "clusters": [
 {
 "identifier": "foo0bar1baz2quux3quux4quuux",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quux4quuux"
```

```
 },
 {
 "identifier": "foo0bar1baz2quux3quux4quuuux",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quux4quuuux"
 },
 {
 "identifier": "foo0bar1baz2quux3quux4quuuuux",
 "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quux4quuuuux"
 }
 ]
}
```

CreateMultiRegionClusters

To create multi-Region linked clusters, use the `create-multi-region-clusters` command. You can issue the command from either Read-Write region in the linked cluster pair.

Sample command

```
aws dsql create-multi-region-clusters \
    --region us-east-1 \
    --linked-region-list us-east-1 us-east-2 \
    --witness-region us-west-2 \
    --client-token test-1
```

If the API operation succeeds, both linked clusters enter the `CREATING` state and cluster creation will proceed asynchronously. To monitor progress you can call the `GetCluster` API in each Region until the return status shows ACTIVE. You can connect to a cluster once both linked clusters become ACTIVE.

> **Note**
>
> During preview, if you encounter a scenario where one cluster is `ACTIVE` and other `FAILED`, we recommend you delete the linked clusters and create them again.

```
{
    "linkedClusterArns": [
```

```
        "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
        "arn:aws:dsql:us-east-2:111122223333:cluster/bar0foo1baz2quux3quuux4"
    ]
}
```

GetCluster on multi-Region clusters

To get information about a multi-Region cluster, use the `get-cluster` command. For multi-Region clusters the response will include the linked cluster ARNs.

Sample command

```
aws dsql get-cluster \
--region us-east-1 \
--identifier your_cluster_id
```

Sample response

```
{
    "identifier": "aaabtjp7shql6wz7w5xqzpxtem",
    "arn": "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
    "status": "ACTIVE",
    "creationTime": "2024-07-17T10:24:23.325000-07:00",
    "deletionProtectionEnabled": true,
    "witnessRegion": "us-west-2",
    "linkedClusterArns": [
        "arn:aws:dsql:us-east-1:111122223333:cluster/foo0bar1baz2quux3quuux4",
        "arn:aws:dsql:us-east-2:111122223333:cluster/bar0foo1baz2quux3quuux4"
    ]
}
```

DeleteMultiRegionClusters

To delete multi-Region clusters, use the `delete-multi-region-clusters` operation from any of the linked cluster Regions.

Note that you can't delete only one Region of a linked cluster pair.

Sample AWS CLI command

```
aws dsql delete-multi-region-clusters \
 --region us-east-1 --linked-cluster-arns "arn:aws:dsql:us-east-2:111122223333:cluster/
bar0foo1baz2quux3quuux4" "arn:aws:dsql:us-east-1:111122223333:cluster/
foo0bar1baz2quux3quuux4"
```

If this API operation succeeds, both clusters enter the `DELETING` state. To determine the exact status of the clusters, use the `get-cluster` API operation on each linked cluster in their corresponding Region.

Sample response

```
{ }
```

Programming with Python

Topics

- Using Aurora DSQL to build an application with Django
- Using Aurora DSQL to build an application with SQLAlchemy
- Using Psycopg2 to interact with Aurora DSQL
- Using Psycopg3 to interact with Aurora DSQL

Using Aurora DSQL to build an application with Django

This section describes how how to create a pet clinic web application with Django that uses Aurora DSQL as a database. This clinic has pets, owners, veterinarians, and specialties

Before you begin, make sure that you have created a cluster in Aurora DSQL. You need the cluster endpoint to build the web application. You must also have installed Python 3.8 or higher and latest AWS SDK for Python (Boto3)

Bootstrap the Django application

1. Create a new directory named `django_aurora_dsql_example`.

```
mkdir django_aurora_dsql_example
cd django_aurora_dsql_example
```

2. Install Django and other dependencies. Create a file named `requirements.txt` and add in the following contents.

```
boto3
botocore
aurora_dsql_django
django
psycopg[binary]
```

3. Use the following commands to create and activate a Python virtual environment.

```
python3 -m venv venv
source venv/bin/activate
```

4. Install the requirements that you defined.

```
pip install --force-reinstall -r requirements.txt
```

5. Verify that you have installed Django. You should see the version of Django that you insalled.

```
python3 -m django --version
```

```
5.1.2 # Your version could be different
```

6. Create a Django project and change your directory to that location.

```
django-admin startproject project
cd project
```

7. Create an application named `pet_clinic`.

```
python3 manage.py startapp pet_clinic
```

8. Django comes installed with default authentication and admin apps, but they don't work with Aurora DSQL. Find the variables in `django_aurora_dsql_example/project/project/settings.py` and set the values like below.

```
ALLOWED_HOSTS = ['*']
```

```
INSTALLED_APPS = ['pet_clinic'] # Make sure that you have the pet_clinic app
 defined here.
MIDDLEWARE = []
TEMPLATES = [
   {
      'BACKEND': 'django.template.backends.django.DjangoTemplates',
      'DIRS': [],
      'APP_DIRS': True,
      'OPTIONS': {
         'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
         ],
      },
   },
]
```

9. Remove the references to the `admin` application in the Django project. From `django_aurora_dsql_example/project/project/urls.py`, remove the path to the admin page.

```
# remove the following line
from django.contrib import admin

# make sure that urlpatterns variable is empty
urlpatterns = []
```

From `django_aurora_dsql_example/project/pet_clinic`, delete the `admin.py` file.

10. Change the database settings so that the application uses the Aurora DSQL cluster instead of the default of SQLite 3.

```
DATABASES = {
   'default': {
      # Provide the endpoint of the cluster
      'HOST': <cluster endpoint>,
      'USER': 'admin',
      'NAME': 'postgres',
      'ENGINE': 'aurora_dsql_django', # This is the custom database adapter for
 Aurora DSQL
      'OPTIONS': {
            'sslmode': 'require',
            'region': 'us-east-2',
```

```
            # Setting password token expirty time is optional. Default is 900s
            'expires_in': 30
            # Setting `aws_profile` name is optional. Default is `default` profile
            # Setting `sslrootcert` is needed if you set 'sslmode': 'verify-full'
        }
    }
}
```

Create the application

Now that you've bootstrapped the Django pet clinic application, you can add models, create views, and run the server.

> **Important**
>
> To run the code, you must have valid AWS credentials.

Create models

As a pet clinic, it needs to account for pets, owners of pets, and veterinarians and their specialties. An owner can visit the veterinarian in the clinic with the pet. The clinic has the following relationships.

- One owner can have many pets.
- A veterinarian can have any number of specialties, and one specialty can be associated with any number of veternarians.

> **Note**
>
> Aurora DSQL doesn't support automatically incrementing the SERIAL type primary key. In these examples, we instead use a UUIDField with a default uuid value as the primary key.

```
from django.db import models
import uuid

# Create your models here.
```

```
class Owner(models.Model):
    # SERIAL Auto incrementing primary keys are not supported. Using UUID instead.
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False
    )
    name = models.CharField(max_length=30, blank=False)
    # This is many to one relation
    city = models.CharField(max_length=80, blank=False)
    telephone = models.CharField(max_length=20, blank=True, null=True, default=None)

    def __str__(self):
        return f'{self.name}'

class Pet(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False
    )
    name = models.CharField(max_length=30, blank=False)
    birth_date = models.DateField()
    owner = models.ForeignKey(Owner, on_delete=models.CASCADE, db_constraint=False,
 null=True)
```

Create the associated tables in your cluster by running the following commands in the `django_aurora_dsql_example/project` directory.

```
# This command generates a file named 0001_Initial.py in django_aurora_dsql_example/
project/pet_clinic directory
python3 manage.py makemigrations pet_clinic
python3 manage.py migrage pet_clinic 0001
```

Create views

Now that we have models and tables, we can create views for each model, and then run CRUD operations with each model.

Note that we do not want to give up upon error immediately. For example, the transaction may fail because of a Optimistic Concurrency Control (OCC) error. Instead of giving up immediately, we

can retry N times. In this example, we are attempting the operation 3 times by default. In order to achieve this a sample `with_retry` method is provided here.

```
from django.shortcuts import render, redirect
from django.views import generic
from django.views.generic import View
from django.http import JsonResponse, HttpResponse, HttpResponseBadRequest
from django.utils.decorators import method_decorator
from django.views.generic import View
from django.views.decorators.csrf import csrf_exempt
from django.db.transaction import atomic
from psycopg import errors
from django.db import Error, IntegrityError
import json, time, datetime

from pet_clinic.models import *

##
# If there is an error, we want to retry instead of giving up immediately.
# initial_wait is the amount of time after with the operation is retried
# delay_factor is the pace at which the retries slow down upon each failure.
# For example an initial_wait of 1 and delay_factor of 2 implies,
# First retry occurs after 1 second, second one after 1*2 = 2 seconds,
# Third one after 2*2 = 4 seconds, forth one after 4*2 = 8 seconds and so on.
##
def with_retries(retries = 3, failed_response = HttpResponse(status=500), initial_wait
 = 1, delay_factor = 2):
    def handle(view):
        def retry_fn(*args, **kwargs):
            delay = initial_wait
            for i in range(retries):
                print(("attempt: %s/%s") % (i+1, retries))
                try:
                    return view(*args, **kwargs)
                except Error as e:
                    print(f"Error: {e}, retrying...")
                    time.sleep(delay)
                    delay *= delay_factor
            return failed_response
        return retry_fn
    return handle

@method_decorator(csrf_exempt, name='dispatch')
```

```
class OwnerView(View):
    @with_retries()
    def get(self, request, id=None, *args, **kwargs):
        owners = Owner.objects
        # Apply filter if specific id is requested.
        if id is not None:
            owners = owners.filter(id=id)
        return JsonResponse(list(owners.values()), safe=False)

    @with_retries()
    @atomic
    def post(self, request, *args, **kwargs):
        data = json.loads(request.body.decode())

        # If id is provided we try updating the existing object
        id = data.get('id', None)
        try:
            owner = Owner.objects.get(id=id) if id is not None else None
        except:
            return HttpResponseBadRequest(("error: check if owner with id `%s` exists")
 % (id))

        name = data.get('name', owner.name if owner else None)
        # Either the name or id must be provided.
        if owner is None and name is None:
            return HttpResponseBadRequest()

        telephone = data.get('telephone', owner.telephone if owner else None)
        city = data.get('city', owner.city if owner else None)

        if owner is None:
            # Owner _not_ present, creating new one
            print(("owner: %s is not present; adding") % (name))
            owner = Owner(name=name, telephone=telephone, city=city)
        else:
            # Owner present, update existing
            print(("owner: %s is present; updating") % (name))
            owner.name = name
            owner.telephone = telephone
            owner.city = city

        owner.save()
        return JsonResponse(list(Owner.objects.filter(id=owner.id).values()),
 safe=False)
```

```
    @with_retries()
    @atomic
    def delete(self, request, id=None, *args, **kwargs):
        if id is not None:
            Owner.objects.filter(id=id).delete()
        return HttpResponse(status=200)

@method_decorator(csrf_exempt, name='dispatch')
class PetView(View):
    @with_retries()
    def get(self, request=None, id=None, *args, **kwargs):
        pets = Pet.objects
        # Apply filter if specific id is requested.
        if id is not None:
            pets = pets.filter(id=id)
        return JsonResponse(list(pets.values()), safe=False)

    @with_retries()
    @atomic
    def post(self, request, *args, **kwargs):
        data = json.loads(request.body.decode())

        # If id is provided we try updating the existing object
        id = data.get('id', None)
        try:
            pet = Pet.objects.get(id=id) if id is not None else None
        except:
            return HttpResponseBadRequest(("error: check if pet with id `%s` exists") %
 (id))

        name = data.get('name', pet.name if pet else None)
        # Either the name or id must be provided.
        if pet is None and name is None:
            return HttpResponseBadRequest()

        birth_date = data.get('birth_date', pet.birth_date if pet else None)
        owner_id = data.get('owner_id', pet.owner.id if pet and pet.owner else None)
        try:
            owner = Owner.objects.get(id=owner_id) if owner_id else None
        except:
            return HttpResponseBadRequest(("error: check if owner with id `%s` exists")
 % (owner_id))
```

```
        if pet is None:
            # Pet _not_ present, creating new one
            print(("pet name: %s is not present; adding") % (name))
            pet = Pet(name=name, birth_date=birth_date, owner=owner)
        else:
            # Pet present, update existing
            print(("pet name: %s is present; updating") % (name))
            pet.name = name
            pet.birth_date = birth_date
            pet.owner = owner

        pet.save()
        return JsonResponse(list(Pet.objects.filter(id=pet.id).values()), safe=False)

    @with_retries()
    @atomic
    def delete(self, request=None, id=None, *args, **kwargs):
        if id is not None:
            Pet.objects.filter(id=id).delete()
        return HttpResponse(status=200)
```

Create paths

We can then create paths so that we can run CRUD operations on the data.

```
from django.contrib import admin
from django.urls import path
from pet_clinic.views import *

urlpatterns = [
    path('owner/', OwnerView.as_view(), name='owner'),
    path('owner/<id>', OwnerView.as_view(), name='owner'),
    path('pet/', PetView.as_view(), name='pet'),
    path('pet/<id>', PetView.as_view(), name='pet'),
]
```

Finally, start the Django application by running the following command.

```
python3 manage.py runserver
```

CRUD operations

Test that your application works by testing the CRUD operations. The following examples demonstrate how to create Owner and Pet objects

```
curl --request POST --data '{"name":"Joe", "city":"Seattle"}' http://0.0.0.0:8000/
owner/
curl --request POST --data '{"name":"Mary", "telephone":"93209753297", "city":"New
 York"}' http://0.0.0.0:8000/owner/
curl --request POST --data '{"name":"Dennis", "city":"Chicago"}' http://0.0.0.0:8000/
owner/
```

```
curl --request POST --data '{"name":"Tom", "birth_date":"2006-10-25"}'
 http://0.0.0.0:8000/pet/
curl --request POST --data '{"name":"luna", "birth_date":"2020-10-10"}'
 http://0.0.0.0:8000/pet/
curl --request POST --data '{"name":"Myna", "birth_date":"2021-09-11"}'
 http://0.0.0.0:8000/pet/
```

Run the following commands to retrieve all of the owners and pets.

```
curl --request GET http://0.0.0.0:8000/owner/
```

```
curl --request GET http://0.0.0.0:8000/pet/
```

The following example demonstrates how to update a specific owner or pet.

```
curl --request POST --data '{"id":"44ca64ed-0264-450b-817b-14386c7df277",
 "city":"Vancouver"}' http://0.0.0.0:8000/owner/
```

```
curl --request POST --data '{"id":"f397b51b-2fdd-441d-b0ac-f115acd74725",
 "birth_date":"2016-09-11"}' http://0.0.0.0:8000/pet/
```

Finally, you can delete an owner or a pet.

```
curl --request DELETE http://0.0.0.0:8000/owner/44ca64ed-0264-450b-817b-14386c7df277
```

```
curl --request DELETE http://0.0.0.0:8000/pet/f397b51b-2fdd-441d-b0ac-f115acd74725
```

Relationships

One-to-many / Many-to-one

These relationships can be achieved by having the foreign key constraint on the field. For example, an owner can have any number of pets. A pet can have only one owner.

```
# An owner can adopt a pet
curl --request POST --data '{"id":"d52b4b69-b5f7-49a9-90af-adfdf10ecc03",
 "owner_id":"0f7cd839-c8ee-436e-baf3-e52aaa51fa65"}' http://0.0.0.0:8000/pet/

# Same owner can have another pet
curl --request POST --data '{"id":"485c8818-d7c1-4965-a024-0e133896c72d",
 "owner_id":"0f7cd839-c8ee-436e-baf3-e52aaa51fa65"}' http://0.0.0.0:8000/pet/

# Deleting the owner deletes pets as ForeignKey is configured with on_delete.CASCADE
curl --request DELETE http://0.0.0.0:8000/owner/0f7cd839-c8ee-436e-baf3-e52aaa51fa65

# Confirm that owner is deleted
curl --request GET http://0.0.0.0:8000/owner/12154d97-0f4c-4fed-b560-6578d46aff6d

# Confirm corresponding pets are deleted
curl --request GET http://0.0.0.0:8000/pet/d52b4b69-b5f7-49a9-90af-adfdf10ecc03
curl --request GET http://0.0.0.0:8000/pet/485c8818-d7c1-4965-a024-0e133896c72d
```

Many-to-Many

To illustrate Many-to-many we can imagine having a list of specialties and a list of veterinarian. A specialty can be attributed to any number of veterinarians and a veterinarian can have any number of specialties. In order to achieve this we will create ManyToMany mapping. As our primary keys are non integer UUIDs, we cannot directly use ManyToMany. We need to define a mapping via custom intermediate table with explicit UUID as the primary key.

One-to-One

To illustrate One-to-One let's imagine that Vet can also be a owner. This imposes one-to-one relationship between the Vet and the owner. Also, not all Vets are owners. We define this by having a OneToOne field named owner in the Vet model and flagging it can be blank or null but it must be unique.

Note

Django treats all AutoFields as integers internally. And Django automatically creates an intermediate table to manage many-to-many mapping with a Auto increment column as primary key. Aurora DSQL does not support this; we will create an intermediate table ourselves instead of letting Django do it automatically.

Define models

```
class Specialty(models.Model):
    name = models.CharField(max_length=80, blank=False, primary_key=True)
    def __str__(self):
        return self.name

class Vet(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False
    )
    name = models.CharField(max_length=30, blank=False)
    specialties = models.ManyToManyField(Specialty, through='VetSpecialties')
    owner = models.OneToOneField(Owner, on_delete=models.SET_DEFAULT,
 db_constraint=False, null=True, blank=True, default=None)
    def __str__(self):
        return f'{self.name}'

# Need to use custom intermediate table because Django considers default primary
# keys as integers. We use UUID as default primary key which is not an integer.
class VetSpecialties(models.Model):
    id = models.UUIDField(
        primary_key=True,
        default=uuid.uuid4,
        editable=False
    )
    vet = models.ForeignKey(Vet, on_delete=models.CASCADE, db_constraint=False)
    specialty = models.ForeignKey(Specialty, on_delete=models.CASCADE,
 db_constraint=False)
```

Define views

Like the views we have created for Owners and Pets, we define the views for Specialties and and Vets. Also, we follow the similar CRUD pattern that we followed for Owners and pets.

```
@method_decorator(csrf_exempt, name='dispatch')
class SpecialtyView(View):
    @with_retries()
    def get(self, request=None, name=None, *args, **kwargs):
        specialties = Specialty.objects
        # Apply filter if specific name is requested.
        if name is not None:
            specialties = specialties.filter(name=name)
        return JsonResponse(list(specialties.values()), safe=False)

    @with_retries()
    @atomic
    def post(self, request=None, *args, **kwargs):
        data = json.loads(request.body.decode())
        name = data.get('name', None)
        if name is None:
            return HttpResponseBadRequest()

        specialty = Specialty(name=name)
        specialty.save()
        return
 JsonResponse(list(Specialty.objects.filter(name=specialty.name).values()), safe=False)

    @with_retries()
    @atomic
    def delete(self, request=None, name=None, *args, **kwargs):
        if id is not None:
            Specialty.objects.filter(name=name).delete()
        return HttpResponse(status=200)

@method_decorator(csrf_exempt, name='dispatch')
class VetView(View):
    @with_retries()
    def get(self, request=None, id=None, *args, **kwargs):
        vets = Vet.objects
        # Apply filter if specific id is requested.
        if id is not None:
            vets = vets.filter(id=id)
        return JsonResponse(list(vets.values()), safe=False)
```

```
    @with_retries()
    @atomic
    def post(self, request, *args, **kwargs):
        data = json.loads(request.body.decode())
        # If id is provided we try updating the existing object
        id = data.get('id', None)
        try:
            vet = Vet.objects.get(id=id) if id is not None else None
        except:
            return HttpResponseBadRequest(("error: check if vet with id `%s` exists") %
(id))

        name = data.get('name', vet.name if vet else None)

        # Either the name or id must be provided.
        if vet is None and name is None:
            return HttpResponseBadRequest()

        owner_id = data.get('owner_id', vet.owner.id if vet and vet.owner else None)
        try:
            owner = Owner.objects.get(id=owner_id) if owner_id else None
        except:
            return HttpResponseBadRequest(("error: check if owner with id `%s` exists")
% (id))

        specialties_list = data.get('specialties', vet.specialties if vet and
vet.specialties else [])
        specialties = []
        for specialty in specialties_list:
            try:
                specialties_obj = Specialty.objects.get(name=specialty)
            except Exception:
                return HttpResponseBadRequest(("error: check if specialty `%s` exists")
% (specialty))
            specialties.append(specialties_obj)

        if vet is None:
            print(("vet name: %s, not present, adding") % (name))
            vet = Vet(name=name, owner_id=owner_id)
        else:
            print(("vet name: %s, present, updating") % (name))
            vet.name = name
            vet.owner = owner
```

```
        # First save the vet so that we have an id. Then we can add specialties.
        # Django needs the id primary key of the parent object before adding relations
        vet.save()

        # Add any specialties provided
        vet.specialties.add(*specialties)
        return JsonResponse(
            {
                'Veterinarian': list(Vet.objects.filter(id=vet.id).values()),
                'Specialties': list(VetSpecialties.objects.filter(vet=vet.id).values())
            }, safe=False)

    @with_retries()
    @atomic
    def delete(self, request, id=None, *args, **kwargs):
        if id is not None:
            Vet.objects.filter(id=id).delete()
        return HttpResponse(status=200)

@method_decorator(csrf_exempt, name='dispatch')
class VetSpecialtiesView(View):
    @with_retries()
    def get(self, request=None, *args, **kwargs):
        data = json.loads(request.body.decode())
        vet_id = data.get('vet_id', None)
        specialty_id = data.get('specialty_id', None)
        specialties = VetSpecialties.objects
        # Apply filter if specific name is requested.
        if vet_id is not None:
            specialties = specialties.filter(vet_id=vet_id)
        if specialty_id is not None:
            specialties = specialties.filter(specialty_id=specialty_id)
        return JsonResponse(list(specialties.values()), safe=False)
```

Update routes

Modify the `django_aurora_dsql_example/project/project/urls.py` and ensure that urlpatterns variable is set like below

```
urlpatterns = [
    path('owner/', OwnerView.as_view(), name='owner'),
    path('owner/<id>', OwnerView.as_view(), name='owner'),
    path('pet/', PetView.as_view(), name='pet'),
```

```
    path('pet/<id>', PetView.as_view(), name='pet'),
    path('vet/', VetView.as_view(), name='vet'),
    path('vet/<id>', VetView.as_view(), name='vet'),
    path('specialty/', SpecialtyView.as_view(), name='specialty'),
    path('specialty/<name>', SpecialtyView.as_view(), name='specialty'),
    path('vet-specialties/<vet_id>', VetSpecialtiesView.as_view(), name='vet-
specialties'),
    path('specialty-vets/<specialty_id>', VetSpecialtiesView.as_view(), name='vet-
specialties'),
]
```

Test many-to-many

```
# Create some specialties
curl --request POST --data '{"name":"Exotic"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Dogs"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Cats"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Pandas"}' http://0.0.0.0:8000/specialty/
```

We can have vets with many specialties and same specialty can be attributed to many vets. If you try adding a specialty that does not exit, an error will be returned.

```
curl --request POST --data '{"name":"Jake", "specialties": ["Dogs", "Cats"]}'
 http://0.0.0.0:8000/vet/
curl --request POST --data '{"name":"Vince", "specialties": ["Dogs"]}'
 http://0.0.0.0:8000/vet/
curl --request POST --data '{"name":"Matt"}' http://0.0.0.0:8000/vet/
# Update Matt to have specialization in Cats and Exotic animals
curl --request POST --data '{"id":"2843be51-a26b-42b6-9e20-c3f2eba6e949",
 "specialties": ["Dogs", "Cats"]}' http://0.0.0.0:8000/vet/
```

Delete

Deleting the specialty will update list of specialties associated with the veterinarian because we have setup the CASCADE delete constraint.

```
# Check the list of vets who has the Dogs specialty attributed
curl --request GET --data '{"specialty_id":"Dogs"}' http://0.0.0.0:8000/vet-
specialties/
```

```
# Delete dogs specialty, in our sample queries there are two vets who has this
 specialty
curl --request DELETE http://0.0.0.0:8000/specialty/Dogs
# We can now check that vets specialties are updated. The Dogs specialty must have been
 removed from the vet's specialties.
curl --request GET --data '{"vet_id":"2843be51-a26b-42b6-9e20-c3f2eba6e949"}'
 http://0.0.0.0:8000/vet-specialties/
```

Test one-to-one

```
# Crate few owners
curl --request POST --data '{"name":"Paul", "city":"Seattle"}' http://0.0.0.0:8000/
owner/
curl --request POST --data '{"name":"Pablo", "city":"New York"}' http://0.0.0.0:8000/
owner/
# Note down owner ids

    # Create some specialties
curl --request POST --data '{"name":"Exotic"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Dogs"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Cats"}' http://0.0.0.0:8000/specialty/
curl --request POST --data '{"name":"Pandas"}' http://0.0.0.0:8000/specialty/

    # Create veterinarians
    # We can create vet who is also a owner
curl --request POST --data '{"name":"Pablo", "specialties": ["Dogs", "Cats"],
 "owner_id": "b60bbdda-6aae-4b82-9711-5743b3667334"}' http://0.0.0.0:8000/vet/
# We can create vets who are not owners
curl --request POST --data '{"name":"Vince", "specialties": ["Exotic"]}'
 http://0.0.0.0:8000/vet/
curl --request POST --data '{"name":"Matt"}' http://0.0.0.0:8000/vet/

# Trying to add a new vet with an already associated owner id will cause integrity
 error
curl --request POST --data '{"name":"Jenny", "owner_id":
 "b60bbdda-6aae-4b82-9711-5743b3667334"}' http://0.0.0.0:8000/vet/

# Deleting the owner will lead to updating of owner field in vet to Null.
curl --request DELETE  http://0.0.0.0:8000/owner/b60bbdda-6aae-4b82-9711-5743b3667334

curl --request GET http://0.0.0.0:8000/vet/603e44b1-cf3a-4180-8df3-2c73fac507bd
```

Using Aurora DSQL to build an application with SQLAlchemy

This section describes how how to create a pet clinic web application with SQLAlchemy that uses Aurora DSQL as a database. This clinic has pets, owners, veterinarians, and specialties.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL.
- Installed Python. You must be running version 3.8 or higher.
- Created an AWS account and configured the credentials and AWS Region.
- Installed the AWS SDK for Python (Boto3).

Setup

See the following steps to set up your environment.

1. In your local environment, create and activate the Python virtual environment with the following commands.

```
python3 -m venv sqlalchemy_venv
source sqlalchemy_venv/bin/activate
```

2. Install the required dependencies.

```
pip install sqlalchemy
pip install "psycopg2-binary>=2.9"
```

> **Note**
>
> Note that SqlAlchemy with Psycopg3 does not work with Aurora DSQL. SqlAlchemy with Psycopg3 uses nested transactions which rely on savepoints as part of the connection setup. Savepoints are not supported by Aurora DSQL

Connect to an Aurora DSQL cluster

The following example demonstrates how to create an Aurora DSQL engine with SQLAlchemy and connect to a cluster in Aurora DSQL.

```
import boto3
from sqlalchemy import create_engine
from sqlalchemy.engine import URL

def create_dsql_engine():
    hostname = "foo0bar1baz2quux3quuux4.c0001.us-east-1.prod.sql.axdb.aws.dev"
    region = "us-east-1"
    client = boto3.client("dsql", region_name=region)

    # The token expiration time is optional, and the default value 900 seconds
    # Use `generate_db_connect_auth_token` instead if you are not connecting as `admin`
 user
    password_token = client.generate_db_connect_admin_auth_token(hostname, region)

    # Example on how to create engine for SQLAlchemy
    url = URL.create("postgresql", username="admin", password=password_token,
        host=hostname, database="postgres")
    # Prefer sslmode = verify-full for production usecases
    engine = create_engine(url, connect_args={"sslmode": "require"})

    return engine
```

Create models

One owner can have many pets, thus creating a one-to-many relationship. A veterinarian can have many specialties, so that is a many-to-many relationship. The following example creates all of these tables and relationships. Aurora DSQL doesn't support SERIAL, so all unique identifiers are based on a universal unique identifier (UUID).

```
## Dependencies for Model class
from sqlalchemy import String
from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import relationship
from sqlalchemy import Column, Date
from sqlalchemy.dialects.postgresql import UUID
from sqlalchemy.sql import text

class Base(DeclarativeBase):
    pass

# Define a Owner table
class Owner(Base):
```

```
    __tablename__ = "owner"

    id = Column(
                "id", UUID, primary_key=True, default=text('gen_random_uuid()')
            )
    name = Column("name", String(30), nullable=False)
    city = Column("city", String(80), nullable=False)
    telephone = Column("telephone", String(20), nullable=True, default=None)

# Define a Pet table
class Pet(Base):
    __tablename__ = "pet"

    id = Column(
                "id", UUID, primary_key=True, default=text('gen_random_uuid()')
            )
    name = Column("name", String(30), nullable=False)
    birth_date = Column("birth_date", Date(), nullable=False)
    owner_id = Column(
                "owner_id", UUID, nullable=True
    )
    owner = relationship("Owner", foreign_keys=[owner_id], primaryjoin="Owner.id ==
 Pet.owner_id")

# Define an association table for Vet and Speacialty
class VetSpecialties(Base):
    __tablename__ = "vetSpecialties"

    id = Column(
                "id", UUID, primary_key=True, default=text('gen_random_uuid()')
            )
    vet_id = Column(
                "vet_id", UUID, nullable=True
    )
    specialty_id = Column(
                "specialty_id", String(80), nullable=True
    )

# Define a Specialty table
class Specialty(Base):
    __tablename__ = "specialty"
    id = Column(
                "name", String(80), primary_key=True
            )
```

```
# Define a Vet table
class Vet(Base):
    __tablename__ = "vet"

    id = Column(
                "id", UUID, primary_key=True, default=text('gen_random_uuid()')
            )
    name = Column("name", String(30), nullable=False)
    specialties = relationship("Specialty", secondary=VetSpecialties.__table__,
        primaryjoin="foreign(VetSpecialties.vet_id)==Vet.id",
        secondaryjoin="foreign(VetSpecialties.specialty_id)==Specialty.id")
```

CRUD examples

You can now run CRUD operations to add, read, update, and delete data. Note that to run these examples, you must have configured AWS credentials.

Run the following example to create all of the necessary tables and modify data inside them.

```
from sqlalchemy.orm import Session
from sqlalchemy import select

def example():
    # Create the engine
    engine = create_dsql_engine()

    # Drop all tables if any
    for table in Base.metadata.tables.values():
        table.drop(engine, checkfirst=True)

    # Create all tables
    for table in Base.metadata.tables.values():
        table.create(engine, checkfirst=True)

    session = Session(engine)
    # Owner-Pet relationship is one to many.
    ## Insert owners
    john_doe = Owner(name="John Doe", city="Anytown")
    mary_major = Owner(name="Mary Major", telephone="555-555-0123", city="Anytown")

    ## Add two pets.
    pet_1 = Pet(name="Pet-1", birth_date="2006-10-25", owner=john_doe)
```

```
    pet_2 = Pet(name="Pet-2", birth_date="2021-7-23", owner=mary_major)

    session.add_all([john_doe, mary_major, pet_1, pet_2])
    session.commit()

    # Read back data for the pet.
    pet_query = select(Pet).where(Pet.name == "Pet-1")
    pet_1 = session.execute(pet_query).fetchone()[0]

    # Get the corresponding owner
    owner_query = select(Owner).where(Owner.id == pet_1.owner_id)
    john_doe = session.execute(owner_query).fetchone()[0]

    # Test: check read values
    assert pet_1.name == "Pet-1"
    assert str(pet_1.birth_date) == "2006-10-25"
    # Owner must be what we have inserted
    assert john_doe.name == "John Doe"
    assert john_doe.city == "Anytown"

    # Vet-Specialty relationship is many to many.
    dogs = Specialty(id="Dogs")
    cats = Specialty(id="Cats")

    ## Insert two vets with specialties, one vet without any specialty
    akua_mansa = Vet(name="Akua Mansa",specialties=[dogs])
    carlos_salazar = Vet(name="Carlos Salazar", specialties=[dogs, cats])

    session.add_all([dogs, cats, akua_mansa, carlos_salazar])
    session.commit()

    # Read back data for the vets.
    vet_query = select(Vet).where(Vet.name == "Akua Mansa")
    akua_mansa = session.execute(vet_query).fetchone()[0]

    vet_query = select(Vet).where(Vet.name == "Carlos Salazar")
    carlos_salazar = session.execute(vet_query).fetchone()[0]

    # Test: check read value
    assert akua_mansa.name == "Akua Mansa"
    assert akua_mansa.specialties[0].id == "Dogs"

    assert carlos_salazar.name == "Carlos Salazar"
    assert carlos_salazar.specialties[0].id == "Cats"
```

```
    assert carlos_salazar.specialties[1].id == "Dogs"
```

Using Psycopg2 to interact with Aurora DSQL

This section describes how to use Psycopg2 to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL.
- Installed Python. You must be running version 3.8 or higher.
- Created an AWS account and configured the credentials and AWS Region.
- Installed the AWS SDK for Python (Boto3).

Before you get started, install the required dependency.

```
pip install "psycopg2-binary>=2.9"
```

Connect to an Aurora DSQL cluster and run queries

```
import psycopg2
import boto3
import os, sys

def main(cluster_endpoint):
    region = 'us-east-1'

    # Generate a password token
    client = boto3.client("dsql", region_name=region)
    password_token = client.generate_db_connect_admin_auth_token(cluster_endpoint,
 region)

    # connection parameters
    dbname = "dbname=postgres"
    user = "user=admin"
    host = f'host={cluster_endpoint}'
    sslmode = "sslmode=verify-full"
    sslrootcert = "sslrootcert=system"
    password = f'password={password_token}'

    # Make a connection to the cluster
```

```
    conn = psycopg2.connect('%s %s %s %s %s %s' % (dbname, user, host, sslmode,
 sslrootcert, password))

    conn.set_session(autocommit=True)

    cur = conn.cursor()

    cur.execute(b"""
        CREATE TABLE IF NOT EXISTS owner(
            id uuid NOT NULL DEFAULT gen_random_uuid(),
            name varchar(30) NOT NULL,
            city varchar(80) NOT NULL,
            telephone varchar(20) DEFAULT NULL,
            PRIMARY KEY (id))"""
        )

    # Insert some rows
    cur.execute("INSERT INTO owner(name, city, telephone) VALUES('John Doe', 'Anytown',
 '555-555-1999')")

    # Read back what we have inserted
    cur.execute("SELECT * FROM owner WHERE name='John Doe'")
    row = cur.fetchone()

    # Verify that the result we got is what we inserted before
    assert row[0] != None
    assert row[1] == "John Doe"
    assert row[2] == "Anytown"
    assert row[3] == "555-555-1999"

    # Placing this cleanup the table after the example. If we run the example
    # again we do not have to worry about data inserted by previous runs
    cur.execute("DELETE FROM owner where name = 'John Doe'")

if __name__ == "__main__":
    # Replace with your own cluster's endpoint
    cluster_endpoint = "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws"
    main(cluster_endpoint)
```

Using Psycopg3 to interact with Aurora DSQL

This section describes how to use Psycopg3 to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL.
- Installed Python. You must be running version 3.8 or higher.
- Created an AWS account and configured the credentials and AWS Region.
- Installed the AWS SDK for Python (Boto3).

Before you get started, install the required dependency.

```
pip install "psycopg[binary]>=3"
```

Connect to an Aurora DSQL cluster and run queries

```
import psycopg
import boto3
import os, sys

def main(cluster_endpoint):
    region = 'us-east-1'

    # Generate a password token
    client = boto3.client("dsql", region_name=region)
    password_token = client.generate_db_connect_admin_auth_token(cluster_endpoint,
 region)

    # connection parameters
    dbname = "dbname=postgres"
    user = "user=admin"
    host = f'host={cluster_endpoint}'
    sslmode = "sslmode=verify-full"
    sslrootcert = "sslrootcert=system"
    password = f'password={password_token}'

    # Make a connection to the cluster
    conn = psycopg.connect('%s %s %s %s %s %s' % (dbname, user, host, sslmode,
 sslrootcert, password))

    conn.set_autocommit(True)

    cur = conn.cursor()
```

```
    cur.execute(b"""
        CREATE TABLE IF NOT EXISTS owner(
            id uuid NOT NULL DEFAULT gen_random_uuid(),
            name varchar(30) NOT NULL,
            city varchar(80) NOT NULL,
            telephone varchar(20) DEFAULT NULL,
            PRIMARY KEY (id))"""
        )

    # Insert some rows
    cur.execute("INSERT INTO owner(name, city, telephone) VALUES('John Doe', 'Anytown',
 '555-555-1999')")

    cur.execute("SELECT * FROM owner WHERE name='John Doe'")
    row = cur.fetchone()

    # Verify that the result we got is what we inserted before
    assert row[0] != None
    assert row[1] == "John Doe"
    assert row[2] == "Anytown"
    assert row[3] == "555-555-1999"

    # Placing this cleanup the table after the example. If we run the example
    # again we do not have to worry about data inserted by previous runs
    cur.execute("DELETE FROM owner where name = 'John Doe'")

if __name__ == "__main__":
    # Replace with your own cluster's endpoint
    cluster_endpoint = "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws"
    main(cluster_endpoint)
```

Programming with Java

Topics

- Using Aurora DSQL to build applications with JDBC, Hibernate, and HikariCP
- Using pgJDBC to interact with Amazon Aurora DSQL

Using Aurora DSQL to build applications with JDBC, Hibernate, and HikariCP

This section describes how how to create a web application with JDBC, Hibernate, and HikariCP that uses Aurora DSQL as a database. This example doesn't cover how to implement `@OneToMany` or `@ManyToMany` relationships, but these relationships in Aurora DSQL work similarly to standard Hibernate implementations. You can use these relationships to model associations between entities in your database. To learn more about how to use these relationships with Hibernate, see Associations in the official Hibernate documentation. As you work with Aurora DSQL, you can follow these guidelines to set up your entity relationships. Note that Aurora DSQL doesn't support foreign keys, so you must use a universally unique identifier (UUID) instead.

Before you begin, make sure that you have completed the following prerequisites:

- Created a cluster in Aurora DSQL.
- Installed Java. You must be running version 1.8 or higher.
- Installed the AWS SDK for Java.
- Configured your AWS credentials.

Setup

To connect to the Aurora DSQL server, you must configure the username, URL endpoint, and password by setting the properties. The following is an example configuration. This example also generates an authentication token, which you can use to connect to your cluster in Aurora DSQL.

```
import org.springframework.boot.autoconfigure.jdbc.DataSourceProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.zaxxer.hikari.HikariDataSource;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.dsql.DsqlUtilities;

@Configuration(proxyBeanMethods = false)
public class DsqlDataSourceConfig {

    @Bean
```

```
    public HikariDataSource dataSource() {
        final DataSourceProperties properties = new DataSourceProperties();

        // Set the username
        properties.setUsername("admin");

        // Set the URL and endpoint
        properties.setUrl("jdbc:postgresql://foo0bar1baz2quux3quuux4.dsql.us-
east-1.on.aws/postgres?ssl=require");

        final HikariDataSource hds =
 properties.initializeDataSourceBuilder().type(HikariDataSource.class).build();

        // Set additional properties
        hds.setMaxLifetime(1500*1000); // pool connection expiration time in milli
 seconds

        // Generate and set the DSQL token
        final DsqlUtilities utilities = DsqlUtilities.builder()
                .region(Region.US_EAST_1)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        // Use generateDbConnectAuthToken when _not_ connecting as `admin` user
        final String token = utilities.generateDbConnectAdminAuthToken(builder ->
                builder.hostname(hds.getJdbcUrl().split("/")[2])
                        .region(Region.US_EAST_1)
                        .expiresIn(Duration.ofMillis(30*1000)) // Token expiration
 time, default is 900 seconds
        );

        hds.setPassword(token);

        return hds;
    }
}
```

Using a UUID as a primary key

Aurora DSQL doesn't support serialized primary keys or identity columns that automatically increment integers that you might find in other relational databases. Instead, we recommend that you use a universally unique identifier (UUID) as the primary key for your identities. To define a primary key, first import the UUID class.

```
import java.util.UUID;
```

You can then define a UUId primary key in your entity class.

```
@Id
@Column(name = "id", updatable = false, nullable = false, columnDefinition = "UUID
 DEFAULT gen_random_uuid()")
private UUID id;
```

Define entity classes

Hibernate can automatically create and validate databases tables based on your entity class definitions. The following example demonstrates how to define an entity class.

```
import java.io.Serializable;
import java.util.UUID;

import jakarta.persistence.Column;
import org.hibernate.annotations.Generated;

import jakarta.persistence.Id;
import jakarta.persistence.MappedSuperclass;

@MappedSuperclass
public class Person implements Serializable {

    @Generated
    @Id
    @Column(name = "id", updatable = false, nullable = false, columnDefinition = "UUID
 DEFAULT gen_random_uuid()")
    private UUID id;

    @Column(name = "first_name")
    @NotBlank
    private String firstName;

    // Getters and setters
    public String getId() {
        return id;
    }

    public void setId(UUID id) {
```

```
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String id) {
        this.firstName = firstName;
    }
}
```

Handle SQL exceptions

To handle specific SQL exceptions, such as 0C001 or 0C000, implement a custom SQLExceptionOverride class. We do not want to evict the connection immediately if we encounter an OCC error.

```
public class DsqlExceptionOverride implements SQLExceptionOverride {
    @Override
    public Override adjudicate(SQLException ex) {
        final String sqlState = ex.getSQLState();

        if ("0C000".equalsIgnoreCase(sqlState) || "0C001".equalsIgnoreCase(sqlState) ||
 (sqlState).matches("0A\\d{3}")) {
            return SQLExceptionOverride.Override.DO_NOT_EVICT;
        }

        return Override.CONTINUE_EVICT;
    }
}
```

Now set the following class in your HikariCP configuration.

```
@Configuration(proxyBeanMethods = false)
public class DsqlDataSourceConfig {

    @Bean
    public HikariDataSource dataSource() {
        final DataSourceProperties properties = new DataSourceProperties();
```

```
        final HikariDataSource hds =
 properties.initializeDataSourceBuilder().type(HikariDataSource.class).build();

        // handle the connection eviction for known exception types.
        hds.setExceptionOverrideClassName(DsqlExceptionOverride.class.getName());

        return hds;
    }
}
```

Using pgJDBC to interact with Amazon Aurora DSQL

This section describes how to use pgJDBC to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL.
- Installed the Java Development Kit (JDK). Make sure that you have version 8 or higher. You can download it from AWS Coretto or use OpenJDK. To verify that you've installed Java and see what version you have, run `java -version`.
- Download and install Maven.
- Installed the AWS SDK for Java 2.x.

Connect to an Aurora DSQL cluster and run queries

```
package org.example;

import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.services.dsql.DsqlUtilities;
import software.amazon.awssdk.regions.Region;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.Duration;
import java.util.Properties;
import java.util.UUID;

public class Example {
```

```
    // Get a connection to Aurora DSQL.
    public static Connection getConnection(String clusterEndpoint, String region)
 throws SQLException {
        Properties props = new Properties();

        // Use the DefaultJavaSSLFactory so that Java's default trust store can be used
        // to verify the server's root cert.
        String url = "jdbc:postgresql://" + clusterEndpoint + ":5432/postgres?
sslmode=verify-full&sslfactory=org.postgresql.ssl.DefaultJavaSSLFactory";

        DsqlUtilities utilities = DsqlUtilities.builder()
                .region(Region.of(region))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        String password = utilities.generateDbConnectAdminAuthToken(builder ->
 builder.hostname(clusterEndpoint)
                .region(Region.of(region)));

        props.setProperty("user", "admin");
        props.setProperty("password", password);
        return DriverManager.getConnection(url, props);
    }

    public static void main(String[] args) {
        // Replace the cluster endpoint with your own
        String clusterEndpoint = "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws";
        String region = "us-east-1";
        try (Connection conn = Example.getConnection(clusterEndpoint, region)) {

            // Create a new table named owner
            Statement create = conn.createStatement();
            create.executeUpdate("CREATE TABLE IF NOT EXISTS owner (id UUID PRIMARY
 KEY, name VARCHAR(255), city VARCHAR(255), telephone VARCHAR(255))");
            create.close();

            // Insert some data
            UUID uuid = UUID.randomUUID();
            String insertSql = String.format("INSERT INTO owner (id, name, city,
 telephone) VALUES ('%s', 'John Doe', 'Anytown', '555-555-1999')", uuid);
            Statement insert = conn.createStatement();
            insert.executeUpdate(insertSql);
            insert.close();
```

```
            // Read back the data and assert they are present
            String selectSQL = "SELECT * FROM owner";
            Statement read = conn.createStatement();
            ResultSet rs = read.executeQuery(selectSQL);
            while (rs.next()) {
                assert rs.getString("id") != null;
                assert rs.getString("name").equals("John Doe");
                assert rs.getString("city").equals("Anytown");
                assert rs.getString("telephone").equals("555-555-1999");
            }

            // Delete some data
            String deleteSql = String.format("DELETE FROM owner where name='John
 Doe'");
            Statement delete = conn.createStatement();
            delete.executeUpdate(deleteSql);
            delete.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Programming with JavaScript

Topics

- Using Node.js to interact with Amazon Aurora DSQL

Using Node.js to interact with Amazon Aurora DSQL

This section describes how to use Node.js to interact with Aurora DSQL.

Before you begin, make sure that you have created a cluster in Aurora DSQL. Also make sure that you have installed Node. You must have installed version 18 or higher. Use the following command to check which version you have.

```
node --version
```

Connect to your Aurora DSQL cluster and run queries

Use the following JavaScript to connect to your cluster in Aurora DSQL.

```
import { DsqlSigner } from "@aws-sdk/dsql-signer";
import pg from "pg";
import assert from "node:assert";
const { Client } = pg;

async function example(clusterEndpoint) {
  let client;
  const region = "us-east-1";
  try {
    // The token expiration time is optional, and the default value 900 seconds
    const signer = new DsqlSigner({
      hostname: clusterEndpoint,
      region,
    });
    const token = await signer.getDbConnectAdminAuthToken();
    client = new Client({
      host: clusterEndpoint,
      user: "admin",
      password: token,
      database: "postgres",
      port: 5432,
      // <https://node-postgres.com/announcements> for version 8.0
      ssl: true
    });

    // Connect
    await client.connect();

    // Create a new table
    await client.query(`CREATE TABLE IF NOT EXISTS owner (
      id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
      name VARCHAR(30) NOT NULL,
      city VARCHAR(80) NOT NULL,
      telephone VARCHAR(20)
    )`);

    // Insert some data
    await client.query("INSERT INTO owner(name, city, telephone) VALUES($1, $2, $3)",
      ["John Doe", "Anytown", "555-555-1900"]
    );
```

```
    // Check that data is inserted by reading it back
    const result = await client.query("SELECT id, city FROM owner where name='John
 Doe'");
    assert.deepEqual(result.rows[0].city, "Anytown")
    assert.notEqual(result.rows[0].id, null)

    await client.query("DELETE FROM owner where name='John Doe'");

  } catch (error) {
    console.error(error);
    raise
  } finally {
    client?.end()
  }
  Promise.resolve()
}

export { example }
```

Programming with C++

Topics

- Using Libpq to interact with Amazon Aurora DSQL

Using Libpq to interact with Amazon Aurora DSQL

This section describes how how to use Libpq to interact with Aurora DSQL.

The example assumes that you are on a linux machine.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL
- Installed the AWS SDK for C++
- Obtained the Libpq library. If you installed postgres, then Libpq is in the paths `../postgres_install_dir/lib` and `../postgres_install_dir/include`. You might have also installed it if you installed the psql client. If you need to get it, you can install it through the package manager.

```
sudo yum install libpq-devel
```

You can also download psql through the official PostgreSQL website, , which includes Libpq.

- Installed the SSL libraries. For example, if you're on Amazon Linux, run the following commands to install the libraries.

```
sudo yum install -y openssl-devel
sudo yum install -y  openssl11-libs
```

You can also download them from the official OpenSSL website.

- Configured your AWS credentials. For more information, see Set and view configuration settings using commands.

Connect to your Aurora DSQL cluster and run queries

Use the following example to generate an authentication token and connect to your Aurora DSQL cluster.

```
#include <libpq-fe.h>
#include <aws/core/Aws.h>
#include <aws/dsql/DSQLClient.h>
#include <iostream>

using namespace Aws;
using namespace Aws::DSQL;
using namespace Aws::DSQL::Model;

std::string generateDBAuthToken(const std::string endpoint, const std::string region) {
    Aws::SDKOptions options;
    Aws::InitAPI(options);
    DSQLClientConfiguration clientConfig;
    clientConfig.region = region;
    DSQLClient client{clientConfig};
    std::string token = "";

    // The token expiration time is optional, and the default value 900 seconds
    // If you aren't using an admin role to connect, use GenerateDBConnectAuthToken
 instead
```

```
    const auto presignedString = client.GenerateDBConnectAdminAuthToken(endpoint,
 region);
    if (presignedString.IsSuccess()) {
        token = presignedString.GetResult();
    } else {
        std::cerr << "Token generation failed." << std::endl;
    }

    Aws::ShutdownAPI(options);
    return token;
}

PGconn* connectToCluster(std::string clusterEndpoint, std::string region) {
    std::string password = generateDBAuthToken(clusterEndpoint, region);

    std::string dbname = "postgres";
    std::string user = "admin";
    std::string sslmode = "require";
    int port = 5432;

    if (password.empty()) {
        std::cerr << "Failed to generate token." << std::endl;
        return NULL;
    }

    char conninfo[4096];
    sprintf(conninfo, "dbname=%s user=%s host=%s port=%i sslmode=%s password=%s",
            dbname.c_str(), user.c_str(), clusterEndpoint.c_str(), port,
 sslmode.c_str(), password.c_str());

    PGconn *conn = PQconnectdb(conninfo);

    if (PQstatus(conn) != CONNECTION_OK) {
        std::cerr << "Error while connecting to the database server: " <<
 PQerrorMessage(conn) << std::endl;
        PQfinish(conn);
       return NULL;
    }

    std::cout << std::endl << "Connection Established: " << std::endl;
    std::cout << "Port: " << PQport(conn) << std::endl;
    std::cout << "Host: " << PQhost(conn) << std::endl;
    std::cout << "DBName: " << PQdb(conn) << std::endl;
```

```
    return conn;
}

void example(PGconn *conn) {

    // Create a table
    std::string create = "CREATE TABLE IF NOT EXISTS owner (id UUID PRIMARY KEY DEFAULT
 gen_random_uuid(), name VARCHAR(30) NOT NULL, city VARCHAR(80) NOT NULL, telephone
 VARCHAR(20))";

    PGresult *createResponse = PQexec(conn, create.c_str());
    ExecStatusType createStatus = PQresultStatus(createResponse);
    PQclear(createResponse);

    if (createStatus != PGRES_COMMAND_OK) {
        std::cerr << "Create Table failed - " << PQerrorMessage(conn) << std::endl;

    }

    // Insert data into the table
    std::string insert = "INSERT INTO owner(name, city, telephone) VALUES('John Doe',
 'Anytown', '555-555-1999')";

    PGresult *insertResponse = PQexec(conn, insert.c_str());
    ExecStatusType insertStatus = PQresultStatus(insertResponse);
    PQclear(insertResponse);

    if (insertStatus != PGRES_COMMAND_OK) {
        std::cerr << "Insert failed - " << PQerrorMessage(conn) << std::endl;
    }

    // Read the data we inserted
    std::string select = "SELECT * FROM owner";

    PGresult *selectResponse = PQexec(conn, select.c_str());
    ExecStatusType selectStatus = PQresultStatus(selectResponse);

    if (selectStatus != PGRES_TUPLES_OK) {
        std::cerr << "Select failed - " << PQerrorMessage(conn) << std::endl;
        PQclear(selectResponse);
        return;
    }

    // Retrieve the number of rows and columns in the result
```

```
    int rows = PQntuples(selectResponse);
    int cols = PQnfields(selectResponse);
    std::cout << "Number of rows: " << rows << std::endl;
    std::cout << "Number of columns: " << cols << std::endl;

    // Output the column names
    for (int i = 0; i < cols; i++) {
        std::cout << PQfname(selectResponse, i) << " \t\t\t ";
    }
    std::cout << std::endl;

    // Output all the rows and column values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            std::cout << PQgetvalue(selectResponse, i, j) << "\t";
        }
        std::cout << std::endl;
    }
    PQclear(selectResponse);
}

int main(int argc, char *argv[]) {
    std::string region = "us-east-1";
    // Replace with your own cluster endpoint
    std::string clusterEndpoint = "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws";

    PGconn *conn = connectToCluster(clusterEndpoint, region);

    if (conn == NULL) {
        std::cerr << "Failed to get connection. Exiting." << std::endl;
        return -1;
    }

    example(conn);

    return 0;
}
```

Programming with Ruby

Topics

- Using Ruby-pg to interact with Amazon Aurora DSQL

- Using Ruby on Rails to interact with Amazon Aurora DSQL

Using Ruby-pg to interact with Amazon Aurora DSQL

This section describes how how to use Ruby-pg to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Configured a `default` profile that contains your AWS credentials that uses the following variables.
  - aws_access_key_id=<your_access_key_id>
  - aws_secret_access_key=<your_secret_access_key>
  - aws_session_token=<your_session_token>

  Your `~/.aws/credentials` file should look like the following.

  ```
  [default]
  aws_access_key_id=<your_access_key_id>
  aws_secret_access_key=<your_secret_access_key>
  aws_session_token=<your_session_token>
  ```
- Created a cluster in Aurora DSQL.
- Installed Ruby. You must have version 2.5 or higher. To check which version you have, run `ruby --version`.
- Installed the required dependencies that are in the Gemfile. To install them, run `bundle install`.

Connect to your Aurora DSQL cluster and run queries

```
require 'pg'
require 'aws-sdk-dsql'

def example()
  cluster_endpoint = 'foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws'
  region = 'us-east-1'
  credentials = Aws::SharedCredentials.new()

  begin
      token_generator = Aws::DSQL::AuthTokenGenerator.new({
```

```
        :credentials => credentials
    })

    # The token expiration time is optional, and the default value 900 seconds
    # if you are not using admin role, use generate_db_connect_auth_token instead
    token = token_generator.generate_db_connect_admin_auth_token({
        :endpoint => cluster_endpoint,
        :region => region
    })

    conn = PG.connect(
      host: cluster_endpoint,
      user: 'admin',
      password: token,
      dbname: 'postgres',
      port: 5432,
      sslmode: 'verify-full',
      sslrootcert: "./root.pem"
    )
rescue => _error
    raise
end

# Create the owner table
conn.exec('CREATE TABLE IF NOT EXISTS owner (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(30) NOT NULL,
  city VARCHAR(80) NOT NULL,
  telephone VARCHAR(20)
)')

# Insert an owner
conn.exec_params('INSERT INTO owner(name, city, telephone) VALUES($1, $2, $3)',
  ['John Doe', 'Anytown', '555-555-0055'])

# Read the result back
result = conn.exec("SELECT city FROM owner where name='John Doe'")

# Raise error if we are unable to read
raise "must have fetched a row" unless result.ntuples == 1
raise "must have fetched right city" unless result[0]["city"] == 'Anytown'

# Delete data we just inserted
conn.exec("DELETE FROM owner where name='John Doe'")
```

```
rescue => error
  puts error.full_message
ensure
  unless conn.nil?
    conn.finish()
  end
end

# Run the example
example()
```

Using Ruby on Rails to interact with Amazon Aurora DSQL

This section describes how how to use Ruby on Rails to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL.
- Rails requires Ruby 3.1.0 or higher. You can download Ruby from the official Ruby website. To check which version of Ruby you have, run `ruby --version`.
- Installed Ruby on Rails. To check which version you have, run `rails --version`. Then run `bundle install` to install the required gems.

Install a connection to Aurora DSQL

Aurora DSQL uses IAM as authentication to establish a connection. You can't provide a password directly to rails through the configuration in the `{root-directory}/config/database.yml` file. Instead, use the `aws_rds_iam` adapter to use an authentication token to connect to Aurora DSQL. The steps below demonstrate how to do so.

Create a file named `{app root directory}/config/initializers/adapter.rb` with the following content.

```
PG::AWS_RDS_IAM.auth_token_generators.add :dsql do
  DsqlAuthTokenGenerator.new
end

require "aws-sigv4"
require 'aws-sdk-dsql'
```

```
# This is our custom DB auth token generator
# use the ruby sdk to generate token instead.
class DsqlAuthTokenGenerator
  def call(host:, port:, user:)
    region = "us-east-1"
    credentials = Aws::SharedCredentials.new()

    token_generator = Aws::DSQL::AuthTokenGenerator.new({
        :credentials => credentials
    })

    # The token expiration time is optional, and the default value 900 seconds
    # if you are not logging in as admin, use generate_db_connect_auth_token instead
    token = token_generator.generate_db_connect_admin_auth_token({
        :endpoint => host,
        :region => region
    })

  end
end

# Monkey-patches to disable unsupported features

require "active_record/connection_adapters/postgresql/schema_statements"

module ActiveRecord::ConnectionAdapters::PostgreSQL::SchemaStatements
  # Aurora DSQL does not support setting min_messages in the connection parameters
  def client_min_messages=(level); end
end

require "active_record/connection_adapters/postgresql_adapter"

class ActiveRecord::ConnectionAdapters::PostgreSQLAdapter

  def set_standard_conforming_strings; end

  # Aurora DSQL does not support running multiple DDL or DDL + DML statements in the
 same transaction
  def supports_ddl_transactions?
    false
  end
end
```

Create the following configuration in the `{app root directory}/config/database.yml` file. The following is an example configuration. You might create a similar configuration for testing purposes or production databases. This configuration automatically creates a new authentication token so you can connect to your database.

```
development:
  <<: *default
  database: postgres

  # The specified database role being used to connect to PostgreSQL.
  # To create additional roles in PostgreSQL see `$ createuser --help`.
  # When left blank, PostgreSQL will use the default role. This is
  # the same name as the operating system user running Rails.
  username: <postgres username> # eg: admin or other postgres users

  # Connect on a TCP socket. Omitted by default since the client uses a
  # domain socket that doesn't need configuration. Windows does not have
  # domain sockets, so uncomment these lines.
  # host: localhost
  # Set to Aurora DSQL cluster endpoint
  # host: <clusterId>.dsql.<region>.on.aws
  host: <cluster endpoint>
  # prefer verify-full for production usecases
  sslmode: require
  # Remember that we defined dsql token generator in the `{app root directory}/config/
initializers/adapter.rb`
  # We are providing it as the token generator to the adapter here.
  aws_rds_iam_auth_token_generator: dsql
  advisory_locks: false
  prepared_statements: false
```

Now you can create a data model. The following example creates a model and a migration file. Change the the model file to explicitly define the primary key of the table.

```
# Execute in the app root directory
bin/rails generate model Owner name:string city:string telephone:string
```

> **Note**
>
> Unlike postgres, Aurora DSQL creates a primary key index by including all columns of the table. This means that active record to search uses all columns of the table instead of just

the primary key. So So the <Entity>.find(<primary key>) won't work because the active record tries to search by using all columns in the primary key index.

To make active record search only using primary keys, set the primary key column explicitly in the model.

```
class Owner < ApplicationRecord
  self.primary_key = "id"
end
```

Generate the schema from the model files in db/migrate.

```
bin/rails db:migrate
```

Finally, disable the plpgsql extension by modifying the {app root directory}/db/schema.rb. In order to disable the plpgsql extension, remove the enable_extension "plgsql" line.

CRUD examples

You can now perform CRUD operations on your database. Run the following example to add owner data to your database.

```
owner = Owner.new(name: "John Smith", city: "Seattle", telephone: "123-456-7890")
owner.save
owner
```

Run the following example to retrieve the data.

```
Owner.find("<owner id>")
```

To update the data, use the following example.

```
Owner.find("<owner id>").update(telephone: "123-456-7891")
```

Finally, you can delete the data.

```
Owner.find("<owner id>").destroy
```

Programming with .NET

Topics

Using .NET to interact with Amazon Aurora DSQL

This section describes how how to use .NET to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL
- Installed .NET. You must have version 8 or higher. To see what version you have, run `dotnet --version`.
- Installed the .NET Npgsql driver.

Connect to your Aurora DSQL cluster

First define a `TokenGenerator` class. This class generates an authentication token, which you can use to connect to your Aurora DSQL cluster.

```
using Amazon.Runtime;
using Amazon.Runtime.Internal;
using Amazon.Runtime.Internal.Auth;
using Amazon.Runtime.Internal.Util;

public static class TokenGenerator
{
    public static string GenerateAuthToken(string? hostname, Amazon.RegionEndpoint
 region)
    {
        AWSCredentials awsCredentials = FallbackCredentialsFactory.GetCredentials();

        string accessKey = awsCredentials.GetCredentials().AccessKey;
        string secretKey = awsCredentials.GetCredentials().SecretKey;
        string token = awsCredentials.GetCredentials().Token;

        const string DsqlServiceName = "dsql";
        const string HTTPGet = "GET";
```

```
        const string HTTPS = "https";
        const string URISchemeDelimiter = "://";
        const string ActionKey = "Action";
        const string ActionValue = "DbConnectAdmin";
        const string XAmzSecurityToken = "X-Amz-Security-Token";

        ImmutableCredentials immutableCredentials = new ImmutableCredentials(accessKey,
secretKey, token) ?? throw new ArgumentNullException("immutableCredentials");
        ArgumentNullException.ThrowIfNull(region);

        hostname = hostname?.Trim();
        if (string.IsNullOrEmpty(hostname))
            throw new ArgumentException("Hostname must not be null or empty.");

        GenerateDsqlAuthTokenRequest authTokenRequest = new
GenerateDsqlAuthTokenRequest();
        IRequest request = new DefaultRequest(authTokenRequest, DsqlServiceName)
        {
            UseQueryString = true,
            HttpMethod = HTTPGet
        };
        request.Parameters.Add(ActionKey, ActionValue);
        request.Endpoint = new UriBuilder(HTTPS, hostname).Uri;

        if (immutableCredentials.UseToken)
        {
            request.Parameters[XAmzSecurityToken] = immutableCredentials.Token;
        }

        var signingResult = AWS4PreSignedUrlSigner.SignRequest(request, null, new
RequestMetrics(), immutableCredentials.AccessKey,
            immutableCredentials.SecretKey, DsqlServiceName, region.SystemName);

        var authorization = "&" + signingResult.ForQueryParameters;
        var url = AmazonServiceClient.ComposeUrl(request);

        // remove the https:// and append the authorization
        return url.AbsoluteUri[(HTTPS.Length + URISchemeDelimiter.Length)..] +
authorization;
    }

    private class GenerateDsqlAuthTokenRequest : AmazonWebServiceRequest
    {
        public GenerateDsqlAuthTokenRequest()
```

```
        {
            ((IAmazonWebServiceRequest)this).SignatureVersion = SignatureVersion.SigV4;
        }
    }
}
```

CRUD examples

Now you can run queries in your Aurora DSQL cluster.

```
using Npgsql;
using Amazon;

class Example
{
    public static async Task Run(string clusterEndpoint)
    {
        RegionEndpoint region = RegionEndpoint.USEast1;

        // Connect to a PostgreSQL database.
        const string username = "admin";
        // The token expiration time is optional, and the default value 900 seconds
        string password = TokenGenerator.GenerateAuthToken(clusterEndpoint, region);
        const string database = "postgres";
        var connString = "Host=" + clusterEndpoint + ";Username=" + username
 + ";Password=" + password + ";Database=" + database + ";Port=" + 5432 +
 ";SSLMode=VerifyFull;";

        var conn = new NpgsqlConnection(connString);
        await conn.OpenAsync();

        // Create a table.
        using var create = new NpgsqlCommand("CREATE TABLE IF NOT EXISTS owner (id
 UUID PRIMARY KEY, name VARCHAR(30) NOT NULL, city VARCHAR(80) NOT NULL, telephone
 VARCHAR(20))", conn);
        create.ExecuteNonQuery();

        // Create an owner.
        var uuid = Guid.NewGuid();
        using var insert = new NpgsqlCommand("INSERT INTO owner(id, name, city,
telephone) VALUES(@id, @name, @city, @telephone)", conn);
        insert.Parameters.AddWithValue("id", uuid);
        insert.Parameters.AddWithValue("name", "John Doe");
```

```
        insert.Parameters.AddWithValue("city", "Anytown");

        insert.Parameters.AddWithValue("telephone", "555-555-0190");

        insert.ExecuteNonQuery();

        // Read the owner.
        using var select = new NpgsqlCommand("SELECT * FROM owner where id=@id", conn);
        select.Parameters.AddWithValue("id", uuid);
        using var reader = await select.ExecuteReaderAsync();
        System.Diagnostics.Debug.Assert(reader.HasRows, "no owner found");

        System.Diagnostics.Debug.WriteLine(reader.Read());

        reader.Close();

        using var delete = new NpgsqlCommand("DELETE FROM owner where id=@id", conn);
        select.Parameters.AddWithValue("id", uuid);
        select.ExecuteNonQuery();

        // Close the connection.
        conn.Close();
    }

    public static async Task Main(string[] args)
    {
        await Run();
    }
}
```

Programming with Rust

Topics

- Using Rust to interact with Amazon Aurora DSQL

Using Rust to interact with Amazon Aurora DSQL

This section describes how how to use Rust to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL
- Configured your AWS credentials. For more information, see Set and view configuration settings using commands.
- Installed Rust. You must have version 1.8.0 or higher. To verify your version, run `rustc --version`.
- Added sqlx to your `Cargo.toml` dependencies. For example, add the following configuration to your dependencies.

```
sqlx = { version = "0.8", features = [ "runtime-tokio", "tls-native-tls" ,
 "postgres"] }
```

- Added the AWS SDK for Rust to your `Cargo.toml` file.

Connect to your Aurora DSQL cluster and run queries

```
use aws_config::{BehaviorVersion, Region};
use aws_sdk_dsql::auth_token::{AuthTokenGenerator, Config};
use rand::Rng;
use sqlx::Row;
use sqlx::postgres::{PgConnectOptions, PgPoolOptions};
use uuid::Uuid;

async fn example(cluster_endpoint: String) -> anyhow::Result<()> {
    let region = "us-east-1";

    // Generate auth token
    let sdk_config = aws_config::load_defaults(BehaviorVersion::latest()).await;
    let signer = AuthTokenGenerator::new(
        Config::builder()
            .hostname(&cluster_endpoint)
            .region(Region::new(region))
            .build()
            .unwrap(),
    );
    let password_token =
 signer.db_connect_admin_auth_token(&sdk_config).await.unwrap();

    // Setup connections
    let connection_options = PgConnectOptions::new()
        .host(cluster_endpoint.as_str())
```

```
        .port(5432)
        .database("postgres")
        .username("admin")
        .password(password_token.as_str())
        .ssl_mode(sqlx::postgres::PgSslMode::VerifyFull);

    let pool = PgPoolOptions::new()
        .max_connections(10)
        .connect_with(connection_options.clone())
        .await?;

    // Create owners table
    // To avoid Optimistic concurrency control (OCC) conflicts
    // Have this table created already.
    sqlx::query(
        "CREATE TABLE IF NOT EXISTS owner (
   id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
   name VARCHAR(255),
   city VARCHAR(255),
   telephone VARCHAR(255)
  )").execute(&pool).await?;

    // Insert some data
    let id = Uuid::new_v4();
    let telephone = rand::thread_rng()
        .gen_range(123456..987654)
        .to_string();
    let result = sqlx::query("INSERT INTO owner (id, name, city, telephone) VALUES ($1,
 $2, $3, $4)")
        .bind(id)
        .bind("John Doe")
        .bind("Anytown")
        .bind(telephone.as_str())
        .execute(&pool)
        .await?;
    assert_eq!(result.rows_affected(), 1);

    // Read data back
    let rows = sqlx::query("SELECT * FROM owner WHERE id=
$1").bind(id).fetch_all(&pool).await?;
    println!("{:?}", rows);

    assert_eq!(rows.len(), 1);
    let row = &rows[0];
```

```
    assert_eq!(row.try_get::<&str, _>("name")?, "John Doe");
    assert_eq!(row.try_get::<&str, _>("city")?, "Anytown");
    assert_eq!(row.try_get::<&str, _>("telephone")?, telephone);

    // Delete some data
    sqlx::query("DELETE FROM owner WHERE name='John Doe'")
        .execute(&pool).await?;

    pool.close().await;
    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let cluster_endpoint = "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws";
    Ok(example(cluster_endpoint).await?)
}
```

Programming with Golang

Topics

- Using Go with Amazon Aurora DSQL

Using Go with Amazon Aurora DSQL

This section describes how how to use Go to interact with Aurora DSQL.

Before you begin, make sure that you have completed the following prerequisites.

- Created a cluster in Aurora DSQL
- Installed Go. To verify that you have installed Go, run `go version`.
- Installed the latest version of AWS SDK for Go.
- Installed the PostgreSQL Go driver with `go get`.

```
go get github.com/jackc/pgx/v5
```

Connect to your Aurora DSQL cluster

Use the following example to generate password token to connect to your Aurora DSQL cluster.

```
import (
 "context"
 "fmt"
 "net/http"
 "os"
 "strings"
 "time"

 _ "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go/aws/credentials"
 "github.com/aws/aws-sdk-go/aws/session"
 v4 "github.com/aws/aws-sdk-go/aws/signer/v4"
 "github.com/google/uuid"
 "github.com/jackc/pgx/v5"
 _ "github.com/jackc/pgx/v5/stdlib"
)

type Owner struct {
 Id        string `json:"id"`
 Name      string `json:"name"`
 City      string `json:"city"`
 Telephone string `json:"telephone"`
}

const (
 REGION = "us-east-1"
)

func GenerateDbConnectAdminAuthToken(creds *credentials.Credentials, clusterEndpoint
 string) (string, error) {
 // the scheme is arbitrary and is only needed because validation of the URL requires
 one.
 endpoint := "https://" + clusterEndpoint
 req, err := http.NewRequest("GET", endpoint, nil)
 if err != nil {
  return "", err
 }
 values := req.URL.Query()
 values.Set("Action", "DbConnectAdmin")
 req.URL.RawQuery = values.Encode()
```

```
 signer := v4.Signer{
  Credentials: creds,
 }
 _, err = signer.Presign(req, nil, "dsql", REGION, 15*time.Minute, time.Now())
 if err != nil {
  return "", err
 }

 url := req.URL.String()[len("https://"):]

 return url, nil
}
```

Now we can write code to connect to your Aurora DSQL cluster.

```
func getConnection(ctx context.Context, clusterEndpoint string) (*pgx.Conn, error) {
 // Build connection URL
 var sb strings.Builder
 sb.WriteString("postgres://")
 sb.WriteString(clusterEndpoint)
 sb.WriteString(":5432/postgres?user=admin&sslmode=verify-full")
 url := sb.String()

 sess, err := session.NewSession()
 if err != nil {
  return nil, err
 }

 creds, err := sess.Config.Credentials.Get()
 if err != nil {
  return nil, err
 }
 staticCredentials := credentials.NewStaticCredentials(
  creds.AccessKeyID,
  creds.SecretAccessKey,
  creds.SessionToken,
 )

 // The token expiration time is optional, and the default value 900 seconds
 // If you are not connecting as admin, use DbConnect action instead
 token, err := GenerateDbConnectAdminAuthToken(staticCredentials, clusterEndpoint)
 if err != nil {
```

```
  return nil, err
 }

 connConfig, err := pgx.ParseConfig(url)
 // To avoid issues with parse config set the password directly in config
 connConfig.Password = token
 if err != nil {
  fmt.Fprintf(os.Stderr, "Unable to parse config: %v\n", err)
  os.Exit(1)
 }

 conn, err := pgx.ConnectConfig(ctx, connConfig)

 return conn, err
}
```

CRUD examples

Now you can run queries in your Aurora DSQL cluster.

```
func example(clusterEndpoint string) error {
 ctx := context.Background()

 // Establish connection
 conn, err := getConnection(ctx, clusterEndpoint)
 if err != nil {
  return err
 }

 // Create owner table
 _, err = conn.Exec(ctx, `
  CREATE TABLE IF NOT EXISTS owner (
   id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
   name VARCHAR(255),
   city VARCHAR(255),
   telephone VARCHAR(255)
  )
 `)
 if err != nil {
  return err
 }

 // insert data
```

```
 query := `INSERT INTO owner (id, name, city, telephone) VALUES ($1, $2, $3, $4)`
 _, err = conn.Exec(ctx, query, uuid.New(), "John Doe", "Anytown", "555-555-0150")

 if err != nil {
  return err
 }

 owners := []Owner{}
 // Define the SQL query to insert a new owner record.
 query = `SELECT id, name, city, telephone FROM owner where name='John Doe'`

 rows, err := conn.Query(ctx, query)
 defer rows.Close()

 owners, err = pgx.CollectRows(rows, pgx.RowToStructByName[Owner])
 fmt.Println(owners)
 if err != nil || owners[0].Name != "John Doe" || owners[0].City != "Anytown" {
  panic("Error retrieving data")
 }

 // Delete some data
 _, err = conn.Exec(ctx, `DELETE FROM owner where name='John Doe'`)
 if err != nil {
  return err
 }

 defer conn.Close(ctx)

 return nil
}

func main() {
 cluster_endpoint := "foo0bar1baz2quux3quuux4.dsql.us-east-1.on.aws";
 err := example(cluster_endpoint)
 if err != nil {
  fmt.Fprintf(os.Stderr, "Unable to run example: %v\n", err)
  os.Exit(1)
 }
}
```

Utilities, tutorials, and sample code in Amazon Aurora DSQL

AWS documentation includes several tutorials that guide you through common Aurora DSQL use cases. Many of these tutorials show you how to use Aurora DSQL with other tools and AWS services. Many of these examples contain sample code that you can access on GitHub.

> **Note**
>
> You can find more tutorials at AWS Database Blog and re:Post.

Tutorials and sample code on GitHub

> **Note**
>
> The links to GitHub repositories might not work until December 4, 2024.

The following tutorials and sample code on GitHub help you performance common tasks in Aurora DSQL.

- Using Benchbase with Aurora DSQL – a branch of the Benchbase open-source benchmarking utility that is verified to work with Aurora DSQL.
- Aurora DSQL loader – this open-source Python script makes it easier for you to load data into Aurora DSQL for your use cases, such as populating tables for testing or transferring data into Aurora DSQL.
- Aurora DSQL samples – `aws-samples/aurora-dsql-samples` repository on GitHub contains code examples of how to connect and use Aurora DSQL in various programming languages using the AWS SDKs, object-relational mappers (ORMs), and web frameworks. The examples demonstrate how to perform common tasks, such as install clients, handle authentication, and perform CRUD operations.

Using Aurora DSQL with the 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 you to build applications as a developer in your preferred language.

- AWS CLI
- AWS SDK for Python (Boto3)
- AWS SDK for JavaScript
- AWS SDK for Java 2.x
- AWS SDK for C++

Using AWS Lambda with Amazon Aurora DSQL

The following sections describe how to use Lambda with Aurora DSQL

Prerequisites

- Authorization to create Lambda functions. For more information, see Getting started with Lambda.
- Authorization to create or modify IAM policy created by Lambda. You need to permissions `iam:CreatePolicy` and `iam:AttachRolePolicy`. For more information, see Actions, resources, and condition keys for IAM .
- You must have installed npm v8.5.3 or higher.
- You must have installed zip v3.0 or higher.

Create a new function in AWS Lambda.

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Create function**.
3. Provide a name, such as `dsql-sample`.
4. Don't edit the default settings to make sure that Lambda creates a new role with basic Lambda permissions.
5. Choose **Create function**.

Authorize your Lambda execution role to connect to your cluster

1. In your Lambda function, choose **Configuration** > **Permissions**.
2. Choose the **role name** to open the execution role in the IAM console.
3. Choose **Add Permissions** > **Create inline policy**, and use the JSON editor.
4. In *Action* paste in the following action to authorize your IAM identity to connect using the admin database role.

```
"Action": ["dsql:DbConnectAdmin"],
```

> **Note**
>
> We're using an admin role to minimize prerequisite steps to get started. You shouldn't use a admin database role for your production applications. See Using database roles with IAM roles to learn how to create custom database roles with authorization that has the fewest permissions to your database.

5. In **Resource**, add your cluster's Amazon Resource Name (ARN). You can also use a wildcard.

```
"Resource": ["*"]
```

6. Choose **Next**.
7. Enter a name for the policy, such as `dsql-sample-dbconnect`.
8. Choose **Create policy**.

Create a package to upload to Lambda.

1. Create a folder named `myfunction`.
2. In the folder, create a new file named `package.json` with the following content.

```
{
  "dependencies": {
    "@aws-sdk/core": "^3.587.0",
    "@aws-sdk/credential-providers": "^3.587.0",
    "@smithy/protocol-http": "^4.0.0",
    "@smithy/signature-v4": "^3.0.0",
    "pg": "^8.11.5"
  }
```

```
}
```

3. In the folder, create a file named `index.mjs` in the directory with the following content.

```
import { formatUrl } from "@aws-sdk/util-format-url";
import { HttpRequest } from "@smithy/protocol-http";
import { SignatureV4 } from "@smithy/signature-v4";
import { fromNodeProviderChain } from "@aws-sdk/credential-providers";
import { NODE_REGION_CONFIG_FILE_OPTIONS, NODE_REGION_CONFIG_OPTIONS } from
 "@smithy/config-resolver";
import { Hash } from "@smithy/hash-node";
import { loadConfig } from "@smithy/node-config-provider";
import pg from "pg";
const { Client } = pg;

export const getRuntimeConfig = (config) => {
  return {
    runtime: "node",
    sha256: config?.sha256 ?? Hash.bind(null, "sha256"),
    credentials: config?.credentials ?? fromNodeProviderChain(),
    region: config?.region ?? loadConfig(NODE_REGION_CONFIG_OPTIONS,
 NODE_REGION_CONFIG_FILE_OPTIONS),
    ...config,
  };
};

// Aurora DSQL requires IAM authentication
// This class generates auth tokens signed using AWS Signature Version 4
export class Signer {
  constructor(hostname) {
    const runtimeConfiguration = getRuntimeConfig({});

    this.credentials = runtimeConfiguration.credentials;
    this.hostname = hostname;
    this.region = runtimeConfiguration.region;

    this.sha256 = runtimeConfiguration.sha256;
    this.service = "dsql";
    this.protocol = "https:";
  }

  async getAuthToken() {
    const signer = new SignatureV4({
      service: this.service,
```

```
      region: this.region,
      credentials: this.credentials,
      sha256: this.sha256,
    });

    // To connect with a custom database role, set Action as "DbConnect"
    const request = new HttpRequest({
      method: "GET",
      protocol: this.protocol,
      hostname: this.hostname,
      query: {
        Action: "DbConnectAdmin",
      },
      headers: {
        host: this.hostname,
      },
    });

    const presigned = await signer.presign(request, {
      expiresIn: 3600,
    });

    // RDS requires the scheme to be removed
    // https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/
UsingWithRDS.IAMDBAuth.Connecting.html
    return formatUrl(presigned).replace(`${this.protocol}//`, "");
  }
}

// To connect with a custom database role, set user as the database role name
async function dsql_sample(token, endpoint) {
  const client = new Client({
    user: "admin",
    database: "postgres",
    host: endpoint,
    password: token,
    ssl: {
      rejectUnauthorized: false
    },
  });

  await client.connect();
  console.log("[dsql_sample] connected to Aurora DSQL!");
```

```
  try {
    console.log("[dsql_sample] attempting transaction.");
    await client.query("BEGIN; SELECT txid_current_if_assigned(); COMMIT;");
    return 200;
  } catch (err) {
    console.log("[dsql_sample] transaction attempt failed!");
    console.error(err);
    return 500;
  } finally {
    await client.end();
  }
}

// https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html
export const handler = async (event) => {
  const endpoint = event.endpoint;
  const s = new Signer(endpoint);
  const token = await s.getAuthToken();
  const responseCode = await dsql_sample(token, endpoint);

  const response = {
    statusCode: responseCode,
    endpoint: endpoint,
  };
  return response;
};
```

4. Use the following commands to create a package.

```
npm install
zip -r pkg.zip .
```

Upload the code package and test your Lambda function

1. In your Lambda function's **Code** tab, choose **Upload from** > **.zip file**
2. Upload the `pkg.zip` you created. For more information, see Deploy Node.js Lambda functions with .zip file archives.
3. In your Lambda function's **Test** tab, paste in the following JSON payload, and modify it to use your cluster ID.

4. In your Lambda function's **Test** tab, use the following *Event JSON* modified to specify your cluster's endpoint.

```
{"endpoint": "replace_with_your_cluster_endpoint"}
```

5. Enter an Event name, such as `dsql-sample-test`. Choose **Save**.
6. Choose **Test**.
7. Choose **Details** to expand the execution response and log output.
8. If it succeeded, the Lambda function execution response should return a 200 status code:

```
{statusCode": 200, "endpoint": "your_cluster_endpoint"}
```

If the database returns an error or if the connection to the database fails, the Lambda function execution response returns a 500 status code.

```
{"statusCode": 500,"endpoint": "your_cluster_endpoint"}
```

Security in Amazon Aurora DSQL

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures 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 DSQL, 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 company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Aurora DSQL. The following topics show you how to configure Aurora DSQL to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Aurora DSQL resources.

Topics

- Cross-service confused deputy prevention
- Security best practices for Amazon Aurora DSQL

AWS managed policies for Amazon Aurora DSQL

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: AmazonAuroraDSQLFullAccess

You can attach `AmazonAuroraDSQLFullAccess` to your users, groups, and roles.

This policy grants permissions that allows full administrative access to Aurora DSQL. Principals with these permissions can create, delete, and update Aurora DSQL clusters, including multi-Region clusters. They can add and remove tags from clusters. They can list clusters and view information about individual clusters. They can see tags attached to Aurora DSQL clusters. They can connect to the database as any user, including admin. They can see any metrics from CloudWatch on your account. They also have permissions to create service-linked roles for the `dsql.amazonaws.com` service, which is required for creating clusters.

Permissions details

This policy includes the following permissions.

- `dsql` – grants principals full access to Aurora DSQL.
- `cloudwatch` – grants permission to publish metric data points to Amazon CloudWatch.
- `iam` – grants permission to create a service-linked role.

You can find the `AmazonAuroraDSQLFullAccess` policy on the IAM console and AmazonAuroraDSQLFullAccess in the AWS Managed Policy Reference Guide.

AWS managed policy: AmazonAuroraDSQLReadOnlyAccess

You can attach `AmazonAuroraDSQLReadOnlyAccess` to your users, groups, and roles.

Allows read access to Aurora DSQL. Principals with these permissions can list clusters and view information about individual clusters. They can see the tags attached to Aurora DSQL clusters. They can retrieve and see any metrics from CloudWatch on your account.

Permissions details

This policy includes the following permissions.

- `dsql` – grants read only permissions to all resources in Aurora DSQL.
- `cloudwatch` – grants permission to retrieve batch amounts of CloudWatch metric data and perform metric math on retrieved data

You can find the `AmazonAuroraDSQLReadOnlyAccess` policy on the IAM console and AmazonAuroraDSQLReadOnlyAccess in the AWS Managed Policy Reference Guide.

AWS managed policy: AmazonAuroraDSQLConsoleFullAccess

You can attach `AmazonAuroraDSQLConsoleFullAccess` to your users, groups, and roles.

Allows full administrative access to Amazon Aurora DSQL via the AWS Management Console. Principals with these permissions can create, delete, and update Aurora DSQL clusters, including multi-Region clusters, with the console. They can list clusters, view information about individual clusters. They can see tags on any resource on your account. They can connect to the database as any user, including the admin. They can see any metrics from CloudWatch on your account. They also have permissions to create service linked roles for the `dsql.amazonaws.com` service, which is required for creating clusters.

You can find the `AmazonAuroraDSQLConsoleFullAccess` policy on the IAM console and AmazonAuroraDSQLConsoleFullAccess in the AWS Managed Policy Reference Guide.

Permissions details

This policy includes the following permissions.

- `dsql` – grants full administrative permissions to all resources in Aurora DSQL via the AWS Management Console.
- `cloudwatch` – grants permission to retrieve batch amounts of CloudWatch metric data and perform metric math on retrieved data
- `tag` – grants permission to returns tag keys and values currently in use in the specified AWS Region for the calling account

You can find the `AmazonAuroraDSQLReadOnlyAccess` policy on the IAM console and AmazonAuroraDSQLReadOnlyAccess in the AWS Managed Policy Reference Guide.

AWS managed policy: AuroraDSQLServiceRolePolicy

You can't attach AuroraDSQLServiceRolePolicy to your IAM entities. This policy is attached to a service-linked role that allows Aurora DSQL to access account resources.

You can find the `AuroraDSQLServiceRolePolicy` policy on the IAM console and AuroraDSQLServiceRolePolicy in the AWS Managed Policy Reference Guide.

Aurora DSQL updates to AWS managed policies

View details about updates to AWS managed policies for Aurora DSQL since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Aurora DSQL Document history page.

Change	Description	Date
Page created	Started tracking managed policies for AWS managed policies related to Amazon Aurora DSQL	December 3, 2024

Data protection in Amazon Aurora DSQL

The AWS shared responsibility model applies to data protection in Amazon Aurora DSQL. 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. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- 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 140-3 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) 140-3.

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 Aurora DSQL 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.

Data encryption

Amazon Aurora DSQL provides a highly durable storage infrastructure designed for mission-critical and primary data storage. Data is redundantly stored on multiple devices across multiple facilities in a Aurora DSQL Region.

Encryption at rest

By default, Aurora DSQL configures encryption at rest for you.

Aurora DSQL owned keys

Aurora DSQL owned keys are not stored in your AWS account. They are part of a collection of KMS keys that Aurora DSQL owns and manages for encrypting data in your clusters. Aurora DSQL uses envelop encryption to encrypt data. These keys are rotated every year (approximately 365 days).

You are not charged a monthly fee or a usage fee for use of AWS owned keys, and they do not count against AWS KMS quotas for your account.

Customer managed keys

Aurora DSQL doesn't support customer-managed keys for encrypting data in your clusters.

Encryption in transit

By default, encryption in transit is configured for you. Aurora DSQL uses mutual TLS (mTLS) to encrypt all traffic between your SQL client and Aurora DSQL.

Encryption and signing of data in transit between AWS CLI, SDK, or API clients and Aurora DSQL endpoints:

- Aurora DSQL provides HTTPS endpoints for encrypting data in transit.
- To protect the integrity of API requests to Aurora DSQL, API calls must be signed by the caller. Calls are signed by an X.509 certificate or the customer's AWS secret access key according to the Signature Version 4 Signing Process (Sigv4). For more information, see Signature Version 4 Signing Process in the *AWS General Reference*.
- Use the AWS CLI or one of the AWS SDKs to make requests to AWS. These tools automatically sign the requests for you with the access key that you specify when you configure the tools.

Key management

See the following to learn about how Aurora DSQL manages keys.

- Aurora DSQL automatically integrates with AWS Key Management Service (AWS KMS) for key management. AWS KMS uses envelope encryption. For more information, see Envelope Encryption.
- Aurora DSQL doesn't support customer managed keys. Aurora DSQL uses multi-tenant keys, with each key being in separate AWS Regions.

Inter-network traffic privacy

Connections are protected both between Aurora DSQL and on-premises applications and between Aurora DSQL and other AWS resources within the same AWS Region.

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 Aurora DSQL 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.

Identity and access management for Amazon Aurora DSQL

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 Aurora DSQL 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 that you do in Aurora DSQL.

Service user – If you use the Aurora DSQL service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Aurora DSQL 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 DSQL, see Troubleshooting Amazon Aurora DSQL identity and access.

Service administrator – If you're in charge of Aurora DSQL resources at your company, you probably have full access to Aurora DSQL. It's your job to determine which Aurora DSQL features and resources your service users should access. You must then submit requests to your IAM 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 DSQL, see How Amazon Aurora DSQL works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Aurora DSQL. To view example Aurora DSQL identity-based policies that you can use in IAM, see Identity-based policy examples for Amazon Aurora DSQL.

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 AWS Signature Version 4 for 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 AWS Multi-factor authentication in IAM 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 Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). 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 Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **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 Create a role for a third-party identity provider (federation) 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*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **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 Cross account resource access in IAM 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 (FAS)** – 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 Create 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 Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) 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*.

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**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

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 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 Define custom IAM permissions with customer managed 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 user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, 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 Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

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.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

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.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

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 (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user 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 Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) 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 user 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 DSQL works with IAM

Before you use IAM to manage access to Aurora DSQL, learn what IAM features are available to use with Aurora DSQL.

IAM features you can use with Amazon Aurora DSQL

IAM feature	Aurora DSQL support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how Aurora DSQL and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Aurora DSQL

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 Define custom IAM permissions with customer managed 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 DSQL

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Resource-based policies within Aurora DSQL

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 DSQL

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.

To see a list of Aurora DSQL actions, see Actions Defined by Amazon Aurora DSQL in the *Service Authorization Reference*.

Policy actions in Aurora DSQL use the following prefix before the action:

```
dsql
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "dsql:action1",
      "dsql:action2"
]
```

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Policy resources for Aurora DSQL

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": "*"
```

To see a list of Aurora DSQL resource types and their ARNs, see Resources Defined by Amazon Aurora DSQL in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by Amazon Aurora DSQL .

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

Policy condition keys for Aurora DSQL

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*.

To see a list of Aurora DSQL condition keys, see Condition Keys for Amazon Aurora DSQL in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions Defined by Amazon Aurora DSQL .

To view examples of Aurora DSQL identity-based policies, see Identity-based policy examples for Amazon Aurora DSQL.

ACLs in Aurora DSQL

Supports 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.

ABAC with Aurora DSQL

Supports ABAC (tags in policies): Partial

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 Define permissions with ABAC authorization 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*.

Using temporary credentials with Aurora DSQL

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 Switch from a user to an IAM 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.

Cross-service principal permissions for Aurora DSQL

Supports forward access sessions (FAS): 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 DSQL

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 Create 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 DSQL functionality. Edit service roles only when Aurora DSQL provides guidance to do so.

Service-linked roles for Aurora DSQL

Supports service-linked roles: No

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 creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon Aurora DSQL

By default, users and roles don't have permission to create or modify Aurora DSQL resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Aurora DSQL, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Amazon Aurora DSQL in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the Aurora DSQL console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Aurora DSQL 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 Validate policies with IAM Access Analyzer 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 Secure API access with MFA 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 DSQL console

To access the Amazon Aurora DSQL console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Aurora DSQL 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 they're trying to perform.

To ensure that users and roles can still use the Aurora DSQL console, also attach the Aurora DSQL `AmazonAuroraDSQLConsoleFullAccess` or `AmazonAuroraDSQLReadOnlyAccess` AWS managed policy to the entities. 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": "*"
        }
    ]
}
```

Troubleshooting Amazon Aurora DSQL identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Aurora DSQL and IAM.

Topics

- I am not authorized to perform an action in Aurora DSQL
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Aurora DSQL resources

I am not authorized to perform an action in Aurora DSQL

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `dsql:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 dsql:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `dsql:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Aurora DSQL.

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 an IAM user named `marymajor` tries to use the console to perform an action in Aurora DSQL. However, the action requires the service to have permissions that are 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's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Aurora DSQL 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 DSQL supports these features, see How Amazon Aurora DSQL 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 Cross account resource access in IAM in the *IAM User Guide*.

Using service-linked roles in Aurora DSQL

Aurora DSQL 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 Aurora DSQL. Service-linked roles are predefined by Aurora DSQL and include all the permissions that the service requires to call AWS services on behalf of your Aurora DSQL cluster.

Service-linked roles make the setup process easier because you don't have to manually add the necessary permissions to use Aurora DSQL. When you create a cluster, Aurora DSQL automatically creates a service-linked role for you. You can delete the service-linked role only after you delete all of your clusters. This protects your Aurora DSQL resources because you can't inadvertently remove permissions needed for access to 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 roles are available in all supported Aurora DSQL Regions.

Service-linked role permissions for Aurora DSQL

Aurora DSQL uses the service-linked role named `AWSServiceRoleForAuroraDsql` – Allows Amazon Aurora DSQL to create and manage AWS resources on your behalf. This service-linked role is attached to the following managed policy: AuroraDSQLServiceRolePolicy.

Create a service-linked role

You don't need to manually create an AuroraDSQLServiceRolePolicy service-linked role. Aurora DSQL creates the service-linked role for you. If the AmazonAuroraDSQLServiceRolePolicy service-linked role has been deleted from your account, Aurora DSQL creates the role when you create a new Aurora DSQL cluster.

Edit a service-linked role

Aurora DSQL doesn't allow you to edit the AuroraDSQLServiceRolePolicy service-linked role. After you create a service-linked role, you can't change the name of the role because various entities might reference the role. However, you can edit the description of the role using the IAM console, the AWS Command Line Interface (AWS CLI), or IAM API.

Delete a service-linked role

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.

Before you can delete a service-linked role for an account, you must delete any clusters in the account.

You can use the IAM console, the AWS CLI, or the IAM API to delete a service-linked role. For more information, see Create a service-linked role in the IAM User Guide.

Supported Regions for Aurora DSQL service-linked roles

Aurora DSQL supports using service-linked roles in all of the Regions where the service is available. For more information, see AWS Regions and endpoints.

Using IAM condition keys with Amazon Aurora DSQL

When you grant permissions in Aurora DSQL you can specify conditions that determine how a permissions policy takes effect. The following are examples of how you can use condition keys in Aurora DSQL permissions policies.

Example 1: Grant permission to create a cluster in a specific AWS Region

The following policy grants permission to create clusters in the US East (N. Virginia) and US East (Ohio) Regions. This policy uses the resource ARN to limit the allowed Regions, so Aurora DSQL can only create clusters only if that ARN is specified in the `Resource` section of the policy.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            # Control where clusters can be created
            "Action": ["CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        }
    ]
}
```

Example 2: Grant permission to create a multi-Region cluster in specific AWS Regions.

The following policy grants permission to create multi-Region clusters in the US East (N. Virginia) and US East (Ohio) Regions. This policy uses the resource ARN to limit the allowed Regions, so Aurora DSQL can only create multi-Region clusters only if that ARN is specified in the `Resource` section of the policy. Note that creating multi-Region clusters also requires `CreateCluster` permission in each specified Region.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {

            "Action": ["CreateMultiRegionClusters"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
        },
        {
            "Action": ["CreateCluster"],
            "Resource": [
                "arn:aws:dsql:us-east-1:*:cluster/*",
                "arn:aws:dsql:us-east-2:*:cluster/*"
            ],
            "Effect": "Allow"
```

```
        }
    ]
}
```

Example 3: Grant permission to create a multi-Region cluster with a specific witness Region.

The following policy uses an Aurora DSQL `dsql:WitnessRegion` condition key and lets a user create multi-Region clusters with a witness Region in US West (Oregon). If you don't specify the `dsql:WitnessRegion` condition, you can use any Region as the witness Region.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": ["CreateMultiRegionClusters"],
            "Resource": "*",
            "Effect": "Allow",
            "Condition": {
                "StringEquals": {
                    "dsql:WitnessRegion": ["us-west-2"]
                }
            }
        },
        {
            "Action": ["CreateCluster"],
            "Resource": "*",
            "Effect": "Allow"
        }
    ]
}
```

Incident response in Amazon Aurora DSQL

Security is the highest priority at AWS. As part of the AWS Cloud shared responsibility model, AWS manages a data center, network, and software architecture that meets the requirements of the most security-sensitive organizations. AWS is responsible for any incident response with respect to the Amazon Aurora DSQL service itself. Also, as an AWS customer, you share a responsibility for maintaining security in the cloud. This means that you control the security you choose to

implement from the AWS tools and features you have access to. In addition, you're responsible for incident response on your side of the shared responsibility model.

By establishing a security baseline that meets the objectives for your applications running in the cloud, you're able to detect deviations that you can respond to. To help you understand the impact that incident response and your choices have on your corporate goals, we encourage you to review the following resources:

- AWS Security Incident Response Guide
- AWS Best Practices for Security, Identity, and Compliance
- Security Perspective of the AWS Cloud Adoption Framework (CAF) whitepaper

Amazon GuardDuty is a managed threat detection service continuously monitoring malicious or unauthorized behavior to help customers protect AWS accounts and workloads and identify suspicious activity potentially before it escalates into an incident. It monitors activity such as unusual API calls or potentially unauthorized deployments indicating possible account or resource compromise or reconnaissance by bad actors. For example, Amazon GuardDuty is able to detect suspicious activity in Amazon Aurora DSQL APIs, such as a user logging in from a new location and creating a new cluster.

Compliance validation for Amazon Aurora DSQL

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. 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 AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config 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.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon Aurora DSQL

The AWS global infrastructure is built around AWS Regions and Availability Zones (AZ). 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 zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures. Aurora DSQL is designed so that you can

take advantage of AWS Regional infrastructure while providing the highest database availability. By default, single-Region clusters in Aurora DSQL have Multi-AZ availability, providing tolerance to major component failures and infrastructure disruptions that might impact access to a full AZ. Multi-Region clusters provide all of the benefits from Multi-AZ resiliency while still providing the strongly consistent database availability, even in cases in which AWS Region is inaccessible to application clients.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Aurora DSQL offers several features to help support your data resiliency and backup needs.

Backup and restore

During preview, Aurora DSQL doesn't support backup and restore.

Aurora DSQL plans to support backup and restore with AWS Backup console, so you can perform a full backup and restore for your single-Region and multi-Region clusters. What is AWS Backup.

Replication

By design, Aurora DSQL commits all write transactions to a distributed transaction log and synchronously replicates all committed log data to user storage replicas in three AZs. Multi-Region clusters provide full cross-Region replication capabilities between read and write Regions. A designated witness Region supports transaction log-only writes and doesn't use any storage. Witness Regions don't have an endpoint. This means that witness Regions store only encrypted transaction logs, require no administration or configuration, and aren't accessible by users.

Aurora DSQL transaction logs and user storage are distributed across with all data presented to Aurora DSQL query processors as a single logical volume. Aurora DSQL automatically splits, merges, and replicates data based on database primary key range and access patterns. Aurora DSQL automatically scales read replicas, both up and down, based on read access frequency.

Cluster storage replicas are distributed across a multi-tenant storage fleet. If a component or AZ becomes impaired, Aurora DSQL automatically redirects access to surviving components and asynchronously repairs missing replicas. Once Aurora DSQL fixes the impaired replicas, Aurora DSQL automatically adds them back to the storage quorum and makes them available to your cluster.

High availability

By default, single-Region and multi-Region clusters in Aurora DSQL are active-active, and you don't need to manually provision, configure, or reconfigure any clusters. Aurora DSQL fully automates cluster recovery, which eliminates the need for traditional primary-secondary failover operations. Replication is always synchronous and done in multiple AZs, so there is no risk of data loss due to replication lag or failover to an asynchronous secondary database during failure recovery.

Single-Region clusters provide a Multi-AZ redundant endpoint that automatically enables concurrent access with strong data consistency across three AZs. This means that user storage replicas on any of these three AZs always return the same result to one or more readers and are always available to receive writes. This strong consistency and Multi-AZ resiliency is available across all Regions for Aurora DSQL multi-Region clusters. This means that multi-Region clusters provide two strongly consistent Regional endpoints, so clients can read or write indiscriminately to either Region with zero replication lag on commit. Aurora DSQL doesn't provide a managed global endpoint for multi-Region clusters, but you can use Amazon Route 53 as a substitute.

Aurora DSQL provides 99.99% availability for single-Region clusters and 99.999% for multi-Region clusters.

Infrastructure Security in Amazon Aurora DSQL

As a managed service, Amazon Aurora DSQL is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Aurora DSQL through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support 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.

Configuration and vulnerability analysis in Amazon Aurora DSQL

AWS handles basic security tasks like guest operating system (OS) and database patching, firewall configuration, and disaster recovery. These procedures have been reviewed and certified by the appropriate third parties. For more details, see the following resources:

- Shared responsibility model
- Amazon Web Services: Overview of security processes (whitepaper)

Cross-service confused deputy prevention

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.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon Aurora DSQL gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

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. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:`*`servicename`*`:*:`*`123456789012`*`:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of `aws:SourceArn` must be ResourceDescription.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Aurora DSQL to prevent the confused deputy problem.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
      "Service": "servicename.amazonaws.com"
    },
    "Action": "servicename:ActionName",
    "Resource": [
      "arn:aws:servicename:::ResourceName/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:servicename:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

Security best practices for Amazon Aurora DSQL

Aurora DSQL provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Use IAM roles to authenticate access to Aurora DSQL

Any users, applications, and other AWS services that access Aurora DSQL must include valid AWS credentials in AWS API and AWS CLI requests. You shouldn't store AWS credentials directly in the application or EC2 instances. These are long-term credentials that aren't automatically rotated. There is significant business impact if these credentials are compromised. An IAM role lets you obtain temporary access keys that you can use to access AWS services and resources.

For more information, see Understanding authentication and authorization for Aurora DSQL.

Use IAM policies for Aurora DSQL base authorization

When you grant permissions, you decide who is getting them, which Aurora DSQL API operations they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM roles and grant permissions to perform operations on Aurora DSQL resources. Also available arepermissions boundaries for IAM entities, which let you set the maximum permissions that an identity-based policy can grant to an IAM entity.

Similar to the root user best practices for your AWS account, don't use the admin role in Aurora DSQL to perform everyday operations. Instead, we recommend that you create custom database roles to manage and connect to your cluster. For more information, see Accessing Aurora DSQL and Understanding authentication and authorization for Aurora DSQL.

Tag your Aurora DSQL resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they let you categorize resources by purpose, owner, environment, or other criteria. The following are some examples.

- Security – used to determine requirements such as encryption.
- Confidentiality – an identifier for the specific data-confidentiality level a resource supports.
- Environment – used to distinguish between development, test, and production infrastructure.

For more information, see Best Practices for Tagging AWS Resources.

Topics

- Detective security best practices for Aurora DSQL
- Preventative security best practices for Aurora DSQL

Detective security best practices for Aurora DSQL

In addition to the following ways to securely use Aurora DSQL, see Security in AWS Well-Architected Tool to learn about how cloud technologies improve your security.

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.

Tag your Aurora DSQL resources for identification and automation

You can assign metadata to your AWS resources in the form of tags. Each tag is a simple label consisting of a customer-defined key and an optional value that can make it easier to manage, search for, and filter resources.

Tagging allows for grouped controls to be implemented. Although there are no inherent types of tags, they enable you to categorize resources by purpose, owner, environment, or other criteria. The following are some examples:

- Security – Used to determine requirements such as encryption.
- Confidentiality – An identifier for the specific data-confidentiality level a resource supports.
- Environment – Used to distinguish between development, test, and production infrastructure.

For more information, see AWS Tagging Strategies.

Preventative security best practices for Aurora DSQL

In addition to the following ways to securely use Aurora DSQL, see Security in AWS Well-Architected Tool to learn about how cloud technologies improve your security.

Use IAM roles to authenticate access to Aurora DSQL

For users, applications, and other AWS services to access Aurora DSQL, they must include valid AWS credentials in their AWS API requests. You should not store AWS credentials directly in the application or EC2 instance. These are long-term credentials that are not automatically rotated,

and therefore could have significant business impact if they are compromised. An IAM role lets you obtain temporary access keys that can be used to access AWS services and resources.

For more information, see Understanding authentication and authorization for Aurora DSQL.

Use IAM policies for Aurora DSQL base authorization

When granting permissions, you decide who is getting them, which Aurora DSQL API operations they are getting permissions for, and the specific actions you want to allow on those resources. Implementing least privilege is key in reducing security risk and the impact that can result from errors or malicious intent.

Attach permissions policies to IAM roles and thereby grant permissions to perform operations on Aurora DSQL resources. Also available are permissions boundaries for IAM entities, which let you set the maximum permissions that an identity-based policy can grant to an IAM entity.

Similar to the root user best practices for your AWS account, don't use the admin role in Aurora DSQL to perform everyday operations. Instead, we recommend that you create custom database roles to manage and connect to your cluster. For more information, see *Accessing Aurora DSQL* and *Authentication and authorization*.

Tagging resources in Aurora DSQL

In AWS, tags are user-defined key-value pairs that you define and associate with Aurora DSQL resources such as clusters. Tags are optional. If you provide a key, the value is optional.

You can use the AWS Management Console, the AWS CLI, or the AWS SDKs to add, list, and delete tags on Aurora DSQL clusters. You can add tags during and after cluster creation using the AWS console. To tag a cluster after creation with the AWS CLI use the `TagResource` operation.

Tagging clusters with a Name

Aurora DSQL creates clusters with a globally unique identifier assigned as the Amazon Resource Name (ARN). If you want to assign a user friendly name to your cluster, we recommend that you use a Tag.

If you create a console with the Aurora DSQL console, Aurora DSQL automatically creates a tag. This tag has a key of **Name** and an automatically generated value that represents the name of the cluster. This value is configurable, so you can assign a more friendly name to your cluster. If a cluster has a Name tag with an associated value, you can see the value throughout the Aurora DSQL console.

Tagging requirements

Tags have the following requirements:

- Keys can't be prefixed with `aws:`.
- Keys must be unique per tag set.
- A key must be between 1 and 128 allowed characters.
- A value must be between 0 and 256 allowed characters.
- Values do not need to be unique per tag set.
- Allowed characters for keys and values are Unicode letters, digits, white space, and any of the following symbols: _ . : / = + - @.
- Keys and values are case sensitive.

Tagging usage notes

When using tags in Aurora DSQL, consider the following.

- When using the AWS CLI or Aurora DSQL API operations, make sure to provide the Amazon Resource Name (ARN) for the Aurora DSQL resource to work with. For more information, see Amazon Resource Name (ARNs) format for Aurora DSQL resources.
- Each resource has one tag set, which is a collection of one or more tags assigned to the resource.
- Each resource can have up to 50 tags per tag set.
- If you delete a resource, any associated tags are deleted.
- You can add tags when you create a resource, you can view and modify tags using the following API operations: `TagResource`, `UntagResource`, and `ListTagsForResource`.
- You can use tags with IAM policies. You can use them to manage access to Aurora DSQL clusters and to control what actions can be applied to those resources. To learn more, see Controlling access to AWS resources using tags.
- You can use tags for various other activities across AWS. To learn more, see Common tagging strategies.

Known issues in Amazon Aurora DSQL

The following list contains known issues with Amazon Aurora DSQL

- Aurora DSQL doesn't complete COUNT(*) operations before transaction timeout for large tables. To retrieve table row count from the system catalog, see Using systems tables and commands in Aurora DSQL.
- While you can run the `CREATE INDEX` command without the `ASYNC` option and insert data before index creation and commit the transaction, doing so might result in missing index entries. We recommend that you use the `ASYNC` option when you create indices. If you use `CREATE INDEX`, confirm that Aurora DSQL has finished creating the index before you `INSERT` into the table.
- Aurora DSQL doesn't currently let you run `GRANT [permission] ON DATABASE`. If you attempt to run that statement, Aurora DSQL returns the error message `ERROR: unsupported object type in GRANT`.
- Aurora DSQL doesn't let non-admin user roles to run the `CREATE SCHEMA` command. You can't run the `GRANT [permission] on DATABASE` command and grant `CREATE` permissions on the database. If a non-admin user role tries to create a schema, Aurora DSQL returns with the error message `ERROR: permission denied for database postgres`.
- Aurora DSQL doesn't support the `FLUSH` command. Aurora DSQL returns transaction errors to any client using the `FLUSH` command. For example, Aurora DSQL doesn't complete queries that use `FLUSH` to run pipelined queries. Most clients use the `SYNC` command to get results over a synchronization point, but not every client does. For example, psycopg3 driver uses `FLUSH` to pipeline queries. The pipelined querying mode from psycopg3 doesn't work with Aurora DSQL.
- Drivers calling `PG_PREPARED_STATEMENTS` might provide an inconsistent view of cached prepared statements for the cluster. You might see more than the expected number of prepared statements per connection for the same cluster and IAM role. Aurora DSQL doesn't preserve statement names that you prepare.
- Clients running on IPv4 only instances might see an incorrect error if connection establishment fails. Some PostgreSQL clients resolve a hostname to both the IPv4 and IPv6 addresses if the server supports dualstack mode and supports connecting to both addresses if the first connection fails. For example, if connecting to the IPv4 address fails because of throttling errors, clients might use IPv6 to connect. If the host doesn't support IPv6 connections, it returns a `NetworkUnreachable` error. However, the underlying cause of the error might be that the host doesn't support IPv6.

- After an Aurora DSQL admin user creates a new schema, it's possible that subsequent `GRANT` and `REVOKE` commands from non-admin users don't reflect for existing cluster connections. This issue can last for the maximum duration of a connection of one hour.
- In rare multi-Region linked-cluster impairment scenarios, it might take longer than expected for transaction commit availability to resume. In general, automated cluster recovery operations can result in transient concurrency control or ocnnection errors. In most cases, you will only see the effects for a percentage of your workload. When you see these transit errors, retry your transaction or reconnect with your client.
- Some SQL clients, such as Datagrip, make expansive calls to system metadata to populate schema information. Aurora DSQL doesn't support all of this information and returns errors. This issue doesn't affect SQL query functionality, but it might affect schema display.
- Aurora DSQL doesn't support nested transactions that rely on savepoints. This impacts the PsycoPG3 driver and tools that utilize nested transactions. We recommend that you use the PsycoPG2 driver.
- You might see the error `Schema Already Exists` if you try to create a schema, but you recently dropped the schema in another transaction. This error occurs because of a stale catalog cache. The workaround is to disconnect and reconnect.
- Queries might fail to recognize newly created schemas and tables and incorrectly report that they don't exist. This error occurs because of a stale catalog cache. The workaround is the disconnect and reconnect.
- An obsolete search path can make it so that Aurora DSQL doesn't discover new objects. Setting a search path to a schema that doesn't exist prevents Aurora DSQL from discovering that schema if you created it in another connection. The workaround is to set the search path again after you create the schema.
- Transactions that contain a query plan with a nested loop join above a merge join can consume more memory than intended and result in an out-of-memory condition.
- Non-admin users can't create objects in the public schema. Only admin users can crete objects in the public schema. The admin user role has permissions to grant read, write, and modify access to these objects to non-admin users, but it cannot grant `CREATE` permissions to the public schema itself. Non-admin users must use different, user-created schemas for object creation.
- Aurora DSQL doesn't support the command `ALTER ROLE [] CONNECTION LIMIT`. Contact AWS support if you need a connection limit increase.
- The admin role has a set of permissions related to database management tasks. By default, these permissions don't extend to objects that other users create. The admin role can't grant or revoke

permissions on these user-created objects to other users. The admin user can grant itself any other role to get the necessary permissions on these objects.

- Aurora DSQL creates the admin role with all new Aurora DSQL clusters. Currently, this role lacks permissions on objects that other users create. This limitation prevents the admin role from granting or revoking permissions on objects that the admin role didn't create.

Cluster quotas and database limits in Amazon Aurora DSQL

The following sections describe the cluster quotas and database limits relevant to Aurora DSQL.

Cluster quotas

Your AWS account has the following cluster quotas in Aurora DSQL.

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
Maximum single-Region clusters per AWS account.	20	Yes	N/A	You have reached the cluster limit.
Maximum multi-Region clusters per AWS account.	5	Yes	N/A	N/A
Maximum storage GB per cluster.	100GB	Yes	DISK_FULL (53100)	Current cluster size exceeds cluster size limit.
Maximum connections per cluster.	1000	Yes	TOO_MANY_CONNECTIONS(53300)	Unable to accept connection, too many open connections.
Maximum connection rate per cluster.	(10, 100)	Yes	CONFIGURED_LIMIT_EXCEEDED(53400)	Unable to accept connection, rate exceeded.

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
Maximum connection duration	60 minutes	No	N/A	N/A

Database limits in Aurora DSQL

The following table describes all database limits in Aurora DSQL.

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
Maximum combined size of the columns used in a primary key	1 Kibibyte	No	54000	ERROR: key size too large
Maximum combined size of the columns in a secondary index	1 Kibibyte	No	54000	ERROR: key size too large
Maximum size of a row in a table	2 Mebibytes	No	54000	ERROR: maximum row size exceeded
Maximum size of a column used in a primary key or secondary index	255 Bytes	No	54000	ERROR: maximum key column size exceeded
Maximum size of a column that	1 Mebibyte	No	54000	ERROR: maximum

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
is not part of an index				column size exceeded
Maximum number of columns that can be used by included in a primary key or a secondary index	8 Column Keys per Primary Key or Index	No	54011	ERROR: more than 8 column keys in an index are not supported
Maximum number of columns in a table	255 Columns per Table	No	54011	ERROR: tables can have at most 255 columns
Maximum number of indexes that can be created for a single table	24	No	54000	ERROR: more than 24 indexes per table are not allowed
Maximum size of all data modified within a write transaction	10 MiB Transacti on Size	No	54000	ERROR: transacti on size limit 10mb exceeded DETAIL: Current transaction size <sizemb> 10mb

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
Maximum number of table and index rows that can be mutated in a single transacti on block	10K rows per transaction, modified by number of secondary indexes	No	54000	ERROR: transacti on row limit exceeded
The base maximum amount of memory to be used by a query operation.	128 MiB per Transaction	No	53200	ERROR: query requires too much temp space, out of memory.
Maximum number of schemas defined within a database	10 Schemas	No	54000	ERROR: more than 10 schemas not allowed
Maximum number of tables that can be created within a database	1000 Tables	No	54000	ERROR: creating more than 1000 tables not allowed
Maximum databases per cluster.	1	No		ERROR: unsupported statement
Maximum transaction time	5 minutes	No	54000	ERROR: transacti on age limit of 300s exceeded

Description	Default Limit	Configurable?	Aurora DSQL error code	Error message
Maximum connection duration	1 hour	No		

Aurora DSQL API reference

In addition to the AWS Management Console and the AWS Command Line Interface (AWS CLI), Aurora DSQL also provides an API interface. You can use the API operations to manage your resources in Aurora DSQL.

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 Aurora DSQL.

Troubleshooting issues in Aurora DSQL

The following topics provide troubleshooting advice for errors and issues that you might encounter when using Aurora DSQL. If you find an issue that is not listed here, please reach out to AWS-distributed-sql-advisory@amazon.com.

Topics

Troubleshooting authentication errors

IAM authentication failed for user "..."

When you generate an Aurora DSQL IAM authentication token, the maximum duration you can set is 1 week. After one week, you can't authenticate with that token.

Additionally, Aurora DSQL rejects your connection request if your assumed role has expired. For example, if you try to connect with a temporary IAM role even if your authentication token hasn't expired, Aurora DSQL will reject the connection request.

To learn more about how IAM works with Aurora DSQL, see Understanding authentication and authorization for Aurora DSQL and AWS Identity and Access Management in Aurora DSQL.

An error occurred (InvalidAccessKeyId) when calling the GetObject operation: The AWS Access Key ID you provided does not exist in our records

IAM rejected your request. For more information, see Why requests are signed.

IAM role <role> does not exist

Aurora DSQL couldn't find your IAM role. For more information, see IAM roles.

IAM role must look like an IAM ARN

See IAM Identifiers - IAM ARNs for more information.

Troubleshooting authorization errors

Role <role> not supported

Aurora DSQL doesn't support the `GRANT` operation. See Supported subsets of PostgreSQL commands in Aurora DSQL.

Cannot establish trust with role <role>

Aurora DSQL doesn't support the `GRANT` operation. See Supported subsets of PostgreSQL commands in Aurora DSQL.

Role <role> does not exist

Aurora DSQL couldn't find specified database user. See Authorize custom database roles to connect to a cluster.

ERROR: permission denied to grant IAM trust with role <role>

To grant access to a database role, you must be connected to your cluster with the admin role. To learn more, see Authorize database roles to use SQL in a database.

ERROR: role <role> must have the LOGIN attribute

Any database roles you create must have the `LOGIN` permission.

To address this error, make sure that you've created the PostgreSQL Role with the `LOGIN` permission. For more information, see CREATE ROLE and ALTER ROLE in the PostgreSQL documentation.

ERROR: role <role> cannot be dropped because some objects depend on it

Aurora DSQL returns an error if you drop a database role with an IAM relationship until you revoke the relationship using `AWS IAM REVOKE`. To learn more, see Revoking authorization.

Troubleshooting SQL errors

Error: Not supported

Aurora DSQL doesn't support all PostgreSQL-based dialect. To learn about what is supported, see Supported PostgreSQL features in Aurora DSQL .

Error: SELECT FOR UPDATE in a read-only transaction is a no-op

You are attempting an operation that isn't allowed in a read-only transaction. To learn more, see Understanding concurrency control in Aurora DSQL.

Error: use `CREATE INDEX ASYNC` instead

To create an index on a table with existing rows, you must use the `CREATE INDEX ASYNC` command. To learn more, see Creating indexes asynchronously in Aurora DSQL.

Troubleshooting OCC errors

OC000 “ERROR: mutation conflicts with another transaction, retry as needed”

OC001 “ERROR: schema has been updated by another transaction, retry as needed”

Your PostgreSQL session had a cached copy of the schema catalog. That cached copy was valid at the time was loaded. Let’s call the time T1 and the version V1.

Another transaction updates the catalog at time T2. Let’s call this V2.

When the original session attempts to read from storage at time T2 it’s still using catalog version V1. Aurora DSQL’s storage layer rejects the request because the latest catalog version at T2 is V2.

When you retry at time T3 from the original session, Aurora DSQL refreshes the catalog cache. The transaction at T3 is using catalog V2. Aurora DSQL will finish the transaction as long as no other catalog changes came through since time T2.

Document history for the Amazon Aurora DSQL User Guide

The following table describes the documentation releases for Aurora DSQL.

Change	Description	Date
Initial release	Initial release of the Amazon Aurora DSQL User Guide.	December 3, 2024