Tuning PostgreSQL parameters in Amazon RDS and Amazon Aurora

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Tuning PostgreSQL parameters in Amazon RDS and Amazon Aurora

Table of Contents

Tuning PostgreSQL parameters in Amazon RDS and Amazon Aurora

Sumana Yanamandra, Ramu Jagini, and Rohit Kapoor, Amazon Web Services (AWS)

February 2024 (document history)

Amazon Aurora PostgreSQL-Compatible Edition and Amazon Relational Database Service (Amazon RDS) for PostgreSQL are sophisticated open-source relational database services that offer a full range of features. You can use these services to set up your PostgreSQL database on a variety of platforms and applications.

Aurora and Amazon RDS offer a simplified way to manage and operate your PostgreSQL databases. They are designed to manage the database infrastructure and provide high availability, durability, and scalability while you focus on application development. However, the default configurations of these services might not be optimal for all workloads. By default, these services are configured to run everywhere with the fewest resources possible and without introducing vulnerabilities. Tuning the parameters can help you achieve better performance, reduce downtime, and improve overall database efficiency. By optimizing the parameters for your specific workload, you can take full advantage of the capabilities that Amazon RDS and Aurora provide and maximize their benefits.

For example, you can improve performance by optimizing Aurora and Amazon RDS for PostgreSQL and configuring their parameters. You should also take performance into consideration when you create database queries. Even when you optimize database settings, the system can perform badly if your queries conduct complete table scans, when they utilize an index, or if they run expensive join or aggregate operations.

This guide is for database developers, database engineers, and administrators who want to tune memory, autovacuum, logging, and logical replication parameters for their PostgreSQL databases. The guide also covers parameters that are specific to Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible. Tuning these parameters can help you optimize database performance and reduce resource usage for your specific workload, resulting in better performance and cost savings.

Using DB and DB cluster parameter groups

Amazon RDS and Aurora can automatically determine the most suitable parameter values for certain settings based on your database instance size. They also support parameter customization for performance tuning through parameter groups for database instances and clusters.

You can use DB and DB cluster parameter groups to modify the parameters that control various aspects of the database engine's behavior, such as memory usage, disk I/O, networking, and locking. By adjusting these parameters, you can optimize the database engine for your specific workload and improve performance.

You can create and configure DB and DB cluster parameter groups by using the AWS Management Console, the AWS Command Line Interface (AWS CLI), or the Amazon RDS API. This guide assumes that you're using the AWS CLI. For console and API instructions, see Working with DB parameter groups and Working with DB cluster parameter groups in the Amazon RDS documentation.

> **Important**
>
> To use the AWS CLI commands provided in this guide, you must first install and configure the AWS CLI.

To create and configure a DB parameter group:

```
# Create a new DB parameter group
aws rds create-db-parameter-group \
  --db-parameter-group-name mydbparamgroup \
  --db-parameter-group-family postgres13 \
  --description "My DB Parameter Group"

# Modify a parameter on the DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <param group name> \
  --parameters "ParameterName=max_connections<parameter-
name>,ParameterValue=<value>,ApplyMethod=immediate"

# Verify DB parameters
aws rds describe-db-parameters \
  --db-parameter-group-name aurora-instance-1
```

To create and configure a DB cluster parameter group:

```
# Create a new DB cluster parameter group
aws rds create-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --db-parameter-group-family postgres12 \
  --description "My new parameter group"

# Modify a parameter on the DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name aws-guide-cluster \
  --parameters "ParameterName=<parameter-name>,ParameterValue=,ApplyMethod=immediate"

# Allocate the new DB cluster parameter to your cluster
aws rds modify-db-cluster  \
  --db-cluster-identifier  \
  --db-cluster-parameter-group-name=-cluster

# Verify cluster parameters
aws rds describe-db-cluster-parameters \
  --db-cluster-parameter-group-name=-cluster
```

> **Note**
>
> Aurora and Amazon RDS provide a default parameter group with preconfigured values that cannot be changed.

Parameter groups can be set as static or dynamic. Dynamic parameters are applied immediately, regardless of whether the `ApplyMethod=immediate` option is enabled. Static parameters require a manual reboot to take effect.

Tuning memory parameters

Tuning memory parameters is an essential task for optimizing the performance of Amazon RDS and Aurora PostgreSQL-Compatible databases. Properly allocating memory for various database operations such as running queries, sorting, indexing, and caching can significantly improve database performance. This section covers some of the most critical memory parameters in Amazon RDS and Aurora PostgreSQL-Compatible, including their default values, formulas for calculating appropriate values, and how to change them. For a full list of parameters, see Working with parameters on your RDS for PostgreSQL DB instance in the Amazon RDS documentation and Amazon Aurora PostgreSQL parameters in the Aurora documentation.

Optimizing these parameters requires a deep understanding of your database workload, as well as the resources available on your Aurora or Amazon RDS DB instance. System performance is influenced by two broad categories of parameters: vital parameters and contingent parameters.

Vital parameters are indispensable parameters that exert a significant and direct influence on the system's performance and are essential to achieving optimal outcomes.

- shared_buffers
- temp_buffers
- effective_cache_size
- work_mem
- maintenance_work_mem

Contingent parameters are scenario and business-specific parameters. They are contingent on other factors and play an indirect yet pivotal role in supporting vital parameters and maximizing overall system performance.

- random_page_cost
- seq_page_cost
- track_activity_query_size
- idle_in_transaction_session_timeout
- statement_timeout
- search_path
- max_connections

These parameters are discussed in more detail in the following sections.

shared_buffers

The `shared_buffers` parameter controls the amount of memory that PostgreSQL uses to cache data in memory. Setting this parameter to an appropriate value can help improve query performance.

For Amazon RDS, the default value for `shared_buffers` is set to `{DBInstanceClassMemory/32768}` bytes, based on the available memory for the DB instance. For Aurora, the default value is set to `{DBInstanceClassMemory/12038,-50003}`, based on the available memory for the DB instance. The optimal value for this parameter depends on several factors, including the size of the database, the number of concurrent connections, and the available instance memory.

> **Note**
>
> When you scale down a DB instance, make sure to adjust `shared_buffers` to match the new memory limits. Otherwise, PostgreSQL might fail to start or you might encounter issues with the PostgreSQL engine. Document all changes for easier adjustments in the future.

AWS CLI syntax

The following command changes `shared_buffers` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify shared_buffers on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=shared_buffers,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify shared_buffers on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters "ParameterName=shared_buffers,ParameterValue=<new-
value>,ApplyMethod=immediate"
```

Type: Static (applying changes requires a reboot)

Default value: `{DBInstanceClassMemory/32768}` bytes in Amazon RDS for PostgreSQL, `{DBInstanceClassMemory/12038,-50003}` in Aurora PostgreSQL-Compatible. In most cases, this equation works out to be about 25 percent of your system's memory. Following this guideline, the `shared_buffers` setting in the parameter group is set up by using PostgreSQL's default units of 8K buffers instead of bytes or kilobytes.

The `shared_buffers` parameter setting can have a significant impact on performance, so we recommend that you test your changes thoroughly to ensure that the value is appropriate for your workload.

Example

Let's say you have a financial services application that is running a PostgreSQL database on Amazon RDS or Aurora. This database is used to store customer transaction data. It has a large number of tables and is accessed by multiple applications on a large number of servers. The application is experiencing slow query performance and high CPU usage. You determine that tuning the `shared_buffers` parameter might help improve performance.

In Amazon RDS for PostgreSQL, the default value of `shared_buffers` is set to `{DBInstanceClassMemory/32768}` bytes of available memory in `db.r5.xlarge` (for example, 3 GB). To determine the appropriate value for `shared_buffers`, you run a series of tests with varying values of `shared_buffers`, starting with the default value of available memory and increasing the value gradually. For each test, you measure the query performance and CPU usage of the database.

Based on the test results, you determine that setting the value of `shared_buffers` to 8 GB results in the best overall query performance and CPU usage for your workload. The value is determined through a combination of testing and analysis of workload characteristics, including the size of the database, the number and complexity of queries, the number of concurrent users, and available system resources. After you make the change, your monitoring systems check the performance of the database to ensure that the new value is appropriate for your workload. You can then fine-tune additional parameters as necessary to further improve performance.

temp_buffers

`temp_buffers` is a key configuration parameter in Aurora PostgreSQL-Compatible and Amazon RDS for PostgreSQL that can significantly impact performance for workloads that involve sorts,

hashes, and aggregate operations on temporary tables. This parameter determines the amount of memory allocated for temporary buffers, which, in turn, affects the efficiency and speed of such operations. If there isn't enough memory allocated for `temp_buffers`, the system might have to use slower, less efficient methods for sorts, hashes, and aggregation operations on temporary tables, leading to suboptimal performance.

AWS CLI syntax

The following command changes `temp_buffers` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify temp_buffers on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=temp_buffers,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify temp_buffers on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=temp_buffers,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 8 MB

For more information about this parameter, see Resource Consumption in the PostgreSQL documentation.

Example

If your workload involves a lot of sorting, hashing, and aggregate operations on temporary tables, `temp_buffers` might not allocate enough memory. In this case, the system might have to perform sort operations on temporary tables that lead to slower, disk-based methods instead of in-memory sorting, hashing, and aggregate operations. This can cause a significant slowdown in query performance, especially for queries that involve large datasets. Increasing the value of `temp_buffers` can ensure that there's enough memory available to perform such operations in memory, leading to a significant improvement in performance.

To find the optimal value of `temp_buffers`, monitor your system performance and identify areas of suboptimal performance. If you notice slow query response times or high CPU utilization,

consider adjusting `temp_buffers`. For example, if your workload involves a lot of temporary tables, increasing the value of `temp_buffers` can help ensure that these tables are stored in memory. This can be much faster than using read/write I/O from storage.

Experiment with different values of `temp_buffers` in small increments and carefully monitor system performance after each change. Analyze the impact of different values on performance and fine-tune the setting based on the specific characteristics of your workload.

effective_cache_size

The `effective_cache_size` parameter specifies the amount of memory that PostgreSQL should assume is available for caching data. Setting this parameter correctly can improve performance by allowing PostgreSQL to make better use of the available memory.

AWS CLI syntax

The following command changes `effective_cache_size` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify effective_cache_size on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=effective_cache_size,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify effective_cache_size on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=effective_cache_size,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `SUM(DBInstanceClassMemory/12038,-50003)` KB

Example

An online learning platform has a large database of course materials, student data, and other content that is frequently accessed by users. The application runs on an Amazon RDS for PostgreSQL `db.r5.xlarge` instance with 32 GB of memory. The application experiences slow performance when users try to read frequently accessed content. After analyzing the database

server's resource usage, you determine that PostgreSQL isn't making optimal use of the available memory.

The `effective_cache_size` parameter in Amazon RDS for PostgreSQL controls the amount of memory used by the server for disk caching. The default value is set to `SUM({DBInstanceClassMemory/12038},-50003)` KB for instance class `db.r5.xlarge`, but this default might not be appropriate for all workloads. In this example, the database server might be storing large amounts of frequently accessed course materials and student data. Increasing the value of the `effective_cache_size` parameter can result in more data being cached in memory, which reduces the number of disk reads required and improves query performance.

When you run a query, Amazon RDS for PostgreSQL first checks whether the data required by the query is already in the cache. If so, the data can be read from memory instead of being read from disk. If the data isn't in the cache, it has to be read from disk, which can be a slow operation.

For the online learning platform, you might decide to set `effective_cache_size` to 16 GB (half of the available memory) after testing and analysis. This value allows PostgreSQL to make better use of the available memory, which reduces the number of disk reads required and improves query performance.

work_mem

The `work_mem` parameter controls the amount of memory that is used by queries for sorting and hashing operations. Its default value is 4 MB. If a query includes multiple operations, it can use up to 4 MB for each operation. Increasing the value of `work_mem` can improve the performance of queries that require sorting or hashing, because these operations require more memory. However, setting this parameter too high can cause excessive memory usage, leading to performance degradation.

To calculate the optimal value for `work_mem`, you can use the following formula:

```
work_mem = (available_memory / (max_connections * work_mem_fraction))
```

where `available_memory` is the total amount of memory available on the server, max_connections is the maximum number of connections allowed, and work_mem_fraction is a fraction that determines how much of the available memory should be allocated to each connection.

AWS CLI syntax

The following command changes `work_mem` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify work_mem on a DB parameter group
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify work_mem on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
   --db-cluster-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 4 MB

Example

A social media analytics tool processes a large amount of data, and queries that involve complex sort and join operations cause high disk I/O and spill to disk. If you increase the value of `work_mem` from 4 MB to 16 MB, PostgreSQL can use more memory for these operations. This reduces the amount of I/O and improves query performance.

maintenance_work_mem

The `maintenance_work_mem` parameter controls the amount of memory used by maintenance operations such as `VACUUM`, `ANALYZE`, and index creation. The default value for this parameter in Amazon RDS and Aurora is 64 MB.

To calculate the appropriate value for this parameter, you can use this formula:

```
maintenance_work_mem = (total_memory - shared_buffers) / (max_connections * 5)
```

Aurora PostgreSQL-Compatible Edition and Amazon RDS for PostgreSQL apply the following formula to set the optimal value:

```
GREATEST({DBInstanceClassMemory/63963136*1024},65536)
```

AWS CLI syntax

The following command changes `maintenance_work_mem` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify maintenance_work_mem on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=maintenance_work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify maintenance_work_mem on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=maintenance_work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 64 MB

Example

Your large-scale application uses a PostgreSQL database that's hosted on Aurora or Amazon RDS. You notice that the database is slow and unresponsive during maintenance activities such as vacuuming and indexing. You can monitor metrics such as memory usage, maintenance operation times, and CPU usage to determine if the current value of `maintenance_work_mem` is causing issues.

To determine the optimal value for `maintenance_work_mem`, you can adjust the parameter and monitor its impact. If memory usage is consistently high or operation times are longer than expected during maintenance operations, increasing `maintenance_work_mem` might help. Conversely, if CPU usage is consistently high during maintenance operations, decreasing `maintenance_work_mem` might help. By iterating through adjustments and testing, you can find the optimal value for `maintenance_work_mem` that provides the best balance for memory usage, maintenance operation times, and CPU usage.

During your investigation, let's say you determine that the default value of 64 MB for `maintenance_work_mem` is too low for your database size. As a result, maintenance operations take longer to complete, cause excessive downtime, and slow down your application's performance. To address this issue, you might decide to tune the `maintenance_work_mem` parameter by

increasing it from 64 MB to 512 MB (which you identify as the optimal value). Applying the change can improve your maintenance operation times by two thirds. For example, a vacuum operation that previously took 30 minutes to complete might now take only 10 minutes. As a result of this optimization, your database can now handle maintenance activities more efficiently.

random_page_cost

The `random_page_cost` parameter helps determine the cost of performing random page access. The query planner in Amazon RDS and Aurora uses this parameter, along with other statistics about the table, to determine the most efficient plan for running a query.

The `seq_page_cost` and `random_page_cost` parameters are closely related and usually used together by the planner to compare the costs of different access methods and decide which one is the most efficient. Therefore, if you change one of these parameters, you should also consider whether the other parameter needs to be adjusted.

In general, the query planner tries to minimize the cost of running a query. The cost is determined by using a combination of the number of disk page reads and the value of `random_page_cost`. A higher value of `random_page_cost` tends to favor sequential scans whereas a lower value tends to favor index scans. A lower value also tends to favor nested loop joins instead of hash joins.

The `random_page_cost` parameter uses the default PostgreSQL engine value (4) unless a value is set in the parameter group or in the local session. You can tune this value depending on the specific characteristics of your server and workload. If most of the indexes used in your workload fit in memory or in the Aurora tiered cache, changing the value of `random_page_cost` to a value that's close to `seq_page_cost` is appropriate.

AWS CLI syntax

The following command changes `random_page_cost` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify random_page_cost on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=random_page_cost,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify random_page_cost on a DB cluster parameter group
```

```
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=random_page_cost,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 4

Example

Let's say you have a database that stores a large amount of data in a table that's frequently queried with filters on non-indexed columns. The queries take a long time to complete, and the query planner doesn't select the most efficient plan for accessing the data.

One way to improve performance would be to decrease the `random_page_cost` parameter. If you set it to 1, the cost of random page access would be four times less expensive than the default value. If you left `random_page_cost` at its default value of 4, random page access would be four times more expensive than sequential page access (as determined by the `seq_page_cost` parameter, which is 1.0 by default). However, in this specific case, random page access might actually be much more expensive depending on the storage type.

Decreasing the value of the `random_page_cost` parameter can make the query planner more likely to select an index-based plan or use a different access method that's better suited to the specific characteristics of the table.

We recommend that you monitor query performance after you change the parameter and make adjustments as needed. You should also check the query planner with an `EXPLAIN` statement to check whether it's selecting an efficient plan.

This is just one example. The optimal setting depends on the specific characteristics of your workload. Also, this is just one aspect of performance tuning; you should also consider other parameters and configuration options that can affect your query performance.

seq_page_cost

The `seq_page_cost` parameter helps determine the cost of performing sequential page access. The query planner in Amazon RDS and Aurora uses this parameter, along with other statistics about the table, to determine the most efficient plan for running a query.

The `seq_page_cost` and `random_page_cost` parameters are closely related and usually used together by the planner to compare the costs of different access methods and decide which one is the most efficient. Therefore, if you change one of these parameters, you should also consider whether the other parameter needs to be adjusted.

When a table is accessed in a sequential manner, PostgreSQL can use the operating system's file system cache to provide faster access. By default, `seq_page_cost` is set to 1.0, which assumes that sequential page access is as cheap as a single disk block read. If you have a table that is primarily accessed in a sequential manner but disk access is slow because it's limited by fewer IOPS, you might want to increase the value of `seq_page_cost` to reflect the additional cost of accessing the disk.

Changing the value of this parameter affects all queries that run in the system, so we recommend that you test your queries with different values to determine the optimal value for your specific use case.

AWS CLI syntax

The following command changes `seq_page_cost` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify seq_page_cost on a DB parameter group
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=seq_page_cost,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify seq_page_cost on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
   --db-cluster-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=seq_page_cost,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 1.0

Example

Let's say you have a database that stores a large amount of data in a table that is primarily accessed sequentially. The table is primarily used for reporting, the queries perform very slowly, and the query planner isn't able to select the most efficient plan for accessing the data.

One way to improve performance would be to decrease the `seq_page_cost` parameter. The default value is 1.0, which assumes that a sequential page access is as cheap as a single disk block read. However, in this specific case, sequential page access might actually be more expensive than that because of the storage type (depending on I/O). If you set `seq_page_cost` to 0.5, the cost of sequential page access is half as expensive as the default value. This change can make the query planner more likely to select a plan that uses a sequential access method that is better suited to the specific characteristics of the table.

We recommend that you monitor query performance after you change the parameter and make adjustments as needed. You should also check the query plan with an `EXPLAIN` statement to check whether it's selecting an efficient plan.

This is just one example. The optimal setting depends on the specific characteristics of your workload. Also, this is just one aspect of performance tuning; you should also consider other parameters and configuration options that affect your query performance.

track_activity_query_size

The `track_activity_query_size` parameter controls the size of the query string that is logged for each active session in the `pg_stat_activity` view. By default, only the first 1,024 bytes of the query string are logged in Amazon RDS for PostgreSQL, and 4,096 bytes are logged in Aurora PostgreSQL-Compatible. If you want to log longer queries, you can set this parameter to a higher value.

AWS CLI syntax

The following command changes `track_activity_query_size` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify track_activity_query_size on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=track_activity_query_size,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify track_activity_query_size on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=track_activity_query_size,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Static (applying changes requires a reboot)

Default value: 1,024 bytes (Amazon RDS for PostgreSQL), 4,096 bytes (Aurora PostgreSQL-Compatible)

Example

Your Amazon RDS for PostgreSQL database is experiencing slow query performance, and you suspect that the issue might be related to long-running queries. You can investigate further by logging longer queries to the `pg_stat_activity` view.

Increasing the value of the `track_activity_query_size` parameter can result in increased logging, which can have a performance impact on the database. We recommend that you set the parameter back to its default 1,024 value after the issue has been resolved.

idle_in_transaction_session_timeout

The `idle_in_transaction_session_timeout` parameter controls the amount of time an idle transaction waits before it's stopped.

The default value for this parameter in Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible is 86,400,000 milliseconds.

AWS CLI syntax

The following command changes `idle_in_transaction_session_timeout` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify idle_in_transaction_session_timeout on a DB parameter group
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=idle_in_transaction_session_timeout,ParameterValue=<new_value>,ApplyMethod=imme

# Modify idle_in_transaction_session_timeout on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
   --db-cluster-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=idle_in_transaction_session_timeout,ParameterValue=<new_value>,ApplyMethod=imme
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 86,400,000 milliseconds (Aurora PostgreSQL-Compatible)

Example

You have an e-commerce application that processes online orders. The application uses a PostgreSQL database that's hosted on Amazon RDS or Aurora. Whenever a customer places an order, the application starts a new transaction to update the inventory and order records.

If a transaction is left idle for a long time, it can prevent other transactions from accessing the same records, leading to performance issues and potentially even application downtime. In addition, idle transactions that aren't properly stopped can consume valuable system resources such as memory and CPU.

To avoid such issues, you can set the `idle_in_transaction_session_timeout` parameter to a value that makes sense for your application. For example, you might set it to 5 minutes (300 seconds) so that any transaction that is left idle for more than 5 minutes will be automatically stopped. This can help to ensure that system resources are used efficiently and that the application can handle a large number of orders without slowing down. By setting the appropriate value for `idle_in_transaction_session_timeout`, you can help ensure that your application performs optimally on Amazon RDS or Aurora.

statement_timeout

The `statement_timeout` parameter sets the maximum amount of time a query can run before it's stopped.

AWS CLI syntax

The following command changes `statement_timeout` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify statement_timeout on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=statement_timeout,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify statement_timeout on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
```

```
  --parameters
"ParameterName=statement_timeout,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 0 milliseconds (no timeout)

Example

Your web application allows users to search through a large database of products. The search queries can sometimes take a long time to complete, causing slow response times for users. To address this issue, you could set the `statement_timeout` parameter to a low value such as 10 seconds, which would force any query that takes longer than 10 seconds to be stopped.

This might seem like a drastic measure, but it can actually be very effective at improving performance. In many cases, long-running queries are caused by poorly optimized SQL queries or inefficient indexes. By setting a low `statement_timeout` value, you can identify these problematic queries and take steps to optimize them.

For example, suppose you discover that a particular search query is consistently timing out. You can use tools such as `EXPLAIN` and `EXPLAIN ANALYZE` to analyze the query and identify any performance bottlenecks. When you have identified the issue, you can take steps to optimize the query by adding new indexes, rewriting the query, or using a different search algorithm. By continuously analyzing and optimizing your SQL queries in this way, you can significantly improve the performance of your application.

search_path

The `search_path` parameter determines the order in which schemas are searched for objects in SQL statements. The default value is `$user, public`, which means that PostgreSQL searches for objects first in the schema that matches the user's name, and then in the public schema.

If you have a large number of schemas or you need to access objects in a specific schema, changing the `search_path` parameter can help improve performance. When you set `search_path` to a specific schema, PostgreSQL can find the objects more quickly without having to search through multiple schemas.

To change the `search_path` parameter in Amazon RDS and Aurora, you can use the following command for `ROLE` level:

```
ALTER ROLE <username> SET search_path = <schema>;
```

AWS CLI syntax

The following command changes `search_path` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify search_path on a DB parameter group
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=search_path,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify search_path on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
   --db-cluster-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=search_path,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `$user, public`

Example

You have a multi-tenant application with separate schemas for each tenant that uses an Amazon RDS for PostgreSQL or Aurora PostgreSQL-Compatible database, and you need to run a query that involves joining data from multiple schemas.

By default, Amazon RDS and Aurora use the search path to determine which schema to use for a given table. The search path is a list of schema names that PostgreSQL searches in order when you refer to a table without qualifying the schema name. By default, Amazon RDS and Aurora first look for the table in the schema that has the same name as the current user, and then look in the public schema.

Let's say that you want to run a query that involves joining tables from multiple schemas, named `tenant1`, `tenant2`, and `tenant3`. To use the tenant schemas, you can include the schema names in the query:

```
SELECT *
```

```
FROM tenant1.table1
JOIN tenant2.table2 ON tenant1.table1.id = tenant2.table2.id
JOIN tenant3.table3 ON tenant2.table2.id = tenant3.table3.id;
```

However, a more efficient method is to change the `search_path` parameter to include the tenant schemas by using the commands in the *AWS CLI syntax* section. You can also use the `SET` command in a PostgreSQL session:

```
SET search_path = tenant1, tenant2, tenant3, public;
```

You can then write the query without qualifying the schema names:

```
SELECT *
FROM table1
JOIN table2 ON table1.id = table2.id
JOIN table3 ON table2.id = table3.id;
```

This can make your query more concise and easier to read, and it can also simplify your application code if you have many queries that involve joining tables from multiple schemas.

max_connections

The `max_connections` parameter sets the maximum number of concurrent connections for your PostgreSQL database.

> **Note**
>
> When you scale down a DB instance, make sure to adjust `max_connections` to match the new memory limits. Otherwise, PostgreSQL might fail to start or you might encounter issues with the PostgreSQL engine. Document all changes for easier adjustments in the future.

AWS CLI syntax

The following command changes `max_connections` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify max_connections on a DB parameter group
```

```
aws rds modify-db-parameter-group \
--db-parameter-group-name <parameter_group_name> \
--parameters
 "ParameterName=max_connections,ParameterValue=<new_value>,ApplyMethod=pending-reboot"

# Modify max_connections on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
--db-cluster-parameter-group-name <parameter_group_name> \
--parameters
 "ParameterName=max_connections,ParameterValue=<new_value>,ApplyMethod=pending-reboot"
```

Type: Static (applying changes requires a reboot)

Default value: `LEAST(DBInstanceClassMemory/9531392,5000)` connections

To optimize the use of `max_connections` in Amazon RDS or Aurora and minimize its impact on performance, consider the following best practices:

- Set the parameter value based on available system resources.
- Monitor connection usage to prevent reaching the limit quickly.
- Use connection pooling to reduce the number of connections required.
- Use Amazon RDS Proxy for connection pooling.

When you tune `max_connections` in Amazon RDS for PostgreSQL or Amazon Aurora PostgreSQL-Compatible, consider the available instance types and their allocated resources, and focus on memory and CPU capacity. Storage and I/O details are managed by AWS, so you can monitor general workload characteristics and system metrics such as `FreeableMemory` through Amazon CloudWatch or the Amazon RDS console to confirm that there's enough memory for connections. Monitor high `CPUUtilization` values, which might indicate that adjustments are needed. Avoid setting `max_connections` too high, because it can impact memory usage and potentially influence I/O indirectly. Keep in mind that each connection consumes memory. To find the right balance, slowly increase `max_connections` and see how it affects your system. Watch for signs of slower performance or higher CPU use. Check if your application still works well. Use features such as read replicas in Aurora to distribute read traffic and reduce the load on the primary instance. Review and adjust `max_connections` regularly based on observed usage patterns to ensure optimal database performance within the given resource constraints.

Tuning autovacuum parameters

Amazon RDS for PostgreSQL databases and Aurora PostgreSQL-Compatible require periodic maintenance known as *vacuuming*. Autovacuum is a built-in PostgreSQL utility that removes outdated or unnecessary data to free up space in the database. The autovacuum process runs the `VACUUM` command in the background at regular intervals.

Tuning autovacuum settings is a crucial step in maintaining the performance, stability, and availability of your Amazon RDS for PostgreSQL or Aurora PostgreSQL-Compatible database system. By adjusting the autovacuum parameters to suit your workload and database size, you can optimize the performance of the autovacuum process and reduce its impact on system resources, thus improving the overall health of your database.

In addition to adjusting the autovacuum settings, it is important to monitor the performance of your database and its components by using the tools and metrics available in Amazon RDS and Aurora. By monitoring performance metrics such as bloat, freeable space, and query run times, you can identify potential issues before they become serious problems, and take appropriate actions to resolve them.

This section discusses the following autovacuum topics and parameters:

- VACUUM and ANALYZE commands
- Checking for bloat
- autovacuum
- autovacuum_work_mem
- autovacuum_naptime
- autovacuum_max_workers
- autovacuum_vacuum_scale_factor
- autovacuum_vacuum_threshold
- autovacuum_analyze_scale_factor
- autovacuum_analyze_threshold
- autovacuum_vacuum_cost_limit

For additional information about autovacuum, see the following links:

- Understanding autovacuum in Amazon RDS for PostgreSQL environments (blog post)

- Working with the PostgreSQL autovacuum on Amazon RDS for PostgreSQL (Amazon RDS documentation)
- Parallel vacuuming in Amazon RDS for PostgreSQL and Amazon Aurora PostgreSQL (blog post)

VACUUM and ANALYZE commands

VACUUM garbage-collects and optionally analyzes a database. For most applications, it's sufficient to let the autovacuum daemon perform vacuuming. However, some administrators might want to modify database parameters for autovacuum, or supplement or replace the daemon's activities by using manually managed VACUUM commands that can be run according to a scheduler.

VACUUM reclaims storage that is occupied by dead tuples. In standard PostgreSQL operations, when tuples are deleted or made obsolete by an update, they aren't physically removed from tables until a VACUUM operation is performed. Therefore, we recommend that you run VACUUM periodically, especially on tables that are updated frequently.

Tuning VACUUM parameters is particularly important in Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible, because these managed database services have different characteristics compared to self-managed PostgreSQL databases. These differences can affect the performance of vacuum operations. Tuning VACUUM parameters is essential to optimize the use of resources and ensure that vacuum operations do not negatively affect the performance and availability of your database system.

Here are some of the parameters that you can use with the VACUUM command in Aurora PostgreSQL-Compatible and Amazon RDS for PostgreSQL:

- FULL
- FREEZE
- VERBOSE
- ANALYZE
- DISABLE_PAGE_SKIPPING
- table_name
- column_name

VACUUM ANALYZE performs a VACUUM operation followed by an ANALYZE operation for each selected table. It provides an efficient way to perform routine maintenance.

Using the VACUUM command without the FULL option reclaims space for reuse. It doesn't require an exclusive lock on the table, so you can run this command during standard reading and writing operations. However, in most cases, the command doesn't return extra space to the operating system but keeps it available for reuse within the same table. VACUUM FULL rewrites the entire contents of the table into a new disk file with no extra space, and allows unused space to be returned to the operating system. This form is much slower and requires an ACCESS EXCLUSIVE lock on each table.

For complete information about these parameters, see the PostgreSQL documentation.

In Aurora and Amazon RDS, autovacuum is a daemon (background utility) process that runs the VACUUM and ANALYZE commands regularly to clean up redundant data in the database and server. Even if you rely on autovacuuming, we recommend that you review and adjust the autovacuum settings discussed in the following sections to ensure optimal performance.

Checking for bloat

The following SQL query examines each table in the XML schema and identifies dead rows (tuples) that waste disk space:

```
SELECT schemaname || '.' || relname as tuplename,
    n_dead_tup,
    (n_dead_tup::float / n_live_tup::float) * 100 as pfrag
FROM pg_stat_user_tables
WHERE schemaname = 'xml' and n_dead_tup > 0 and n_live_tup > 0 order by pfrag desc;
```

If this query returns a high percentage (pfrag) of dead tuples, you can use the VACUUM command to reclaim space.

To monitor data sizes before and after transactions, run the following query in the shell after connecting to a specific database:

```
SELECT pg_size_pretty(pg_relation_size('table_name'));
```

autovacuum

You can set autovacuum globally by using the autovacuum configuration parameter, or you can change it on a per-table basis by setting the autovacuum_enabled column of the pg_class table to true or false for a specific table.

When you enable autovacuum on a table, the database server periodically scans the table for dead rows and tuples and removes them in the background, without any intervention from the database administrator. This helps to keep the table small, improve query performance, and reduce the size of backups.

AWS CLI syntax

The following command enables `autovacuum` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters "ParameterName=autovacuum,ParameterValue=true,ApplyMethod=immediate"

# Modify autovacuum on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters "ParameterName=autovacuum,ParameterValue=true,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: Enabled

You can also disable or enable autovacuum on a specific table by using **psql**:

```
ALTER TABLE <table_name> SET (autovacuum_enabled = true);
```

Too much vacuuming can affect performance, so it's important to monitor the performance of the autovacuum process as well as the performance of your database, and adjust the settings as needed.

Example

Your PostgreSQL database has a table that receives a high volume of write and delete operations. Without autovacuum, this table would eventually become filled with dead rows (that is, rows that have been marked for deletion but haven't yet been physically removed from the table). These dead rows would take up space on the disk, slow down queries, and increase the size of backups. You can enable autovacuum on the table to automatically scan for dead rows and remove them to mitigate these issues.

autovacuum_work_mem

`autovacuum_work_mem` is a PostgreSQL configuration parameter that controls the amount of memory used by the autovacuum process when it performs table maintenance tasks such as vacuuming or analysis.

In Aurora and Amazon RDS, you can adjust the value of `autovacuum_work_mem` to optimize performance.

AWS CLI syntax

The following command enables `autovacuum_work_mem` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_work_mem on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_work_mem on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_work_mem,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `GREATEST({DBInstanceClassMemory/32768},131072)` KB in Aurora PostgreSQL-Compatible, 64 MB in Amazon RDS for PostgreSQL. However, the default value might vary depending on the specific version of Amazon RDS or Aurora you're using.

Example

Your Amazon RDS for PostgreSQL database has a large table that is frequently updated. Over time, you notice that the database becomes slower, and you suspect that autovacuum is taking too long to complete.

As part of your investigation, you check the system logs, use the `pg_stat_activity` view to see which queries and processes are currently running, check the `pg_stat_user_tables` view to see statistics for each table, use the `pg_settings` view to compare the value of

autovacuum_work_mem to the available memory on the system, and monitor memory usage for spikes. After you gather this information, you can set autovacuum_work_mem to the optimal value that your workload needs. To find the right balance between memory usage and performance, you might decide to set it to one fourth of the available memory on the system. After you change the value, you monitor the performance of the database and might see that autovacuum completes much faster than before and your database performs faster overall.

autovacuum_naptime

The autovacuum_naptime parameter controls the time interval between successive runs of the autovacuum process. The default value is 15 seconds for Amazon RDS for PostgreSQL and 5 seconds for Aurora PostgreSQL-Compatible.

For example, let's say that your Amazon RDS for PostgreSQL database has a table that receives a high volume of write and delete operations. If you keep the default setting, the frequent autovacuum scans will be disruptive to this highly transactive table. If you set this parameter to a high value, there will be a longer interval between successive scans, and dead rows will be removed less frequently.

You can use autovacuum_naptime to manage the load caused by the vacuum process, especially if you have a busy server that already has a high CPU or I/O load. The longer you set the nap time, the less frequently autovacuum runs, which reduces the load on the server. However, setting autovacuum_naptime to a very high value can cause your PostgreSQL tables to grow and dead rows to accumulate, leading to a performance decrease. We recommend that you monitor the performance of the autovacuum process and adjust the autovacuum_naptime setting as needed.

AWS CLI syntax

The following command changes autovacuum_naptime for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_naptime on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_naptime,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_naptime on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
```

```
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_naptime,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 15 seconds (Amazon RDS for PostgreSQL), 5 seconds (Aurora PostgreSQL-Compatible)

autovacuum_max_workers

The `autovacuum_max_workers` parameter controls the maximum number of worker processes that the autovacuum process can create. Each worker process is responsible for vacuuming or analyzing a single table.

For example, let's say that you have a large database with many tables that are frequently updated and deleted. If you set the `autovacuum_max_workers` to a low value such as 1, only one table can be vacuumed at a time, and it takes longer for all tables to be cleaned. If you set `autovacuum_max_workers` to a high value such as 8, up to eight tables can be vacuumed simultaneously. This can make the cleaning process faster for databases that contain many tables.

AWS CLI syntax

The following command changes `autovacuum_max_workers` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_max_workers on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_max_workers,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_max_workers on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_max_workers,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Static (applying changes requires a reboot)

Default value: `GREATEST(DBInstanceClassMemory/64371566592,3)` workers

Increasing the `autovacuum_max_workers` setting can increase the load on the server, which can impact performance if you don't have enough resources. The optimal setting depends on the specific requirements of your database, its size, and the number of tables it contains. We recommend that you experiment with different values and monitor performance to find the optimal setting for your use case.

autovacuum_vacuum_scale_factor

The `autovacuum_vacuum_scale_factor` configuration parameter controls how aggressive the autovacuum process should be when vacuuming a table.

The vacuum scale factor is a fraction of the total number of tuples in a table that must be modified before autovacuum cleans the table. The default value is 0.1 (that is, 10 percent of the tuples must be modified). For example, if a table has 1,000,000 tuples, and 100,000 of those tuples are marked as dead or deleted, autovacuum vacuums the table, depending on the value of `autovacuum_vacuum_threshold` as a controlling factor.

The `autovacuum_vacuum_scale_factor` parameter helps you control how frequently the vacuum process runs. If a table receives a lot of write operations, you might want to lower the vacuum scale factor so autovacuum runs more frequently, and keep the table smaller. Conversely, if a table receives few write operations, you might want to raise the vacuum scale factor so autovacuum runs less frequently, and save resources.

AWS CLI syntax

The following command changes `autovacuum_vacuum_scale_factor` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_vacuum_scale_factor on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_vacuum_scale_factor,ParameterValue=<new_value>,ApplyMethod=immediate

# Modify autovacuum_vacuum_scale_factor on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_vacuum_scale_factor,ParameterValue=<new_value>,ApplyMethod=immediate
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 0.1

The `autovacuum_vacuum_scale_factor` parameter works in conjunction with the `autovacuum_vacuum_threshold`, `autovacuum_vacuum_cost_limit, and autovacuum_naptime` parameters. For additional information about this parameter, see the AWS blog post Understanding autovacuum in Amazon RDS for PostgreSQL environments.

autovacuum_vacuum_threshold

The `autovacuum_vacuum_threshold` parameter controls the minimum number of tuple update or delete operations that must occur on a table before autovacuum vacuums it. This setting can be useful to prevent unnecessary vacuuming on tables that do not have a high rate of these operations. The default value is 50, which is the PostgreSQL engine default, for both Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible.

For example, let's say you have a table with 100,000 rows and `autovacuum_vacuum_threshold` is set to 50. If the table receives only 49 updates or deletes, autovacuum won't vacuum it. If the table receives 50 or more updates or deletes, autovacuum will vacuum it, depending on the value of `autovacuum_vacuum_scale_factor` multiplied by the number of table rows as a controlling factor.

Setting this parameter too high can cause the table to grow and dead rows to accumulate, which can affect performance.

AWS CLI syntax

The following command changes `autovacuum_vacuum_threshold` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_vacuum_threshold on a DB parameter group
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=autovacuum_vacuum_threshold,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_vacuum_threshold on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
   --db-cluster-parameter-group-name <parameter_group_name> \
```

```
    --parameters
  "ParameterName=autovacuum_vacuum_threshold,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 50 operations

The `autovacuum_vacuum_threshold` parameter works in conjunction with the `autovacuum_vacuum_scale_factor`, `autovacuum_vacuum_cost_limit,` and `autovacuum_naptime` parameters. The optimal settings depend on the specific requirements of your database and table size.

For additional information about this parameter, see the AWS blog post Understanding autovacuum in Amazon RDS for PostgreSQL environments.

autovacuum_analyze_scale_factor

The `autovacuum_analyze_scale_factor` parameter controls how aggressive the autovacuum process should be when analyzing (collecting statistics about the distribution of data in a table).

The autovacuum process uses this parameter to calculate a threshold based on the number of tuples in a table. If the number of tuple inserts, updates, or deletes exceeds this threshold, autovacuum analyzes the table. The default value is 0.05 (that is, 5 percent of the tuples must be modified) for both Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible.

For example, let's say that your table has 1,000,000 tuples and you keep the default `autovacuum_analyze_scale_factor` value at 0.05. If the table receives 50,000 or more updates or deletes, autovacuum vacuums it, depending on the `autovacuum_analyze_threshold` value and adding the number of table rows as a controlling factor.

AWS CLI syntax

The following command changes `autovacuum_analyze_scale_factor` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_analyze_scale_factor on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
```

```
  --parameters
 "ParameterName=autovacuum_analyze_scale_factor,ParameterValue=<new_value>,ApplyMethod=immediat

# Modify autovacuum_analyze_scale_factor on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_analyze_scale_factor,ParameterValue=<new_value>,ApplyMethod=immediat
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 0.05 (5 percent)

It's essential for the query planner to collect statistics in order to make informed decisions, such as how to access the data and how to organize it, so we recommend that you monitor the performance of the autovacuum process and adjust the settings as needed to ensure that statistics are up to date.

The `autovacuum_analyze_scale_factor` parameter works in conjunction with the `autovacuum_analyze_threshold`, `autovacuum_analyze_cost_limit,` and `autovacuum_naptime` parameters. The optimal setting depends on the specific requirements of your database and table size and the frequency of updates. For additional information about this parameter, see the AWS blog post Understanding autovacuum in Amazon RDS for PostgreSQL environments.

autovacuum_analyze_threshold

The `autovacuum_analyze_threshold` parameter is similar to `autovacuum_vacuum_threshold`. It controls the minimum number of tuple inserts, updates, or deletes that must occur on a table before autovacuum analyzes it. This setting can be useful to prevent unnecessary vacuuming on tables that don't have a high rate of these operations. The default value is 50, which is the PostgreSQL engine default, for both Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible.

For example, let's say you have a table with 100,000 rows and you keep the `autovacuum_analyze_threshold` default at 50. If the table receives only 49 inserts, updates, or deletes, autovacuum won't analyzes it. If the table receives 50 or more inserts, updates, or deletes, autovacuum will analyze it, keeping the value of `autovacuum_analyze_scale_factor` multiplied by the number of table rows as a controlling factor.

AWS CLI syntax

The following command changes `autovacuum_analyze_threshold` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_analyze_threshold on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_analyze_threshold,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_analyze_threshold on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_analyze_threshold,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 50 operations

This parameter works in conjunction with the `autovacuum_analyze_scale_factor` parameter, so take both settings into account when you configure autovacuum.

It's essential for the query planner to collect statistics in order to make informed decisions, such as how to access the data and how to organize it. Setting `autovacuum_analyze_threshold` too high can cause the statistics to become stale, leading to poor performance. We recommend that you monitor the performance of the autovacuum process and adjust the settings as needed.

For additional information about this parameter, see the AWS blog post Understanding autovacuum in Amazon RDS for PostgreSQL environments.

autovacuum_vacuum_cost_limit

The `autovacuum_vacuum_cost_limit` parameter controls the amount of CPU and I/O resources that an autovacuum worker can consume.

Limiting the resource usage of autovacuum processes can help prevent them from consuming too much CPU or disk I/O, which might impact the performance of other queries that run on the same system. The parameter specifies a *cost limit*, which is a unit of work that the worker is allowed

to perform before it must pause and check to see if it's still under the limit. For example, if the parameter is set to 2,000, a worker is allowed to process 2,000 units of work before pausing.

You can set the `autovacuum_vacuum_cost_limit` parameter by using the `SET` command in a PostgreSQL session, or use an AWS CLI command.

AWS CLI syntax

The following command changes `autovacuum_vacuum_cost_limit` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify autovacuum_vacuum_cost_limit on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_vacuum_cost_limit,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify autovacuum_vacuum_cost_limit on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=autovacuum_vacuum_cost_limit,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `GREATEST({log(DBInstanceClassMemory/21474836480)*600},200)` units of work

If you set the value of `autovacuum_vacuum_cost_limit` too high, the autovacuum process might consume too many resources and slow down other queries. If you set it too low, the autovacuum process might not reclaim enough space, which causes the table to become larger over time. It's essential to find the right balance that works for your system.

This parameter affects only the autovacuum process, not the manual `VACUUM` commands. Also, it only applies to the autovacuum processes for `VACUUM` but not for `ANALYZE`.

Tuning logging parameters

Tuning logging parameters in PostgreSQL helps ensure that you are collecting the right information without producing large logs that overwhelm your system.

Optimizing logging parameters is crucial for balancing log details with system performance and disk usage. You can customize the following logging parameters to capture the appropriate level of detail in the logs, to diagnose issues, and to investigate incidents effectively while minimizing the impact on system performance and disk usage.

- rds.force_autovacuum_logging
- rds.force_admin_logging_level
- log_duration
- log_min_duration_statement
- log_error_verbosity
- log_statement
- log_statement_stats
- log_min_error_statement
- log_min_messages
- log_temp_files
- log_connections
- log_disonnections

These parameters are discussed in more detail in the following sections.

> **Warning**
>
> The best settings for these parameters depend on your organization's policies and compliance requirements. However, enabling logging parameters can result in a large number of logs and messages, which can use up storage and affect performance, especially for a busy database. We recommend that you use these parameters carefully. For example, you might decide to enable them temporarily to narrow down a problem with a slow-performing SQL statement, and turn them off when the monitoring period is over.

rds.force_autovacuum_logging

The `rds.force_autovacuum_logging` parameter (available only in Amazon RDS for PostgreSQL) controls whether autovacuum actions are logged in the server log. Its values are `disabled`, `debug5`, `debug4`, `debug3`, `debug2`, `debug1`, `info`, `notice`, `warning`, `error`, `log`, `fatal`, `panic`. The default value is `warning`.

When you enable `rds.force_autovacuum_logging`, all actions of the autovacuum process, such as when the process starts, when it ends, and how many rows it vacuums, are logged. This is helpful for debugging or troubleshooting autovacuum performance issues.

AWS CLI syntax

The following command changes `rds.force_autovacuum_logging` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify rds.force_autovacuum_logging on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=rds.force_autovacuum_logging,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify rds.force_autovacuum_logging on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=rds.force_autovacuum_logging,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `warning`

Example

You can use the `rds.force_autovacuum_logging` parameter to analyze the performance of autovacuum on a table that has a very high write rate. For example, if your table receives a large number of write and delete operations per second, and you experience slow performance, you can enable the parameter to log the start and end times of each autovacuum run and to determine how many rows were vacuumed. This can provide valuable information about how frequently autovacuum is running, how long it takes to run, and how many rows it vacuums. You can then use

this information to fine-tune autovacuum settings such as `autovacuum_vacuum_scale_factor`, `autovacuum_vacuum_threshold`, and `autovacuum_naptime` to optimize performance.

rds.force_admin_logging_level

The `rds.force_admin_logging_level` parameter (available only in Amazon RDS for PostgreSQL) controls the level of detail in the logs that are produced by administrative operations such as vacuuming, analysis, and reindexing. It accepts the values `debug5`, `debug4`, `debug3`, `debug2`, `debug1`, `log`, `info`, `notice`, `warning`, `error`, `log`, `fatal`, and `off` (default). The optimal setting depends on your use case. For example, if you are troubleshooting an issue, you might want to set the parameter to a debug level. Otherwise, you can use the `log`, `info`, or `warning` setting.

If you set `rds.force_admin_logging_level` to `debug1`, you can log detailed information for a reindexing operation, such as the start and end times, the number of rows processed, and any errors or warnings that occur during the process. This can provide valuable information about how the reindexing process is performing and help you troubleshoot any issues that occur.

AWS CLI syntax

The following command changes `rds.force_admin_logging_level` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify rds.force_admin_logging_level on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=rds.force_admin_logging_level,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify rds.force_admin_logging_level on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=rds.force_admin_logging_level,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `off`

Example

You can use `rds.force_admin_logging_level` to monitor and analyze the performance of administrative operations across multiple tables in a large database. For example, let's say you have a large database with many tables, and you want to optimize the performance of these tables by regularly running vacuuming and analysis operations on them. By setting the `rds.force_admin_logging_level` parameter to `info` or `log`, you can log the start and end times of each operation and the tables that were affected. You can use this information to track the performance of administrative operations across different tables and identify the tables that might require more frequent or more aggressive maintenance.

Some of the logging levels generate a large number of log files and messages that can fill up disk space quickly, especially if you have a busy database. We recommend that you use this parameter carefully and turn it off when the monitoring period is over.

log_duration

The `log_duration` parameter controls whether the duration of each query (that is, the time it takes to run) is logged with the query. When you set this parameter to on, the time it takes to run each query is included in the log output along with the query text. The time is measured in milliseconds.

The main use case for the `log_duration` parameter is to help with performance tuning and troubleshooting. By logging the duration of each query, you can identify queries that are taking the longest to run, and then focus your efforts on optimizing those queries. This can help you identify and fix performance bottlenecks, and help improve the overall performance of your database.

AWS CLI syntax

The following command changes `log_duration` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_duration on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_duration,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_duration on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
```

```
  --parameters
 "ParameterName=log_duration,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `off`

Example

You might use this parameter if you suspect that a specific query or set of queries is causing performance problems. By enabling the `log_duration` parameter and examining the log output, you can see which queries are taking the longest to run and then take appropriate action, such as optimizing indexes, adding new indexes, or rewriting the query.

Enabling `log_duration` can increase the volume of log output. We recommend that you use it only when needed and turn it off during standard operations to avoid filling up the storage or making the logs hard to read.

log_min_duration_statement

The `log_min_duration_statement` parameter controls the minimum amount of time, in milliseconds, that a SQL statement runs before it is logged.

This parameter helps you identify long-running queries that might cause performance issues. You can set it to a threshold value (a runtime that is considered too long for a specific workload) to capture queries that exceed that threshold and identify potential performance bottlenecks. For an example use case, see Using logging parameters to capture bind variables later in this guide.

AWS CLI syntax

The following command changes `log_min_duration_statement` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_min_duration_statement on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_min_duration_statement,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_ min_duration_statement on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
```

```
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_min_duration_statement,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: 1 (disabled, which is the PostgreSQL engine default)

Example

The following command logs any statement that takes longer than 100 milliseconds to run:

```
aws rds modify-db-parameter-group \
   --db-parameter-group-name <parameter_group_name> \
   --parameters
 "ParameterName=log_min_duration_statement,ParameterValue=100,ApplyMethod=immediate"
```

log_error_verbosity

The `log_error_verbosity` parameter controls the level of detail included in the log output for errors and messages that are logged at the error level or higher. This parameter can take one of three values: `terse`, `default`, or `verbose`.

- `terse` includes only the message text, the error level, and the file and line number where the error occurred.
- `default` includes the message text, the error level, the file and line number, and the error context.
- `verbose` includes the message text, the error level, the file and line number, the error context, and the full error message.

Set the parameter to `verbose` to get the most detailed information for troubleshooting and debugging in a non-production environment. In a production environment, you might want to set it to `default` or `terse` so it provides only essential information and doesn't fill up log storage with too many details.

AWS CLI syntax

The following command changes `log_error_verbosity` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_error_verbosity on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_error_verbosity,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_error_verbosity on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_error_verbosity,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `default`

log_statement

The `log_statement` parameter controls which SQL statements are logged in the server log. The parameter can take one of the following values:

- none (default) doesn't log any statements
- `ddl` logs only data definition language (DDL) statements such as `CREATE TABLE` and `ALTER TABLE`
- `mod` logs only data-modifying statements such as `INSERT`, `UPDATE`, and `DELETE`
- `all` logs all SQL statements

You can use the `log_statement` parameter to control the amount of information written to the log by documenting only the specific types of statements that are relevant to your use case.

AWS CLI syntax

The following command changes `log_statement` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_statement on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
```

```
  --parameters
 "ParameterName=log_statement,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_statement on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_statement,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: none

Example

In a production environment, you might want to set `log_statement` to `ddl` to log only DDL statements and track any changes made to the database schema. In a development environment, you might want to set the parameter to `all` to log all statements to help with debugging and troubleshooting. For another example use case, see Using logging parameters to capture bind variables later in this guide.

Enabling `log_statement` can increase the volume of log output, so use it only when needed and turn it off to avoid filling up storage or making the logs difficult to read.

We recommend that you monitor your system and adjust the value of this parameter to achieve the appropriate balance between the amount of information logged and the storage and performance of the system.

log_statement_stats

The `log_statement_stats` parameter controls whether the statistics that are associated with running a SQL statement are logged with the statement. When you turn this parameter on, statistics such as the number of rows affected, the number of disk blocks read and written, and the time it takes to run the statement are included in the log output.

You can use the `log_statement_stats` parameter to gather additional information about the performance of individual statements and the overall workload. By logging statement statistics, you can identify patterns in query performance and resource usage, and use that information to optimize your database and improve overall performance.

AWS CLI syntax

The following command changes `log_statement_stats` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_statement_stats on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_statement_stats,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_statement_stas on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_statement_stats,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `off` (PostgreSQL engine default); use 0 or 1 (Boolean) to set in parameter groups

Example

You can use `log_statement_stats` to analyze the behavior of a specific query, see how it uses resources such as CPU, memory, and disk I/O, and identify whether the query can be optimized. You can also use this parameter to see if a specific table is read frequently (which might indicate the need to create an index on a specific column) or if a table is scanned too often.

Enabling `log_statement_stats` can increase the volume of log output, so use it only when needed and turn it off to avoid filling up storage or making the logs difficult to read.

log_min_error_statement

The `log_min_error_statement` parameter controls which SQL statements that result in an error will be logged. Its values are `debug5`, `debug4`, `debug3`, `debug2`, `debug1`, `info`, `notice`, `warning`, `error`, `log`, `fatal`, and `panic`. These settings control the amount of information that is written to the log so you can filter out messages of lower severity. You can set this parameter to a higher severity level to reduce the amount of log output and find important messages more easily.

AWS CLI syntax

The following command changes `log_min_error_statement` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_min_error_statement on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_min_error_statement,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_min_error_statement on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_min_error_statement,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `error` (PostgreSQL engine default)

Example

You might consider using `log_min_error_statement` when you troubleshoot a specific issue and want to see error messages from SQL statements that are causing errors.

log_min_messages

The `log_min_messages` parameter controls the severity level that is written to the log. You can set the parameter to debug5, debug4, debug3, debug2, debug1, `info`, `notice`, `warning`, `error`, `log`, `fatal`, or `panic`. These settings control the amount of information that is written to the log so you can filter out messages of lower severity. You can set this parameter to a higher severity level to reduce the amount of log output and find important messages more easily.

AWS CLI syntax

The following command changes `log_min_messages` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_min_messages on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
```

```
  --parameters
 "ParameterName=log_min_messages,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_min_messages on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_min_messages,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `notice`

Example

If you're troubleshooting a specific issue and you want to see all error messages, you might set this parameter to `error` to log only errors and higher-level severity issues. If you're interested in monitoring the performance of the system, you might set this parameter to `info` to see more detailed information such as the duration and statistics of each statement.

Setting `log_min_messages` to a higher severity level decreases the volume of logs. We recommend that you tune this parameter depending on your specific use case, the size of the log you want to check, and the amount of disk space you have.

log_temp_files

The `log_temp_files` parameter controls the logging of temporary file names and sizes. It applies to temporary files created for purposes such as sorts, hashes, and temporary query results. When this parameter is enabled, a log entry is generated for each temporary file upon deletion, including its file size, in bytes. You can set this parameter to 0 (zero) for comprehensive logging of all temporary file information, or to a positive value to log files that exceed that size (in kilobytes, if units aren't specified). This can be useful for identifying and resolving performance bottlenecks or other issues related to temporary storage. By default, logging of temporary files is disabled.

AWS CLI syntax

The following command changes `log_temp_files` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_temp_files on a DB parameter group
```

```
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_temp_files,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_temp_files on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_temp_files,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: -1 (PostgreSQL engine default)

Example

You might enable this parameter if you suspect that the system is using too much temporary storage, or that temporary files are not being deleted properly. When you examine the log output, you can see the queries or operations that are generating temporary files and how these files are being used.

Some queries or operations create a large number of temporary files, so enabling `log_temp_files` might impact the overall performance of your system.

log_connections

The `log_connections` parameter controls whether connections to the database are logged. When you set this parameter to on, the log contains information about each successful connection to the database, such as the client's IP address, the username, the database name, and the date and time of the connection.

You can use the `log_connections` parameter to monitor and troubleshoot connections to the database. You can see the users, applications, terminals, and bots that connect to the database, where they're connecting from, and how often. This information can be useful for identifying and resolving connection-related issues or tracking usage patterns.

AWS CLI syntax

The following command changes `log_connections` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_connections on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_connections,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_connections on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_connections,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `off` (PostgreSQL engine default)

Example

You can use this parameter if you suspect that too many connections to the database, or a specific user or IP address that's connecting too frequently, is affecting performance. By enabling the `log_connections` parameter and examining the log output, you can see the number and details of all connections.

Before you enable this parameter, check your organization's policies and consider the security implications of logging IP addresses and usernames.

log_disconnections

The `log_disconnections` parameter controls the logging of disconnections from the database. When you set this parameter to on, it logs information about the end of each session, such as the client's IP address, the username, the database name, and the date and time of the disconnection.

You can use the `log_disconnections` parameter to monitor and troubleshoot database session terminations. You can see the users, applications, terminals, and bots that disconnect from the database, when, and why. For example, you can review unexpected terminations such as a crash or administrator-initiated disconnections. This information can be useful for identifying and resolving issues related to disconnections or tracking usage patterns.

AWS CLI syntax

The following command changes `log_disconnections` for a specific DB parameter group. This change applies to all instances or clusters that use the parameter group.

```
# Modify log_disconnections on a DB parameter group
aws rds modify-db-parameter-group \
  --db-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_disconnections,ParameterValue=<new_value>,ApplyMethod=immediate"

# Modify log_disconnections on a DB cluster parameter group
aws rds modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name <parameter_group_name> \
  --parameters
 "ParameterName=log_disconnections,ParameterValue=<new_value>,ApplyMethod=immediate"
```

Type: Dynamic (changes are applied immediately if you set `ApplyMethod=immediate`)

Default value: `off` (PostgreSQL engine default)

Example

You might use `log_disconnections` if you suspect that there are too many users disconnecting from the database, or that a specific user or IP address is disconnecting too frequently. By enabling the `log_disconnections` parameter and examining the log output, you can see the number and details of all disconnections, including who, when, and whether any errors were raised before the disconnection.

Before you enable this parameter, check your organization's policies and consider the security implications of logging IP addresses and usernames.

Using logging parameters to capture bind variables

A typical use case for capturing bind variables in PostgreSQL is to debug and performance-tune SQL queries. A bind variable lets you pass data to a query when you run it. By capturing the bind variables, you can see the input data that was passed to a query, which can help you identify any issues with the data or with query performance. Capturing the bind variables can also help you audit the input data and detect potential security risks or malicious activity.

There are a several ways to capture bind variables for PostgreSQL. One method is to enable the `debug_print_parse` and `debug_print_rewritten` parameters. This causes PostgreSQL to

send the parsed and rewritten versions of SQL statements, along with the bound variables, to the server log.

- `debug_print_parse`: When you enable this parameter, the parse tree of incoming queries is printed to the server log. This can be useful for understanding the structure of a query and the values of any bound parameters.
- `debug_print_rewritten`: When you enable this parameter, the rewritten forms of incoming queries are printed to the server log. This can be useful for understanding how the query planner interprets a query and the values of any bound parameters.

You can use two additional parameters in Amazon RDS and Aurora to capture bind variables in your PostgreSQL databases:

- `log_min_duration_statement`: This parameter sets the minimum duration of a statement before it is logged, in milliseconds. When a statement takes longer than the specified duration, its bind values are included in the log output.
- `log_statement`: This parameter controls which SQL statements are logged. Set this parameter to `all` or bind to include the bound values in the log. Increasing the logging level affects performance, so we recommend that you revert changes after troubleshooting.

You can also use the `pg_stat_statements` extension, which provides performance statistics for all SQL statements run by a server, including the query text and the bound values. This extension allows you to use **pgAdmin** or similar tools to monitor and analyze query performance.

Another option is to use the `pg_bind_parameter_status()` function to get the values of bound parameters from a prepared statement or to use the `pg_get_parameter_status (paramname)` function to retrieve the status or value of a specific runtime parameter.

Additionally, you can use third-party tools such as **pgBadger** to analyze the PostgreSQL logs and extract the bind variables and other information for further analysis.

Tuning replication parameters

In PostgreSQL, you can replicate data changes from one PostgreSQL database to another by using logical replication instead of physical, file-based replication. Logical replication uses the write-ahead log (WAL) to capture changes and supports the replication of selected tables or entire databases.

Amazon RDS for PostgreSQL and Aurora PostgreSQL-Compatible both support logical replication, so you can set up a highly available and scalable database architecture that can handle read and write traffic from multiple sources. These services use **pglogical**, which is an open-source PostgreSQL extension, to implement logical replication.

Tuning logical replication in Aurora and Amazon RDS is important for achieving optimal performance, scalability, and availability. You can tune the parameters in the **pglogical** extension to manage the performance of logical replication. For example, you can:

- Improve the performance of replication by increasing the number of worker processes or adjusting their memory allocation.
- Reduce the risk of replication lag by adjusting the frequency of synchronization between the source and replica databases.
- Optimize the use of resources by adjusting the memory and CPU allocation of the worker processes.
- Ensure that the replication process does not cause undue impact on the performance of the source database.

You can use the following parameters in Aurora and Amazon RDS to control and configure logical replication:

- `max_replication_slots` sets the maximum number of replication slots that can be created on the server. A replication slot is a named, persistent reservation for a replication connection to send WAL data to a replica.
- `max_wal_senders` sets the maximum number of simultaneously connected WAL sender processes. WAL sender processes are used to stream the WAL from the primary server to the replica.
- `wal_sender_timeout` sets the maximum time, in milliseconds, that a WAL sender waits for a response from the replica before giving up and reconnecting.

- `wal_receiver_timeout` sets the maximum time, in milliseconds, that a replica waits for WAL data from the primary database before timing out.
- `log_replication_commands`, when set to on, runs the replication-related SQL statements.

When you enable the `rds.logical_replication` parameter (by setting it to 1) the `wal_level` parameter is set to `logical`, which means that all changes made to the database are written to the WAL in a format that can be read and applied to a replica. This setting is required to enable logical replication. This setting also allows for the replication of `SELECT` statements.

Setting `wal_level` to `logical` can increase the amount of data written to the WAL, and therefore to disk, which can affect system performance. We recommend that you consider available disk space and system performance when enabling logical replication.

Example

You want to replicate data from your primary database to a secondary database for backup and disaster recovery purposes. However, the secondary database has a high volume of read operations, so you want to make sure that the replication process is as fast and efficient as possible without compromising data integrity.

The default values for logical replication in Amazon RDS and Aurora prioritize consistency over performance, so they might not be optimal for this use case. To optimize your logical replication settings for speed and efficiency, you can customize the parameters as follows:

- Increase `max_replication_slots` from 10 (default for Amazon RDS) or 20 (default for Aurora) to 30 to accommodate potential future growth and replication needs.
- Increase `max_wal_senders` from 10 (default) to 20 to ensure that there are enough WAL sender processes to keep up with replication demand.
- Decrease `wal_sender_timeout` from 30 seconds (default) to 15 seconds to ensure that idle WAL sender processes are terminated more quickly, which frees up resources for active replication.
- Decrease `wal_receiver_timeout` from 30 seconds (default) to 15 seconds to ensure that idle WAL receiver processes are terminated more quickly, which frees up resources for active replication.
- Increase `max_logical_replication_workers` from 4 (default) to 8 to ensure that there are enough logical replication worker processes to keep up with replication demand.

These optimizations provide faster and more efficient data replication while maintaining data integrity and security.

For example, if a disaster were to occur and the primary database became unavailable, the secondary database would already have the latest data available due to the optimized replication process. This would enable your business operations to continue providing critical services without interruption.

Best practices

Tuning logical replication with huge workloads can be a complex task that depends on a variety of factors, including the size of the dataset, the number of tables being replicated, the number of replicas, and the available resources. Here are a few general tips for tuning logical replication with huge workloads:

- **Monitor replication lag.** The replication lag is the time difference between the primary server and standby servers. Monitoring replication lag can help you identify potential bottlenecks and take action to improve replication performance. You can use the `pg_current_wal_lsn()` function to check the current replication lag.
- **Tune WAL settings.** The `pg_logical` extension uses WAL to transmit changes from the primary server to the standby server. If the WAL settings aren't tuned properly, replication can become slow and unreliable. Make sure to set the `max_wal_senders` and `max_replication_slots` parameters to adequate values, depending on your workloads.
- **Have an indexing strategy.** Having proper indexes on the primary server can help improve the performance of the logical replication, reduce the I/O on the primary server, and reduce the load on the system.
- **Use parallel replication.** Using parallel replication can help increase replication speed by allowing multiple parallel worker processes to replicate data. This feature is available on PostgreSQL 12 and later.

Next steps

After you optimize the memory, replication, autovacuum, and logging parameters for your Amazon RDS for PostgreSQL or Aurora PostgreSQL-Compatible database, consider these steps to further improve the performance of your database:

- **Monitor your database.** Keep track of your database performance over time by using built-in monitoring tools or third-party solutions. Monitor key performance metrics such as CPU utilization, disk I/O, memory usage, and query runtimes to identify potential bottlenecks and areas for improvement.
- **Continuously tune parameters.** As your workload evolves, continue to monitor and adjust your database parameters to ensure optimal performance. Regularly check system logs, error messages, and performance metrics to identify new tuning opportunities.
- **Implement caching.** Use caching to reduce the number of queries that hit the database. You can implement caching at the application level by using tools such as Memcached or Redis, or you can use Amazon ElastiCache to provide an in-memory cache for your database.
- **Optimize your queries.** Poorly designed queries can significantly impact database performance. Use `EXPLAIN` and other query tuning tools to identify slow queries, optimize them, and eliminate any unnecessary queries.

By following these guidelines, you can optimize the performance of your Aurora or Amazon RDS for PostgreSQL database and ensure that it meets the needs of your application with improved database performance, increased reliability, reduced downtime, better security, and cost savings. By optimizing the configuration parameters to suit your workload, you can ensure that your database is running efficiently and using resources effectively, leading to better performance and a more responsive application. Additionally, properly configured parameters can reduce the likelihood of errors and vulnerabilities, resulting in increased reliability and better security. This can translate to cost savings in terms of reduced maintenance and downtime, as well as better overall user experience and satisfaction.

Resources

- Amazon Aurora PostgreSQL parameters, Part 1: Memory and query plan management (AWS blog post)
- Amazon Aurora PostgreSQL parameters, Part 2: Replication, security, and logging (AWS blog post)
- Amazon Aurora PostgreSQL parameters, Part 3: Optimizer parameters (AWS blog post)
- Amazon Aurora PostgreSQL parameters, Part 4: ANSI compatibility options (AWS blog post)
- Working with Amazon Aurora PostgreSQL (AWS documentation)
- Working with Amazon RDS for PostgreSQL (AWS documentation)
- Monitoring DB load with Performance Insights on Amazon RDS (AWS documentation)
- Using Amazon CloudWatch metrics (AWS documentation)
- pg_stats_statements PostgreSQL documentation)

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Updated information about memory and autovacuum parameters	Updated the description of the random_page_cost parameter; added missing units to the default values for memory and autovacuum parameters; updated the AWS CLI syntax for the max_connections parameter.	February 27, 2024
Updated information about `autovacuum`	Corrected the autovacuum default setting (enabled).	December 27, 2023
Updated information about `max_connections`	Updated the max_connections section with new guidance on tuning this parameter.	November 15, 2023
Initial publication	—	October 31, 2023

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, AWS Panorama offers devices that add CV to on-premises camera networks, and Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more

information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the “2021-05-27 00:15:37” date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries.

Detective guardrails detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners,

migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata.

The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your

organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.