AWS RDS PostgreSQL

Administration

John Rey Goh

Contents

Amazon RDS for PostgreSQL

You can create:

- DB instances
- DB snapshots
- point-in-time restores
- backups.

DB instances running PostgreSQL support:

- Multi-AZ deployments
- read replicas
- Provisioned IOPS
- can be created inside a virtual private cloud (VPC)
- You can also use Secure Socket Layer (SSL) to connect to a DB instance running PostgreSQL

Common management tasks for Amazon RDS for PostgreSQL

Setting up Amazon RDS for first-time use

- ✓ Sign up for an AWS account
- ✓ Create a user with administrative access
- ✓ Assign access to additional users
- ✓ Grant programmatic access
- ✓ Determine requirements

The basic building block of Amazon RDS is the DB instance. In a DB instance, you create your databases. A DB instance provides a network address called an endpoint. Your applications use this endpoint to connect to your DB instance. When you create a DB instance, you specify details like storage, memory, database engine and version, network configuration, security, and maintenance periods. You control network access to a DB instance through a security group.

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

- Resource requirements – What are the memory and processor requirements for your application or service? You use these settings to help you determine what DB instance class to use. For specifications about DB instance classes, see DB instance classes.

- VPC, subnet, and security group – Your DB instance will most likely be in a virtual private cloud (VPC). To connect to your DB instance, you need to set up security group rules. These rules are set up differently depending on what kind of VPC you use and how you use it. For example, you can use: a default VPC or a user-defined VPC.

The following list describes the rules for each VPC option:

Default VPC – If your AWS account has a default VPC in the current AWS Region, that VPC is configured to support DB instances. If you specify the default VPC when you create the DB instance, do the following:

Make sure to create a VPC security group that authorizes connections from the application or service to the Amazon RDS DB instance. Use the Security Group option on the VPC console or the AWS CLI to create VPC security groups.

Specify the default DB subnet group. If this is the first DB instance you have created in this AWS Region, Amazon RDS creates the default DB subnet group when it creates the DB instance.

User-defined VPC – If you want to specify a user-defined VPC when you create a DB instance, be aware of the following:

Make sure to create a VPC security group that authorizes connections from the application or service to the Amazon RDS DB instance. Use the Security Group option on the VPC console or the AWS CLI to create VPC security groups.

The VPC must meet certain requirements in order to host DB instances, such as having at least two subnets, each in a separate Availability Zone. For information, see Amazon VPC VPCs and Amazon RDS.

Make sure to specify a DB subnet group that defines which subnets in that VPC can be used by the DB instance.

- High availability – Do you need failover support? On Amazon RDS, a Multi-AZ deployment creates a primary DB instance and a secondary standby DB instance in another Availability Zone for failover support. We recommend Multi-AZ deployments for production workloads to maintain high availability. For development and test purposes, you can use a deployment that isn't Multi-AZ.

- IAM policies – Does your AWS account have policies that grant the permissions needed to perform Amazon RDS operations? If you are connecting to AWS using IAM credentials, your IAM account must have IAM policies that grant the permissions required to perform Amazon RDS operations.
- Open ports – What TCP/IP port does your database listen on? The firewalls at some companies might block connections to the default port for your database engine. If your company firewall blocks the default port, choose another port for the new DB instance. When you create a DB instance that listens on a port you specify, you can change the port by modifying the DB instance.

- AWS Region – What AWS Region do you want your database in? Having your database in close proximity to your application or web service can reduce network latency. For more information, see Regions, Availability Zones, and Local Zones.

- DB disk subsystem – What are your storage requirements? Amazon RDS provides three storage types:

General Purpose (SSD)
Provisioned IOPS (PIOPS)
Magnetic (also known as standard storage)

Provide access to your DB instance in your VPC by creating a security group

VPC security groups provide access to DB instances in a VPC. They act as a firewall for the associated DB instance, controlling both inbound and outbound traffic at the DB instance level. DB instances are created by default with a firewall and a default security group that protect the DB instance.

Before you can connect to your DB instance, you must add rules to a security group that enable you to connect. Use your network and configuration information to create rules to allow access to your DB instance.

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

You can configure connectivity between an Amazon EC2 instance a DB instance when you create the DB instance.

You can set up network connectivity between an Amazon EC2 instance and a DB instance automatically when you create the DB instance.

- To create a VPC security group

1. Sign in to the AWS Management Console and open the Amazon VPC console

2. In the upper-right corner of the AWS Management Console, choose the AWS Region where you want to create your VPC security group and DB instance. In the list of Amazon VPC resources for that AWS Region, you should see at least one VPC and several subnets. If you don't, you don't have a default VPC in that AWS Region.

3. In the navigation pane, choose Security Groups→ Choose Create security group.

In Basic details, enter the Security group name and Description. For VPC, choose the VPC that you want to create your DB instance in.

In Inbound rules, choose Add rule.

For Type, choose Custom TCP.

For Port range, enter the port value to use for your DB instance.

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

If you need to add more IP addresses or different port ranges, choose Add rule and enter the information for the rule.

(Optional) In Outbound rules, add rules for outbound traffic. By default, all outbound traffic is allowed.

Choose Create security group.

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

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

After you have completed the setup requirements, you can create a DB instance using your requirements and security group. To do so, follow the instructions in Creating an Amazon RDS DB instance.

Understanding Amazon RDS DB instances
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.DBInstanceClass.html

Setting up high availability and failover support

Multi-AZ deployments can have one standby or two standby DB instances. When the deployment has one standby DB instance, it's called a Multi-AZ DB instance deployment. A Multi-AZ DB instance deployment has one standby DB instance that provides failover support, but doesn't serve read traffic. When the deployment has two standby DB instances, it's called a Multi-AZ DB cluster deployment. A Multi-AZ DB cluster deployment has standby DB instances that provide failover support and can also serve read traffic.

You can use the AWS Management Console to determine whether a Multi-AZ deployment is a Multi-AZ DB instance deployment or a Multi-AZ DB cluster deployment. In the navigation pane, choose Databases, and then choose a DB identifier.

A Multi-AZ DB instance deployment has the following characteristics:
There is only one row for the DB instance.
The value of Role is Instance or Primary.

The value of Multi-AZ is Yes.

A Multi-AZ DB cluster deployment has the following characteristics:
There is a cluster-level row with three DB instance rows under it.
For the cluster-level row, the value of Role is Multi-AZ DB cluster.
For each instance-level row, the value of Role is Writer instance or Reader instance.
For each instance-level row, the value of Multi-AZ is 3 Zones.

Amazon RDS provides high availability and failover support for DB instances using Multi-AZ deployments with a single standby DB instance. This type of deployment is called a Multi-AZ DB instance deployment. Amazon RDS uses several different technologies to provide this failover support. Multi-AZ deployments for MariaDB, MySQL, Oracle, PostgreSQL, and RDS Custom for SQL Server DB instances use the Amazon failover technology. Microsoft SQL Server DB instances use SQL Server Database Mirroring (DBM) or Always On Availability Groups (AGs).

In a Multi-AZ DB instance deployment, Amazon RDS automatically provisions and maintains a synchronous standby replica in a different Availability Zone. The primary DB instance is synchronously replicated across Availability Zones to a standby replica to provide data redundancy and minimize latency spikes during system backups.

Note: The high availability option isn't a scaling solution for read-only scenarios. You can't use a standby replica to serve read traffic. To serve read-only traffic, use a Multi-AZ DB cluster or a read replica instead.

To convert to a Multi-AZ DB instance deployment with the RDS console

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Databases, and then choose the DB instance that you want to modify.
3. From Actions, choose Convert to Multi-AZ deployment.
4. On the confirmation page, choose Apply immediately to apply the changes immediately. Choosing this option doesn't cause downtime, but there is a possible performance impact. Alternatively, you can choose to apply the update during the next maintenance window. For more information, see Schedule modifications setting.
5. Choose Convert to Multi-AZ.

Failover process for Amazon RDS

If a planned or unplanned outage of your DB instance results from an infrastructure defect, Amazon RDS automatically switches to a standby replica in another Availability Zone if you have turned on Multi-AZ.

The time that it takes for the failover to complete depends on the database activity and other conditions at the time the primary DB instance became unavailable.

Failover times are typically 60–120 seconds.

However, large transactions or a lengthy recovery process can increase failover time. When the failover is complete, it can take additional time for the RDS console to reflect the new Availability Zone.

Understanding the Amazon Virtual Private Cloud (VPC) network

Your DB instance is in a virtual private cloud (VPC). A VPC is a virtual network that is logically isolated from other virtual networks in the AWS Cloud. Amazon VPC makes it possible for you to launch AWS resources, such as an Amazon RDS DB instance or Amazon EC2 instance, into a VPC. The VPC can either be a default VPC that comes with your account or one that you create. All VPCs are associated with your AWS account.

Your default VPC has three subnets that you can use to isolate resources inside the VPC. The default VPC also has an internet gateway that can be used to provide access to resources inside the VPC from outside the VPC.

Working with a DB instance in a VPC
Here are some tips on working with a DB instance in a VPC:

1. Your VPC must have at least two subnets. These subnets must be in two different Availability Zones in the AWS Region where you want to deploy your DB instance. A subnet is a segment of a VPC's IP address range that you can specify and that you can use to group DB instances based on your security and operational needs.

2. For Multi-AZ deployments, defining a subnet for two or more Availability Zones in an AWS Region allows Amazon RDS to create a new standby in another Availability Zone as needed. Make sure to do this even for Single-AZ deployments, just in case you want to convert them to Multi-AZ deployments at some point.

3. The DB subnet group for a Local Zone can have only one subnet.

4. If you want your DB instance in the VPC to be publicly accessible, make sure to turn on the VPC attributes DNS hostnames and DNS resolution.

5. Your VPC must have a DB subnet group that you create. You create a DB subnet group by specifying the subnets you created. Amazon RDS chooses a subnet and an IP address within that subnet group to associate with your DB instance. The DB instance uses the Availability Zone that contains the subnet.

6. Your VPC must have a VPC security group that allows access to the DB instance.

7. The CIDR blocks in each of your subnets must be large enough to accommodate spare IP addresses for Amazon RDS to use during maintenance activities, including failover and compute scaling. For example, a range such as 10.0.0.0/24 and 10.0.1.0/24 is typically large enough.

8. A VPC can have an instance tenancy attribute of either default or dedicated. All default VPCs have the instance tenancy attribute set to default, and a default VPC can support any DB instance class.

9. If you choose to have your DB instance in a dedicated VPC where the instance tenancy attribute is set to dedicated, the DB instance class of your DB instance must be one of the approved Amazon EC2 dedicated instance types. For example, the r5.large EC2 dedicated instance corresponds to the db.r5.large DB instance class. For information about instance tenancy in a VPC, see Dedicated instances in the Amazon Elastic Compute Cloud User Guide.

10. When you set the instance tenancy attribute to dedicated for a DB instance, it doesn't guarantee that the DB instance will run on a dedicated host.

11. When an option group is assigned to a DB instance, it's associated with the DB instance's VPC. This linkage means that you can't use the option group assigned to a DB instance if you attempt to restore the DB instance into a different VPC.

12. If you restore a DB instance into a different VPC, make sure to either assign the default option group to the DB instance, assign an option group that is linked to that VPC, or create a new option group and assign it to the DB instance. With persistent or permanent options, such as Oracle TDE, you must create a new option group that includes the persistent or permanent option when restoring a DB instance into a different VPC.

Setting up read-only read replicas (primary and standbys)

When you create a read replica, Amazon RDS takes a DB snapshot of your source DB instance and begins replication. The source DB instance experiences a very brief I/O suspension when the DB snapshot operation begins. The I/O suspension typically lasts about one second. You can avoid the I/O suspension if the source DB instance is a Multi-AZ deployment, because in that case the snapshot is taken from the secondary DB instance.

An active, long-running transaction can slow the process of creating the read replica. We recommend that you wait for long-running transactions to complete before creating a read replica. If you create multiple read replicas in parallel from the same source DB instance, Amazon RDS takes only one snapshot at the start of the first create action.

When creating a read replica, there are a few things to consider. First, you must enable automatic backups on the source DB instance by setting the backup retention period to a value other than 0. This requirement also applies to a read replica that is the source DB instance for another read replica. To enable automatic backups on an RDS for MySQL read replica, first create the read replica, then modify the read replica to enable automatic backups.

To create a read replica from a source DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Databases.

3. Choose the DB instance that you want to use as the source for a read replica.
4. For Actions, choose Create read replica.
5. For DB instance identifier, enter a name for the read replica.
6. Choose your instance configuration. We recommend that you use the same or larger DB instance class and storage type as the source DB instance for the read replica.
7. For AWS Region, specify the destination Region for the read replica.
8. For Storage, specify the allocated storage size and whether you want to use storage autoscaling.
9. If your source DB instance isn't on the latest storage configuration, the Upgrade storage file system configuration option is available. You can enable this setting to upgrade the storage file system of the read replica to the preferred configuration. For more information, see Upgrading the storage file system for a DB instance.
10. For Availability, choose whether to create a standby of your replica in another Availability Zone for failover support for the replica.
11. Creating your read replica as a Multi-AZ DB instance is independent of whether the source database is a Multi-AZ DB instance.
12. Specify other DB instance settings. For information about each available setting, see Settings for DB instances.
13. Choose Create read replica.
14. After the read replica is created, you can see it on the Databases page in the RDS console. It shows Replica in the Role column.

Promoting a read replica to be a standalone DB instance

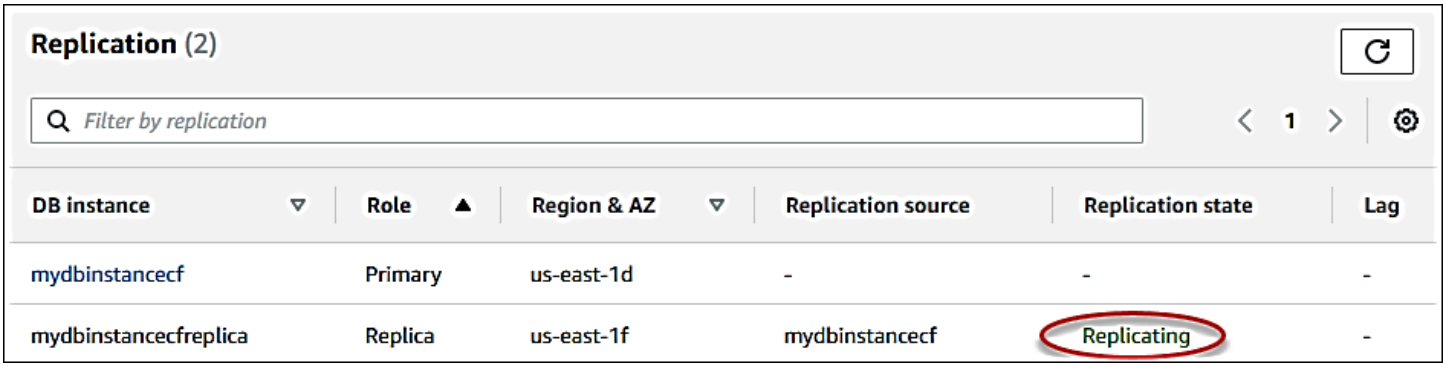
You can promote a read replica into a standalone DB instance. If a source DB instance has several read replicas, promoting one of the read replicas to a DB instance has no effect on the other replicas.

When you promote a read replica, RDS reboots the DB instance before making it available. The promotion process can take several minutes or longer to complete, depending on the size of the read replica.

To promote a read replica to a standalone DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the Amazon RDS console, choose Databases.
3. The Databases pane appears. Each read replica shows Replica in the Role column.
4. Choose the read replica that you want to promote.
5. For Actions, choose Promote.
6. On the Promote Read Replica page, enter the backup retention period and the backup window for the newly promoted DB instance.
7. When the settings are as you want them, choose Continue.
8. On the acknowledgment page, choose Promote Read Replica.

You can monitor the status of a read replica in several ways. The Amazon RDS console shows the status of a read replica in the Replication section of the Connectivity & security tab in the read replica details. To view the details for a read replica, choose the name of the read replica in the list of DB instances in the Amazon RDS console.



Understanding security groups

VPC security groups control the access that traffic has in and out of a DB instance. By default, network access is turned off for a DB instance. You can specify rules in a security group that allow access from an IP address range, port, or security group. After ingress rules are configured, the same rules apply to all DB instances that are associated with that security group. You can specify up to 20 rules in a security group.

You can associate a security group with a DB instance by using Modify on the RDS console

Setting up parameter groups and features

Associating a DB parameter group with a DB instance

You can create your own DB parameter groups with customized settings. You can associate a DB parameter group with a DB instance using the AWS Management Console, the AWS CLI, or the RDS API. You can do so when you create or modify a DB instance.

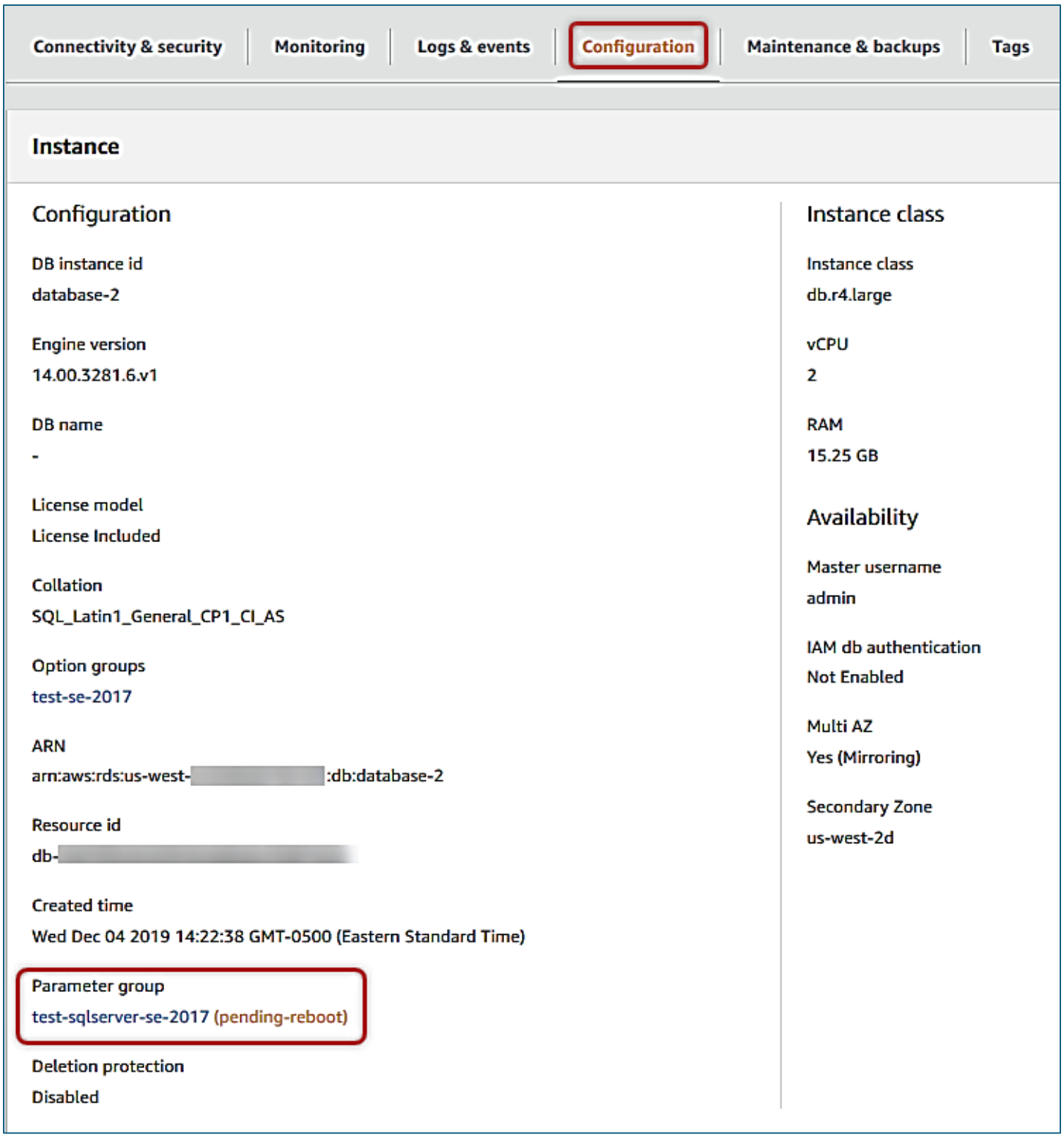
To associate a DB parameter group with a DB instance

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Databases, and then choose the DB instance that you want to modify.
3. Choose Modify. The Modify DB instance page appears.
4. Change the DB parameter group setting.
5. Choose Continue and check the summary of modifications.
6. (Optional) Choose Apply immediately to apply the changes immediately. Choosing this option can cause an outage in some cases. For more information, see Schedule modifications setting.
7. On the confirmation page, review your changes. If they are correct, choose Modify DB instance to save your changes.
8. Or choose Back to edit your changes or Cancel to cancel your changes.

Modifying parameters in a DB parameter group

You can modify parameter values in a customer-created DB parameter group; you can't change the parameter values in a default DB parameter group. Changes to parameters in a customer-created DB parameter group are applied to all DB instances that are associated with the DB parameter group.

Changes to some parameters are applied to the DB instance immediately without a reboot. Changes to other parameters are applied only after the DB instance is rebooted. The RDS console shows the status of the DB parameter group associated with a DB instance on the Configuration tab. For example, suppose that the DB instance isn't using the latest changes to its associated DB parameter group. If so, the RDS console shows the DB parameter group with a status of pending-reboot. To apply the latest parameter changes to that DB instance, manually reboot the DB instance.


Connectivity & security
Monitoring
Logs & events
Configuration
Maintenance & backups
Tags
Instance
Configuration
DB instance id
database-2
Engine version
14.00.3281.6.v1
DB name
-
License model
License Included
Collation
SQL_Latin1_General_CP1_CI_AS
Option groups
test-se-2017
ARN
arn:aws:rds:us-west-:db:database-2
Resource id
db-
Created time
Wed Dec 04 2019 14:22:38 GMT-0500 (Eastern Standard Time)
Parameter group
test-sqlserver-se-2017 (pending-reboot)
Deletion protection
Disabled
Instance class
Instance class
db.r4.large
vCPU
2
RAM
15.25 GB
Availability
Master username
admin
IAM db authentication
Not Enabled
Multi AZ
Yes (Mirroring)
Secondary Zone
us-west-2d

To modify the parameters in a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Parameter groups.
3. In the list, choose the name of the parameter group that you want to modify.
4. For Parameter group actions, choose Edit.
5. Change the values of the parameters that you want to modify. You can scroll through the parameters using the arrow keys at the top right of the dialog box.
6. You can't change values in a default parameter group.
7. Choose Save changes.

To reset parameters in a DB parameter group to their default values

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Parameter groups.
3. In the list, choose the parameter group.
4. For Parameter group actions, choose Edit.
5. Choose the parameters that you want to reset to their default values. You can scroll through the parameters using the arrow keys at the top right of the dialog box.
6. You can't reset values in a default parameter group.
7. Choose Reset and then confirm by choosing Reset parameters.

To copy a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Parameter groups.
3. In the list, choose the custom parameter group that you want to copy.
4. For Parameter group actions, choose Copy.
5. In New DB parameter group identifier, enter a name for the new parameter group.
6. In Description, enter a description for the new parameter group.
7. Choose Copy.

To list all DB parameter groups for an AWS account

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Parameter groups.
3. The DB parameter groups appear in a list.

To delete a DB parameter group

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Parameter groups.
3. The DB parameter groups appear in a list.
4. Choose the name of the parameter groups to be deleted.
5. Choose Actions and then Delete.
6. Review the parameter group names and then choose Delete.

Connecting to your PostgreSQL DB instance

After Amazon RDS provisions your DB instance, you can use any standard SQL client application to connect to the instance. Before you can connect, the DB instance must be available and accessible. Whether you can connect to the instance from outside the VPC depends on how you created the Amazon RDS DB instance:

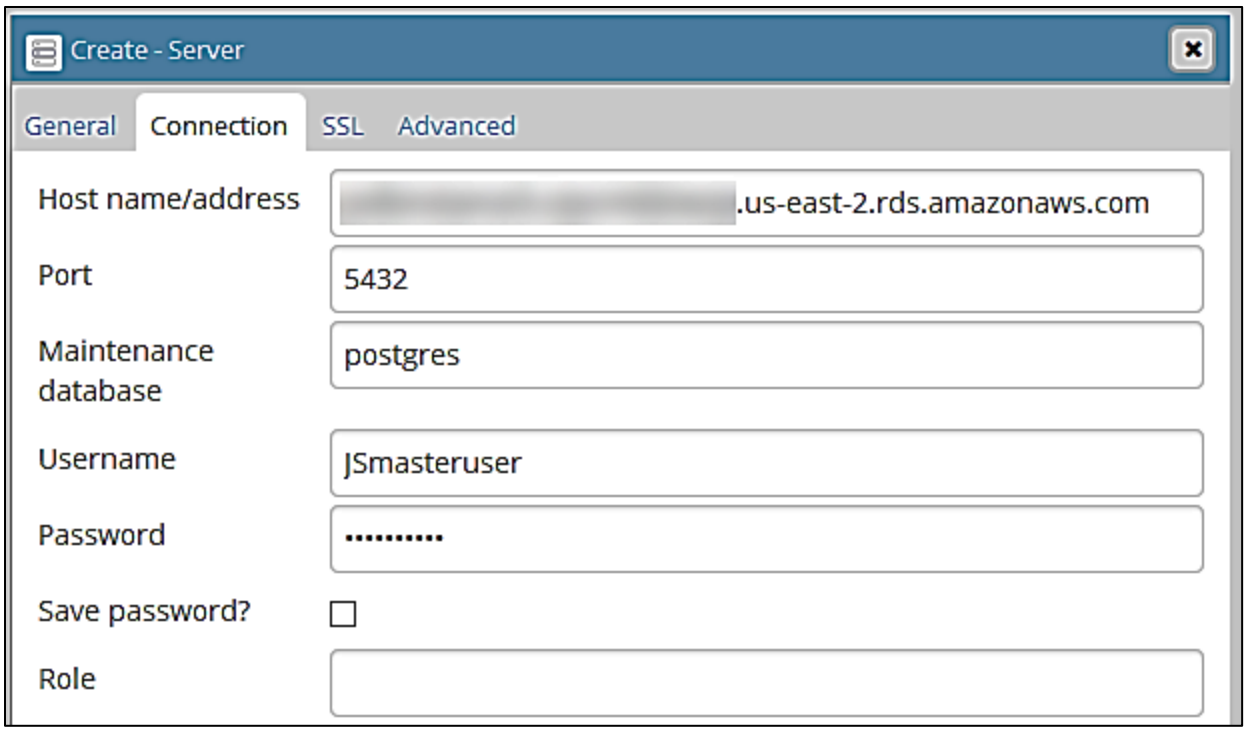
- If you created your DB instance as public, devices and Amazon EC2 instances outside the VPC can connect to your database.

- If you created your DB instance as private, only Amazon EC2 instances and devices inside the Amazon VPC can connect to your database.

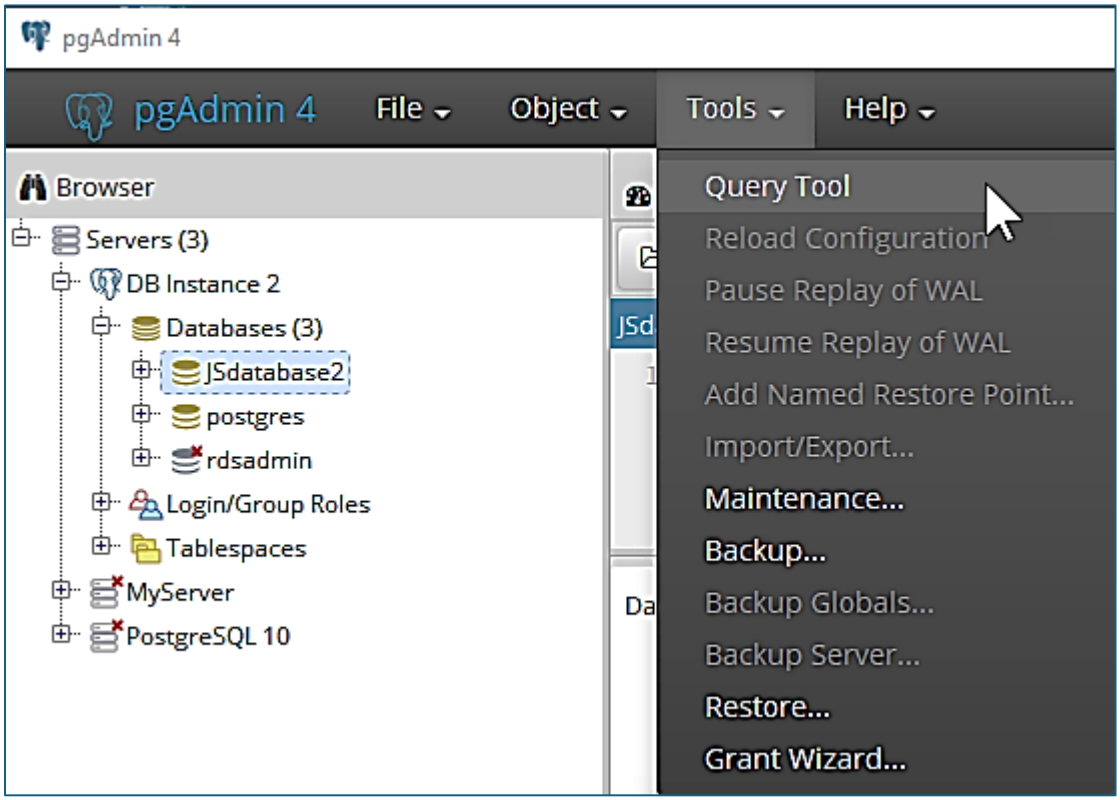
To check whether your DB instance is public or private, use the AWS Management Console to view the Connectivity & security tab for your instance. Under Security, you can find the "Publicly accessible" value, with No for private, Yes for public.

To connect to your RDS for PostgreSQL DB instance using pgAdmin

1. Launch the pgAdmin application on your client computer.
2. On the Dashboard tab, choose Add New Server.
3. In the Create - Server dialog box, type a name on the General tab to identify the server in pgAdmin.
4. On the Connection tab, type the following information from your DB instance:
5. For Host, type the endpoint, for example mypostgresql.c6c8dntfzzhgv0.us-east-2.rds.amazonaws.com.
6. For Port, type the assigned port.
7. For Username, type the user name that you entered when you created the DB instance (if you changed the 'master username' from the default, postgres).
8. For Password, type the password that you entered when you created the DB instance.



9. Choose Save.
10. If you have any problems connecting, see Troubleshooting connections to your RDS for PostgreSQL instance.
11. To access a database in the pgAdmin browser, expand Servers, the DB instance, and Databases. Choose the DB instance's database name.

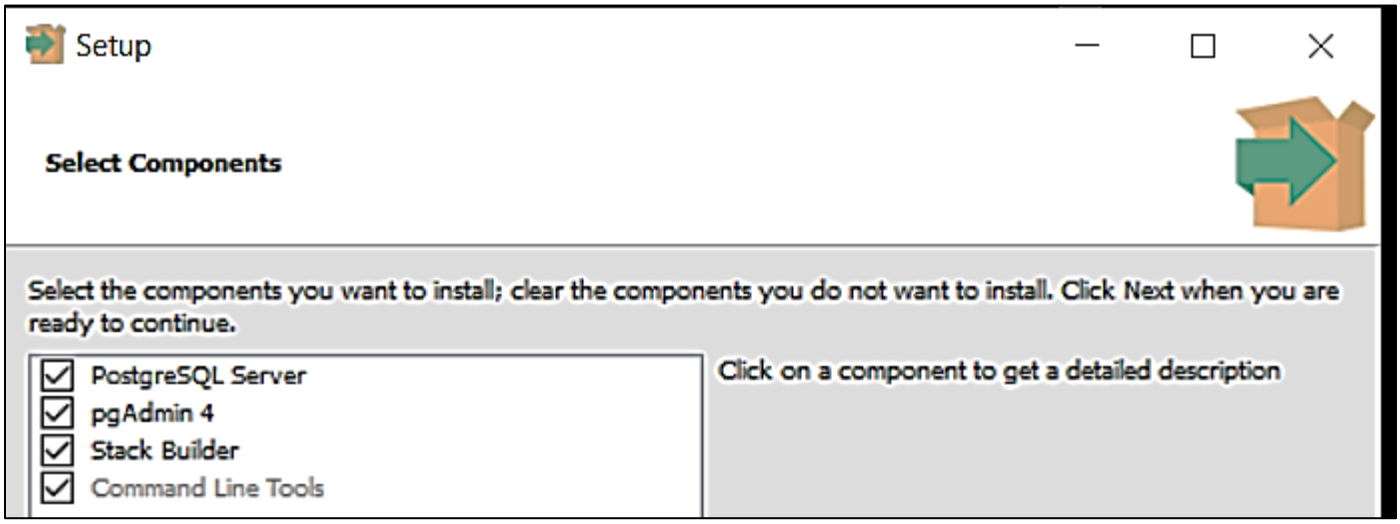


12. To open a panel where you can enter SQL commands, choose Tools, Query Tool.

Using psql to connect to your RDS for PostgreSQL DB instance

1. You can use a local instance of the psql command line utility to connect to a RDS for PostgreSQL DB instance. You need either PostgreSQL or the psql client installed on your client computer.

2. You can download the PostgreSQL client from the PostgreSQL. Get the full installer from here: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads then you can choose to install only the command-line tools as follows:



3. To connect to your RDS for PostgreSQL DB instance using psql, you need to provide host (DNS) information, access credentials, and the name of the database.

4. Use one of the following formats to connect to your RDS for PostgreSQL DB instance. When you connect, you're prompted for a password. For batch jobs or scripts, use the --no-password option. This option is set for the entire session.

5. A connection attempt with --no-password fails when the server requires password authentication and a password is not available from other sources. For more information, see the psql documentation

6. If this is the first time you are connecting to this DB instance, or if you didn't yet create a database for this RDS for PostgreSQL instance, you can connect to the postgres database using the 'master username' and password.

7. For Unix, use the following format.

```
psql \
  --host=<DB instance endpoint> \
  --port=<port> \
  --username=<master username> \
  --password \
  --dbname=<database name>
```

8. For Windows, use the following format.

```
psql ^
  --host=<DB instance endpoint> ^
  --port=<port> ^
  --username=<master username> ^
  --password ^
  --dbname=<database name>
```

For example, the following command connects to a database called mypgdb on a PostgreSQL DB instance called mypostgresql using fictitious credentials.

```
psql --host=mypostgresql.c6c8mwvfdgv0.us-west-2.rds.amazonaws.com --port=5432 --
username=awsuser --password --dbname=mypgdb
```

Backing up and restoring your DB instance

Automated backups occur daily during the preferred backup window.

If you don't specify a preferred backup window when you create the DB instance or Multi-AZ DB cluster, Amazon RDS assigns a default 30-minute backup window. This window is selected at random from an 8-hour block of time for each AWS Region.

Region Name	Region	Time Block
US East (Ohio)	us-east-2	03:00–11:00 UTC
US East (N. Virginia)	us-east-1	03:00–11:00 UTC
US West (N. California)	us-west-1	06:00–14:00 UTC
US West (Oregon)	us-west-2	06:00–14:00 UTC
Africa (Cape Town)	af-south-1	03:00–11:00 UTC
Asia Pacific (Hong Kong)	ap-east-1	06:00–14:00 UTC
Asia Pacific (Hyderabad)	ap-south-2	06:30–14:30 UTC
Asia Pacific (Jakarta)	ap-southeast-3	08:00–16:00 UTC
Asia Pacific (Melbourne)	ap-southeast-4	11:00–19:00 UTC
Asia Pacific (Mumbai)	ap-south-1	16:30–00:30 UTC
Asia Pacific (Osaka)	ap-northeast-3	00:00–08:00 UTC
Asia Pacific (Seoul)	ap-northeast-2	13:00–21:00 UTC
Asia Pacific (Singapore)	ap-southeast-1	14:00–22:00 UTC
Asia Pacific (Sydney)	ap-southeast-2	12:00–20:00 UTC
Asia Pacific (Tokyo)	ap-northeast-1	13:00–21:00 UTC
Canada (Central)	ca-central-1	03:00–11:00 UTC
Canada West (Calgary)	ca-west-1	18:00–02:00 UTC
China (Beijing)	cn-north-1	06:00–14:00 UTC
China (Ningxia)	cn-northwest-1	06:00–14:00 UTC
Europe (Frankfurt)	eu-central-1	20:00–04:00 UTC
Europe (Ireland)	eu-west-1	22:00–06:00 UTC

Europe (London)	eu-west-2	22:00–06:00 UTC
Europe (Milan)	eu-south-1	02:00–10:00 UTC
Europe (Paris)	eu-west-3	07:29–14:29 UTC
Europe (Spain)	eu-south-2	02:00–10:00 UTC
Europe (Stockholm)	eu-north-1	23:00–07:00 UTC
Europe (Zurich)	eu-central-2	02:00–10:00 UTC
Israel (Tel Aviv)	il-central-1	03:00–11:00 UTC
Middle East (Bahrain)	me-south-1	06:00–14:00 UTC
Middle East (UAE)	me-central-1	05:00–13:00 UTC
South America (São Paulo)	sa-east-1	23:00–07:00 UTC
AWS GovCloud (US-East)	us-gov-east-1	17:00–01:00 UTC
AWS GovCloud (US-West)	us-gov-west-1	06:00–14:00 UTC

For MariaDB, MySQL, Oracle, and PostgreSQL, I/O activity isn't suspended on your primary during backup for Multi-AZ deployments because the backup is taken from the standby.

Backup retention period

You can set the backup retention period when you create a DB instance or Multi-AZ DB cluster. If you create a DB instance using the Amazon RDS API or the AWS CLI and if you don't set the backup retention period, the default backup retention period is one day. If you create a DB instance using the console, the default backup retention period is seven days.

After you create a DB instance or cluster, you can modify the backup retention period. You can set the backup retention period of a DB instance to between 0 and 35 days. Setting the backup retention period to 0 disables automated backups. For a Multi-AZ DB cluster, you can set the backup retention period to between 1 and 35 days. Manual snapshot limits (100 per Region) don't apply to automated backups.

Important: An outage occurs if you change the backup retention period of a DB instance from 0 to a nonzero value or from a nonzero value to 0.

RDS doesn't include time spent in the stopped state when the backup retention period is calculated. Automated backups aren't created while a DB instance or cluster is stopped. Backups can be retained longer than the backup retention period if a DB instance has been stopped.

To enable automated backups immediately

1. Sign in to the AWS Management Console and open the Amazon RDS
2. In the navigation pane, choose Databases, and then choose the DB instance or Multi-AZ DB cluster that you want to modify.
3. Choose Modify.
4. For Backup retention period, choose a positive nonzero value, for example 3 days.
5. Choose Continue.
6. Choose Apply immediately.
7. Choose Modify DB instance or Modify cluster to save your changes and enable automated backups.

Viewing automated backups
To view your automated backups, choose Automated backups in the navigation pane.

Retaining automated backups

Note: You can only retain automated backups of DB instances, not Multi-AZ DB clusters.

When you delete a DB instance, you can choose to retain automated backups. Automated backups can be retained for a number of days equal to the backup retention period configured for the DB instance at the time when you delete it.

Retained automated backups contain system snapshots and transaction logs from a DB instance. They also include your DB instance properties like allocated storage and DB instance class, which are required to restore it to an active instance.

Retained automated backups and manual snapshots incur billing charges until they're deleted.

You can retain automated backups for RDS instances running the Db2, MariaDB, MySQL, PostgreSQL, Oracle, and Microsoft SQL Server engines.

Retention period
The system snapshots and transaction logs in a retained automated backup expire the same way that they expire for the source DB instance. Because there are no new snapshots or logs created for this instance, the retained automated backups eventually expire completely.

Viewing retained backups
To view your retained automated backups, choose Automated backups in the navigation pane, then choose Retained. To view individual snapshots associated with a retained automated backup, choose Snapshots in the navigation pane.

Restoring a DB instance to a specified time

- You can restore a DB instance to a specific point in time, creating a new DB instance without modifying the source DB instance.

- When you restore a DB instance to a point in time, you can choose the default virtual private cloud (VPC) security group. Or you can apply a custom VPC security group to your DB instance.

- Restored DB instances are automatically associated with the default DB parameter and option groups. However, you can apply a custom parameter group and option group by specifying them during a restore.

- If the source DB instance has resource tags, RDS adds the latest tags to the restored DB instance.

- Note: We recommend that you restore to the same or similar DB instance size—and IOPS if using Provisioned IOPS storage—as the source DB instance. You might get an error if, for example, you choose a DB instance size with an incompatible IOPS value.

To restore a DB instance to a specified time

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Automated backups.
3. The automated backups are displayed on the Current Region tab.
4. Choose the DB instance that you want to restore.
5. For Actions, choose Restore to point in time.
6. The Restore to point in time window appears.
7. Choose Latest restorable time to restore to the latest possible time, or choose Custom to choose a time.
8. If you chose Custom, enter the date and time to which you want to restore the instance.
9. Note: Times are shown in your local time zone, which is indicated by an offset from Coordinated Universal Time (UTC). For example, UTC-5 is Eastern Standard Time/Central Daylight Time.
10. For DB instance identifier, enter the name of the target restored DB instance. The name must be unique.
11. Choose other options as needed, such as DB instance class, storage, and whether you want to use storage autoscaling.
12. Choose Restore to point in time.

To delete a retained automated backup

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Automated backups.
3. On the Retained tab, choose the retained automated backup that you want to delete.
4. For Actions, choose Delete.
5. On the confirmation page, enter delete me and choose Delete.

Retention costs

The cost of a retained automated backup is the cost of total storage of the system snapshots that are associated with it. There is no additional charge for transaction logs or instance metadata. All other pricing rules for backups apply to restorable instances.

For example, suppose that your total allocated storage of running instances is 100 GB. Suppose also that you have 50 GB of manual snapshots plus 75 GB of system snapshots associated with a retained automated backup. In this case, you are charged only for the additional 25 GB of backup storage, like this: (50 GB + 75 GB) – 100 GB = 25 GB.

To disable automated backups immediately

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Databases, and then choose the DB instance or Multi-AZ DB cluster that you want to modify.
3. Choose Modify.
4. For Backup retention period, choose 0 days.
5. Choose Continue.
6. Choose Apply immediately.
7. Choose Modify DB instance or Modify cluster to save your changes and disable automated backups.

Monitoring the activity and performance of your DB instance

Amazon RDS integrates with Amazon CloudWatch to display a variety of RDS DB instance metrics in the RDS console. For descriptions of these metrics, see Metrics reference for Amazon RDS.

For your DB instance, the following categories of metrics are monitored:

- CloudWatch – Shows the Amazon CloudWatch metrics for RDS that you can access in the RDS console. You can also access these metrics in the CloudWatch console. Each metric includes a graph that shows the metric monitored over a specific time span. For a list of CloudWatch metrics, see Amazon CloudWatch metrics for Amazon RDS.

- Enhanced monitoring – Shows a summary of operating-system metrics when your RDS DB instance has turned on Enhanced Monitoring. RDS delivers the metrics from Enhanced Monitoring to your Amazon CloudWatch Logs account. Each OS metric includes a graph showing the metric monitored over a specific time span. For an overview, see Monitoring OS metrics with Enhanced Monitoring. For a list of Enhanced Monitoring metrics, see OS metrics in Enhanced Monitoring.

- OS Process list – Shows details for each process running in your DB instance.

- Performance Insights – Opens the Amazon RDS Performance Insights dashboard for a DB instance. For an overview of Performance Insights, see Monitoring DB load with Performance Insights on Amazon RDS. For a list of Performance Insights metrics, see Amazon CloudWatch metrics for Amazon RDS Performance Insights.

Amazon RDS now provides a consolidated view of Performance Insights and CloudWatch metrics in the Performance Insights dashboard. Performance Insights must be turned on for your DB instance to use this view. You can choose the new monitoring view in the Monitoring tab or Performance Insights in the navigation pane.

Viewing Amazon RDS events

You can retrieve the following event information for your Amazon RDS resources:

- Resource name
- Resource type
- Time of the event
- Message summary of the event

You can access events in the following parts of the AWS Management Console:

- The Events tab, which shows events from the past 24 hours.
- The Recent events table in the Logs & events section in the Databases tab, which can show events for up to the past 2 weeks.

Notes:

✓ You can also retrieve events by using the describe-events AWS CLI command, or the DescribeEvents RDS API operation. If you use the AWS CLI or the RDS API to view events, you can retrieve events for up to the past 14 days.

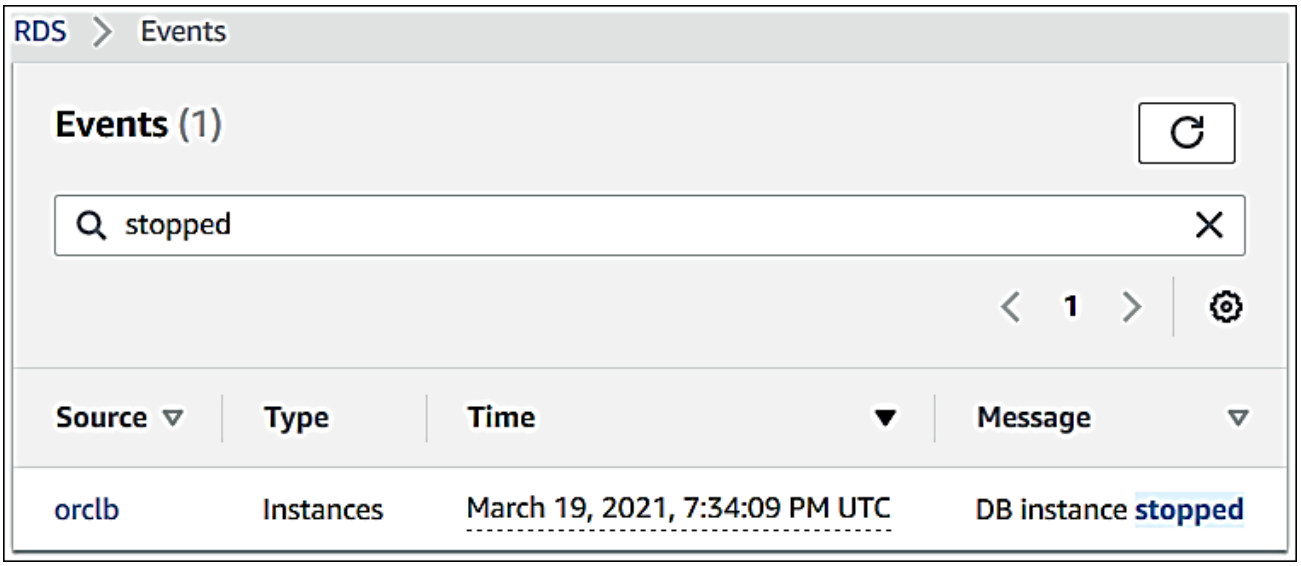
✓ If you need to store events for longer periods of time, you can send Amazon RDS events to EventBridge. For more information, see Creating a rule that triggers on an Amazon RDS event

✓ For descriptions of the Amazon RDS events, see Amazon RDS event categories and event messages.

✓ To access detailed information about events using AWS CloudTrail, including request parameters, see CloudTrail events.

To view all Amazon RDS events for the past 24 hours in the Console

1. Sign in to the AWS Management Console and open the Amazon RDS console
2. In the navigation pane, choose Events.
3. The available events appear in a list.
4. (Optional) Enter a search term to filter your results.
5. The following example shows a list of events filtered by the characters stopped.



Working with log files

RDS for PostgreSQL logs database activities to the default PostgreSQL log file. For an on-premises PostgreSQL DB instance, these messages are stored locally in log/postgresql.log. For an RDS for PostgreSQL DB instance, the log file is available on the Amazon RDS instance. Also, you must use the Amazon RDS Console to view or download its contents. The default logging level captures login failures, fatal server errors, deadlocks, and query failures.

AWS provides two managed PostgreSQL options: Amazon RDS for PostgreSQL and Amazon Aurora PostgreSQL. One of the common questions that new PostgreSQL users ask is how to capture the database activity logs for debugging and monitoring purposes.

Each RDS and Aurora PostgreSQL instance is associated with a parameter group that contains the engine specific configurations. The engine configurations also include several parameters that control the PostgreSQL logging behavior. AWS provides the parameter groups with default configuration settings to use for your instances.

PostgreSQL log file

The log files store the engine logs that the RDS and Aurora PostgreSQL instances generate. PostgreSQL provides a few parameters when choosing the naming convention and rotation policy of the log file. These parameters provide the ability to generate periodic or fixed max size log files.

The parameter log_filename specifies the name of the log file. Aurora PostgreSQL 9.6 and RDS PostgreSQL allow the following two file naming options:

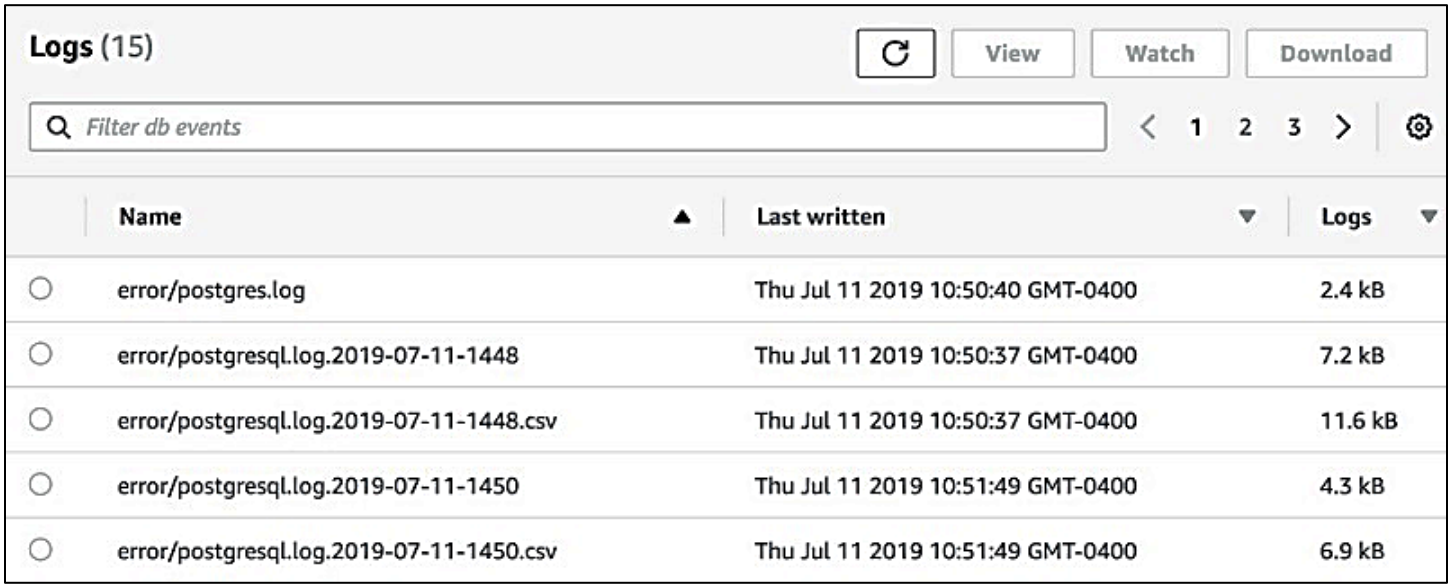
- postgresql.log.%Y-%m-%d – This option uses a unique log file name for each day. For example, postgresql.log.2019-04-01.
- postgresql.log.%Y-%m-%d-%H – This option uses a unique log file name for each hour of the day. For example, postgresql.log.2019-04-01-12.

Aurora PostgreSQL 10 introduced an additional log file name option, postgresql.log.%Y-%m-%d-%H%M. This option allows you to use a unique log file name with minute of the day. For example, the name of a new log file that starts at 12:30pm will be postgresql.log.2019-04-01-1230.

PostgreSQL creates and starts using a new log file when the conditions specified by parameters log_rotation_age or log_rotation_size are met. The parameter log_rotation_age specifies the maximum age (in minutes) for any log files. After this duration, PostgreSQL generates a new log file using the file naming convention. Similarly, the parameter log_rotation_size specifies the maximum size (in KB) of the log file. When the log file reaches this size, PostgreSQL generates a new log file using the file naming convention.

RDS and Aurora PostgreSQL do not overwrite the existing log files. For example, assume you are using the daily log naming convention (postgresql.log.%Y-%m-%d) and the log_rotation_size is 1 MB. In this example, when the file reaches 1 MB and the date has not changed, PostgreSQL continues writing to the

same file without truncating it. The parameter log_truncate_on_rotation, which is hardcoded to false in RDS and Aurora PostgreSQL, controls this behavior.



Configuring logging parameters

PostgreSQL users can enable additional logging by tuning the engine parameters. This section provides some of the key parameters you can set to capture more engine activity logs. For more information about additional logging parameters, see the section Error Reporting and Logging in the PostgreSQL documentation.

Increasing database logging impacts storage size, I/O use, and CPU use. Because of this, test these changes before deploying them in production.

rds.log_retention_period

This parameter specifies the retention period of the RDS and Aurora PostgreSQL engine log files (in minutes). By default, the logs are retained for 4320 minutes (three days). The max allowed value for this parameter is 10080 minutes (seven days). If you need to retain the logs for longer, you must move them out of this default location. Part 2 of this post discusses how to export the log files.

log_connections

By default, new connection details are not logged. To log all new client connection details, set this parameter to 1. The following output shows an example log for a new client connection.

Standard error format
user=postgres database=logtestdb SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, compression=off)

CSV format
2019-03-10 03:51:56 UTC:10.0.0.123(52820):postgres@logtestdb:[18161]:LOG: connection authorized: 2019-03-10 03:51:56.158 UTC,"postgres","logtestdb",18161,"10.0.0.123:52820",5c8489dc.46f1,2,"authentication",2019-03-10 03:51:56 UTC,5/17090128,0,LOG,00000,"connection authorized: user=postgres database=logtestdb SSL enabled (protocol=TLSv1.2, cipher=ECDHE-RSA-AES256-GCM-SHA384, compression=off)",,,,,,,,,""

The connection logs can help determine the source, username, frequency, and time of the connections. This information can be useful in troubleshooting activities or security audits. Enabling this logging can help you answer questions such as how many new connections are received per day, what the count of connections coming from each host or user is, or if SSL is used for all connections.

In general, it is recommended to keep the connection logging enabled. Applications frequently use connection poolers, which result in connections opening one time and staying open for longer. When a smaller number of new connections open, fewer connection log messages are captured in the log files.

log_disconnections

By default, disconnection details are not logged. To log all client disconnections, set this parameter to 1. The following output shows an example log for a client disconnection.

Standard error format
2019-03-10 03:52:14 UTC:10.0.0.123(52820):postgres@logtestdb:[18161]:LOG: disconnection: session time: 0:00:18.720 user=postgres database=logtestdb host=10.0.0.123 port=52820

CSV format
2019-03-10 03:51:56 UTC,,0,LOG,00000,"disconnection: session time: 0:00:18.720 user=postgres database=logtestdb host=10.0.0.123 port=52820",,,,,,,,,"psql"

You can use the disconnection information to determine exactly how long a user was connected to the database. This information is useful for auditing purposes. In general, it is recommended to keep the disconnection logging enabled.

log_temp_files

Queries use temporary files on disk if an operation cannot be completed using the allowed memory (work_mem). These operations involve storing sorts, hashes, and temporary query results. The performance of disk access is much slower than the memory access; therefore, whenever the query needs to use temporary files, there is a performance impact. Because of this, it is important to enable temporary file usage logging. You can enable this logging by setting log_temp_files parameter to 0 (to log all temporary files) or to any positive number (default units KB). This positive number specifies the temporary file size threshold beyond which it is logged.

For example, you get the following log message when a query uses a temporary file that is greater than or equal to the threshold size specified by log_temp_files.

Standard error format
2019-03-10 03:58:18 UTC:10.0.0.123(52834):postgres@logtestdb:[20175]:LOG: temporary file: path "base/pgsql_tmp/pgsql_tmp20175.0", size 14032896
2019-03-10 03:58:18 UTC:10.0.0.123(52834):postgres@logtestdb:[20175]:STATEMENT: SELECT DISTINCT id FROM logtest1;

CSV format
2019-03-10 03:58:18.879 UTC,"postgres","logtestdb",20175,"10.0.0.123:52834",5c848a8c.4ecf,4,"SELECT",2019-03-10 03:54:52 UTC,5/17090205,0,LOG,00000,"temporary file: path ""base/pgsql_tmp/pgsql_tmp20175.0"", size 14032896",,,,,,"SELECT DISTINCT id FROM logtest1;",,,"psql"

The log message shows the size of the temporary file generated by the query. Using this information, you may increase the work_mem, if possible, or tune the query to minimize the temporary file usage.

log_lock_waits

Setting this parameter to 1 enables logging sessions that are stuck in a locked state for a duration longer than the duration set for deadlock detection (1 second by default). This information helps determine if session locking is causing a performance issue.

For example, when one transaction blocks another transaction, you get the following log message.

Standard error format
2019-03-10 04:01:22 UTC:10.0.0.123(52806):postgres@logtestdb:[9732]:LOG: process 9732 still waiting for ShareLock on transaction 113942590 after 1000.045 ms
2019-03-10 04:01:22 UTC:10.0.0.123(52806):postgres@logtestdb:[9732]:DETAIL: Process holding the lock: 20175. Wait queue: 9732.
2019-03-10 04:01:22 UTC:10.0.0.123(52806):postgres@logtestdb:[9732]:CONTEXT: while updating tuple (10809,11) in relation "logtest1"
2019-03-10 04:01:22 UTC:10.0.0.123(52806):postgres@logtestdb:[9732]:STATEMENT: update logtest1 set date = now() where id=1;

CSV format
2019-03-10 04:01:22.866 UTC,"postgres","logtestdb",9732,"10.0.0.123:52806",5c847d81.2604,1,"UPDATE waiting",2019-03-10 02:59:13 UTC,6/909514,113942591,LOG,00000,"process 9732 still waiting for ShareLock on transaction 113942590 after 1000.045 ms","Process holding the lock: 20175. Wait queue: 9732.",,,,"while updating tuple (10809,11) in relation ""logtest1""","update logtest1 set date = now() where id=1;",,,"psql"

This log message shows the query waiting for the lock and also provides the PID of the blocking session. This information helps debug performance issues. Using this information, you can identify the transactions that are blocking each other and can optimize them to avoid or reduce

the locking. For example, two transactions that attempt to update the same row lock each other. This error message can help you isolate these transactions so that you can improve them.

log_autovacuum_min_duration

The auto-vacuum daemon process runs in the background as per the configurations and prevents bloat from accumulating in the database. This functionality is important, but it comes at a cost in terms of CPU, memory, and IO resource usage. The parameter log_autovacuum_min_duration helps gain visibility into when this background process runs and for which tables. Setting this parameter to 0 starts logging information of all auto-vacuum and auto-analyze runs. Setting the parameter to any positive value (units milliseconds) creates logs for only the auto-vacuum and auto-analyze runs that take more than this time.

For example, you get the following log message for the auto-vacuum and auto-analyze runs after setting this parameter and rds.force_autovacuum_logging_level (explained in the following section) set to log.

Standard error format
2019-03-10 04:07:12 UTC::@:[29679]:LOG: automatic vacuum of table "logtestdb.public.logtest1": index scans: 0
pages: 0 removed, 10811 remain, 0 skipped due to pins, 0 skipped frozen
tuples: 1000001 removed, 1000000 remain, 0 are dead but not yet removable, oldest xmin: 113942594
buffer usage: 21671 hits, 0 misses, 1 dirtied
avg read rate: 0.000 MB/s, avg write rate: 0.003 MB/s
system usage: CPU: user: 0.12 s, system: 0.00 s, elapsed: 2.30 s
2019-03-10 04:07:14 UTC::@:[29679]:LOG: automatic analyze of table "logtestdb.public.logtest1"
system usage: CPU: user: 0.06 s, system: 0.00 s, elapsed: 1.17 s

CSV format
2019-03-10 04:07:12.938 UTC,,,29679,,5c848d6e.73ef,1,,2019-03-10 04:07:10 UTC,7/15686043,0,LOG,00000,"automatic vacuum of table ""logtestdb.public.logtest1"": index scans: 0
pages: 0 removed, 10811 remain, 0 skipped due to pins, 0 skipped frozen
tuples: 1000001 removed, 1000000 remain, 0 are dead but not yet removable, oldest xmin: 113942594
buffer usage: 21671 hits, 0 misses, 1 dirtied
avg read rate: 0.000 MB/s, avg write rate: 0.003 MB/s
system usage: CPU: user: 0.12 s, system: 0.00 s, elapsed: 2.30 s",,,,,,,,,""
2019-03-10 04:07:14.109 UTC,,,29679,,5c848d6e.73ef,2,,2019-03-10 04:07:10 UTC,7/15686044,113942594,LOG,00000,"automatic analyze of table ""logtestdb.public.logtest1""
system usage: CPU: user: 0.06 s, system: 0.00 s, elapsed: 1.17 s",,,,,,,,,""

This log message tells you when the auto-vacuum was running, the time it took to complete, and the cleanups it performed. This information is useful when you are investigating performance issues such as high I/O or CPU load during peak hours, and table bloat increasing because auto-vacuum can't process old rows due to locks.

rds.force_autovacuum_logging_level

In RDS and Aurora PostgreSQL, logging auto-vacuum and auto-analyze processes is disabled by default. The auto-vacuum logging parameter log_autovacuum_min_duration does not work until you set this parameter to the desired values. The allowed values for this parameter are disabled, debug5, debug4, debug3, debug2, debug1, info, notice, warning, error, log, fatal, and panic.

When you set this parameter to a value other than disabled, such as log, the PostgreSQL engine starts logging the auto-vacuum and auto-analyze activity as per the threshold that the parameter log_autovacuum_min_duration set.

log_min_duration_statement

This parameter provides a valuable feature that enables logging slow queries. When set to a value (units milliseconds) other than -1 (means disabled), the database starts logging all queries that take at least the specified number of milliseconds.

For example, when a query takes time greater than or equal to the threshold that log_min_duration_statement specifies, you get the following log message.

Standard error format
2019-03-10 04:05:23 UTC:10.0.0.123(52834):postgres@logtestdb:[20175]:LOG: duration: 100.507 ms statement: SELECT count(*) FROM logtest1 where value<10;

CSV format
2019-03-10 04:05:23.199
UTC,"postgres","logtestdb",20175,"10.0.0.123:52834",5c848a8c.4ecf,6,"SELECT",2019-03-10 03:54:52 UTC,5/0,0,LOG,00000,"duration: 100.507 ms statement: SELECT count(*) FROM logtest1 where value<10;",,,,,,,,,"psql"

In addition to providing the duration of the query, the log message also provides information about who ran the query, from which source IP, and at what time. This information helps identify the source of the query and determine the next steps.

This parameter can be helpful in many scenarios. For example, if all your queries are expected to complete within 1 second, you can set this parameter to 1 second to capture query that takes more than this time. Similarly, when working on a query performance improvement task, you may want to start by focusing on the slowest queries and work on them first. You can do this by setting this parameter to a cutoff value and start capturing all queries that are taking more than this cutoff time. After you have optimized all these queries, you can reduce this parameter further and start working on the next set of slow queries.

log_statement

This is another useful query logging feature. The supported parameter values are as follows:

"none" – This feature is off.
"ddl" – Logs all DDL statements.
"mod" – Logs all INSERT, UPDATE, and DELETE statements.
"all" – Logs all statements.

It is a good idea to set this parameter to ddl to capture all schema change-related statements for auditing purposes.

For example, dropping a column captures the following log message.

Standard error format
2019-03-10 04:08:47 UTC:10.0.0.123(52834):postgres@logtestdb:[20175]:LOG: statement: ALTER TABLE logtest1 DROP COLUMN date;

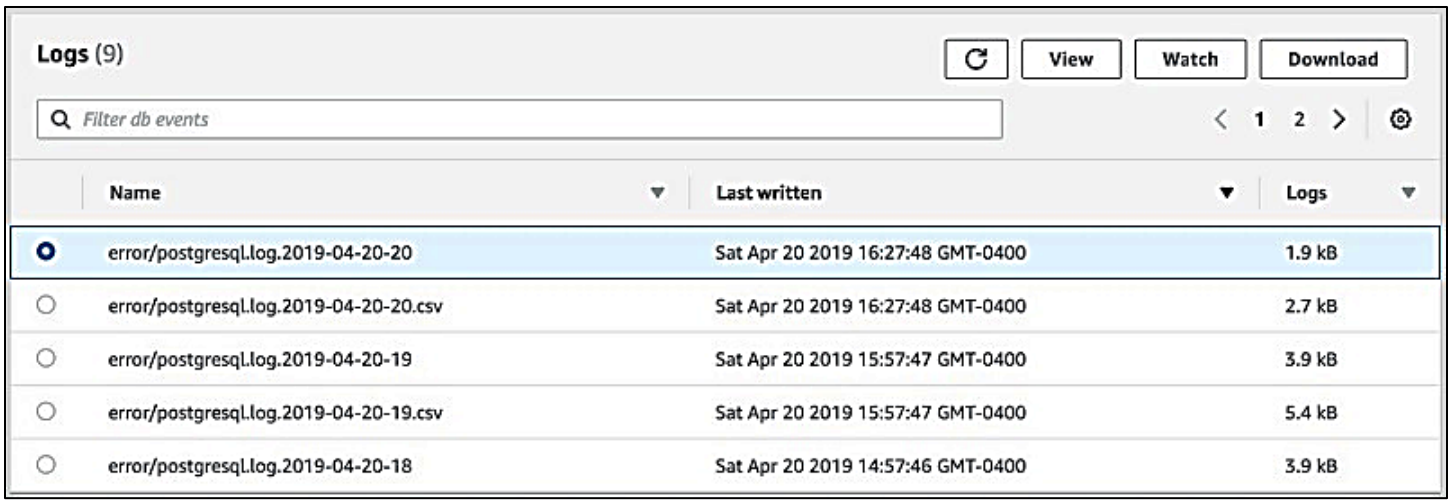
CSV format
2019-03-10 04:08:47.606
UTC,"postgres","logtestdb",20175,"10.0.0.123:52834",5c848a8c.4ecf,8,"idle",2019-03-10 03:54:52 UTC,5/17090210,0,LOG,00000,"statement: ALTER TABLE logtest1 DROP COLUMN date;",,,,,,,,,"psql"

rds.force_admin_logging_level

RDS and Aurora PostgreSQL have an internal superuser named rdsadmin. This user is only used for database management activities. For example, you might forget the password for the master user and want to reset it. The rdsadminuser performs the password reset when you change your password from the AWS Management Console. This user's activities aren't captured even after setting the parameters discussed in previous sections. If there is a requirement to capture the activities from this user, enable this by setting rds.force_admin_logging_level to log. When you set this parameter, the PostgreSQL engine starts capturing all queries executed by this admin user in any of the user databases.
Beyond native logging parameters

The most direct method to access the PostgreSQL log files is through the AWS Management Console. Complete the following steps:

1. Open the Amazon RDS
2. Choose your RDS/Aurora instance.
3. Choose Logs and events.
4. In Logs, choose the required log file.
5. Choose either View, Watch, or Download.



Turning on query logging for your RDS for PostgreSQL DB instance

You can collect more detailed information about your database activities, including queries, queries waiting for locks, checkpoints, and many other details by setting some of the parameters listed in the following table. This topic focuses on logging queries.

Parameter	Default	Description
log_connections	–	Logs each successful connection.
log_disconnections	–	Logs the end of each session and its duration.
log_checkpoints	1	Logs each checkpoint.
log_lock_waits	–	Logs long lock waits. By default, this parameter isn't set.
log_min_duration_sample	–	(ms) Sets the minimum execution time above which a sample of statements is logged. Sample size is set using the log_statement_sample_rate parameter.
log_min_duration_statement	–	Any SQL statement that runs atleast for the specified amount of time or longer gets logged. By default, this parameter isn't set. Turning on this parameter can help you find unoptimized queries.
log_statement	–	Sets the type of statements logged. By default, this parameter isn't set, but you can change it to all, ddl, or mod to specify the types of SQL statements that you want logged. If you specify anything other than none for this parameter, you should also take additional steps to prevent the exposure of passwords in the log files.
log_statement_sample_rate	–	The percentage of statements exceeding the time specified in log_min_duration_sample to be logged, expressed as a floating point value between 0.0 and 1.0.
log_statement_stats	–	Writes cumulative performance statistics to the server log.

Using logging to find slow performing queries

You can log SQL statements and queries to help find slow performing queries. You turn on this capability by modifying the settings in the log_statement and log_min_duration parameters as outlined in this section.

Before turning on query logging for your RDS for PostgreSQL DB instance, you should be aware of possible password exposure in the logs and how to mitigate the risks.

Following, you can find reference information about the log_statement and log_min_duration parameters.

log_statement

This parameter specifies the type of SQL statements that should get sent to the log. The default value is none. If you change this parameter to all, ddl, or mod, be sure to apply recommended actions to mitigate the risk of exposing passwords in the logs. For more information, see Mitigating risk of password exposure when using query logging.

all
Logs all statements. This setting is recommended for debugging purposes.

ddl
Logs all data definition language (DDL) statements, such as CREATE, ALTER, DROP, and so on.

mod
Logs all DDL statements and data manipulation language (DML) statements, such as INSERT, UPDATE, and DELETE, which modify the data.

none
No SQL statements get logged. We recommend this setting to avoid the risk of exposing passwords in the logs.

log_min_duration_statement

Any SQL statement that runs atleast for the specified amount of time or longer gets logged. By default, this parameter isn't set. Turning on this parameter can help you find unoptimized queries.

–1–2147483647

The number of milliseconds (ms) of runtime over which a statement gets logged.

To set up query logging

These steps assume that your RDS for PostgreSQL DB instance uses a custom DB parameter group.

1. Set the log_statement parameter to all. The following example shows the information that is written to the postgresql.log file with this parameter setting.

```
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:LOG: statement: SELECT feedback,
s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:LOG: QUERY STATISTICS
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:DETAIL: ! system usage stats:
! 0.017355 s user, 0.000000 s system, 0.168593 s elapsed
! [0.025146 s user, 0.000000 s system total]
! 36644 kB max resident size
! 0/8 [0/8] filesystem blocks in/out
! 0/733 [0/1364] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 19/0 [27/0] voluntary/involuntary context switches
2022-10-05 22:05:52 UTC:52.95.4.1(11335):postgres@labdb:[3639]:STATEMENT: SELECT feedback,
s.sentiment,s.confidence
FROM support,aws_comprehend.detect_sentiment(feedback, 'en') s
ORDER BY s.confidence DESC;
2022-10-05 22:05:56 UTC:52.95.4.1(11335):postgres@labdb:[3639]:ERROR: syntax error at or near
"ORDER" at character 1
2022-10-05 22:05:56 UTC:52.95.4.1(11335):postgres@labdb:[3639]:STATEMENT: ORDER BY s.confidence
DESC;
---------------------- END OF LOG ----------------------
```

2. Set the log_min_duration_statement parameter. The following example shows the information that is written to the postgresql.log file when the parameter is set to 1.

   Queries that exceed the duration specified in the log_min_duration_statement parameter are logged. The following shows an example. You can view the log file for your RDS for PostgreSQL DB instance in the Amazon RDS Console.

```
2022-10-05 19:05:19 UTC:52.95.4.1(6461):postgres@labdb:[6144]:LOG: statement: DROP table
comments;
2022-10-05 19:05:19 UTC:52.95.4.1(6461):postgres@labdb:[6144]:LOG: duration: 167.754 ms
2022-10-05 19:08:07 UTC::@:[355]:LOG: checkpoint starting: time
2022-10-05 19:08:08 UTC::@:[355]:LOG: checkpoint complete: wrote 11 buffers (0.0%); 0 WAL file(s)
added, 0 removed, 0 recycled; write=1.013 s, sync=0.006 s, total=1.033 s; sync files=8, longest=0.004 s,
average=0.001 s; distance=131028 kB, estimate=131028 kB
---------------------- END OF LOG ----------------------
```

Understanding the best practices for PostgreSQL DB instances

Watch Video here: https://youtu.be/3JLPWOoiVB8

Of two important areas where you can improve performance with RDS for PostgreSQL, one is when loading data into a DB instance. Another is when using the PostgreSQL autovacuum feature.

Loading data into a PostgreSQL DB instance
When loading data into an Amazon RDS for PostgreSQL DB instance, modify your DB instance settings and your DB parameter group values. Set these to allow for the most efficient importing of data into your DB instance.

1. Modify your DB instance settings to the following:
   - Disable DB instance backups (set backup_retention to 0)
   - Disable Multi-AZ

2. Modify your DB parameter group to include the following settings. Also, test the parameter settings to find the most efficient settings for your DB instance.
   - Increase the value of the maintenance_work_mem parameter. For more information about PostgreSQL resource consumption parameters, see the PostgreSQL documentation
   - Increase the value of the max_wal_size and checkpoint_timeout parameters to reduce the number of writes to the write-ahead log (WAL) log.
   - Disable the synchronous_commit parameter.
   - Disable the PostgreSQL autovacuum parameter.
   - Make sure that none of the tables you're importing are unlogged. Data stored in unlogged tables can be lost during a failover. For more information, see CREATE TABLE UNLOGGED

3. Use the pg_dump -Fc (compressed) or pg_restore -j (parallel) commands with these settings.
4. After the load operation completes, return your DB instance and DB parameters to their normal settings.

Working with the PostgreSQL autovacuum feature
The autovacuum feature for PostgreSQL databases is a feature that we strongly recommend you use to maintain the health of your PostgreSQL DB instance. Autovacuum automates the execution of the VACUUM and ANALYZE command Using autovacuum is required by PostgreSQL, not imposed by Amazon RDS, and its use is critical to good performance. The feature is enabled by default for all new Amazon RDS for PostgreSQL DB instances, and the related configuration parameters are appropriately set by default.

Autovacuum is not a "resource free" operation, but it works in the background and yields to user operations as much as possible. When enabled, autovacuum checks for tables that have had a large number of updated or deleted tuples. It also protects against loss of very old data due to transaction ID wraparound. For more information, see Preventing transaction ID wraparound failures

.

Autovacuum should not be thought of as a high-overhead operation that can be reduced to gain better performance. On the contrary, tables that have a high velocity of updates and deletes will quickly deteriorate over time if autovacuum is not run.

Important: Not running autovacuum can result in an eventual required outage to perform a much more intrusive vacuum operation. In some cases, an RDS for PostgreSQL DB instance might become unavailable because of an over-conservative use of autovacuum. In these cases, the PostgreSQL database shuts down to protect itself. At that point, Amazon RDS must perform a single-user-mode full vacuum directly on the DB instance. This full vacuum can result in a multi-hour outage. Thus, we strongly recommend that you do not turn off autovacuum, which is turned on by default.

The autovacuum parameters determine when and how hard autovacuum works. Theautovacuum_vacuum_threshold and autovacuum_vacuum_scale_factor parameters determine when autovacuum is run. The autovacuum_max_workers, autovacuum_nap_time, autovacuum_cost_limit, and autovacuum_cost_delay parameters determine how hard autovacuum works.

The following query shows the number of "dead" tuples in a table named table1:

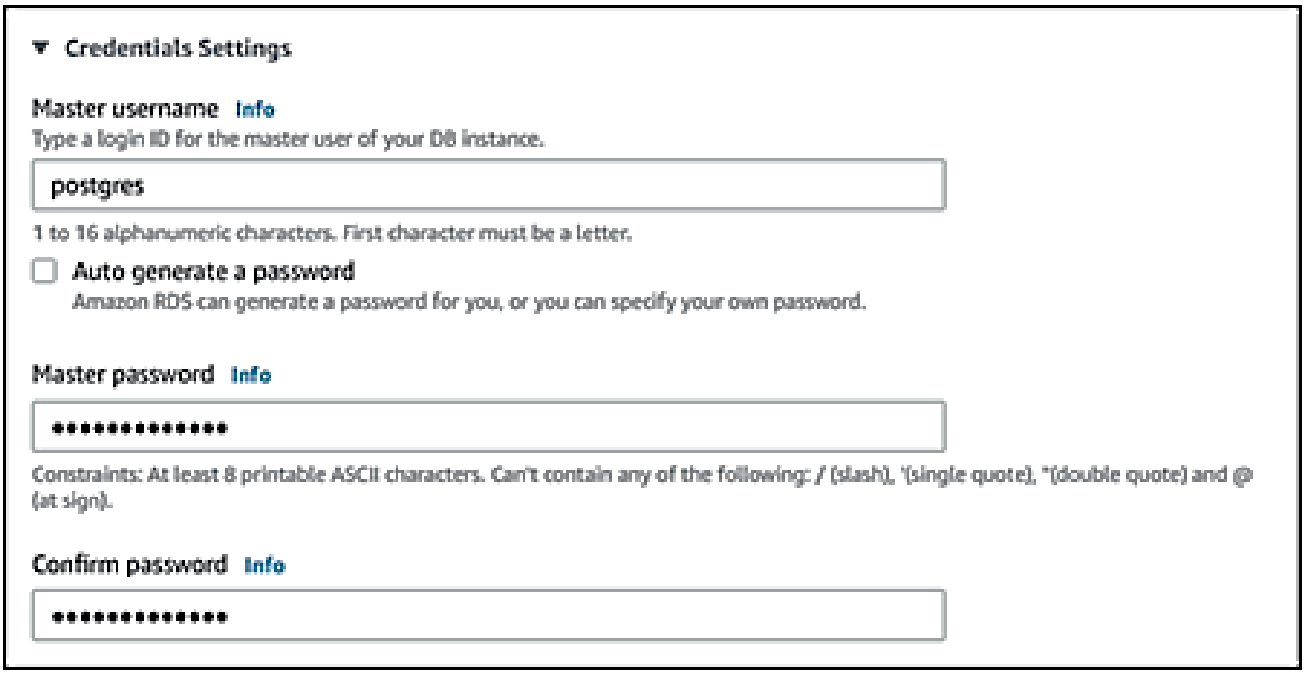
```
SELECT relname, n_dead_tup, last_vacuum, last_autovacuum FROM
pg_catalog.pg_stat_all_tables
WHERE n_dead_tup > 0 and relname = 'table1';
```

The results of the query will resemble the following:

```
relname | n_dead_tup | last_vacuum | last_autovacuum
---------+------------+-------------+-----------------
 tasks   |   81430522 |             |
(1 row)
```

Understanding PostgreSQL roles and permissions

When you create an RDS for PostgreSQL DB instance using the AWS Management Console, an administrator account is created at the same time. By default, its name is postgres, as shown in the following screenshot:



Understanding the rds_superuser role

In PostgreSQL, a role can define a user, a group, or a set of specific permissions granted to a group or user for various objects in the database. PostgreSQL commands to CREATE USER and CREATE GROUP have been replaced by the more general, CREATE ROLE with specific properties to distinguish database users. A database user can be thought of as a role with the LOGIN privilege.

Note: The CREATE USER and CREATE GROUP commands can still be used.

The postgres user is the most highly privileged database user on your RDS for PostgreSQL DB instance. It has the characteristics defined by the following CREATE ROLE statement.

```
CREATE ROLE postgres WITH LOGIN NOSUPERUSER INHERIT CREATEDB CREATEROLE NOREPLICATION VALID UNTIL 'infinity'
```

The properties NOSUPERUSER, NOREPLICATION, INHERIT, and VALID UNTIL 'infinity' are the default options for CREATE ROLE, unless otherwise specified.

By default, postgres has privileges granted to the rds_superuser role, and permissions to create roles and databases. The rds_superuser role allows the postgres user to do the following:

- Add extensions that are available for use with Amazon RDS
- Create roles for users and grant privileges to users.
- Create databases.
- Grant rds_superuser privileges to user roles that don't have these privileges, and revoke privileges as needed. We recommend that you grant this role only to those users who perform superuser tasks. In other words, you can grant this role to database administrators (DBAs) or system administrators.

- Grant (and revoke) the rds_replication role to database users that don't have the rds_superuser role.
- Grant (and revoke) the rds_password role to database users that don't have the rds_superuser role.
- Obtain status information about all database connections by using the pg_stat_activity view. When needed, rds_superuser can stop any connections by using pg_terminate_backend or pg_cancel_backend.

In the CREATE ROLE postgres… statement, you can see that the postgres user role specifically disallows PostgreSQL superuser permissions. RDS for PostgreSQL is a managed service, so you can't access the host OS, and you can't connect using the PostgreSQL superuser account. Many of the tasks that require superuser access on a stand-alone PostgreSQL are managed automatically by Amazon RDS.

The rds_superuser role is one of several predefined roles in an RDS for PostgreSQL DB instance.

In the following list, you find some of the other predefined roles that are created automatically for a new RDS for PostgreSQL DB instance. Predefined roles and their privileges can't be changed. You can't drop, rename, or modify privileges for these predefined roles. Attempting to do so results in an error.

- rds_password – A role that can change passwords and set up password constraints for database users. The rds_superuser role is granted with this role by default, and can grant the role to database users.
- rdsadmin – A role that's created to handle many of the management tasks that the administrator with superuser privileges would perform on a standalone PostgreSQL database. This role is used internally by RDS for PostgreSQL for many management tasks.
- rdstopmgr – A role that's used internally by Amazon RDS to support Multi-AZ deployments.

To see all predefined roles, you can connect to your RDS for PostgreSQL DB instance and use the ff metacommand:

```
psql \du
```

```
List of roles
 Role name          |    Attributes                          |    Member of
--------------------+----------------------------------------+---------------------------------
postgres            | Create role, Create DB          +      | {rds_superuser}
                    | Password valid until infinity          |
rds_superuser       | Cannot login                           | {pg_monitor,pg_signal_backend,
                    |                    +                   |  rds_replication,rds_password}
…
```

In the output, you can see that rds_superuser isn't a database user role (it can't login), but it has the privileges of many other roles. You can also see that database user postgres is a member of the rds_superuser role. As mentioned previously, postgres is the default value in the Amazon RDS console's Create database page. If you chose another name, that name is shown in the list of roles instead.

Controlling user access to the PostgreSQL database

New databases in PostgreSQL are always created with a default set of privileges in the database's public schema that allow all database users and roles to create objects. These privileges allow database users to connect to the database, for example, and create temporary tables while connected.

To better control user access to the databases instances that you create on your RDS for PostgreSQL DB instance, we recommend that you revoke these default public privileges. After doing so, you then grant specific privileges for database users on a more granular basis, as shown in the following procedure.
To set up roles and privileges for a new database instance

Suppose you're setting up a database on a newly created RDS for PostgreSQL DB instance for use by several researchers, all of whom need read-write access to the database.

1. Use psql (or pgAdmin) to connect to your RDS for PostgreSQL DB instance:

```
psql --host=your-db-instance.666666666666.aws-region.rds.amazonaws.com --port=5432 --
username=postgres --password
```

   When prompted, enter your password. The psql client connects and displays the default administrative connection database, postgres=>, as the prompt.

2. To prevent database users from creating objects in the public schema, do the following:

```
postgres=> REVOKE CREATE ON SCHEMA public FROM PUBLIC;
REVOKE
```

3. Next, you create a new database instance:

```
postgres=> CREATE DATABASE lab_db;
CREATE DATABASE
```

4. Revoke all privileges from the PUBLIC schema on this new database.

```
postgres=> REVOKE ALL ON DATABASE lab_db FROM public;
REVOKE
```

5. Create a role for database users.

```
postgres=> CREATE ROLE lab_tech;
CREATE ROLE
```

6. Give database users that have this role the ability to connect to the database.

```
postgres=> GRANT CONNECT ON DATABASE lab_db TO lab_tech;
GRANT
```

7. Grant all users with the lab_tech role all privileges on this database.

```
postgres=> GRANT ALL PRIVILEGES ON DATABASE lab_db TO lab_tech;
GRANT
```

8. Create database users, as follows:

```
postgres=> CREATE ROLE lab_user1 LOGIN PASSWORD 'change_me';
CREATE ROLE

postgres=> CREATE ROLE lab_user2 LOGIN PASSWORD 'change_me';
CREATE ROLE
```

9. Grant these two users the privileges associated with the lab_tech role:

```
postgres=> GRANT lab_tech TO lab_user1;
GRANT ROLE

postgres=> GRANT lab_tech TO lab_user2;
GRANT ROLE
```

At this point, lab_user1 and lab_user2 can connect to the lab_db database. This example doesn't follow best practices for enterprise usage, which might include creating multiple database instances, different schemas, and granting limited permissions.

The recommended approach for setting up fine-grained access control in PostgreSQL is as follows:

1. Use the master user to create roles per application or use case, like readonly and readwrite.
2. Add permissions to allow these roles to access various database objects. For example, the readonly role can only run SELECT queries.
3. Grant the roles the least possible permissions required for the functionality.
4. Create new users for each application or distinct functionality, like app_user and reporting_user.
5. Assign the applicable roles to these users to quickly grant them the same permissions as the role. For example, grant the readwrite role to app_user and grant the readonly role to reporting_user.
6. At any time, you can remove the role from the user in order to revoke the permissions.

Users, groups, and roles

Users, groups, and roles are the same thing in PostgreSQL, with the only difference being that users have permission to log in by default. The CREATE USER and CREATE GROUP statements are actually aliases for the CREATE ROLE statement.

To create a PostgreSQL user, use the following SQL statement:

```
CREATE USER myuser WITH PASSWORD 'secret_passwd';
```

You can also create a user with the following SQL statement:

```
CREATE ROLE myuser WITH LOGIN PASSWORD 'secret_passwd';
```

Both of these statements create the exact same user. This new user does not have any permissions other than the default permissions available to the public role. All new users and roles inherit permissions from the public role. The following section provides more details about the public role.

Public schema and public role

When a new database is created, PostgreSQL by default creates a schema named public and grants access on this schema to a backend role named public. All new users and roles are by default granted this public role, and therefore can create objects in the public schema.

PostgreSQL uses a concept of a search path. The search path is a list of schema names that PostgreSQL checks when you don't use a qualified name of the database object. For example, when you select from a table named "mytable", PostgreSQL looks for this table in the schemas listed in the search path. It chooses the first match it finds. By default, the search path contains the following schemas:

```
postgres=# show search_path;
  search_path
----------------
 "$user", public
(1 row)
```

The first name "$user" resolves to the name of the currently logged in user. By default, no schema with the same name as the user name exists. So the public schema becomes the default schema whenever an unqualified object name is used. Because of this, when a user tries to create a new table without specifying the schema name, the table gets created in the public schema. As mentioned earlier, by default, all users have access to create objects in the public schema, and therefore the table is created successfully.

This becomes a problem if you are trying to create a read-only user. Even if you restrict all privileges, the permissions inherited via the public role allow the user to create objects in the public schema.

To fix this, you should revoke the default create permission on the public schema from the public role using the following SQL statement:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Make sure that you are the owner of the public schema or are part of a role that allows you to run this SQL statement.

The following statement revokes the public role's ability to connect to the database:

```
REVOKE ALL ON DATABASE mydatabase FROM PUBLIC;
```

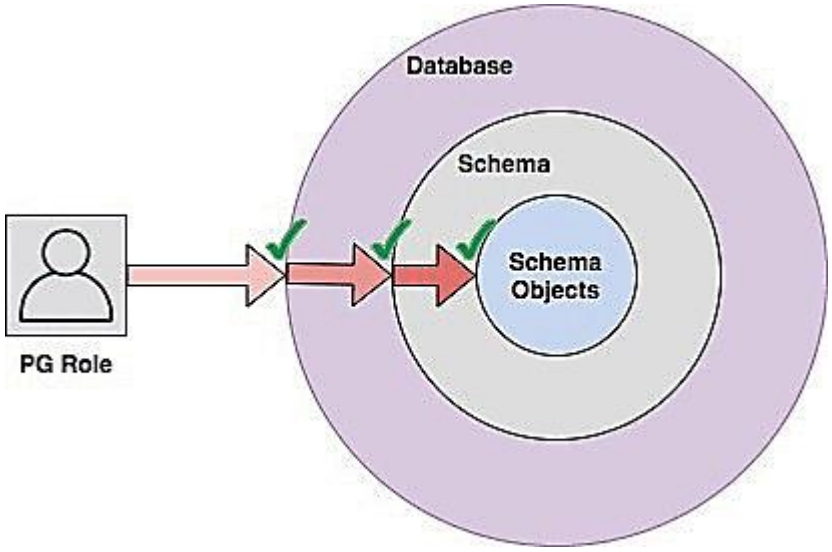
This makes sure that users can't connect to the database by default unless this permission is explicitly granted.

Revoking permissions from the public role impacts all existing users and roles. Any users and roles that should be able to connect to the database or create objects in the public schema should be granted the

permissions explicitly before revoking any permissions from the public role in the production environment.

Creating database roles

The following sections document the process of creating new roles and granting them permissions to access various database objects. Permissions must be granted at the database, schema, and schema object level. For example, if you need to grant access to a table, you must also make sure that the role has access to the database and schema in which the table exists. If any of the permissions are missing, the role cannot access the table.



Read-only role

The first step is to create a new role named readonly using the following SQL statement:

```
CREATE ROLE readonly;
```

This is a base role with no permissions and no password. It cannot be used to log in to the database.

Grant this role permission to connect to your target database named “mydatabase”:

```
GRANT CONNECT ON DATABASE mydatabase TO readonly;
```

The next step is to grant this role usage access to your schema. Let’s assume the schema is named myschema:

```
GRANT USAGE ON SCHEMA myschema TO readonly;
```

This step grants the readonly role permission to perform some activity inside the schema. Without this step, the readonly role cannot perform any action on the objects in this schema, even if the permissions were granted for those objects.

The next step is to grant the readonly role access to run select on the required tables.

```
GRANT SELECT ON TABLE mytable1, mytable2 TO readonly;
```

If the requirement is to grant access on all the tables and views in the schema, then you can use the following SQL:

```
GRANT SELECT ON ALL TABLES IN SCHEMA myschema TO readonly;
```

The preceding SQL statement grants SELECT access to the readonly role on all the existing tables and views in the schema myschema. Note that any new tables that get added in the future will not be accessible by the readonly user. To help ensure that new tables and views are also accessible, run the following statement to grant permissions automatically:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO readonly;
```

Read/write role

The process of adding a read/write role is very similar to the read-only role process covered previously.

The first step is creating a role:

```
CREATE ROLE readwrite;
```

Grant this role permission to connect to your target database:

```
GRANT CONNECT ON DATABASE mydatabase TO readwrite;
```

Grant schema usage privilege:

```
GRANT USAGE ON SCHEMA myschema TO readwrite;
```

If you want to allow this role to create new objects like tables in this schema, then use the following SQL instead of the one preceding:

```
GRANT USAGE, CREATE ON SCHEMA myschema TO readwrite;
```

The next step is to grant access to the tables. As mentioned in the previous section, the grant can be on individual tables or all tables in the schema. For individual tables, use the following SQL:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE mytable1, mytable2 TO readwrite;
```

For all the tables and views in the schema, use the following SQL:

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

To automatically grant permissions on tables and views added in the future:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT, INSERT, UPDATE, DELETE
ON TABLES TO readwrite;
```

For read/write roles, there is normally a requirement to use sequences also. You can give selective access as follows:

```
GRANT USAGE ON SEQUENCE myseq1, myseq2 TO readwrite;
```

You can also grant permission to all sequences using the following SQL statement:

```
GRANT USAGE ON ALL SEQUENCES IN SCHEMA myschema TO readwrite;
```

To automatically grant permissions to sequences added in the future:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT USAGE ON SEQUENCES TO
readwrite;
```

Creating database users

With the roles in place, the process of creating users is simplified. Just create the user and grant it one of the existing roles. Here are the SQL statements for this process:

```
CREATE USER myuser1 WITH PASSWORD 'secret_passwd';
GRANT readonly TO myuser1;
```

This grants myuser1 the same permissions as the readonly role.

Similarly, you can grant read and write access to a user by granting the readwrite role. The PostgreSQL CREATE USER documentation contains more details about the parameters you can set while creating a user. For example, you can specify an expiry time for the user or allow the user to create databases.
Managing user passwords

After creating a user, you must provide these credentials to the application so that it can access the database. It is essential to make sure that these credentials are not hardcoded in the source code or placed in some shared configuration files as clear text.

AWS provides a solution for this with AWS Secrets Manager. Using Secrets Manager, you can store the credentials and then use AWS Identity and Access Management (IAM) to allow only certain IAM users and roles to read the credentials. For the steps involved in this, see Creating and Managing Secrets with AWS Secrets Manager in the AWS Secrets Manager User Guide.

In addition to storing the credentials, a very useful feature that Secrets Manager provides is database user password rotation. You can use this feature to set up a policy to automatically change the password at a certain frequency.

Revoking or changing user permissions

Using the method documented previously, it becomes very easy to revoke privileges from a user. For example, you can remove the readwrite permission from myuser1 using the following SQL statement:

```
REVOKE readwrite FROM myuser1;
```

Similarly, you can grant a new role as follows:

```
GRANT readonly TO myuser1;
```

Monitoring usage

You can monitor user activity by setting PostgreSQL logging parameters available in the RDS parameter groups. For example, you can set the log_connections and log_disconnections parameters to capture all new connections and disconnections. After setting these parameters in the parameter group, you will see the following messages in the log files:

```
2018-11-09 21:08:39 UTC:XX-XX-XX-XX.amazon.com(27585):myuser@mydb:[18014]:LOG:
connection authorized: user=myuser database=mydb SSL enabled (protocol=TLSv1.2,
cipher=ECDHE-RSA-AES256-GCM-SHA384, compression=off)

2018-11-09 21:09:19 UTC:XX-XX-XX-XX.amazon.com(27585):myuser@mydb:[18014]:LOG:
disconnection: session time: 0:00:39.649 user=myuser database=mydb host=XX-XX-XX-
XX.amazon.com port=27585
```

If you require more detailed session or object-level custom auditing information, then you can use the pgAudit extension. The steps to configure pgAudit with Amazon RDS and Aurora PostgreSQL are available in Working with the pgaudit Extension in the Amazon RDS User Guide.

Increasing database logging does impact storage size, I/O use, and CPU use. Because of this, it is important that you test these changes before deploying them in production.

Checking the granted roles

You can use the following query to get a list of all the database users and roles along with a list of roles that have been granted to them:

```
SELECT
    r.rolname,
    ARRAY(SELECT b.rolname
        FROM pg_catalog.pg_auth_members m
        JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid)
        WHERE m.member = r.oid) as memberof
FROM pg_catalog.pg_roles r
WHERE r.rolname NOT IN ('pg_signal_backend','rds_iam',
                'rds_replication','rds_superuser',
                'rdsadmin','rdsrepladmin')
ORDER BY 1;
```

Here is sample output from a test RDS instance:

```
  rolname    |   memberof
-------------+-----------------
 app_user     | {readwrite}
 postgres     | {rds_superuser}
 readonly     | {}
 readwrite    | {}
 reporting_user | {readonly}
```

Note that a user can be member of multiple roles with distinct or overlapping permissions. In this case, the user gets a sum of all the permissions.

You can also use the catalog table pg_roles to check attributes like password expiry date or number of parallel connections allowed.

Delegating and controlling user password management

As a DBA, you might want to delegate the management of user passwords. Or, you might want to prevent database users from changing their passwords or reconfiguring password constraints, such as password lifetime. To ensure that only the database users that you choose can change password settings, you can turn on the restricted password management feature. When you activate this feature, only those database users that have been granted the rds_password role can manage passwords.
Note

To use restricted password management, your RDS for PostgreSQL DB instance must be running PostgreSQL 10.6 or higher.

By default, this feature is off, as shown in the following:

```
postgres=> SHOW rds.restrict_password_commands;
 rds.restrict_password_commands
--------------------------------
 off
(1 row)
```

To turn on this feature, you use a custom parameter group and change the setting for rds.restrict_password_commands to 1.

Be sure to reboot your RDS for PostgreSQL DB instance so that the setting takes effect.

With this feature active, rds_password privileges are needed for the following SQL commands:

```
CREATE ROLE myrole WITH PASSWORD 'mypassword';
CREATE ROLE myrole WITH PASSWORD 'mypassword' VALID UNTIL '2023-01-01';
ALTER ROLE myrole WITH PASSWORD 'mypassword' VALID UNTIL '2023-01-01';
ALTER ROLE myrole WITH PASSWORD 'mypassword';
ALTER ROLE myrole VALID UNTIL '2023-01-01';
ALTER ROLE myrole RENAME TO myrole2;
```

Renaming a role (ALTER ROLE myrole RENAME TO newname) is also restricted if the password uses the MD5 hashing algorithm.

With this feature active, attempting any of these SQL commands without the rds_password role permissions generates the following error:

```
ERROR: must be a member of rds_password to alter passwords
```

We recommend that you grant the rds_password to only a few roles that you use solely for password management. If you grant rds_password privileges to database users that don't have rds_superuser privileges, you need to also grant them the CREATEROLE attribute.

Make sure that you verify password requirements such as expiration and needed complexity on the client side. If you use your own client-side utility for password related changes, the utility needs to be a member of rds_password and have CREATE ROLE privileges.

Working with the PostgreSQL autovacuum on Amazon RDS for PostgreSQL

We strongly recommend that you use the autovacuum feature to maintain the health of your PostgreSQL DB instance. Autovacuum automates the start of the VACUUM and the ANALYZE commands. It checks for tables with a large number of inserted, updated, or deleted tuples. After this check, it reclaims storage by removing obsolete data or tuples from the PostgreSQL database.

By default, autovacuum is turned on for the Amazon RDS for PostgreSQL DB instances that you create using any of the default PostgreSQL DB parameter groups. These include default.postgres10, default.postgres11, and so on. All default PostgreSQL DB parameter groups have an rds.adaptive_autovacuum parameter that's set to 1, thus activating the feature. Other configuration parameters associated with the autovacuum feature are also set by default. Because these defaults are somewhat generic, you can benefit from tuning some of the parameters associated with the autovacuum feature for your specific workload.

Allocating memory for autovacuum

One of the most important parameters influencing autovacuum performance is the maintenance_work_mem

parameter. This parameter determines how much memory that you allocate for autovacuum to use to scan a database table and to hold all the row IDs that are going to be vacuumed. If you set the value of the maintenance_work_mem parameter too low, the vacuum process might have to scan the table multiple times to complete its work. Such multiple scans can have a negative impact on performance.

When doing calculations to determine the maintenance_work_mem parameter value, keep in mind two things:

- The default unit is kilobytes (KB) for this parameter.
- The maintenance_work_mem parameter works in conjunction with the autovacuum_max_workers parameter. If you have many small tables, allocate more autovacuum_max_workers and less maintenance_work_mem. If you have large tables (say, larger than 100 GB), allocate more memory and fewer worker processes. You need to have enough memory allocated to succeed on your biggest table. Each autovacuum_max_workers can use the memory that you allocate. Thus, make sure that the combination of worker processes and memory equal the total memory that you want to allocate.

- In general terms, for large hosts set the maintenance_work_mem parameter to a value between one and two gigabytes (between 1,048,576 and 2,097,152 KB). For extremely large hosts, set the parameter to a value between two and four gigabytes (between 2,097,152 and 4,194,304 KB). The value that you set for this parameter depends on the workload. Amazon RDS has updated its default for this parameter to be kilobytes calculated as follows.

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

Reducing the likelihood of transaction ID wraparound

In some cases, parameter group settings related to autovacuum might not be aggressive enough to prevent transaction ID wraparound. To address this, RDS for PostgreSQL provides a mechanism that adapts the autovacuum parameter values automatically. Adaptive autovacuum parameter tuning is a feature for RDS for PostgreSQL.

Adaptive autovacuum parameter tuning is turned on by default for RDS for PostgreSQL instances with the dynamic parameter rds.adaptive_autovacuum set to ON. We strongly recommend that you keep this turned on. However, to turn off adaptive autovacuum parameter tuning, set the rds.adaptive_autovacuum parameter to 0 or OFF.

Determining if the tables in your database need vacuuming

You can use the following query to show the number of unvacuumed transactions in a database. The datfrozenxid column of a database's pg_database row is a lower bound on the normal transaction IDs appearing in that database. This column is the minimum of the per-table relfrozenxid values within the database.

```
SELECT datname, age(datfrozenxid) FROM pg_database ORDER BY age(datfrozenxid) desc limit 20;
```

For example, the results of running the preceding query might be the following.

```
datname    | age
mydb       | 1771757888
template0  | 1721757888
template1  | 1721757888
rdsadmin   | 1694008527
postgres   | 1693881061
(5 rows)
```

When the age of a database reaches 2 billion transaction IDs, transaction ID (XID) wraparound occurs and the database becomes read-only. You can use this query to produce a metric and run a few times a day. By default, autovacuum is set to keep the age of transactions to no more than 200,000,000 (autovacuum_freeze_max_age).

A sample monitoring strategy might look like this:

- Set the autovacuum_freeze_max_age value to 200 million transactions.
- If a table reaches 500 million unvacuumed transactions, that triggers a low-severity alarm. This isn't an unreasonable value, but it can indicate that autovacuum isn't keeping up.

- If a table ages to 1 billion, this should be treated as an alarm to take action on. In general, you want to keep ages closer to autovacuum_freeze_max_age for performance reasons. We recommend that you investigate using the recommendations that follow.
- If a table reaches 1.5 billion unvacuumed transactions, that triggers a high-severity alarm. Depending on how quickly your database uses transaction IDs, this alarm can indicate that the system is running out of time to run autovacuum. In this case, we recommend that you resolve this immediately.

If a table is constantly breaching these thresholds, modify your autovacuum parameters further. By default, using VACUUM manually (which has cost-based delays disabled) is more aggressive than using the default autovacuum, but it is also more intrusive to the system as a whole.

We recommend the following:

- Be aware and turn on a monitoring mechanism so that you are aware of the age of your oldest transactions.
- For busier tables, perform a manual vacuum freeze regularly during a maintenance window, in addition to relying on autovacuum. For information on performing a manual vacuum freeze, see Performing a manual vacuum freeze.

Determining which tables are currently eligible for autovacuum

Often, it is one or two tables in need of vacuuming. Tables whose relfrozenxid value is greater than the number of transactions in autovacuum_freeze_max_age are always targeted by autovacuum. Otherwise, if the number of tuples made obsolete since the last VACUUM exceeds the vacuum threshold, the table is vacuumed.

The autovacuum threshold is defined as:

```
Vacuum-threshold = vacuum-base-threshold + vacuum-scale-factor * number-of-tuples
```

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is pg_class.reltuples.

While you are connected to your database, run the following query to see a list of tables that autovacuum sees as eligible for vacuuming.

```
WITH vbt AS (SELECT setting AS autovacuum_vacuum_threshold FROM
pg_settings WHERE name = 'autovacuum_vacuum_threshold'),
vsf AS (SELECT setting AS autovacuum_vacuum_scale_factor FROM
pg_settings WHERE name = 'autovacuum_vacuum_scale_factor'),
fma AS (SELECT setting AS autovacuum_freeze_max_age FROM pg_settings WHERE name =
'autovacuum_freeze_max_age'),
sto AS (select opt_oid, split_part(setting, '=', 1) as param,
split_part(setting, '=', 2) as value from (select oid opt_oid, unnest(reloptions) setting from pg_class) opt)
SELECT '"'||ns.nspname||'"."'||c.relname||'"' as relation,
pg_size_pretty(pg_table_size(c.oid)) as table_size,
age(relfrozenxid) as xid_age,
```

```
coalesce(cfma.value::float, autovacuum_freeze_max_age::float) autovacuum_freeze_max_age,
(coalesce(cvbt.value::float, autovacuum_vacuum_threshold::float) +
coalesce(cvsf.value::float,autovacuum_vacuum_scale_factor::float) * c.reltuples)
AS autovacuum_vacuum_tuples, n_dead_tup as dead_tuples FROM
pg_class c join pg_namespace ns on ns.oid = c.relnamespace
join pg_stat_all_tables stat on stat.relid = c.oid join vbt on (1=1) join vsf on (1=1) join fma on (1=1)
left join sto cvbt on cvbt.param = 'autovacuum_vacuum_threshold' and c.oid = cvbt.opt_oid
left join sto cvsf on cvsf.param = 'autovacuum_vacuum_scale_factor' and c.oid = cvsf.opt_oid
left join sto cfma on cfma.param = 'autovacuum_freeze_max_age' and c.oid = cfma.opt_oid
WHERE c.relkind = 'r' and nspname <> 'pg_catalog'
AND (age(relfrozenxid) >= coalesce(cfma.value::float, autovacuum_freeze_max_age::float)
OR coalesce(cvbt.value::float, autovacuum_vacuum_threshold::float) +
coalesce(cvsf.value::float,autovacuum_vacuum_scale_factor::float) *
c.reltuples <= n_dead_tup)
ORDER BY age(relfrozenxid) DESC LIMIT 50;
```

Determining if autovacuum is currently running and for how long

If you need to manually vacuum a table, make sure to determine if autovacuum is currently running. If it is, you might need to adjust parameters to make it run more efficiently, or turn off autovacuum temporarily so that you can manually run VACUUM.

Use the following query to determine if autovacuum is running, how long it has been running, and if it is waiting on another session.

```
SELECT datname, usename, pid, state, wait_event, current_timestamp - xact_start AS xact_runtime,
query
FROM pg_stat_activity
WHERE upper(query) LIKE '%VACUUM%'
ORDER BY xact_start;
```

After running the query, you should see output similar to the following.

```
 datname | usename  |  pid  | state  | wait_event |     xact_runtime      | query
 --------+----------+-------+--------+------------+-----------------------+-----------------------------------------------------
-----------------------------------------------------
 mydb    | rdsadmin | 16473 | active |            | 33 days 16:32:11.600656 | autovacuum: VACUUM ANALYZE
public.mytable1 (to prevent wraparound)
 mydb    | rdsadmin | 22553 | active |            | 14 days 09:15:34.073141 | autovacuum: VACUUM ANALYZE
public.mytable2 (to prevent wraparound)
 mydb    | rdsadmin | 41909 | active |            | 3 days 02:43:54.203349  | autovacuum: VACUUM ANALYZE
public.mytable3
 mydb    | rdsadmin |   618 | active |            | 00:00:00                | SELECT datname, usename, pid, state,
wait_event, current_timestamp - xact_start AS xact_runtime, query+
         |          |       |        |            |                         | FROM pg_stat_activity                                +
         |          |       |        |            |                         | WHERE query like '%VACUUM%'
+
         |          |       |        |            |                         | ORDER BY xact_start;                                 +
```

Several issues can cause a long-running autovacuum session (that is, multiple days long). The most common issue is that your maintenance_work_mem parameter value is set too low for the size of the table or rate of updates.

We recommend that you use the following formula to set the maintenance_work_mem parameter value.

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

Short running autovacuum sessions can also indicate problems:

- It can indicate that there aren't enough autovacuum_max_workers for your workload. In this case, you need to indicate the number of workers.
- It can indicate that there is an index corruption (autovacuum crashes and restarts on the same relation but makes no progress). In this case, run a manual vacuum freeze verbose table to see the exact cause.

Performing a manual vacuum freeze

You might want to perform a manual vacuum on a table that has a vacuum process already running. This is useful if you have identified a table with an age approaching 2 billion transactions (or above any threshold you are monitoring).

The following steps are guidelines, with several variations to the process. For example, during testing, suppose that you find that the maintenance_work_mem parameter value is set too small and that you need to take immediate action on a table. However, perhaps you don't want to bounce the instance at the moment.

Using the queries in previous sections, you determine which table is the problem and notice a long running autovacuum session. You know that you need to change the maintenance_work_mem parameter setting, but you also need to take immediate action and vacuum the table in question. The following procedure shows what to do in this situation.

To manually perform a vacuum freeze

1. Open two sessions to the database containing the table you want to vacuum. For the second session, use "screen" or another utility that maintains the session if your connection is dropped.
2. In session one, get the process ID (PID) of the autovacuum session running on the table.

   Run the following query to get the PID of the autovacuum session.

   ```
   SELECT datname, usename, pid, current_timestamp - xact_start
   AS xact_runtime, query
   FROM pg_stat_activity WHERE upper(query) LIKE '%VACUUM%' ORDER BY
   xact_start;
   ```

3. In session two, calculate the amount of memory that you need for this operation. In this example, we determine that we can afford to use up to 2 GB of memory for this operation, so we set maintenance_work_mem for the current session to 2 GB.

```
SET maintenance_work_mem='2 GB';
```

4. In session two, issue a vacuum freeze verbose command for the table. The verbose setting is useful because, although there is no progress report for this in PostgreSQL currently, you can see activity.

```
\timing on
```

```
INFO:  vacuuming "public.pgbench_branches"
INFO:  index "pgbench_branches_pkey" now contains 50 row versions in 2 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  index "pgbench_branches_test_index" now contains 50 row versions in 2 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  "pgbench_branches": found 0 removable, 50 nonremovable row versions
   in 43 out of 43 pages
DETAIL:  0 dead row versions cannot be removed yet.
There were 9347 unused item pointers.
0 pages are entirely empty.
CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
Time: 2.765 ms
```

5. In session one, if autovacuum was blocking the vacuum session, you see in pg_stat_activity that waiting is "T" for your vacuum session. In this case, you need to end the autovacuum process as follows.

```
SELECT pg_terminate_backend('the_pid');
```

   At this point, your session begins. It's important to note that autovacuum restarts immediately because this table is probably the highest on its list of work.

6. Initiate your vacuum freeze verbose command in session two, and then end the autovacuum process in session one.

Reindexing a table when autovacuum is running

If an index has become corrupt, autovacuum continues to process the table and fails. If you attempt a manual vacuum in this situation, you receive an error message like the following.

```
postgres=>  vacuum freeze pgbench_branches;
ERROR: index "pgbench_branches_test_index" contains unexpected
  zero page at block 30521
HINT: Please REINDEX it.
```

When the index is corrupted and autovacuum is attempting to run on the table, you contend with an already running autovacuum session. When you issue a REINDEX command, you take out an exclusive lock on the table. Write operations are blocked, and also read operations that use that specific index.

To reindex a table when autovacuum is running on the table

1. Open two sessions to the database containing the table that you want to vacuum. For the second session, use "screen" or another utility that maintains the session if your connection is dropped.
2. In session one, get the PID of the autovacuum session running on the table.
3. Run the following query to get the PID of the autovacuum session.

```
SELECT datname, usename, pid, current_timestamp - xact_start
AS xact_runtime, query
FROM pg_stat_activity WHERE upper(query) like '%VACUUM%' ORDER BY
xact_start;
```

4. In session two, issue the reindex command.

```
\timing on
```

Timing is on.

```
reindex index pgbench_branches_test_index;
```

REINDEX
Time: 9.966 ms

5. In session one, if autovacuum was blocking the process, you see in pg_stat_activity that waiting is "T" for your vacuum session. In this case, you end the autovacuum process.

```
SELECT pg_terminate_backend('the_pid');
```

At this point, your session begins. It's important to note that autovacuum restarts immediately because this table is probably the highest on its list of work.

6. Initiate your command in session two, and then end the autovacuum process in session 1.

Managing autovacuum with large indexes

As part of its operation, autovacuum performs several vacuum phases while running on a table. Before the table is cleaned up, all of its indexes are first vacuumed. When removing multiple large indexes, this phase consumes a significant amount of time and resources. Therefore, as a best practice, be sure to control the number of indexes on a table and eliminate unused indexes.

For this process, first check the overall index size. Then, determine if there are potentially unused indexes that can be removed as shown in the following examples.

To check the size of the table and its indexes

```
postgres=> select pg_size_pretty(pg_relation_size('pgbench_accounts'));
pg_size_pretty
6404 MB
```

```
(1 row)

postgres=> select pg_size_pretty(pg_indexes_size('pgbench_accounts'));
pg_size_pretty
11 GB
(1 row)
```

In this example, the size of indexes is larger than the table. This difference can cause performance issues as the indexes are bloated or unused, which impacts the autovacuum as well as insert operations.

To check for unused indexes

Using the pg_stat_user_indexes view, you can check how frequently an index is used with the idx_scan column. In the following example, the unused indexes have the idx_scan value of 0.

```
postgres=> select * from pg_stat_user_indexes where relname = 'pgbench_accounts' order by idx_scan desc;
```

```
relid  | indexrelid | schemaname | relname         | indexrelname         | idx_scan | idx_tup_read | idx_tup_fetch
-------+------------+------------+-----------------+----------------------+----------+--------------+---------------
16433  | 16454      | public     | pgbench_accounts | index_f             | 6        | 6            | 0
16433  | 16450      | public     | pgbench_accounts | index_b             | 3        | 199999       | 0
16433  | 16447      | public     | pgbench_accounts | pgbench_accounts_pkey | 0      | 0            | 0
16433  | 16452      | public     | pgbench_accounts | index_d             | 0        | 0            | 0
16433  | 16453      | public     | pgbench_accounts | index_e             | 0        | 0            | 0
16433  | 16451      | public     | pgbench_accounts | index_c             | 0        | 0            | 0
16433  | 16449      | public     | pgbench_accounts | index_a             | 0        | 0            | 0
(7 rows)
```

```
postgres=> select schemaname, relname, indexrelname, idx_scan from pg_stat_user_indexes where relname = 'pgbench_accounts' order by idx_scan desc;
```

```
schemaname  | relname         | indexrelname         | idx_scan
------------+-----------------+----------------------+----------
public      | pgbench_accounts | index_f             | 6
public      | pgbench_accounts | index_b             | 3
public      | pgbench_accounts | pgbench_accounts_pkey | 0
public      | pgbench_accounts | index_d             | 0
public      | pgbench_accounts | index_e             | 0
public      | pgbench_accounts | index_c             | 0
public      | pgbench_accounts | index_a             | 0
(7 rows)
```

Note: These statistics are incremental from the time that the statistics are reset. Suppose you have an index that is only used at the end of a business quarter or just for a specific report. It's possible that this index hasn't been used since the statistics were reset.

Indexes that are used to enforce uniqueness won't have scans performed and shouldn't be identified as unused indexes. To identify the unused indexes, you should have in-depth knowledge of the application and its queries.

To check when the stats were last reset for a database, use pg_stat_database

```
postgres=> select datname, stats_reset from pg_stat_database where datname = 'postgres';
```

```
datname  | stats_reset
----------+-------------------------------
postgres | 2022-11-17 08:58:11.427224+00
(1 row)
```

Vacuuming a table as quickly as possible

RDS for PostgreSQL 12 and higher

If you have too many indexes in a large table, your DB instance could be nearing transaction ID wraparound (XID), which is when the XID counter wraps around to zero. Left unchecked, this situation could result in data loss. However, you can quickly vacuum the table without cleaning up the indexes. In RDS for PostgreSQL 12 and higher, you can use VACUUM with the INDEX_CLEANUP clause.

```
postgres=> VACUUM (INDEX_CLEANUP FALSE, VERBOSE TRUE) pgbench_accounts;
```

```
INFO: vacuuming "public.pgbench_accounts"
INFO: table "pgbench_accounts": found 0 removable, 8 nonremovable row versions in 1 out of 819673 pages
DETAIL: 0 dead row versions cannot be removed yet, oldest xmin: 7517
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.01 s.
```

If an autovacuum session is already running, you must terminate it to begin the manual VACUUM. For information on performing a manual vacuum freeze, see Performing a manual vacuum freeze Note

Skipping index cleanup regularly might cause index bloat, which impacts the overall scan performance. As a best practice, use the preceding procedure only to prevent transaction ID wraparound.

Setting table-level autovacuum parameters

You can set autovacuum-related storage parameters at a table level, which can be better than altering the behavior of the entire database. For large tables, you might need to set aggressive settings and you might not want to make autovacuum behave that way for all tables.

The following query shows which tables currently have table-level options in place.

```
SELECT relname, reloptions
FROM pg_class
WHERE reloptions IS NOT null;
```

An example where this might be useful is on tables that are much larger than the rest of your tables. Suppose that you have one 300-GB table and 30 other tables less than 1 GB. In this case, you might set some specific parameters for your large table so you don't alter the behavior of your entire system.

```
ALTER TABLE mytable set (autovacuum_vacuum_cost_delay=0);
```

Doing this turns off the cost-based autovacuum delay for this table at the expense of more resource usage on your system. Normally, autovacuum pauses for autovacuum_vacuum_cost_delay each time autovacuum_cost_limit is reached. For more details, see the PostgreSQL documentation about cost-based vacuuming

Logging autovacuum and vacuum activities

Information about autovacuum activities is sent to the postgresql.log based on the level specified in the rds.force_autovacuum_logging_level parameter. Following are the values allowed for this parameter and the PostgreSQL versions for which that value is the default setting:

- disabled (PostgreSQL 10, PostgreSQL 9.6)
- debug5, debug4, debug3, debug2, debug1
- info (PostgreSQL 12, PostgreSQL 11)
- notice
- warning (PostgreSQL 13 and above)
- error, log, fatal, panic

The rds.force_autovacuum_logging_level works with the log_autovacuum_min_duration parameter. The log_autovacuum_min_duration parameter's value is the threshold (in milliseconds) above which autovacuum actions get logged. A setting of -1 logs nothing, while a setting of 0 logs all actions. As with rds.force_autovacuum_logging_level, default values for log_autovacuum_min_duration are version dependent, as follows:

- 10000 ms – PostgreSQL 14, PostgreSQL 13, PostgreSQL 12, and PostgreSQL 11
- (empty) – No default value for PostgreSQL 10 and PostgreSQL 9.6

We recommend that you set rds.force_autovacuum_logging_level to WARNING. We also recommend that you set log_autovacuum_min_duration to a value from 1000 to 5000. A setting of 5000 logs activity that takes longer than 5,000 milliseconds. Any setting other than -1 also logs messages if the autovacuum action is skipped because of a conflicting lock or concurrently dropped relations.

To troubleshoot issues, you can change the rds.force_autovacuum_logging_level parameter to one of the debug levels, from debug1 up to debug5 for the most verbose information. We recommend that you use debug settings for short periods of time and for troubleshooting purposes only.

Note: PostgreSQL allows the rds_superuser account to view autovacuum sessions in pg_stat_activity. For example, you can identify and end an autovacuum session that is blocking a command from running, or running slower than a manually issued vacuum command.

Improving query performance for RDS for PostgreSQL with Amazon RDS Optimized Reads

You can achieve faster query processing for RDS for PostgreSQL with Amazon RDS Optimized Reads. An RDS for PostgreSQL DB instance or Multi-AZ DB cluster that uses RDS Optimized Reads can achieve up to 50% faster query processing compared to one that doesn't use it.

Optimized Reads is available by default on RDS for PostgreSQL versions 15.2 and higher, 14.7 and higher, and 13.10 and higher.

When you use an RDS for PostgreSQL DB instance or Multi-AZ DB cluster that has RDS Optimized Reads turned on, it achieves up to 50% faster query performance using the local Non-Volatile Memory Express (NVMe) based solid state drive (SSD) block-level storage. You can achieve faster query processing by placing the temporary tables that are generated by PostgreSQL on the local storage, which reduces the traffic to Elastic Block Storage (EBS) over the network.

In PostgreSQL, temporary objects are assigned to a temporary namespace that drops automatically at the end of the session. The temporary namespace while dropping removes any objects that are session-dependent, including schema-qualified objects, such as tables, functions, operators, or even extensions.

In RDS for PostgreSQL, the temp_tablespaces parameter is configured for this temporary work area where the temporary objects are stored.

The following queries return the name of the tablespace and its location.

```
postgres=> show temp_tablespaces;
temp_tablespaces
---------------------
rds_temp_tablespace
(1 row)
```

The rds_temp_tablespace is a tablespace configured by RDS that points to the NVMe local storage. You can always switch back to Amazon EBS storage by modifying this parameter in the Parameter group using the AWS Management Console to point to any tablespace other than rds_temp_tablespace. For more information, see Modifying parameters in a DB parameter group. You can also use the SET command to modify the value of the temp_tablespaces parameter to pg_default at the session level using SET command. Modifying the parameter redirects the temporary work area to Amazon EBS. Switching back to

Amazon EBS helps when the local storage for your RDS instance or cluster isn't sufficient to perform a specific SQL operation.

```
postgres=> SET temp_tablespaces TO 'pg_default';
SET

postgres=> show temp_tablespaces;

 temp_tablespaces
------------------
 pg_default
```

Using RDS Optimized Reads

When you provision an RDS for PostgreSQL DB instance with one of the NVMe based DB instance classes in a Single-AZ DB instance deployment, Multi-AZ DB instance deployment, or Multi-AZ DB cluster deployment, the DB instance automatically uses RDS Optimized Reads.

To turn on RDS Optimized Reads, do one of the following:

- Create an RDS for PostgreSQL DB instance or Multi-AZ DB cluster using one of the NVMe based DB instance classes. For more information, see Creating an Amazon RDS DB instance.
- Modify an existing RDS for PostgreSQL DB instance or Multi-AZ DB cluster to use one of the NVMe based DB instance classes. For more information, see Modifying an Amazon RDS DB instance.

RDS Optimized Reads is available in all AWS Regions where one or more of the DB instance classes with local NVMe SSD storage are supported. For more information, see DB instance classes.

To switch back to a non-optimized reads RDS instance, modify the DB instance class of your RDS instance or cluster to the similar instance class that only supports EBS storage for your database workloads. For example, if the current DB instance class is db.r6gd.4xlarge, choose db.r6g.4xlarge to switch back. For more information, see Modifying an Amazon RDS DB instance.

Monitoring DB instances that use RDS Optimized Reads

You can monitor DB instances that use RDS Optimized Reads using the following CloudWatch metrics:

- FreeLocalStorage
- ReadIOPSLocalStorage
- ReadLatencyLocalStorage
- ReadThroughputLocalStorage
- WriteIOPSLocalStorage
- WriteLatencyLocalStorage
- WriteThroughputLocalStorage

These metrics provide data about available instance store storage, IOPS, and throughput. For more information about these metrics, see Amazon CloudWatch instance-level metrics for Amazon RDS.

To monitor current usage of your local storage, log in to your database using the following query:

```
SELECT
  spcname AS "Name",
  pg_catalog.pg_size_pretty(pg_catalog.pg_tablespace_size(oid)) AS "size"
FROM
  pg_catalog.pg_tablespace
WHERE
  spcname IN ('rds_temp_tablespace');
```

Tuning with wait events for RDS for PostgreSQL

Wait events are an important tuning tool for RDS for PostgreSQL. When you can find out why sessions are waiting for resources and what they are doing, you're better able to reduce bottlenecks. You can use the information in this section to find possible causes and corrective actions. This section also discusses basic PostgreSQL tuning concepts.

A wait event is an indication that the session is waiting for a resource. For example, the wait event Client:ClientRead occurs when RDS for PostgreSQL is waiting to receive data from the client. Sessions typically wait for resources such as the following.

- Single-threaded access to a buffer, for example, when a session is attempting to modify a buffer
- A row that is currently locked by another session
- A data file read
- A log file write

For example, to satisfy a query, the session might perform a full table scan. If the data isn't already in memory, the session waits for the disk I/O to complete. When the buffers are read into memory, the session might need to wait because other sessions are accessing the same buffers. The database records the waits by using a predefined wait event. These events are grouped into categories.

By itself, a single wait event doesn't indicate a performance problem. For example, if requested data isn't in memory, reading data from disk is necessary. If one session locks a row for an update, another session waits for the row to be unlocked so that it can update it. A commit requires waiting for the write to a log file to complete. Waits are integral to the normal functioning of a database.

On the other hand, large numbers of wait events typically show a performance problem. In such cases, you can use wait event data to determine where sessions are spending time. For example, if a report that typically runs in minutes now takes hours to run, you can identify the wait events that contribute the most to total wait time. If you can determine the causes of the top wait events, you can sometimes make changes that improve performance. For example, if your session is waiting on a row that has been locked by another session, you can end the locking session.

The following lists the wait events for RDS for PostgreSQL that most commonly indicate performance problems, and summarizes the most common causes and corrective actions..

Client:ClientRead
This event occurs when RDS for PostgreSQL is waiting to receive data from the client.

Client:ClientWrite
This event occurs when RDS for PostgreSQL is waiting to write data to the client.

CPU
This event occurs when a thread is active in CPU or is waiting for CPU.

IO:BufFileRead and IO:BufFileWrite
These events occur when RDS for PostgreSQL creates temporary files.

IO:DataFileRead
This event occurs when a connection waits on a backend process to read a required page from storage because the page isn't available in shared memory.

IO:WALWrite
This event occurs when RDS for PostgreSQL is waiting for the write-ahead log (WAL) buffers to be written to a WAL file.

Lock:advisory
This event occurs when a PostgreSQL application uses a lock to coordinate activity across multiple sessions.

Lock:extend
This event occurs when a backend process is waiting to lock a relation to extend it while another process has a lock on that relation for the same purpose.

Lock:Relation
This event occurs when a query is waiting to acquire a lock on a table or view that's currently locked by another transaction.

Lock:transactionid
This event occurs when a transaction is waiting for a row-level lock.

Lock:tuple
This event occurs when a backend process is waiting to acquire a lock on a tuple.

LWLock:BufferMapping (LWLock:buffer_mapping)
This event occurs when a session is waiting to associate a data block with a buffer in the shared buffer pool.

LWLock:BufferIO (IPC:BufferIO)
This event occurs when RDS for PostgreSQL is waiting for other processes to finish their input/output (I/O) operations when concurrently trying to access a page.

LWLock:buffer_content (BufferContent)
This event occurs when a session is waiting to read or write a data page in memory while another session has that page locked for writing.

LWLock:lock_manager (LWLock:lockmanager)
This event occurs when the RDS for PostgreSQL engine maintains the shared lock's memory area to allocate, check, and deallocate a lock when a fast path lock isn't possible.

Timeout:PgSleep
This event occurs when a server process has called the pg_sleep function and is waiting for the sleep timeout to expire.

Timeout:VacuumDelay
This event indicates that the vacuum process is sleeping because the estimated cost limit has been reached.