RDS for PostgreSQL

Best Practices

Issue	01
Date	2025-09-04

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Best Practices

This chapter describes best practices for working with RDS for PostgreSQL and provides operational guidelines that you can follow when using this service.

Table 1-1 RDS for PostgreSQL best practices

Reference	Description
Creating a Cross-Region DR Relationship for an RDS for PostgreSQL Instance	Describes how to create a cross-region DR relationship for an RDS for PostgreSQL instance.
RDS for PostgreSQL Publications and Subscriptions	Describes publications and subscriptions of RDS for PostgreSQL.
User-Defined Data Type Conversion	Describes how to create user-defined data type conversion functions in RDS for PostgreSQL.
Using Client Drivers to Implement Failover and Read/Write Splitting	Describes how to use client drivers to enable failover and read/write splitting.
Other Extension Plug-Ins	Describes how to use the open-source pg_waldump to parse WAL logs of RDS for PostgreSQL.
Best Practices for Using PoWA	Describes how to use PoWA to monitor the performance of RDS for PostgreSQL instances.
Best Practices for Using pg_dump	Describes how to use pg_dump to back up data.
Best Practices for Using PgBouncer	Describes how to install, configure, and use PgBouncer.
Database Naming Rules	Describes how to create read-only users for RDS for PostgreSQL instances.
RDS for PostgreSQL Table Design	Describes how to design RDS for PostgreSQL table structures to match your workloads.

Reference	Description
RDS for PostgreSQL Permissions Management	Describes how to manage user privileges for RDS for PostgreSQL instances.
Troubleshooting WAL Accumulation	Describes how to troubleshoot WAL accumulation of RDS for PostgreSQL instances.
Updating, Deleting, or Inserting Data Records at a Time	Describes how to efficiently insert, update, and delete data records at a time for RDS for PostgreSQL instances.
Using Event Triggers to Implement the DDL Recycle Bin, Firewalls, and Incremental Synchronization	Describes how to use PostgreSQL event triggers to implement the DDL recycle bin, firewalls, and incremental synchronization.
Creating Replication Slots to Enable CDC	Describes how to enable CDC for an RDS for PostgreSQL instance.
Read/Write Splitting with Pgpool	Describes how to use pgpool to implement read/write splitting for RDS for PostgreSQL DB instances and read replicas.
User Preference Recommendation Systems	Describes how to design a user recommendation system with RDS for PostgreSQL.
Suggestions on RDS for PostgreSQL Metric Alarm Configurations	Describes how to configure RDS for PostgreSQL metric alarm rules.
Security Best Practices	Provides guidance on RDS for PostgreSQL security configurations.

2 Creating a Cross-Region DR Relationship for an RDS for PostgreSQL Instance

2.1 Overview

Scenarios

You can create a cross-region DR relationship for your DB instance. If the production instance fails or the service system breaks down due to other force majeure factors, the DR instance in another region ensures that the production data is not lost and the production system continues to run without interruption. This enhances system availability.

This practice includes the following tasks:

- Create an RDS for PostgreSQL instance.
- Create a cross-region DR relationship for the RDS for PostgreSQL instance.

Prerequisites

- You have registered with Huawei Cloud and completed real-name authentication.
- Your account balance is greater than or equal to $0 USD.

Constraints

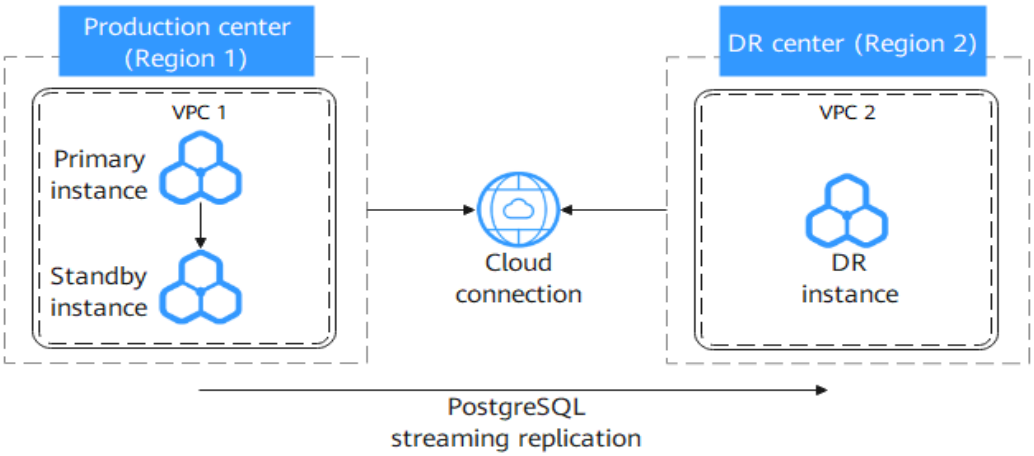
- The primary DB instance and DR instance are running properly and are deployed in different clouds or regions. The primary DB instance is deployed in primary/standby mode and the DR instance is deployed in standalone mode.
- Before configuring disaster recovery for the DR instance, you must configure it for the primary instance. Otherwise, the DR relationship cannot be established.
- The specifications of the DR instance are at least equal to those of the primary DB instance.

- Cross-cloud or cross-region DR is supported only for RDS for PostgreSQL 12 and later versions.
- The underlying architecture and major version of the DR instance must be the same as those of the primary DB instance.
- If the DR instance's minor version differs from the primary DB instance, it will automatically update to be the same as that of the primary DB instance after the DR relationship is established.
- Cross-cloud or cross-region DR relationships cannot be established across major versions.
- The DR instance can be promoted to primary and the DR replication status can be queried only after the DR relationship between the primary DB instance and DR instance is established.
- Ensure that the primary DB instance and DR instance are in the regions where Cloud Connect or Virtual Private Network (VPN) has been rolled out.
- DR instances do not support point-in-time recovery (PITR) or CBR snapshot-based backups. Perform such operations on the primary instance if needed.

How Cross-Region DR Works

Two RDS for PostgreSQL instances are deployed in two data centers, one in the production center and the other in the DR center. RDS replicates data from the primary instance in the production center to the DR instance in the DR center, keeping data synchronous across the regions. Before using this function, ensure that Cloud Connect can be used to connect the two regions.

Figure 2-1 Diagram



Service List

- Cloud Connect
- Virtual Private Cloud (VPC)
- Relational Database Service (RDS)

Notes on Usage

- The resource planning in this practice is for demonstration only. Adjust it as needed.

- The end-to-end test data in this practice is for reference only.

2.2 Resource Planning

Table 2-1 Resource planning

Category	Subcategory	Planned Value	Description
Production center VPC	VPC name	vpc-pg-01	Specify a name that is easy to identify.
	Region	CN-Hong Kong	To reduce network latency, select the region nearest to you.
	AZ	az1	-
	Subnet	192.168.10.0/24	Select a subnet with sufficient network resources.
	Subnet name	subnet-2aa1	Specify a name that is easy to identify.
DR center VPC	VPC name	vpc-pg-02	Specify a name that is easy to identify.
	Region	AP-Singapore	To reduce network latency, select the region nearest to you.
	AZ	az1	-
	Subnet	192.168.20.0/24	Select a subnet with sufficient network resources.
	Subnet name	subnet-a388	Specify a name that is easy to identify.
RDS for PostgreSQL instance in the production center	DB instance name/ID	rds-pg-01 04**in03	Specify a name that is easy to identify.
	Region	CN-Hong Kong	To reduce network latency, select the region nearest to you.
	DB engine version	PostgreSQL 12	-
	Private IP address	192.168.10.117	-
	DB instance type	Primary/Standby	Select **Primary/Standby** for the production instance.
	Storage type	Cloud SSD	-

Category	Subcategory	Planned Value	Description
	AZ	az1, az3	-
	Instance class	Dedicated 2 vCPUs | 4 GB	-
	Storage space	40 GB	-
RDS for PostgreSQL instance in the DR center	DB instance name/ID	rds-pg-02 5f**in03	Specify a name that is easy to identify.
	Region	AP-Singapore	To reduce network latency, select the region nearest to you.
	DB engine version	PostgreSQL 12	-
	Private IP address	192.168.20.69	-
	DB instance type	Single	Select **Single** for the DR instance.
	Storage type	Cloud SSD	-
	AZ	az1	-
	Instance class	Dedicated, 8 vCPUs | 16 GB	The CPU and memory specifications of the DR instance must be greater than or equal to those of the primary instance.
	Storage space	100 GB	The storage space of the DR instance must be greater than or equal to that of the primary instance.

2.3 Preparing an RDS for PostgreSQL Instance in the Production Center

This section describes how to create a VPC, a security group, and an RDS for PostgreSQL instance in the production center.

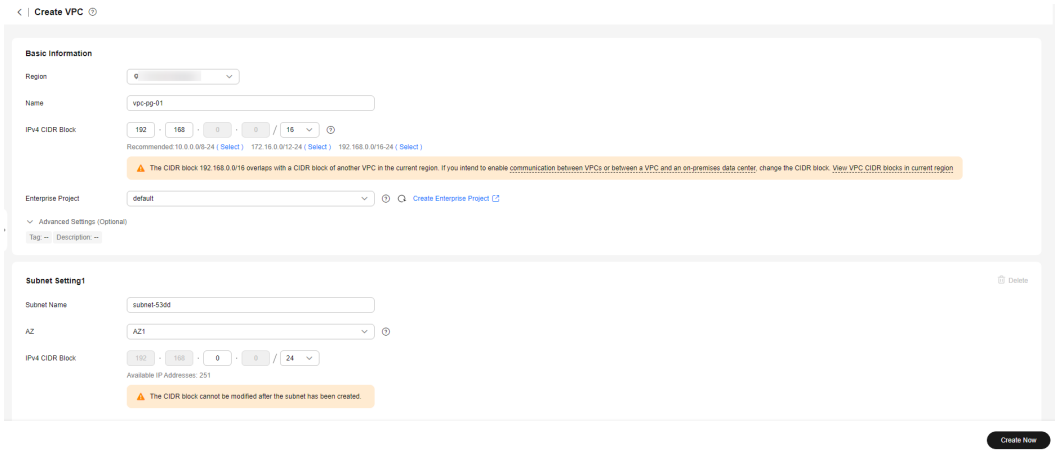
- **Step 1: Create a VPC and Security Group**
- **Step 2: Create an RDS for PostgreSQL Instance**

Step 1: Create a VPC and Security Group

Step 1 Go to the **Create VPC** page.

Step 2 Select **CN-Hong Kong** for **Region**. Configure the basic information, subnet, and IP address.

Figure 2-2 Creating a VPC

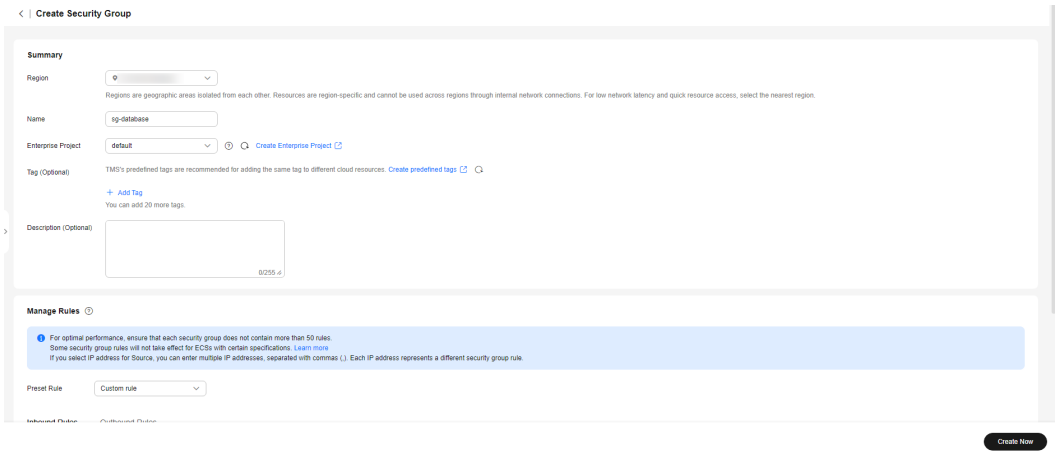


Step 3 Click **Create Now**.

Step 4 In the navigation pane of **Network Console**, choose **Access Control** > **Security Groups**.

Step 5 Click **Create Security Group**.

Figure 2-3 Creating a security group



Step 6 Click **Create Now**.

----End

Step 2: Create an RDS for PostgreSQL Instance

Step 1 Go to the **Buy DB Instance** page.

Step 2 Select **CN-Hong Kong** for **Region**. Configure the instance information and click **Buy**.

Figure 2-4 Selecting a DB engine version

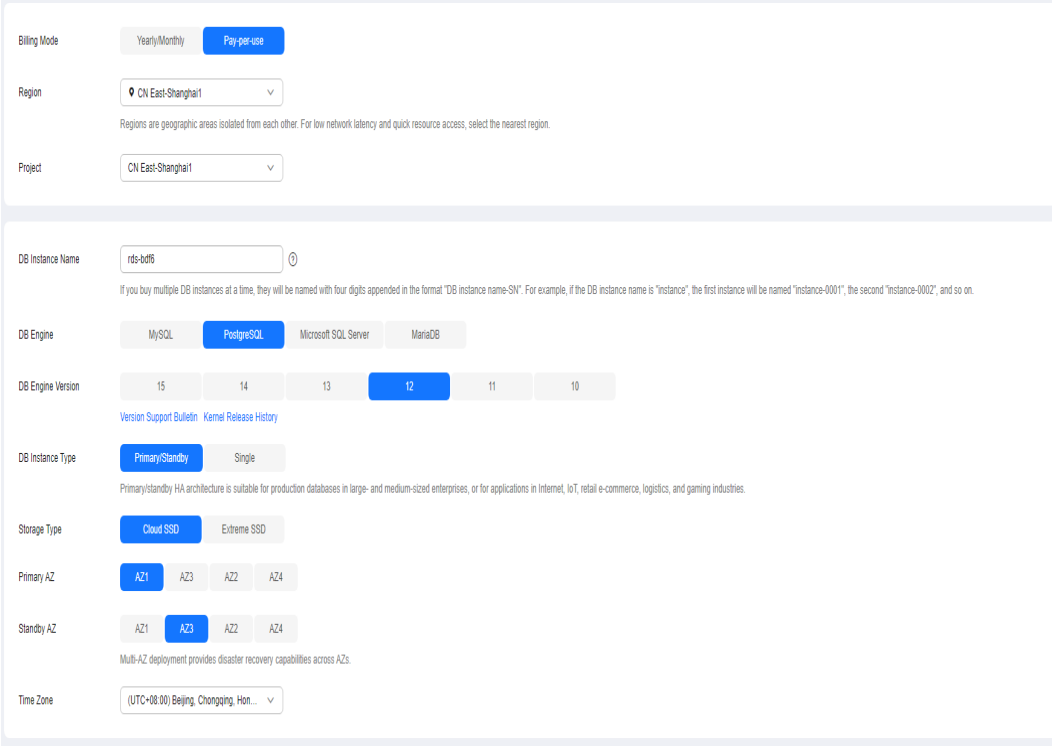


Figure 2-5 Selecting an instance class

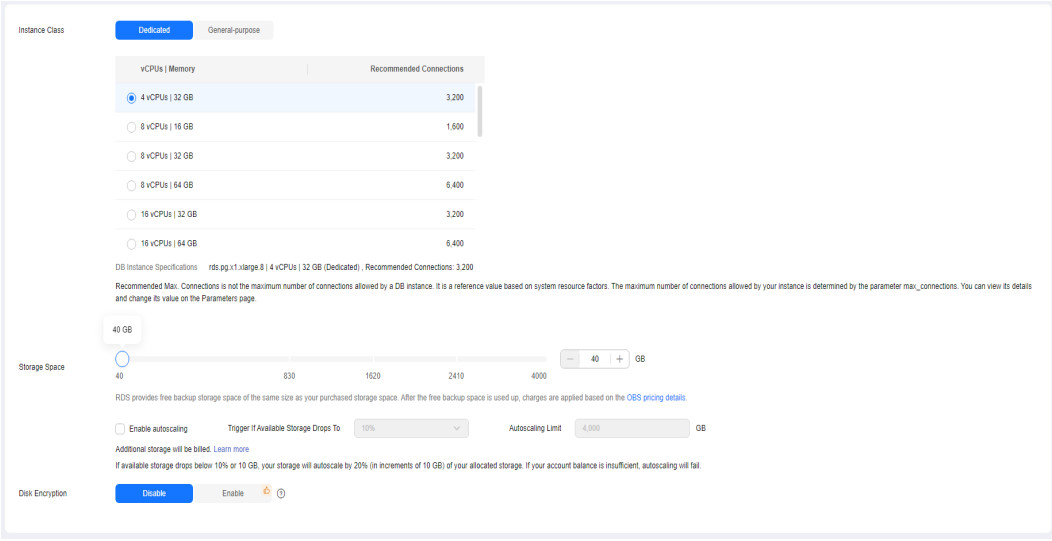


Figure 2-6 Configuring network information as planned

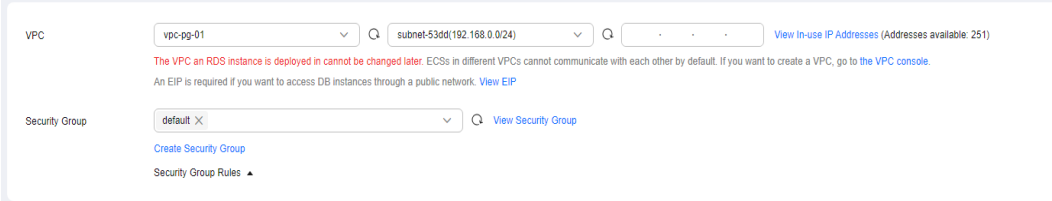
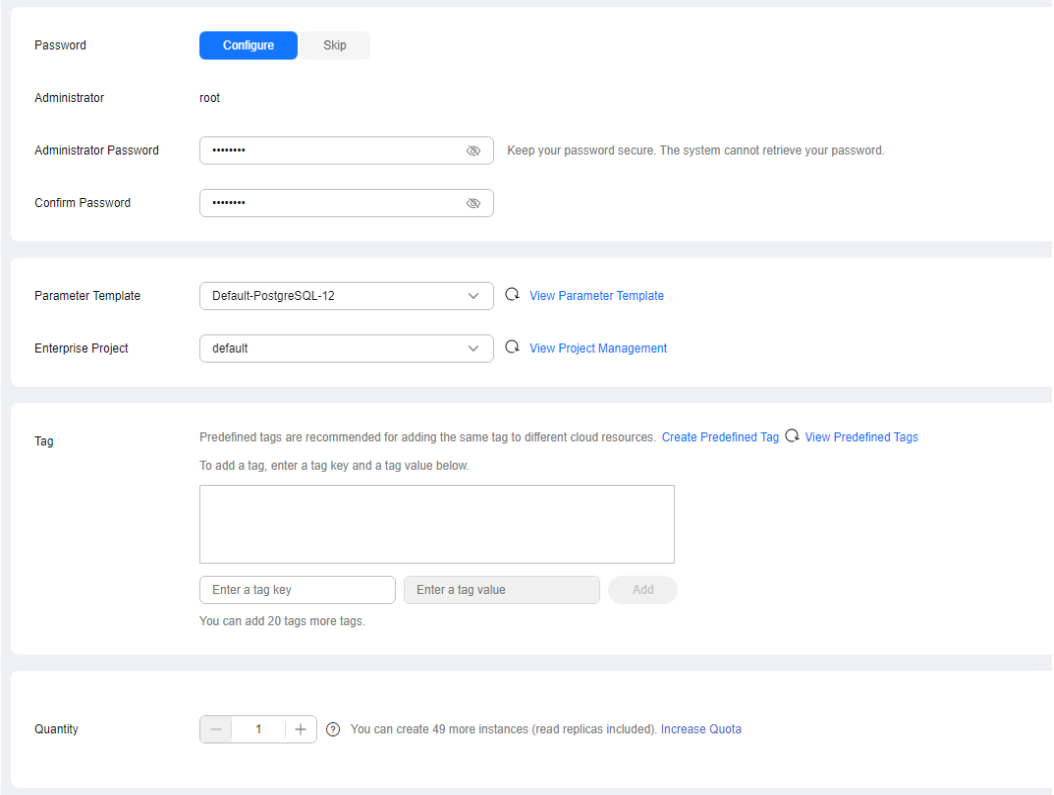


Figure 2-7 Setting an administrator password



Step 3 Confirm the settings.

- To modify your settings, click **Previous**.
- If you do not need to modify your settings, click **Submit**.

----End

2.4 Preparing an RDS for PostgreSQL Instance in the DR Center

This section describes how to create a VPC, a security group, and an RDS for PostgreSQL instance in the DR center.

NOTICE

- The VPC subnet CIDR block of the DR instance must be different from that of the production instance. This is the prerequisite for cross-region network connection.
- The security groups in the production center and DR center must allow access from the database ports in the VPC subnet CIDR blocks to each other.

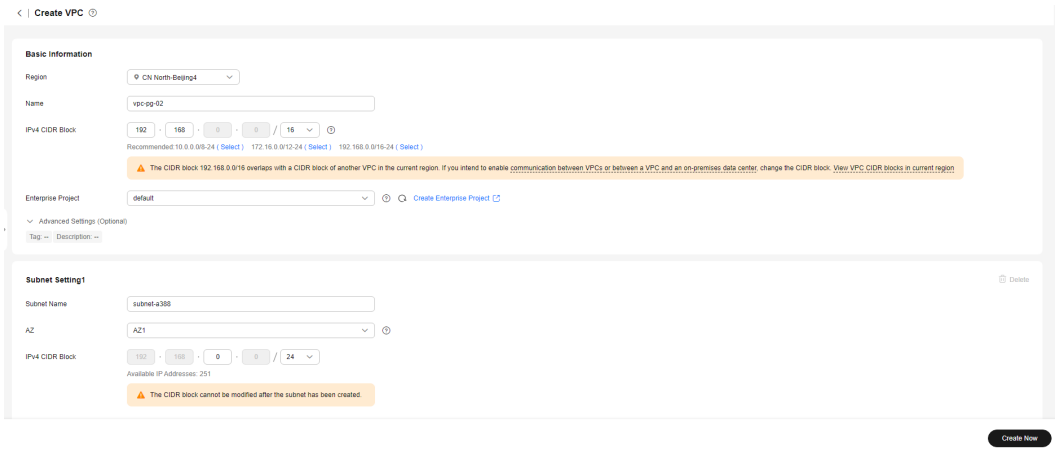
- **Step 1: Create a VPC and Security Group**
- **Step 2: Create an RDS for PostgreSQL Instance**

Step 1: Create a VPC and Security Group

Step 1 Go to the **Create VPC** page.

Step 2 Select **AP-Singapore** for **Region**. Configure the basic information, subnet, and IP address.

Figure 2-8 Creating a VPC

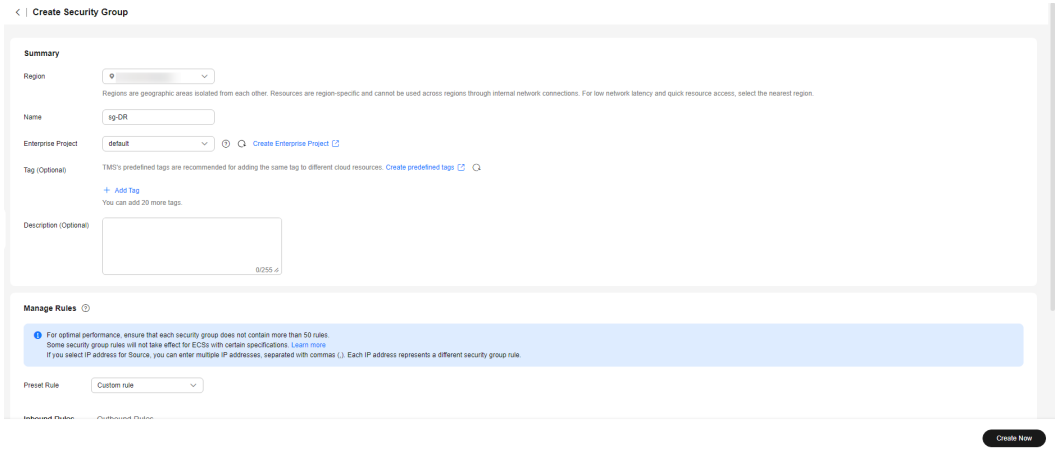


Step 3 Click **Create Now**.

Step 4 In the navigation pane of **Network Console**, choose **Access Control** > **Security Groups**.

Step 5 Click **Create Security Group**.

Figure 2-9 Creating a security group



Step 6 Click **Create Now**.

----End

Step 2: Create an RDS for PostgreSQL Instance

Step 1 Go to the **Buy DB Instance** page.

Step 2 Select **AP-Singapore** for **Region**. Configure the instance information and click **Buy**.

Figure 2-10 Selecting a DB engine version

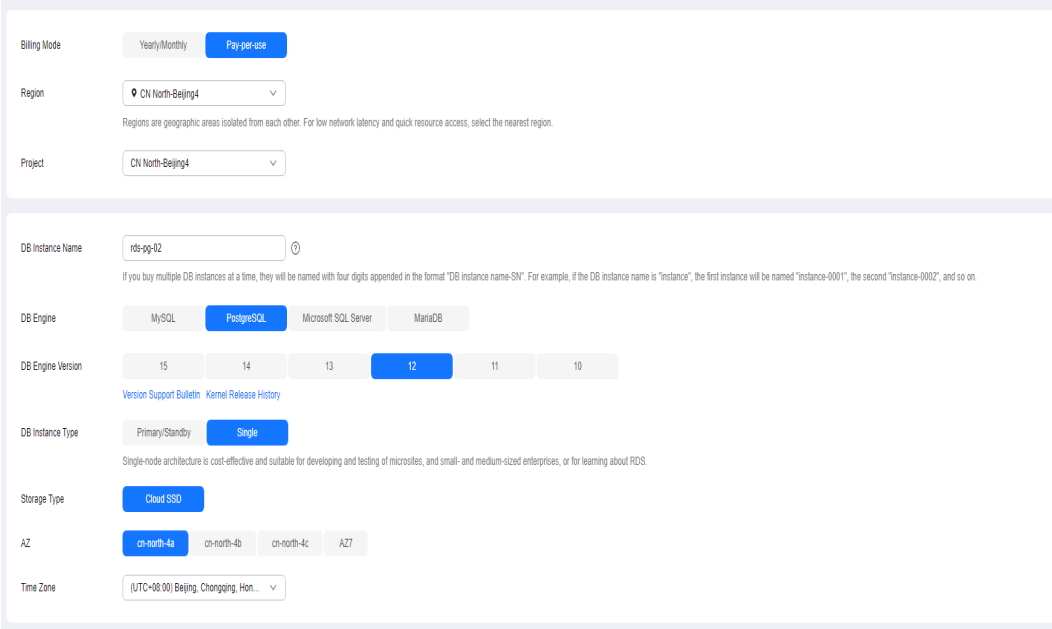


Figure 2-11 Selecting an instance class

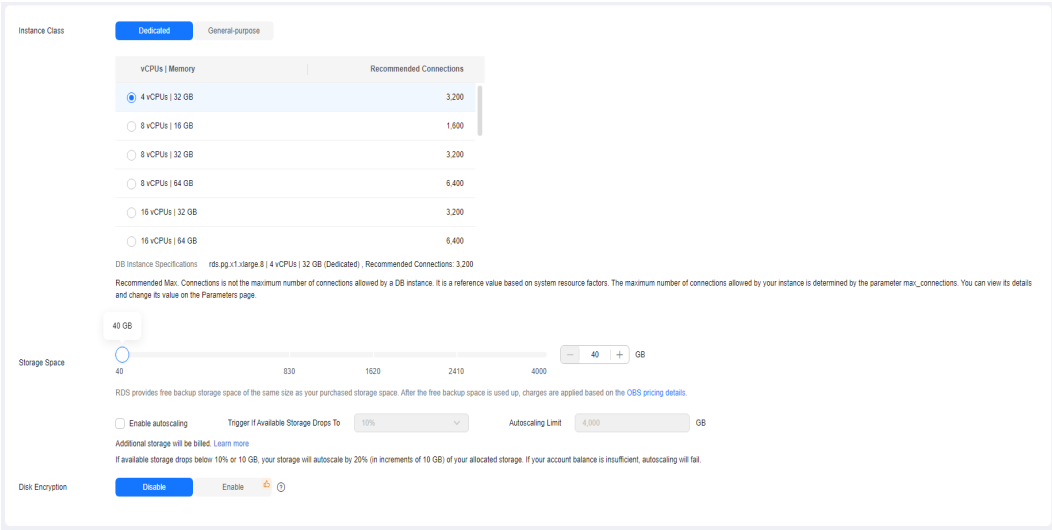


Figure 2-12 Configuring network information as planned

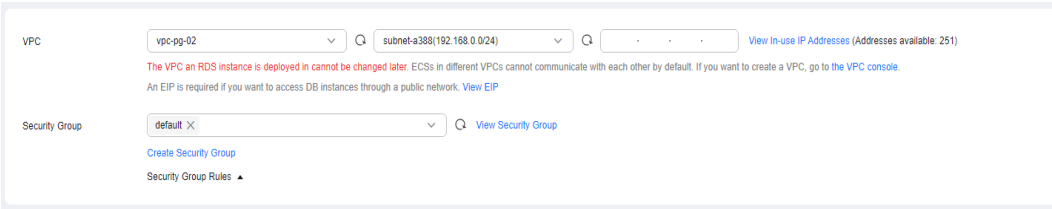
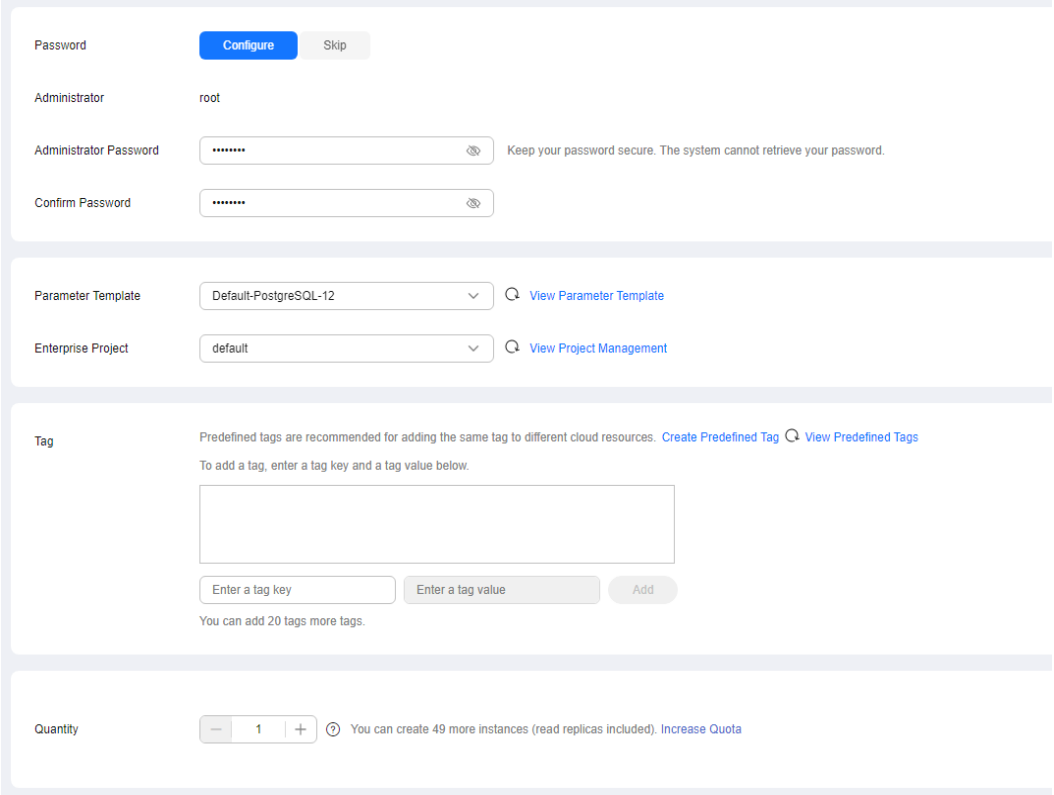


Figure 2-13 Setting an administrator password



Step 3 Confirm the settings.

- To modify your settings, click **Previous**.
- If you do not need to modify your settings, click **Submit**.

----End

2.5 Configuring Cross-Region Network Connectivity

Before setting up a DR relationship, you need to configure cross-region network connectivity. For details, see **Method 1: Using Cloud Connect to Connect VPCs in Different Regions** or **Method 2: Using VPN to Connect VPCs in Different Regions**.

You are advised to set a bandwidth size based on the transaction log generation rate metric. The bandwidth must be greater than or equal to 10 times the maximum value of this metric. This is because the unit of the network bandwidth is Mbit/s and that of the transaction log generation rate is MB/s.

For example, if the maximum transaction log generation rate is 10 MB/s, you are advised to set network bandwidth to 100 Mbit/s so that it is sufficient enough for the DR instance to synchronize data from the primary instance in a timely manner.

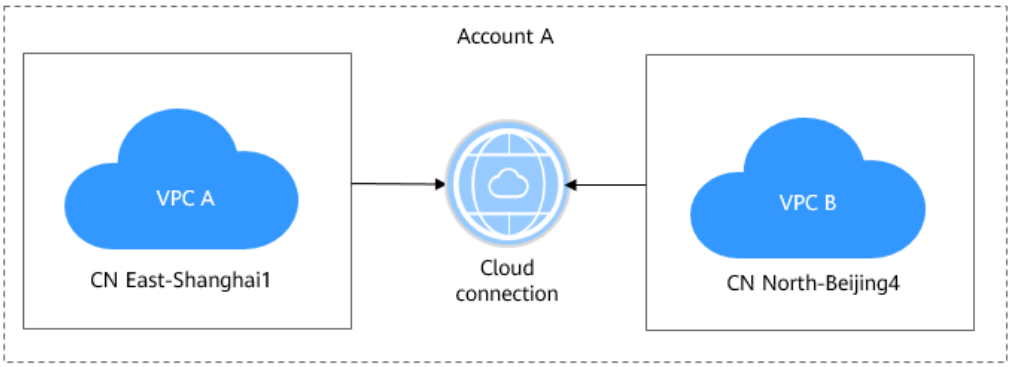
After the network is connected, you need to configure the security groups for the primary instance and DR instance to allow traffic from each other. For details, see **Configuring Security Groups**.

Method 1: Using Cloud Connect to Connect VPCs in Different Regions

Before setting up a DR relationship, you need to configure cross-region network connectivity.

You can use **Cloud Connect** to connect VPCs across regions.

Figure 2-14 Communication between VPCs in the same account but different regions



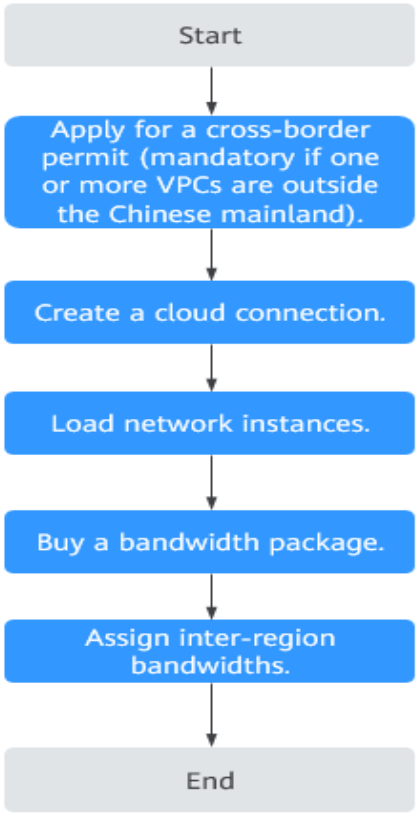
NOTICE

Ensure that the primary and DR instances are in the **regions where cloud connections are available**.

Ensure that the VPC subnets to which the primary and DR instances belong allow access from each other.

For details about how to enable communication between VPCs in different regions, see **Using a Cloud Connection to Connect VPCs in Different Regions**.

Figure 2-15 Flowchart



Method 2: Using VPN to Connect VPCs in Different Regions

You can use **Virtual Private Network (VPN)** to enable communication between VPCs across regions.

NOTICE

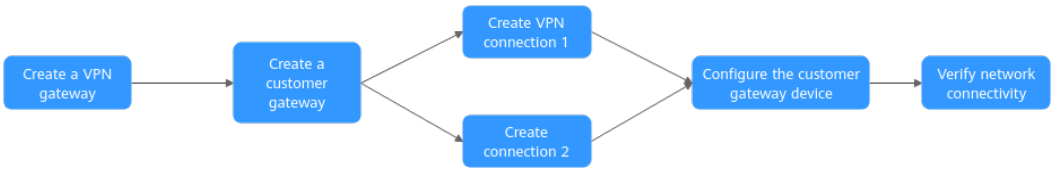
Ensure that the primary and DR instances are in the **regions where VPN is available**.

After configuring the VPN service, you need to contact the VPN customer service to configure the network.

Ensure that the VPC subnets to which the primary and DR instances belong allow access from each other.

For details about how to configure a VPN connection, see **Overview**.

Figure 2-16 Flowchart



Configuring Security Groups

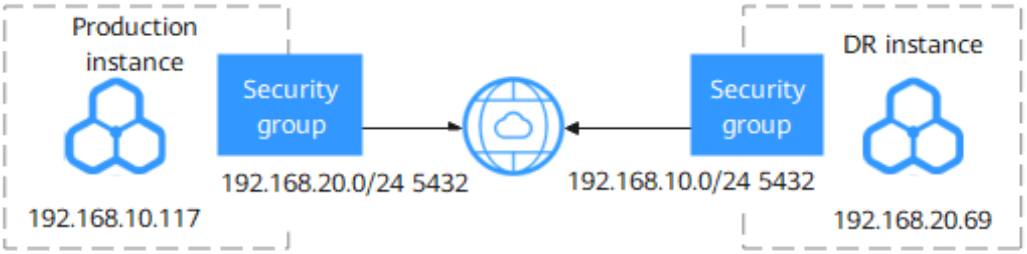
After connecting two VPCs in different regions, you need to configure security groups for the primary and DR instances so that ports in different VPC CIDR blocks can communicate with each other.

Suppose that there are two instances listed in **Table 2-2** and they use the default port 5432. The firewall configurations for them are as shown in **Figure 2-17**.

Table 2-2 CIDR block configurations

Instance	VPC CIDR Block	IP Address
Production instance	192.168.10.0/24	192.168.10.117
DR instance	192.168.20.0/24	192.168.20.69

Figure 2-17 Firewall configurations



2.6 Creating a DR Relationship

Scenarios

After a cross-region DR relationship is created, if the region where the primary instance is located encounters a natural disaster and the primary instance cannot be connected, you can promote the DR instance in another region to primary. To connect to the new primary instance, you only need to change the connection address on the application side.

Precautions

- Before using this function, ensure that the network between the DB instances in two different regions is connected. You can use **Cloud Connect** or **Virtual Private Network (VPN)** to connect the VPCs in different regions.
- Before using this function, ensure that the primary instance and DR instance are available and are deployed in different regions. The primary instance uses a primary/standby deployment and the DR instance uses a standalone deployment.
- The vCPUs, memory, and storage space of the DR instance must be greater than or equal to those of the primary instance.

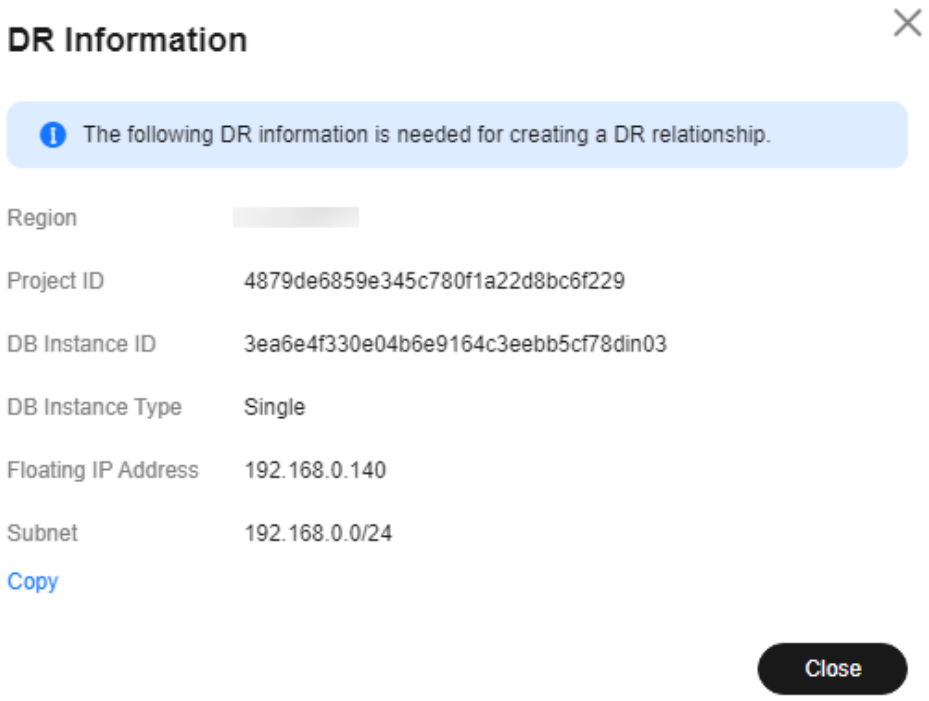
- The underlying architecture and major version of the DR instance must be the same as those of the primary DB instance.
- Cross-cloud or cross-region DR relationships cannot be established across major versions.
- After the API for configuring DR for the primary instance is called, you cannot change the instance class or perform a primary/standby switchover until the DR relationship is set up.
- After a DR relationship is set up, you can change the instance class of the DR instance. To use this function, **submit a service ticket**.
- After changing the database port or private IP address of the primary instance, you need to re-establish the DR relationship.
- After a DR instance is set up, a minor version upgrade cannot be performed.
- RDS for PostgreSQL 12 and later versions support cross-region DR.
- Modifying a parameter of the primary instance does not modify that of the DR instance. You need to modify the parameter of the DR instance separately.
- RDS for PostgreSQL DR instances do not support point-in-time recovery (PITR) or CBR snapshot-based backups. Perform such operations on the primary instance if needed.

Procedure

Step 1 Paste the configuration information of the DR instance to the production instance and configure DR for the production instance.

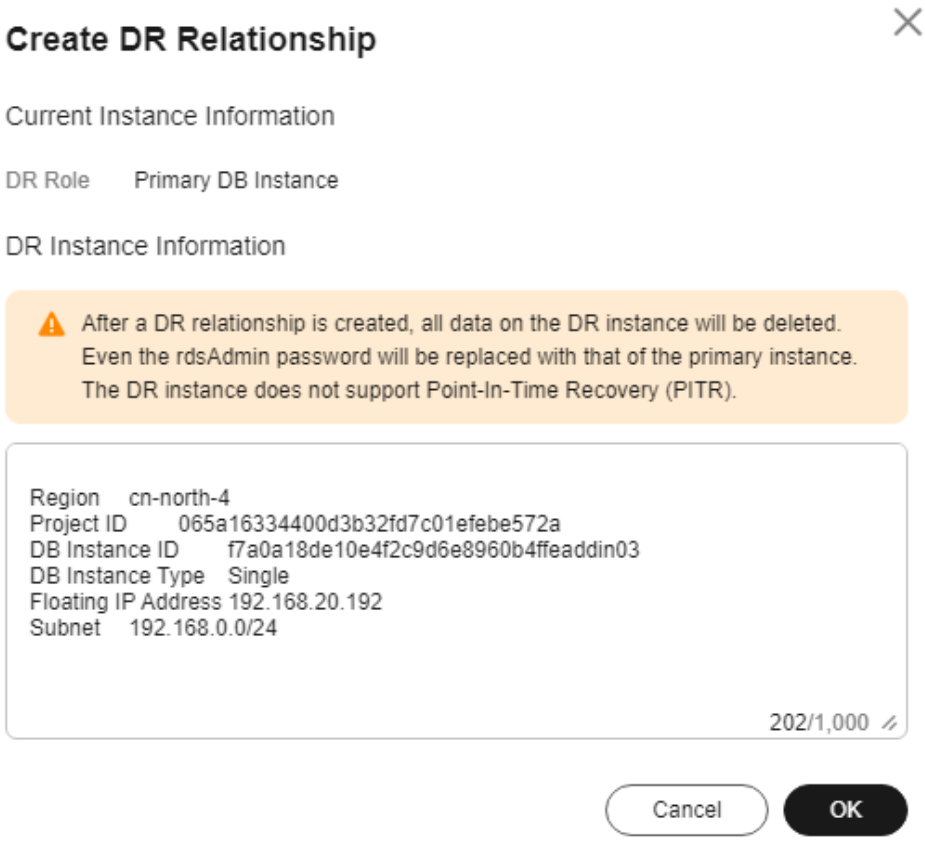
1. **Log in to the management console**.
2. Click  in the upper left corner and select the region where the DR instance is located, for example, AP-Singapore.
3. Click  in the upper left corner of the page and choose **Databases** > **Relational Database Service**.
4. On the **Instances** page, click the DR instance name to go to the **Overview** page.
5. Click **DR Information**.
6. In the displayed dialog box, click **Copy**.

Figure 2-18 Copying DR instance configuration information



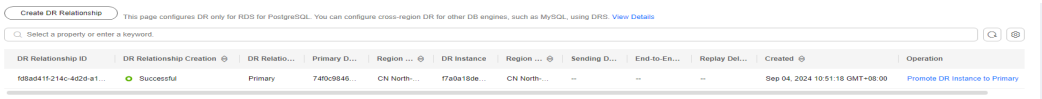
7. Click in the upper left corner of the management console and select the region where the production instance is located, for example, CN-Hong Kong.
8. On the **Instances** page, locate the production instance and choose **More** > **View DR Details** in the **Operation** column.
9. Click **Create DR Relationship**. In the displayed dialog box, paste the DR information copied from **Step 1.6** to the text box and click **OK** to configure DR for the production instance.

Figure 2-19 Pasting DR instance information



10. On the **DR Management** page of the production instance, check whether the DR is configured. If the value of **DR Relationship Creation** is **Successful**, the DR is successfully configured for the production instance. Perform subsequent operations only after this task is successfully executed.

Figure 2-20 Checking whether the DR is configured



Step 2 Paste the information of the production instance to the DR instance and configure DR for the DR instance.

1. On the **Instances** page, click the production instance name to go to the **Overview** page.
2. Click **DR Information**.
3. In the displayed dialog box, click **Copy**.
4. Click in the upper left corner and select the region where the DR instance is located, for example, AP-Singapore.
5. On the **Instances** page, locate the DR instance and choose **More** > **View DR Details** in the **Operation** column.

6. Click **Create DR Relationship**. In the displayed dialog box, paste the DR information copied from **Step 2.3** to the text box and click **OK** to configure DR for the DR instance.
7. On the **DR Management** page, check whether the DR is configured. If the value of **DR Relationship Creation** is **Successful**, the DR is successfully configured for the DR instance.
8. On the **DR Management** page, you can view the DR replication status, sending delay, end-to-end delay, and replay delay.

----End

2.7 Promoting a DR Instance to Primary

Scenarios

If the region where the primary instance is located suffers a natural disaster and the primary instance cannot be connected, you can promote the DR instance in another region to primary. To connect to the new primary instance, you only need to change the connection address on the application side.

Precautions

After a DR instance is promoted to primary, the DR relationship between it and the original primary instance is removed.

Procedure

Step 1 **Log in to the management console**.

Step 2 Click  in the upper left corner and select the region where the DR instance is located, for example, AP-Singapore.

Step 3 Click  in the upper left corner of the page and choose **Databases** > **Relational Database Service**.

Step 4 On the **Instances** page, click the name of the DR instance.

Step 5 In the navigation pane, choose **DR Management**.

Step 6 In the DR relationship list, click **Promote DR Instance to Primary** in the **Operation** column.

Step 7 In the displayed dialog box, click **OK**.

Step 8 Check the task execution result on the **Task Center** page. If the task status is **Completed**, the promotion is successful.

Step 9 Change the database connection address on the application side and manually switch workloads over to the new primary instance.

----End

2.8 Removing a DR Relationship

Scenarios

If a DR relationship is no longer needed, you can remove it.

Precautions

Only the DR relationship that has been successfully established can be removed. You must first remove the DR relationship of the DR instance and then that of the primary instance. Otherwise, an alarm may be generated.

Procedure

Step 1 **Log in to the management console**.

Step 2 Remove the DR relationship of the DR instance.

1. On the **Overview** page of the primary instance, click **DR Information**.
2. In the displayed dialog box, click **Copy**.
3. On the **Instances** page, click the name of the DR instance.
4. In the navigation pane, choose **DR Management**.
5. In the DR relationship list, click **Remove DR Relationship** in the **Operation** column.
6. Paste the copied DR information to the dialog box.
7. Check the task execution result on the **DR Management** page. If the list is deleted, the task is successfully executed.

Step 3 Remove the DR relationship of the primary instance by referring to **Step 2**.

----End

2.9 FAQs

- Question 1: What should I do if I fail to configure DR for a DR instance?

  Check whether the security groups of the production center and DR center allow traffic from the ports of the VPC subnets each other. If a VPN connection is used, check whether the VPCs are connected as instructed in **Method 2: Using VPN to Connect VPCs in Different Regions**.
- Question 2: Why did the system display a message indicating that there is a route conflict in the current VPC when I use Cloud Connect to connect VPCs?

  Rectify the fault by referring to **What Can I Do If There Is a Route Conflict When I Load a Network Instance to a Cloud Connection?**

3 RDS for PostgreSQL Publications and Subscriptions

Logical Definition

A publication can be defined on any primary physical replication server. The node where a publication is defined is called the publisher. A publication is a set of changes generated from a table or a group of tables, and might also be described as a change set or replication set. Each publication exists in only one database.

A subscription is the downstream side of logical replication. The node where a subscription is defined is called the subscriber. A subscription defines the connection to another database and the set of publications (one or more) to which it wants to subscribe. The subscriber database behaves in the same way as any other RDS for PostgreSQL instance (primary) and can be used as a publisher for other databases by defining its own publications.

Required Permissions

- To create a publication, the publisher must have the replication permission.
- When creating a publication for ALL TABLES, ensure that the publisher uses the **root** user of the initial or later versions for privilege escalation.
- When creating or deleting a subscription, ensure that the subscriber uses the **root** user of the initial or later versions for privilege escalation.
- When creating a publication or subscription, ensure that the publisher and subscriber are in the same VPC.

For details about **root** privilege escalation of each version, see **Privileges of the root User**.

Restrictions on Publications

- Publications may currently only contain tables (indexes, sequence numbers, and materialized views cannot be published). Each table can be added to multiple publications if needed.
- One publication can have multiple subscribers.
- ALL TABLES can be used to publish all tables.

- Multiple publications can be created in a given database, but each publication must have a unique name. The created publications can be obtained by querying pg_publication.
- Publications can choose to limit the changes they produce to any combination of INSERT, UPDATE, DELETE, and TRUNCATE, similar to how triggers are fired by particular event types. By default, all operation types are replicated.

  Example: To publish the UPDATE and DELETE operations on the **t1** table:

  ```
  CREATE PUBLICATION update_delete_only FOR TABLE t1
     WITH (publish = 'update, delete') ;
  ```

- Replica identity: A published table must have a replica identity configured in order to be able to replicate UPDATE and DELETE operations. If **nothing** is set for the replica identity, subsequent UPDATE or DELETE operations will cause an error on the publisher.

  You can obtain the replica identity of a table from **pg_class.relreplident**.

  relreplident is of character type and identifies columns used to form "replica identity" for rows: d = default, f = all columns, i = index, and n = nothing.

  To check whether a table has an index constraint that can be used as a replica identity, run the following:

  ```
  SELECT quote_ident(nspname) || '.' || quote_ident(relname) AS name, con.ri AS keys,
        CASE relreplident WHEN 'd' THEN 'default' WHEN 'n' THEN 'nothing' WHEN 'f' THEN
  'full' WHEN 'i' THEN 'index' END AS replica_identity
  FROM pg_class c JOIN pg_namespace n ON c.relnamespace = n.oid, LATERAL (SELECT
  array_agg(contype) AS ri FROM pg_constraint WHERE conrelid = c.oid) con
  WHERE relkind = 'r' AND nspname NOT IN ('pg_catalog', 'information_schema', 'monitor',
  'repack', 'pg_toast')
  ORDER BY 2,3;
  ```

- Command for changing a replica identity

  The replica identity of a table can be changed using **ALTER TABLE**.

  ```
  ALTER TABLE table_name REPLICA IDENTITY
  { DEFAULT | USING INDEX index_name | FULL | NOTHING };
  -- There are four forms:
  ALTER TABLE t_normal REPLICA IDENTITY DEFAULT;                    -- The primary key is
  used as the replica identity. If there is no primary key, the replica identity is set to FULL.
  ALTER TABLE t_normal REPLICA IDENTITY FULL;                       -- The entire row is used
  as the replica identity.
  ALTER TABLE t_normal REPLICA IDENTITY USING INDEX t_normal_v_key; -- A unique index
  is used as the replica identity.
  ALTER TABLE t_normal REPLICA IDENTITY NOTHING;                    -- No replica identity is
  set.
  ```

- Precautions for using replica identities
  - If a table has a primary key, the replica identity can be set to DEFAULT.
  - If a table does not have a primary key but has a non-null unique index, the replica identity can be set to INDEX.
  - If a table does not have a primary key or a non-null unique index, the replica identity can be set to FULL. This, however, is very inefficient and should only be used as a fallback if no other solution is possible.
  - In all cases other than those mentioned above, logical replication cannot be implemented. The output information is insufficient, and an error may be reported.

- **If a table with replica identity "nothing" is added to logical replication, deleting or updating the table will cause an error on the publisher.**

Restrictions on Subscriptions

- To ensure that failover slots are used, failover slots must be created on the publisher and associated with the existing replication slots using **create_slot = false**.

  CREATE SUBSCRIPTION sub1 CONNECTION 'host=192.168.0.1 port=5432 user=user1 dbname=db1' PUBLICATION pub_name with (create_slot = false,slot_name = FailoverSlot_name);
- Logical replication does not replicate DDL changes, so the tables in the publication set must already exist on the subscriber.
- Multiple subscriptions can be created in a given database. These subscriptions can come from one or more publishers.
- A given table of a subscriber cannot accept multiple publications from the same source.
- When creating a subscription or altering a subscription, you can use **enable** to enable the subscription or **disable** to suspend the subscription.
- To delete a subscription, use **DROP SUBSCRIPTION**. Note that after a subscription is deleted, the local table and data are not deleted, but upstream information of the subscription is no longer received.

NOTICE

If a subscription is associated with a replication slot, **DROP SUBSCRIPTION** cannot be executed inside a transaction block. You can use **ALTER SUBSCRIPTION** to disassociate the subscription from the replication slot.

To completely delete a subscription, perform the following steps:

a. Query the replication slot associated with the subscription on the subscriber.

   select subname,subconninfo,subslotname from pg_subscription where subname = 'sub2';

   - **subname** indicates the subscriber name.
   - **subconninfo** indicates information about the connected remote host.
   - **subslotname** indicates the replication slot name of the remote host.

b. On the subscriber, disassociate the subscription from the replication slot and delete the subscription.

   ALTER SUBSCRIPTION subname SET (slot_name = NONE);

   DROP SUBSCRIPTION subname;

c. Delete the associated replication slot at the publisher.

   select pg_drop_replication_slot(' slot_name);

Syntax Reference

- Publications

  CREATE PUBLICATION is used to create a publication, **DROP PUBLICATION** is used to delete a publication, and **ALTER PUBLICATION** is used to modify a publication.

  After a publication is created, tables can be added or removed dynamically using **ALTER PUBLICATION**. Such operations are all transactional.

- Subscriptions

  CREATE SUBSCRIPTION is used to create a subscription, **DROP SUBSCRIPTION** is used to delete a subscription, and **ALTER SUBSCRIPTION** is used to modify a subscription.

  After creating a subscription, you can use **ALTER SUBSCRIPTION** to suspend or resume the subscription at any time. Deleting and recreating a subscription results in the loss of synchronized information, which means that related data needs to be synchronized again.

For details, see the official documentation. PostgreSQL 13 is used as an example.

- Creating a publication: **https://www.postgresql.org/docs/13/sql-createpublication.html**
- Deleting a publication: **https://www.postgresql.org/docs/13/sql-droppublication.html**
- Modifying a publication: **https://www.postgresql.org/docs/13/sql-alterpublication.html**

4 User-Defined Data Type Conversion

Description

There are three data type conversion modes for PostgreSQL: implicit conversion, assignment conversion, and explicit conversion. They correspond to i (Implicit), a (Assignment), and e (Explicit) in the pg_cast system catalog.

- Implicit conversion: a conversion from low bytes to high bytes of the same data type, for example, from **int** to **bigint**
- Assignment conversion: a conversion from high bytes to low bytes of the same data type, for example, from **smallint** to **int**
- Explicit conversion: a conversion between different data types

How to Use

1. Before converting data types, you can run the following command to check whether RDS for PostgreSQL supports data type conversion:

```
select * from pg_catalog.pg_cast ;
oid  | castsource | casttarget | castfunc | castcontext | castmethod
-------+------------+------------+----------+-------------+------------
 11277 |         20 |         21 |      714 | a           | f
 11278 |         20 |         23 |      480 | a           | f
 11279 |         20 |        700 |      652 | i           | f
 11280 |         20 |        701 |      482 | i           | f
......
```

2. Run the following command to check whether **int4** can be converted to **text**:

```
select * from pg_catalog.pg_cast where castsource = 'int4'::regtype and casttarget = 'bool'::regtype;
  oid  | castsource | casttarget | castfunc | castcontext | castmethod
-------+------------+------------+----------+-------------+------------
 11311 |         23 |         16 |     2557 | e           | f
(1 row)
```

The conversion is supported, and the conversion type is implicit conversion.

If no built-in conversion functions are available, customize a conversion function to support the conversion. For details, see **User-Defined Data Type Conversion**.

User-Defined Data Type Conversion

- Use double colons (::) to perform a forcible conversion.

```
select '10'::int,'2023-10-05'::date;
 int4 |    date
------+------------
```



```
  10 | 2023-10-05
(1 row)
```

- Use the CAST function to convert the type.

```
select CAST('10' as int),CAST('2023-10-05' as date);
 int4 |    date
------+------------
   10 | 2023-10-05
(1 row)
```

- Customize a data type conversion.

  For details, see **https://www.postgresql.org/docs/14/sql-createcast.html**.

  NOTICE

  Adding a custom type conversion will affect the existing execution plans of RDS for PostgreSQL. Therefore, customizing type conversions are not recommended.

  – Conversion between time and character types

    CREATE CAST(varchar as date) WITH INOUT AS IMPLICIT;

  – Conversion between boolean types and numeric types

    create cast(boolean as numeric) with INOUT AS IMPLICIT;

  – Conversion between numeric types and character types

    create cast(varchar as numeric) with INOUT AS IMPLICIT;

  Example: Convert **text** to **date**.

```
create or replace function public.text_to_date(text) returns date as
$$
  select to_date($1,'yyyy-mm-dd');
$$
language sql strict;

create cast (text as date) with function public.text_to_date(text) as implicit;

select text '2023-09-09' + 1;
  ?column?
------------
 2023-09-10
(1 row)
```


5 Using Client Drivers to Implement Failover and Read/Write Splitting

Since PostgreSQL 10 (libpq.so.5.10), libpq has been supporting failover and read/write splitting, and Java Database Connectivity (JDBC) has been supporting read/write splitting, failover, and load balancing.

PostgreSQL client drivers are backward compatible. Even RDS for PostgreSQL 9.5 and 9.6 instances can be connected through the libpq driver of the latest version to implement failover.

NOTE

In this section, failover refers to the failover of read-only workloads.

- libpq is a C application programming interface (API) to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries.
- JDBC is an API used in Java to define how client programs access databases. In PostgreSQL, JDBC supports failover and load balancing.

Table 5-1 Functions supported by libpq and JDBC

Driver	Read/Write Splitting	Load Balancing	Failover
libpq	√	×	√
JDBC	√	√	√

Using libpq for Failover and Read/Write Splitting

You can use libpq functions to connect to multiple databases. If one database fails, workloads are automatically switched to another available database.

postgresql://[user[:password]@][netloc][:port][,...][/dbname][?param1=value1&...]

Example: Connect to one primary RDS for PostgreSQL instance and two read replicas. Read requests will not fail as long as there is at least one available instance.

postgres://
<instance_ip>:*<instance_port>*,*<instance_ip>*:*<instance_port>*,*<instance_ip>*:*<instance_port>*/*<database_name>*?**target_session_attrs=any**

Table 5-2 Parameter description

Parameter	Description	Example Value
<instance_ip>	IP address of the DB instance.	If you attempt to access the instance from an ECS, set *instance_ip* to the floating IP address displayed on the **Overview** page of the instance. If you attempt to access the instance through an EIP, set *instance_ip* to the EIP that has been bound to the instance.
<instance_port>	Database port of the DB instance.	Set this parameter to the database port displayed on the **Overview** page. Default value: **5432**
<database_name>	Name of the database to be connected.	The default management database is **postgres**. You can enter the database name based on the site requirements.
target_session_attrs	Type of the database to be connected.	• **any** (default): libpq can connect to any database. If the connection is interrupted due to a fault in the database, libpq will attempt to connect to another database to implement failover. • **read-write**: libpq can only connect to a database that supports both read and write. libpq attempts a connection to the first database you specified. If this database supports only read or write operations, libpq disconnects from it and attempts to connect to the second one and so on until it connects to a database that supports both read and write. • **read-only**: libpq can only connect to a read-only database. libpq attempts a connection to the first database you specified. If this database is not a read-only database, libpq disconnects from it and attempts to connect to the second one and so on until it connects to a read-only database. This value is not supported in RDS for PostgreSQL 13 (libpq.so.5.13) or earlier versions.

For details about libpq and related parameters, see **Connection Strings**.

You can use the **pg_is_in_recovery()** function in your application to determine whether the connected database is a primary instance (indicated by **f**) or a read replica to implement read/write splitting.

The following is an example of Python code (psycopg2 a wrapper for libpq):

```
// There will be security risks if the username and password used for authentication are directly written into code. Store the username and password in ciphertext in the configuration file or environment variables.
// In this example, the username and password are stored in the environment variables. Before running this example, set environment variables EXAMPLE_USERNAME_ENV and EXAMPLE_PASSWORD_ENV as needed.

import psycopg2
import os

username = os.getenv("EXAMPLE_USERNAME_ENV")
password = os.getenv("EXAMPLE_PASSWORD_ENV")
conn = psycopg2.connect(database=<database_name>,host=<instance_ip>, user=username,
password=password, port=<instance_port>, target_session_attrs="read-write")
cur = conn.cursor()
cur.execute("select pg_is_in_recovery()")
row = cur.fetchone()
print("recovery =", row[0])
```

Using JDBC for Failover and Read/Write Splitting

You can define multiple databases (hosts and ports) in the connection URL and separate them with commas (,). JDBC will attempt to connect to them in sequence until the connection is successful. If the connection fails, an error message is displayed.

jdbc:postgresql://node1,node2,node3/${database}?targetServerType=preferSecondary&loadBalanceHosts=true

Example:

jdbc:postgresql://*<instance_ip>***:***<instance_port>***,***<instance_ip>***:***<instance_port>***,***<instance_ip>***:***<instance_port>***/***<database_name>***?targetServerType=preferSecondary&loadBalanceHosts=true**

For details about the Java code, see **Connecting to an RDS for PostgreSQL Instance Through JDBC**.

Table 5-3 Parameter description

Parameter	Description	Example Value
targetServerType	Type of the database to be connected.	• **any**: any database. • **primary**: primary database (writable and readable). For versions earlier than JDBC 42.2.0, use the parameter value **master**. • **secondary**: secondary database (readable). For versions earlier than JDBC 42.2.0, use the parameter value **slave**. • **preferSecondary**: The secondary database is preferred. If no secondary database is available, the primary database is connected. For versions earlier than JDBC 42.2.0, use the parameter value **preferSlave**.
loadBalanceHosts	Sequence of databases to be connected.	• **False** (default): Databases are connected in the sequence defined in the URL. • **True**: Databases are randomly connected.

NOTE

To distinguish between the primary and secondary databases, check whether data can be written to the database. If yes, it is a primary database. If no, it is a secondary database. You can use the **pg_is_in_recovery()** function to determine whether a database is a primary database. For details, see **Using libpq for Failover and Read/Write Splitting**.

To implement read/write splitting, you need to configure two data sources. For the first data source, set **targetServerType** to **primary** to process write requests. For the second data source:

- If there is only one read replica, set **targetServerType** to **preferSecondary** to process read requests. Assume that the IP addresses of the primary instance and read replica are 10.1.1.1 and 10.1.1.2, respectively.

  jdbc:postgresql://10.1.1.2:5432,10.1.1.1:5432/${database}?targetServerType=preferSecondary

- If there are two read replicas, set **targetServerType** to **any** to process read requests. Assume that the IP addresses of the read replicas are 10.1.1.2 and 10.1.1.3, respectively.

  jdbc:postgresql://10.1.1.2:5432,10.1.1.3:5432/${database}?targetServerType=any&loadBalanceHosts=true

6 Other Extension Plug-Ins

In addition to mandatory plug-ins pg_stat_statements, btree_gist, and PoWA, the following plug-ins are used for collecting new performance indicators:

- pg_qualstats
- pg_stat_kcache
- pg_wait_sampling
- pg_track_settings
- hypopg

Each of the plug-ins can extend different performance metrics.

Currently, only pg_track_settings is supported on Huawei Cloud.

pg_track_settings Plug-In Extension

Step 1 Log in to the management console.

Step 2 Click  in the upper left corner and select a region.

Step 3 Click  in the upper left corner of the page and choose **Databases** > **Relational Database Service**.

Step 4 On the **Instances** page, locate the target DB instance and click **Log In** in the **Operation** column.

Alternatively, click the instance name on the **Instances** page. On the displayed **Basic Information** page, click **Log In** in the upper right corner of the page.

Step 5 On the displayed login page, enter the correct username and password and click **Log In**.

Step 6 Select the powa database and run the SQL command to create pg_track_settings.

```
select control_extension('create', 'pg_track_settings');
```

Step 7 Create a PostgreSQL database (powa-repository) on the ECS, and install and activate pg_track_settings to collect performance metrics.

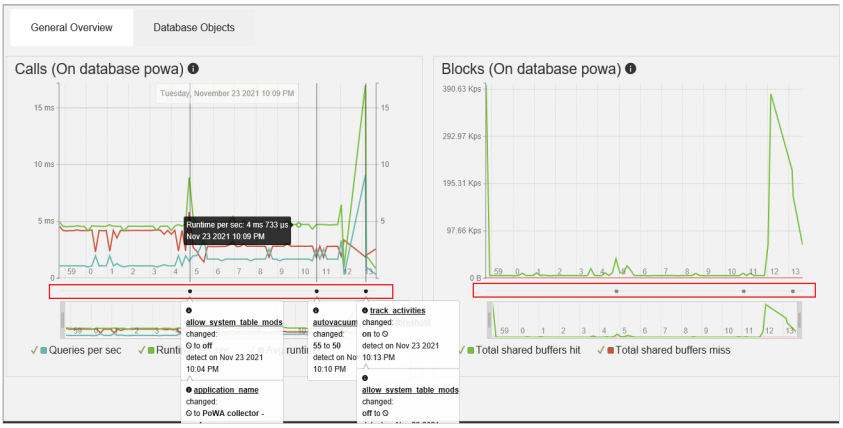
```
# pg_track_settings
cd /home/postgres/env
wget https://github.com/rjuju/pg_track_settings/archive/refs/tags/2.0.1.tar.gz
```



```
mv 2.0.1.tar.gz pg_track_settings.2.0.1.tar.gz
tar -xzvf pg_track_settings.2.0.1.tar.gz
cd pg_track_settings-2.0.1
make && make install
# powa-repository
psql -d powa
powa=# create extension pg_track_settings ;
CREATE EXTENSION
# Activate the pg_track_settings collection function for the target instance.
dbpowa=# select powa_activate_extension(1, 'pg_track_settings');
powa_activate_extension
-------------------------
t
(1 row)
```

Step 8 Verify the pg_track_settings plug-in extension.

Change the value of the **autovacuum_analyze_threshold** parameter on the target instance to **55**. The default value is **50**. After about 5 minutes, you can view the modification record on the PoWA, as shown in the following figure.



The contents in the three boxes in the preceding figure are as follows:

- The time when pg_track_settings is activated and the database parameter value at that time.
- The time when the **autovacuum_analyze_threshold** parameter is modified, its original value, and changed value.
- The time when pg_track_settings is canceled and the database parameter value at that time.

----End

7 Best Practices for Using PoWA

7.1 Overview

PoWA is an open-source system used to monitor the performance of RDS for PostgreSQL databases. It consists of the PoWA-archivist, PoWA-collector, and PoWA-web components and obtains performance data through other extensions installed in the RDS for PostgreSQL databases. The key components are as follows:

- PoWA-archivist: the PostgreSQL extension for collecting performance data obtained by other extensions.
- PoWA-collector: the daemon that gathers performance metrics from remote PostgreSQL instances on a dedicated repository server.
- PoWA-web: the web-based user interface displaying performance metrics collected by the PoWA-collector.
- Other extensions: the sources of performance metric data. They are installed on the target PostgreSQL database.
- PoWA: the system name.

Security Risk Warning

The following security risks may exist during PoWA deployment and configuration.

- (Remote mode) When configuring instance performance metric information to be collected in powa-repository, you need to enter the IP address, **root** username, and connection password of the target instance. You can query related information in the **powa_servers** table. The connection password is displayed in plaintext.
- In the **PoWA-collector** configuration file, the powa-repository connection information does not contain the connection password. It means that the powa-repository connection configuration item for PoWA-collector must be trust.
- In the **PoWA-web** configuration file, the **root** username and connection password of powa-repository (remote mode) or DB instance (local mode) are optional and stored in plaintext.

Before using the PoWA, you need to be aware of the preceding security risks. For details about how to harden security, see the **official PoWA documentation**.

Other Supported Extensions

In addition to mandatory extensions pg_stat_statements, btree_gist, and PoWA, the following extensions are used for collecting extended performance metrics:

- pg_qualstats
- pg_stat_kcache
- pg_wait_sampling
- pg_track_settings
- hypopg

Each of the extensions can collect different extended performance metrics. For details about the extensions supported by RDS for PostgreSQL, see **Supported Extensions**.

7.2 Supported Performance Metrics

7.2.1 Database Performance Metrics

General Overview

Figure 7-1 General Overview

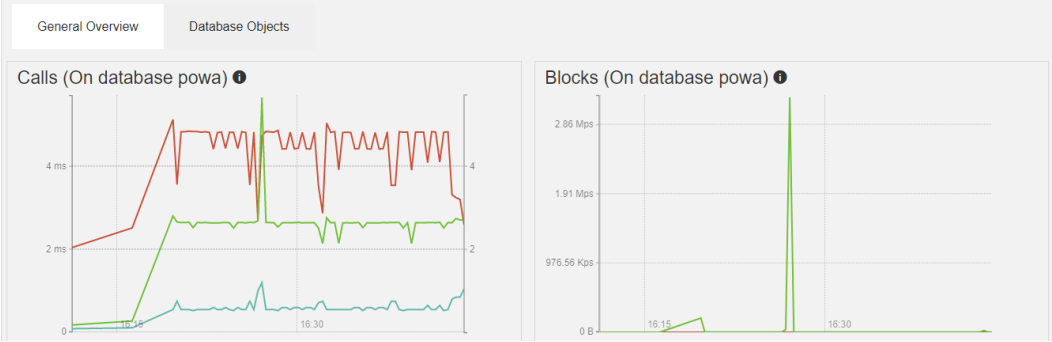


Table 7-1 Calls field description

Field	Description
Queries per sec	Number of queries executed per second
Runtime per sec	Total duration of queries executed per second
Avg runtime	Average query duration

Table 7-2 Blocks field description

Field	Description
Total shared buffers hit	Amount of data found in shared buffers
Total shared buffers miss	Amount of data found in OS cache or read from disk

Database Objects

Figure 7-2 Database Objects

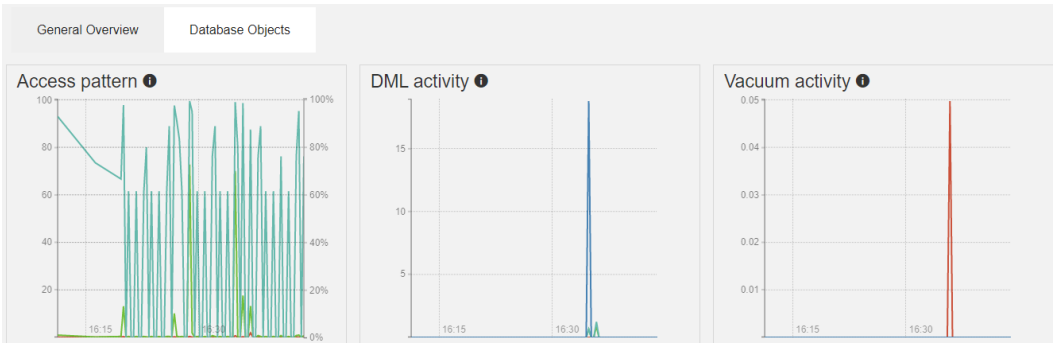


Table 7-3 Access pattern field description

Field	Description
Index scans ratio	Ratio of index scans to sequential scans
Index scans	Number of index scans per second
Sequential scans	Number of sequential scans per second

Table 7-4 DML activity field description

Field	Description
Tuples inserted	Number of tuples inserted per second
Tuples updated	Number of tuples updated per second
Tuples HOT updated	Number of heap-only tuples (HOT) updated per second
Tuples deleted	Number of tuples deleted per second

Table 7-5 Vacuum activity field description

Field	Description
# Vacuum	Number of vacuums per second
# Autovacuum	Number of autovacuums per second
# Analyze	Number of analyses per second
# Autoanalyze	Number of autoanalyses per second

Details for all databases

Figure 7-3 Details for all databases

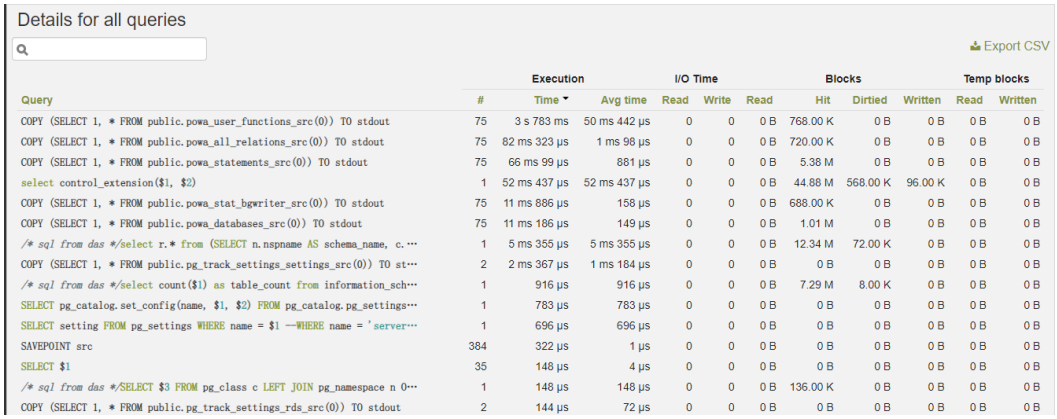


Table 7-6 Details for all databases field description

Field	Description
Query	SQL statement to be executed
(Execution) #	Number of times that the SQL statement is executed
(Execution) Time	Total execution time of the SQL statement
(Execution) Avg time	Average time for executing the SQL statement
(I/O Time) Read	Read I/O wait time
(I/O Time) Write	Write I/O wait time
(Blocks) Read	Number of disk read pages
(Blocks) Hit	Number of hit pages in the shared buffer
(Blocks) Dirtied	Number of dirty pages
(Blocks) Written	Number of disk write pages
(Temp blocks) Read	Number of disk temporary read pages

Field	Description
(Temp blocks) Write	Number of disk temporary write pages

7.2.2 Instance Performance Metrics

General Overview

Figure 7-4 General Overview metrics

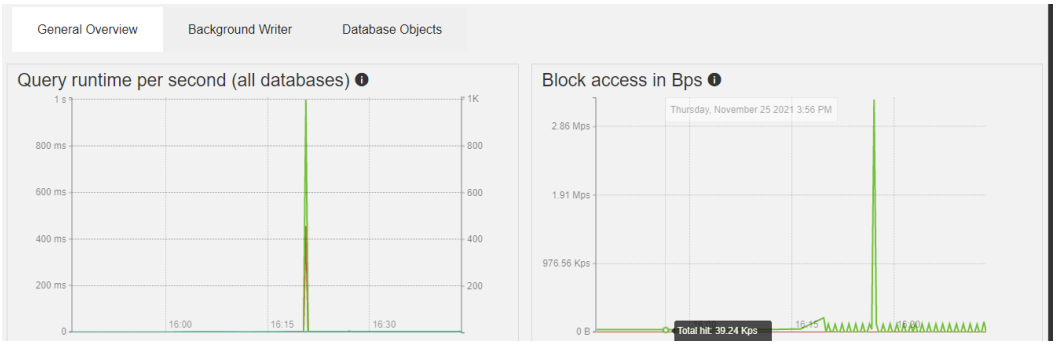


Table 7-7 Query runtime per second (all databases) field description

Field	Description
Queries per sec	Number of queries executed per second
Runtime per sec	Total duration of queries executed per second
Avg runtime	Average query duration

Table 7-8 Block access in Bps field description

Field	Description
Total hit	Amount of data found in shared buffers
Total read	Amount of data found in OS cache or read from disk

Background Writer

Figure 7-5 Background Writer metrics

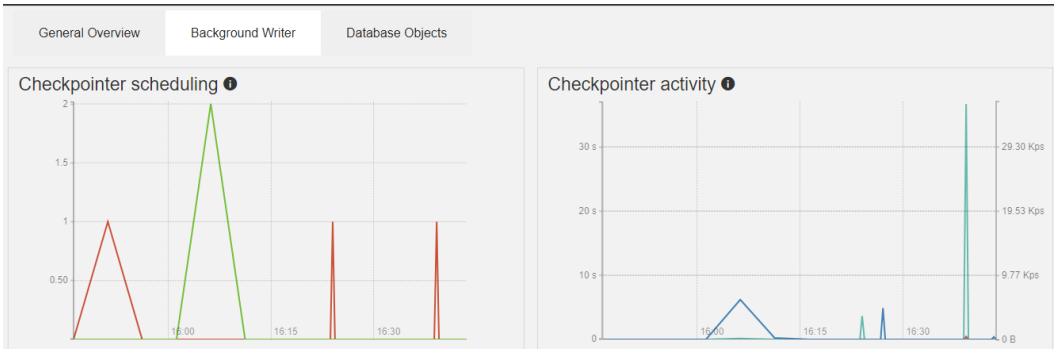


Table 7-9 Checkpointer scheduling field description

Field	Description
of requested checkpoints	Number of requested checkpoints that have been performed
of scheduled checkpoints	Number of scheduled checkpoints that have been performed

Table 7-10 Checkpointer activity field description

Field	Description
Buffers alloc	Number of buffers allocated
Sync time	Total amount of time that has been spent in the portion of checkpoint processing where files are synchronized to disk, in milliseconds
Write time	Total amount of time that has been spent in the portion of checkpoint processing where files are written to disk, in milliseconds

Table 7-11 Background writer field description

Field	Description
Maxwritten clean	Number of times the background writer stopped a cleaning scan because it had written too many buffers
Buffers clean	Number of buffers written by the background writer

Table 7-12 Backends field description

Field	Description
Buffers backend fsync	Number of times a backend had to execute its own fsync call (normally the background writer handles those even when the backend does its own write)
Buffers backend	Number of buffers written directly by a backend

Database Objects

Figure 7-6 Database Objects metrics

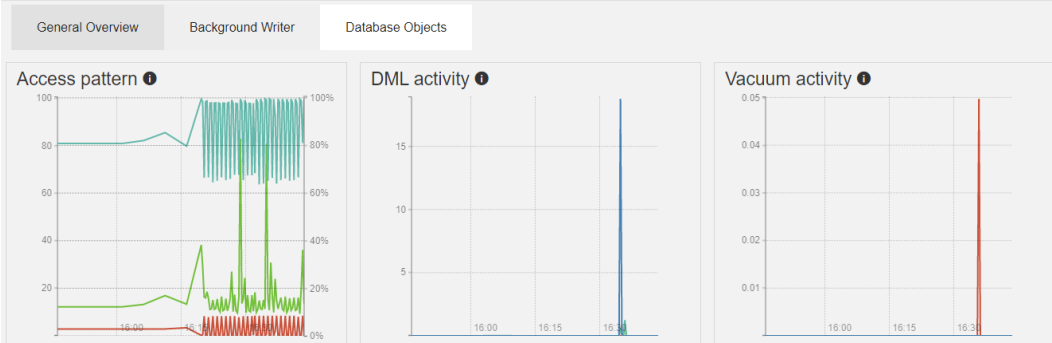


Table 7-13 Access pattern field description

Field	Description
Index scans ratio	Ratio of index scans to sequential scans
Index scans	Number of index scans per second
Sequential scans	Number of sequential scans per second

Table 7-14 DML activity field description

Field	Description
Tuples inserted	Number of tuples inserted per second
Tuples updated	Number of tuples updated per second
Tuples HOT updated	Number of heap-only tuples (HOT) updated per second
Tuples deleted	Number of tuples deleted per second

Table 7-15 Vacuum activity field description

Field	Description
# Vacuum	Number of vacuums per second
# Autovacuum	Number of autovacuums per second
# Analyze	Number of analyses per second
# Autoanalyze	Number of autoanalyses per second

Details for all databases

Figure 7-7 Details for all databases metrics

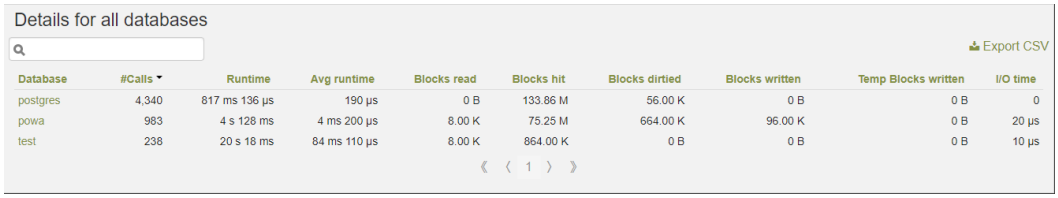


Table 7-16

Field	Description
Database	Database name
#Calls	Total number of executed SQL statements
Runtime	Total runtime of the SQL statement
Avg runtime	Average runtime of the SQL statement
Blocks read	Number of pages read from the disk
Blocks hit	Number of hit pages in the shared buffer
Blocks dirtied	Number of dirty pages
Blocks written	Number of disk write pages
Temp Blocks written	Number of disk temporary write pages
I/O time	I/O wait time

7.3 PoWA Deployment

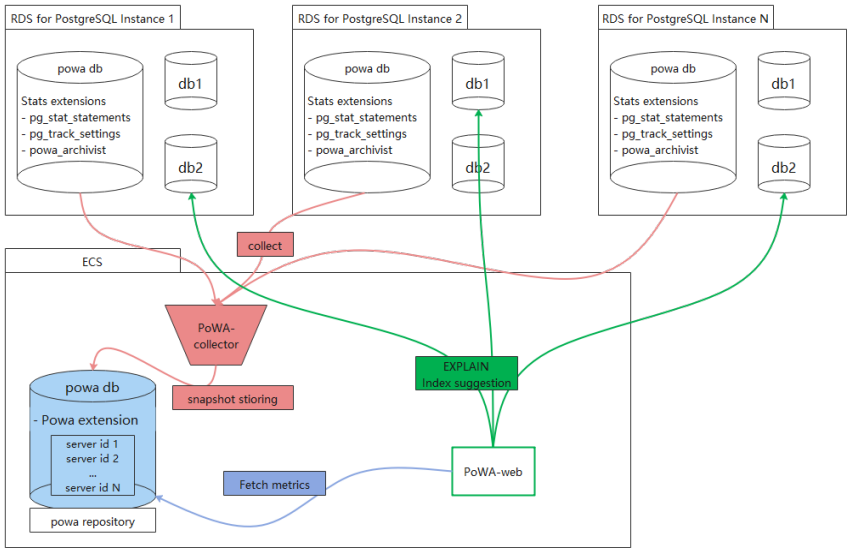
7.3.1 Deploying PoWA for an RDS for PostgreSQL Instance

To remotely deploy PoWA on Huawei Cloud, there must be an ECS with PoWA-archivist, PoWA-collector, and PoWA-web installed on it. This section describes how to install PoWA-archivist, PoWA-collector, and PoWA-web.

Architecture

The remote deployment architecture is as follows:

Figure 7-8 Remote deployment architecture



Preparations

- An RDS for PostgreSQL 12.6 instance has been created.
- An ECS has been created and associated with an EIP. In this example, the ECS uses the CentOS 8.2 64-bit image.

Installing Python3

PoWA-collector and PoWA-web must be installed in a Python3 environment. You can use pip3 to install them to facilitate the installation. In this example, Python 3.6.8 is installed on the ECS by default. The latest PoWA version fails to be installed. For details about how to install the latest version, see **Installing Python 3.9.9**.

Installing PoWA-archivist

1. Run the **wget** command to obtain the **PoWA-archivist source code**.
   ```
   wget https://github.com/powa-team/powa-archivist/archive/refs/tags/REL_4_1_2.tar.gz
   ```
2. Decompress the downloaded **REL_4_1_2.tar.gz** package.
3. Install PoWA-archivist to the decompressed directory.
   ```
   make && make install
   ```


Installing PoWA-collector and PoWA-web

1. Switch to the RDS for PostgreSQL database user. Take user **postgres** as an example.
   ```
   su - postgres
   ```
2. Install PoWA-collector and PoWA-web. psycopg2 is mandatory for installing them.
   ```
   pip install psycopg2
   pip install powa-collector
   pip install powa-web
   ```

After the installation is complete, check the following path tree. If the following information is displayed, the PoWA-collector and PoWA-web have been installed.

```
/home/postgres/.local/bin
├── powa-collector.py
├── powa-web
└── __pycache__
```

Creating the PoWA Extension

Step 1 Log in to the powa database of the RDS for PostgreSQL instance as user **root**. (If the powa database does not exist, create it first.)

Step 2 Create the PoWA extension in the powa database.

```
select control_extension('create', 'pg_stat_statements');
select control_extension('create', 'btree_gist');
select control_extension('create', 'powa');
```

----End

FAQs

Q: What should I do if the error message "python setup.py build_ext --pg-config / path/to/pg_config build" is displayed when the **pip install psycopg2** command is executed?

A: You need to add the **bin** and **lib** paths of RDS for PostgreSQL to environment variables and run the **pip install psycopg2** command.

Installing Python 3.9.9

1. Prepare the environment.

   Perform the following operations in sequence. Otherwise, Python 3.9.9 may fail to be installed (for example, SSL component dependency fails). As a result, PoWA-collector and PoWA-web fail to be installed.

   ```
   yum install readline* -y
   yum install zlib* -y
   yum install gcc-c++ -y
   yum install sqlite* -y
   yum install openssl-* -y
   yum install libffi* -y
   ```
2. Install **Python 3.9.9**.
   a. Run the following commands as user **root**:
      ```
      mkdir env
      cd env
      wget https://www.python.org/ftp/python/3.9.9/Python-3.9.9.tgz
      tar -xzvf Python-3.9.9.tgz
      ```



```
cd Python-3.9.9
./configure --prefix=/usr/local/python3.9.9
make && make install
```

b. Create a soft link.

```
ln -s /usr/local/python3.9.9/bin/python3.9 /usr/bin/python
ln -s /usr/local/python3.9.9/bin/pip3.9 /usr/bin/pip
```

3. Check whether the installation is successful.

a. Verify the installation, especially the SSL function.

```
[root@ecs-ad4d Python-3.9.9]# python
Python 3.9.9 (main, Nov 25 2021, 12:36:32)
[GCC 8.4.1 20200928 (Red Hat 8.4.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
import ssl
import urllib.request
context = ssl._create_unverified_context()
urllib.request.urlopen('https://www.example.com/',context=context).read()
```

b. If any command output is displayed, the installation is successful. Run the following command to exit:

```
quit()
```

7.3.2 Deploying PoWA on a Self-Managed PostgreSQL Instance

This section describes how to deploy PoWA on a self-managed PostgreSQL database built on an ECS.

Preparations

There is a self-managed PostgreSQL instance:

- Version: PostgreSQL 12.6
- Administrator account: **postgres**
- PostgreSQL dedicated storage database: powa-repository
- Data path: **/home/postgres/data**

Deploying PoWA

Step 1 Add **pg_stat_statements** to **shared_preload_libraries** in the **/home/postgres/data/postgresql.conf** file.

```
shared_preload_libraries = 'pg_stat_statements'         # (change requires restart)
```

Step 2 Restart the database.

```
pg_ctl restart -D /home/postgres/data/
```

Step 3 Log in to the database as the **postgres** user, create database **powa**, and install related extensions.

NOTICE

The created database must be named **powa**. Otherwise, an error is reported and certain functions do not take effect while PoWA is running.


```
[postgres@ecs-ad4d ~]$ psql -U postgres -d postgres
psql (12.6)
Type "help" for help.
postgres=# create database powa;
CREATE DATABASE
postgres=# \c powa
You are now connected to database "powa" as user "postgres".
powa=# create extension pg_stat_statements ;
CREATE EXTENSION
powa=# create extension btree_gist ;
CREATE EXTENSION
powa=# create extension powa;
CREATE EXTENSION
```

Step 4 Configure the instance whose performance metrics need to be collected.

1. Add the instance information.

```
powa=# select powa_register_server(
    hostname => '192.168.0.1',
    alias => 'myInstance',
    port => 5432,
    username => 'user1',
    password => '**********',
    frequency => 300);
     powa_register_server
    ----------------------
     t
    (1 row)
```

2. Obtain the information about the target instance from the **powa_servers** table.

```
powa=# select * from powa_servers;
id | hostname | alias | port | username |  password  | dbname | frequency | powa_coalesce | retention |
allow_ui_connection |version
----+---------------+------------+------+----------+------------+--------+-----------+---------------+-----------
+------------------
0 |           | <local>|  0 |          |           |        |        -1 |   100         | 00:00:00  | t           |
1 | 192.168.0.1 | myInstance | user1 | 5432     | ********** | powa   |       300 |           100 | 1 day       |
t            |
(2 rows)
```

NOTICE

The preceding operations involve important privacy information such as the IP address, **root** account, and plaintext password of the target instance.

Before using this extension, assess its security risks.

----End

Configuring PoWA-collector

Start the PoWA-collector.

```
cd /home/postgres/.local/bin
./powa-collector.py &
```

When PoWA-collector starts, it searches for configuration files in the following sequence:

1. /etc/powa-collector.conf
2. ~/.config/powa-collector.conf

3. ~/.powa-collector.conf
4. ./powa-collector.conf

The configuration files must contain the following options:

- **repository.dsn**: URL. It is used to notify PoWA-collector of how to connect to the dedicated storage database (powa-repository).
- **debug**: Boolean type. It specifies whether to enable PoWA-collector in debugging mode.

Take the configuration file **./powa-collector.conf** as an example.

```
{
   "repository": {
   "dsn": "postgresql://postgres@localhost:5432/powa"
   },
   "debug": true
}
```

No password is configured in the PoWA-collector configuration. Therefore, you need to set the connection policy in the **pg_hba.conf** file of the **powa-repository** database to **trust** (password-free connection).

Configuring PoWA-web

Start the PoWA-web.

```
cd /home/postgres/.local/bin
./powa-web &
```

When PoWA-web starts, it searches for configuration files in the following sequence:

1. /etc/powa-web.conf
2. ~/.config/powa-web.conf
3. ~/.powa-web.conf
4. ./powa-web.conf

Take the configuration file **./powa-web.conf** as an example.

```
# cd /home/postgres/.local/bin
# vim ./powa-web.conf
# Write the configuration information and save it.
servers={
      'main': {
        'host': 'localhost',
        'port': '5432',
        'database': 'powa',
        'username': 'postgres',
        'query': {'client_encoding': 'utf8'}
      }
}
cookie_secret="SECRET_STRING"
```

In this section, the connection policy in the **pg_hab.conf** file of the powa-repository database is set to **trust** (password-free connection). Therefore, the password is not configured.

7.4 Viewing Metric Details on PoWA

After PoWA is deployed and PoWA-collector and PoWA-web are started, you can log in to PoWA using a browser to view metric details of the monitored instances.

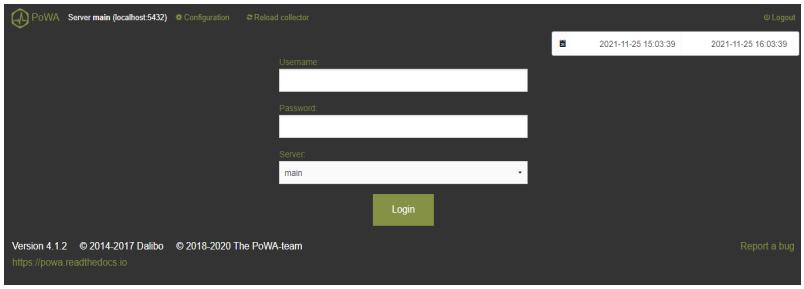
Accessing PoWA

Step 1 Use a browser to access PoWA.

NOTE

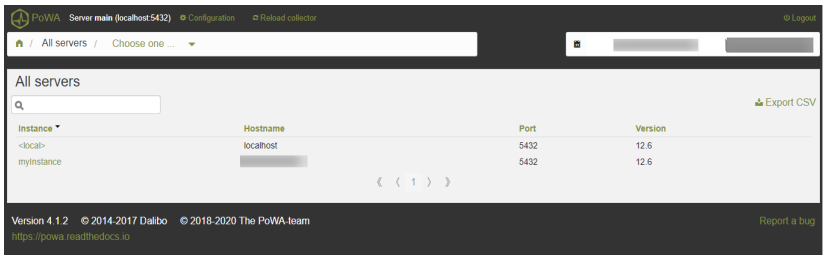
- There is no port option in the **powa-web.conf** file. The default value **8888** is used.
- URL: **http://***ECS_IP_address*:**8888/**

Figure 7-9 Accessing PoWA



Step 2 Enter the username and password, and click **Login**.

Figure 7-10 PoWA home page



In this example, PoWA collects information about two PostgreSQL instances.

- **<local>**: PostgreSQL database built on an ECS, which is used as the powa-repository database.
- **myinstance**: RDS for PostgreSQL instance, which is used as the target for performance data collection. (**myinstance** is the instance alias registered in powa-repository.)

Step 3 Click an instance to view its performance metrics.

----End

Viewing Metric Details

PoWA can collect and display various performance metrics. The following describes how to view the slow SQL metric.

Step 1 **Log in to the management console**.

Step 2 Click in the upper left corner and select a region.

Step 3 Click in the upper left corner of the page and choose **Databases** > **Relational Database Service**.

Step 4 On the **Instances** page, locate the target DB instance and click **Log In** in the **Operation** column.

Alternatively, click the instance name on the **Instances** page. On the displayed page, click **Log In** in the upper right corner of the page.

Step 5 On the displayed login page, enter the username and password and click **Log In**.

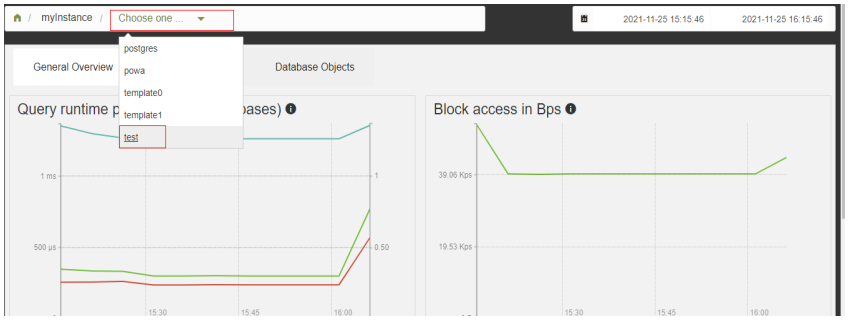
Step 6 In the database list of the **Home** page, click **Create Database**.

Step 7 On the displayed page, enter a database name (for example, **test**) and select a character set.

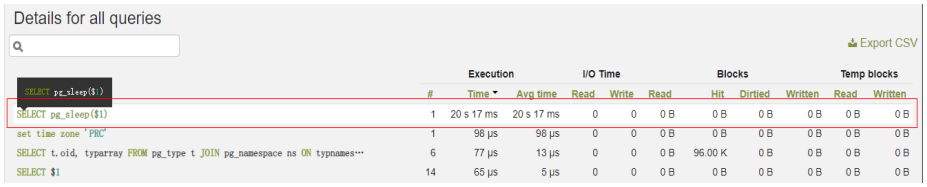
Step 8 Choose **SQL Operations** > **SQL Query** to execute a slow SQL statement, for example, **SELECT pg_sleep($1)**, in the **test** database.

Step 9 After about 5 minutes, on the PoWA home page, select the target DB instance and select the **test** database.

Figure 7-11 PoWA home page



In **Details for all queries**, check that the execution time of the **SELECT pg_sleep($1)** statement is 20s.



----End

Installing Other Extensions to Collect Performance Metrics

The following steps take pg_track_settings as an example.

Step 1 **Log in to the management console**.

Step 2 Click in the upper left corner and select a region.

Step 3 Click in the upper left corner of the page and choose **Databases** > **Relational Database Service**.

Step 4 On the **Instances** page, locate the target DB instance and click **Log In** in the **Operation** column.

Alternatively, click the instance name on the **Instances** page. On the displayed page, click **Log In** in the upper right corner of the page.

Step 5 On the displayed login page, enter the username and password and click **Log In**.

Step 6 Select the **powa** database and run the following SQL command to create pg_track_settings:

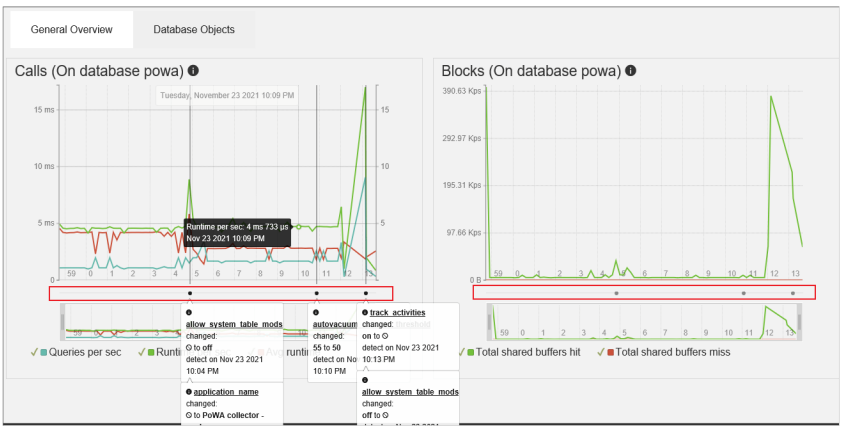
```
select control_extension('create', 'pg_track_settings');
```

Step 7 Create a PostgreSQL database (powa-repository) on the ECS, and install and activate **pg_track_settings** to collect performance metrics.

```
# pg_track_settings
cd /home/postgres/env
wget https://github.com/rjuju/pg_track_settings/archive/refs/tags/2.0.1.tar.gz
mv 2.0.1.tar.gz pg_track_settings.2.0.1.tar.gz
tar -xzvf pg_track_settings.2.0.1.tar.gz
cd pg_track_settings-2.0.1
make && make install
# powa-repository
psql -d powa
powa=# create extension pg_track_settings ;
CREATE EXTENSION
# Activate pg_track_settings for the target instance.
dbpowa=# select powa_activate_extension(1, 'pg_track_settings');
powa_activate_extension
-------------------------
t
(1 row)
```

Step 8 Verify the pg_track_settings extension.

Change the value of the **autovacuum_analyze_threshold** parameter on the target instance to **55**. The default value is **50**. After about 5 minutes, you can view the modification record on the PoWA page, as shown in the following figure:



The contents in the three boxes in the preceding figure include:

- The time when pg_track_settings was activated and the database parameter value at that time.
- The time when the **autovacuum_analyze_threshold** parameter was modified, its original value, and new value.
- The time when pg_track_settings was canceled and the database parameter value at that time.

----End

8 Best Practices for Using pg_dump

Description

pg_dump is a native tool for backing up a PostgreSQL database. The file created by pg_dump can be a SQL script file or an archive file. For details, see **pg_dump**.

- SQL script file: It is a plain-text file that contains the SQL commands required to rebuild a database to the state when it was backed up.
- Archive file: It must be used with pg_restore to rebuild a database. This format allows pg_restore to select the data to be restored.

Precautions

pg_dump dumps a single database, schemas, or tables. Only tables, data, and functions can be exported. Before restoration, you need to create a database and account in the target instance in advance.

- **--format=custom**: The dump file is in binary format, which can be used only by pg_restore. You can restore specific tables from a dump file.
- **--format=plain**: The dump file is in plain-text format. To restore from such a plain-text file, connect to the database and execute the file.

Constraints

Before using pg_dump and pg_restore, ensure that the versions of the source and target databases are the same to avoid compatibility issues. Incompatible versions may cause data loss or restoration errors.

Preparing Test Data

```
# Create a database.
create database dump_database;

# Log in to the database.
\c dump_database

# Create table 1 and insert data into the table.
create table dump_table(id int primary key, content char(50));
insert into dump_table values(1,'aa');
insert into dump_table values(2,'bb');
```



```
# Create table 2 and insert data into the table.
create table dump_table2(id int primary key, content char(50));
insert into dump_table2 values(1,'aaaa');
insert into dump_table2 values(2,'bbbb');
```

Using pg_dump to Export a Database to a SQL File

Syntax

```
pg_dump --username=<DB_USER> --host=<DB_IPADDRESS> --port=<DB_PORT> --format=plain --
file=<BACKUP_FILE><DB_NAME>
```

- *DB_USER* indicates the database username.
- *DB_IPADDRESS* indicates the database address.
- *DB_PORT* indicates the database port.
- *BACKUP_FILE* indicates the name of the file to be exported.
- *DB_NAME* indicates the name of the database to be exported.
- **--format** indicates the format of the exported file. **plain** (default) indicates a plain-text file that contains SQL scripts. For details about other options, see **pg_dump**.

Examples

- Export a database to a SQL file (INSERT statements).
  ```
  $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --inserts --
  file=backup.sql dump_database
  Password for user root:
  ```
- Export all table schemas from a database to a SQL file.
  ```
  $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --schema-only --
  file=backup.sql dump_database
  Password for user root:
  ```
- Export all table data from a database to a SQL file.
  ```
  $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --data-only --
  file=backup.sql dump_database
  Password for user root:
  ```

After the commands in any of the above examples are executed, a **backup.sql** file will be generated as follows:

```
[rds@localhost ~]$ ll backup.sql
-rw-r----- 1 rds rds 5657 May 24 09:21 backup.sql
```

Using pg_dump to Export Specified Tables from a Database to a SQL File

Syntax

```
pg_dump --username=<DB_USER> --host=<DB_ADDRESS> --port=<DB_PORT> --format=plain --
file=<BACKUP_FILE> <DB_NAME> --table=<TABLE_NAME>
```

- *DB_USER* indicates the database username.
- *DB_ADDRESS* indicates the database address.
- *DB_PORT* indicates the database port.
- *BACKUP_FILE* indicates the name of the file to be exported.
- **DB_NAME** indicates the name of the database to be migrated.
- *TABLE_NAME* indicates the name of the specified table in the database to be migrated.

- **--format** indicates the format of the exported file. **plain** (default) indicates a plain-text file that contains SQL scripts. For details about other options, see **pg_dump**.

Examples

- Export a single table from a database to a SQL file.
```
 $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --file=backup.sql dump_database --table=dump_table
 Password for user root
```
- Export multiple tables from a database to a SQL file.
```
 $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --file=backup.sql dump_database --table=dump_table --table=dump_table2
 Password for user root:
```
- Export all tables starting with **ts**_ from a database to a SQL file.
```
 $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --file=backup.sql dump_database --table=ts_*
 Password for user root:
```
- Export all tables excluding those starting with **ts**_ from a database to a SQL file.
```
 $ pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --file=backup.sql dump_database -T=ts_*
 Password for user root:
```

After the commands in any of the above examples are executed, a **backup.sql** file will be generated as follows:

```
[rds@localhost ~]$ ll backup.sql
-rw-r----- 1 rds rds 5657 May 24 09:21 backup.sql
```

Using pg_dump to Export Data of a Specific Schema

Syntax

```
pg_dump --username=<DB_USER> --host=<DB_IPADDRESS> --port=<DB_PORT> --format=plain --schema=<SCHEMA> <DB_NAME> --table=<BACKUP_FILE>
```

- *DB_USER* indicates the database username.
- *DB_IPADDRESS* indicates the database address.
- *DB_PORT* indicates the database port.
- *BACKUP_FILE* indicates the name of the file to be exported.
- *DB_NAME* indicates the name of the database to be exported.
- *SCHEMA* indicates the name of the schema to be exported.
- **--format** indicates the format of the exported file. **plain** (default) indicates a plain-text file that contains SQL scripts. For details about other options, see **pg_dump**.

Examples

- Export all data of the public schema from a database.
```
pg_dump --username=root --host=192.168.61.143 --port=5432 --format=plain --schema=public dump_database  > backup.sql
```
- Export all data except the public schema from a database in a customized compression format.
```
pg_dump --username=root --host=192.168.61.143 --port=5432 --format=custom -b -v -N public dump_database > all_sch_except_pub.backup
```


Restoring Data

To restore from a plain-text SQL script file, run the **psql** command. See the following example commands:

```
# Restore a specific database.
psql --username=root --host=192.168.61.143 --port=5432  backup_database  < backup.sql

# Restore a specific table.
psql --username=root --host=192.168.61.143 --port=5432  backup_database --
table=dump_table  < backup.sql

# Restore a specific schema.
psql --username=root --host=192.168.61.143 --port=5432  backup_database --schema=public  <
backup.sql
```

NOTE

Before the restoration, create a database named **backup_database** in the target database.

To restore from other file formats, use pg_restore. **pg_restore** is used to restore a PostgreSQL database in any non-plain-text format dumped by pg_dump.

```
pg_restore --username=root --host=192.168.61.143 --port=5432 --dbname=backup_database --
format=custom all_sch_except_pub.backup --verbose
```

FAQs

1. What should I do if an error about insufficient permissions is reported for pg_dump?

   Solution:

   Check whether the **root** user is used to export data. If any other user account is used, an error about insufficient permissions will be reported. If the **root** user is used and an error is still reported, check the database version. You can run pg_dump commands as the **root** user only when the kernel version support **root** privilege escalation. For details about the kernel versions that support **root** privilege escalation, see **Privileges of the root User**.

2. Why an error was reported for functions such as control_extension after I imported a dump file to the target RDS for PostgreSQL database?

   Solution:

   That's because the target database contains these functions. This error can be ignored.

9 Best Practices for Using PgBouncer

Introduction to PgBouncer

PgBouncer is a lightweight connection pooler for PostgreSQL. It can:

- Cache connections to PostgreSQL. When a connection request is received, an idle process is allocated. PostgreSQL does not need to fork a new process to establish a connection. No resources need to be used for creating a new process and establishing a connection.
- Improve the connection usage and prevent excessive invalid connections from consuming too many database resources and causing high CPU usage.
- Restrict client connections to prevent excessive or malicious connection requests.

It is lightweight because:

- It uses libevent for socket communication, improving the communication efficiency.
- It uses C language and only 2 KB of memory is consumed by each connection.

PgBouncer supports the following types of connection pooling:

- Session pooling: PgBouncer does not reclaim the allocated connection until the client session ends.
- Transaction pooling: PgBouncer reclaims the allocated connection after the transaction is complete. The client only has exclusive access to a connection during a transaction. Non-transaction requests do not have exclusive connections.
- Statement pooling: PgBouncer reclaims the connection anytime a database request completes. In this mode, the client cannot use transactions. Using transactions in this case will cause data inconsistency.

The default pooling type for PgBouncer is session. You are advised to change it to transaction.

Installation and Configuration

Before deploying PgBouncer on the cloud, **purchase an ECS**. To reduce network latency, you are advised to select the same VPC and subnet as those of the

backend RDS instance for the ECS. After the purchase is complete, log in to the ECS to set up the environment.

1. PgBouncer is based on libevent, so you need to install the **libevent-devel** and **openssl-devel** dependencies.
```
yum install -y libevent-devel
yum install -y openssl-devel
```
2. After that, download the source code from the PgBouncer official website and compile the code and install PgBouncer as a regular user.
```
su - pgbouncer
tar -zxvf pgbouncer-1.19.0.tar.gz
cd  pgbouncer-1.19.0
./configure --prefix=/usr/local
make
make install
```
3. Create the following directories to store the files (such as logs and process IDs) generated by PgBouncer:
```
mkdir -p /etc/pgbouncer/
mkdir -p /var/log/pgbouncer/
mkdir -p /var/run/pgbouncer/
```
4. Before starting PgBouncer, build the configuration file **pgbouncer.ini**.
```
[databases]
* = host=127.0.0.1 port=5432
[pgbouncer]
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid
listen_addr = *
listen_port = 6432
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
admin_users = postgres
stats_users = stats, postgres
pool_mode = transaction
server_reset_query = DISCARD ALL
max_client_conn = 100
default_pool_size = 20
;; resolve: unsupported startup parameter: extra_float_digits
;;ignore_startup_parameters = extra_float_digits
```
   For details about the parameters in the configuration file, see the **official PgBouncer documentation**.

Starting PgBouncer

PgBouncer cannot be started as **root**. It has to be started as a regular user.

```
pgbouncer -d /etc/pgbouncer/pgbouncer.ini
```

After it is started, run **netstat -tunlp | grep pgbouncer** to check the listening port of the connection pool and then connect to the DB instance.

```
psql -U root -d postgres -h 127.0.0.1 -p 6432
Password for user root:
psql (12.13)
Type "help" for help.
postgres=> \l
                                  List of databases
   Name    |   Owner   | Encoding |   Collate   |    Ctype    |    Access privileges
-----------+-----------+----------+-------------+-------------+-------------------------
 postgres  | pgbouncer | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | pgbouncer | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/pgbouncer
           |           |          |             |             | pgbouncer=CTc/pgbouncer
 template1 | pgbouncer | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/pgbouncer
```


Stopping PgBouncer

You can run the **kill** command to stop it.

```
kill `cat /var/run/pgbouncer/pgbouncer.pid`
cat /var/run/pgbouncer/pgbouncer.pid | xargs kill -9
```

PgBouncer Management

PgBouncer provides a virtual database **pgbouncer**, which provides a database operation interface like PostgreSQL. It is not a real database, but a command line interface virtualized by PgBouncer. To log in to this virtual database, run the following command:

```
psql -p 6432 -d pgbouncer
```

If some configuration parameters are modified, you do not need to restart PgBouncer but run **reload** for the modifications to be applied.

```
pgbouncer=# reload;
RELOAD
```

After login, you can run **show help** to check the command help, run **show clients** to check the client connection information, and run **show pools** to check the connection pool information.

An Example for Read/Write Splitting

PgBouncer cannot automatically parse or split read and write requests. Read and write requests need to be distinguished on the application side.

1. Modify the database information in the **pgbouncer.ini** file and add the connection configurations of the primary instance and read replica to the file. In this example, the parameters are set as follows:

   ```
   [databases]
   ;; * = host=127.0.0.1 port=5432
   # The connection information of the read replica.
   mydb_read: host=10.7.131.69 port=5432 dbname=postgres user=root password=***
   # The connection information of the primary instance.
   mydb_write:  host=10.8.115.171 port=5432 dbname=postgres user=root password=***
   [pgbouncer]
   logfile = /var/log/pgbouncer/pgbouncer.log
   pidfile = /var/run/pgbouncer/pgbouncer.pid
   listen_addr = *
   listen_port = 6432
   auth_type = md5
   auth_file = /etc/pgbouncer/userlist.txt
   admin_users = postgres
   stats_users = stats, postgres
   pool_mode = transaction
   server_reset_query = DISCARD ALL
   max_client_conn = 100
   default_pool_size = 20
   ;; resolve: unsupported startup parameter: extra_float_digits
   ;;ignore_startup_parameters = extra_float_digits
   ```

2. Check whether the primary instance and read replica can be connected. The primary instance and read replica have been connected using psql and read/write splitting is supported.

   ```
   psql -U root -d mydb_write -h 127.0.0.1 -p 6432
   Password for user root:
   psql  (14.6)
   mydb_write=>  SELECT pg_is_in_recovery();
   ```



```
 pg_is_in_recovery
-----------------------
 f
(1 row)
psql -U root -d mydb_read -h 127.0.0.1 -p 6432
Password for user root:
psql  (14.6)
mydb_read=>  SELECT pg_is_in_recovery();
pg_is_in_recovery
-----------------------
 t
(1 row)
```


10 Database Naming Rules

- The keyword length of RDS for PostgreSQL is limited to 63 bytes. Therefore, it is recommended that the length of a database name be no more than 30 characters.
- A database name can contain only lowercase letters, underscores (_), and digits. Do not use reserved keywords in database names or begin a database name with pg, digits, or underscores (_). For details about reserved keywords, see the **official documentation**.

11 RDS for PostgreSQL Table Design

RDS for PostgreSQL excels in processing complex online transaction processing (OLTP) transactions and supports NoSQL (JSON, XML, or hstore) and geographic information system (GIS) data types. It has earned a reputation for reliability and data integrity, and is widely used for websites, location-based applications, and complex data object processing.

This topic uses an e-commerce platform as an example to describe how to design table structures, and how to create databases and tables and process application requests.

Scenarios

An e-commerce platform requires database table structures matching its core workloads (online shopping, order management, and product statistics analysis) to ensure the consistency and integrity of mission-critical business data and speed up transaction processing when there are many concurrent requests.

Table Design

- User table (**t_customers**)

  This table stores user information. Each user has a record in the table, and each user has a unique user ID (**cust_id**).

  Table 11-1 User table

Field Name	Field Type	Description
cust_id	SERIAL	User ID, which is the primary key.
cust_nickname	VARCHAR(50)	User nickname, with the NOT NULL constraint added.
cust_gender	VARCHAR(10)	User gender.
cust_birthday	DATE	User birthday.

Field Name	Field Type	Description
cust_address	TEXT	User shipping address, with the NOT NULL constraint added.

- Offering table (**t_goods**)

  This table stores offering information. Each offering has a record in the table, and each offering has a unique offering ID (**item_id**).

  Table 11-2 Offering table

Field Name	Field Type	Description
item_id	SERIAL	Offering ID, which is the primary key.
item_name	VARCHAR(100)	Offering name, with the NOT NULL constraint added.
item_category	VARCHAR(50)	Offering category, with the NOT NULL constraint added.
item_desc	TEXT	Offering description.
item_price	DECIMAL(10,2)	Offering unit price, with the NOT NULL constraint added.
stock_quantity	INTEGER	Stock quantity, with the NOT NULL constraint added.

- Order table (**t_orders**)

  This table stores order information, which is used to associate users and offerings. Each order has a record in this table, and each order has a unique order ID (**trans_id**).

  Table 11-3 Order table

Field Name	Field Type	Description
trans_id	SERIAL	Order ID, which is the primary key.
cust_id	INTEGER	Order user ID, which is associated with the user ID (**cust_id**) in the user table (**t_customers**).

Field Name	Field Type	Description
item_id	INTEGER	Order offering ID, which is associated with the offering ID (**item_id**) in the offering table (**t_goods**).
purchase_quantity	INTEGER	Offering purchase quantity, with the NOT NULL constraint added.
total_price	DECIMAL(10,2)	Offering total price, with the NOT NULL constraint added.
order_time	TIMESTAMP	Order generation time, with the DEFAULT constraint added.
order_status	VARCHAR(20)	Order status, with the NOT NULL constraint added.

Creating Tables

1. Create tables in a database.
   - Create the **t_customers** table.
     ```
     CREATE TABLE t_customers (
       cust_id SERIAL PRIMARY KEY,
       cust_nickname VARCHAR(50) NOT NULL,
       cust_gender VARCHAR(10),
       cust_birthday DATE,
       cust_address TEXT NOT NULL
     );
     ```
   - Create the **t_goods** table.
     ```
     CREATE TABLE t_goods(
       item_id SERIAL PRIMARY KEY,
       item_name VARCHAR(100) NOT NULL,
       item_category VARCHAR(50) NOT NULL,
       item_desc TEXT,
       item_price DECIMAL(10,2) NOT NULL,
       stock_quantity INTEGER NOT NULL
     );
     ```
   - Create the **t_orders** table.
     ```
     CREATE TABLE t_orders (
       trans_id SERIAL PRIMARY KEY,
       cust_id INTEGER REFERENCES t_customers(cust_id),
       item_id INTEGER REFERENCES t_goods(item_id),
       purchase_quantity INTEGER NOT NULL,
       total_price DECIMAL(10,2) NOT NULL,
       order_time TIMESTAMP NOT NULL DEFAULT NOW(),
       order_status VARCHAR(20) NOT NULL
     );
     ```
2. Assume that some offerings are released in the mall, users have registered accounts on the platform and purchased some offerings, and the data has

been sent back to the system database. Insert the following test data into the database:

```
-- User data
INSERT INTO t_customers (cust_nickname, cust_gender, cust_birthday, cust_address) VALUES
  ('Rich Man', 'Female', '1995-08-12', 'Haidian District, Beijing'),
  ('Superman', 'Male', '1998-03-25', 'Pudong New Area, Shanghai');
-- Offering data
INSERT INTO t_goods(item_name, item_category, item_desc, item_price, stock_quantity) VALUES
  ('Smartphone X', 'Electronics', 'Latest Smartphone', 5999.00, 22),
  ('Wireless Headset Pro', 'Electronics', 'Hot Products', 1299.00, 200),
('Cotton T-shirt', 'Clothing', 'Pure Cotton', 199.00, 300);
-- Order data
INSERT INTO t_orders(cust_id, item_id, purchase_quantity, total_price, order_status) VALUES
  (1, 1, 1, 5999.00, 'Paid'),
  (1, 2, 2, 2598.00, 'Shipped'),
  (2, 3, 5, 995.00, 'Completed');
```

Examples

- User **Superman** queries all of his orders (**cust_id** is **2**).

  ```
  SELECT m.item_name, t.purchase_quantity, t.total_price, t.order_time, t.order_status
  FROM t_customers c
  JOIN t_orders t ON c.cust_id = t.cust_id
  JOIN t_goods m ON t.item_id = m.item_id
  WHERE c.cust_id = 2;
  ```

  The command output is as follows:

  ```
  item_name | purchase_quantity | total_price |         order_time         | order_status
  ------------+-------------------+-------------+----------------------------+--------------
   Cotton T-shirt   |                 5 |      995.00 | 2025-07-31 15:04:03.593379 | Completed
  (1 row)
  ```

- The platform collects statistics on the sales volume of each offering.

  ```
  SELECT m.item_name, SUM(t.purchase_quantity) AS total_sold
  FROM t_goods m
  LEFT JOIN t_orders t ON m.item_id = t.item_id
  GROUP BY m.item_name
  ORDER BY total_sold DESC;
  ```

  The command output is as follows:

  ```
  item_name  | total_sold
  -------------+------------
   Cotton T-shirt     |          5
   Wireless headset Pro |          2
  Smartphone X   |          1
  (3 rows)
  ```

- Query electronics whose stock quantity is less than 50.

  ```
  SELECT item_name, stock_quantity
  FROM t_goods
  WHERE stock_quantity < 50
  AND item_category = 'Electronics';
  ```

  The command output is as follows:

  ```
  item_name | stock_quantity
  ------------+----------------
   Smartphone X |             22
  (1 row)
  ```


12 RDS for PostgreSQL Permissions Management

Basic Concepts

PostgreSQL manages user permissions using two concepts, role and privilege.

- Role

  A role is a collection of privileges. It can be thought of as either a user or a user group.

  - A login role is a role that has the LOGIN privilege.
  - A group role does not have the LOGIN privilege. It is used to manage privileges. Other roles can be added to the group to inherit privileges.
  - By default, all roles belong to a role named **public** (a special group role). Pay attention to its default privileges.

- Privilege

  Privileges are permissions to perform operations for database objects (such as tables, functions, and schemas). Common privileges include:

  - Table/View: SELECT (query), INSERT, UPDATE, DELETE, and TRUNCATE (clear)
  - Database: CONNECT and CREATE (create schema)
  - Schema: CREATE (create object) and USAGE (access object)
  - Function/Stored procedure: EXECUTE
  - Sequence: USAGE (use) and UPDATE (modify)

Principles

RDS for PostgreSQL uses roles to manage user permissions. A role by itself does not have the LOGIN privilege. You can create a user and assign a Login role to the user. Then the user can log in to databases and inherit the privileges the role has. If the role's privileges change, the user's privileges change as well.

Suggestions

- A privileged account **root** is provided for your RDS for PostgreSQL instance by default. This account has all privileges on your instance and can be used only by senior database administrators (DBAs).
- The project manager can create a resource account to manage roles and can also create multiple roles to enable fine-grained permissions management.
- You can create user accounts. These accounts can be used to log in to and perform operations on databases.
- If your project (*prj*) has multiple schemas, you are advised to divide role privileges by schema, for example, *{prj}_{role}_{schema}*_readonly and *{prj}_{role}_{schema}*_write. Do not put tables in the **public** schema. By default, all users have the CREATE and USAGE privileges for this schema.

Example of Privilege Design

- Senior DBAs have the privileged account **root** for RDS for PostgreSQL instances.
- The project manager has a resource account **db_prj_owner**. It is used to manage accounts and roles.
- The project name is **db_prj**, and the new schemas are **db_prj**, **db_prj_1**, and **db_prj2**.

The following table describes the privileges the resource account and roles have.

Table 12-1 Privileges

User/Role	Privileges for Tables in the Schemas	Privileges for Stored Procedures in the Schemas
root is a privileged account. This account is created by default after an instance is created.	• DDL: CREATE, DROP, and ALTER* • DQL: SELECT* • DML: UPDATE, INSERT, and DELETE	• DDL: CREATE, DROP, and ALTER* • DQL: SELECT and the privilege for calling stored procedures
db_prj_owner is the only resource account of a project.	• DDL: CREATE, DROP, and ALTER* • DQL: SELECT* • DML: UPDATE, INSERT, and DELETE	• DDL: CREATE, DROP, and ALTER* • DQL: SELECT and the privilege for calling stored procedures
db_prj_role1_readwrite (role)	• DQL: SELECT* • DML: UPDATE, INSERT, and DELETE	DQL: SELECT and the privilege for calling stored procedures. If a stored procedure contains DDL statements, a permission error is reported.

User/Role	Privileges for Tables in the Schemas	Privileges for Stored Procedures in the Schemas
db_prj_role2_readonly (role)	DQL: SELECT	DQL: SELECT and the privilege for calling stored procedures. If a stored procedure contains DDL or DML statements, a permission error is reported.

Procedure

1. Create resource account **db_prj_owner** and roles for your project.

   DBAs use the privileged account **root** to perform the following operations.

   ```
   ---db_prj_owner is the username of the owner. The password is only an example. Change it as required.
   CREATE USER db_prj_owner WITH LOGIN PASSWORD 'XXXXXXX';

   CREATE ROLE db_prj_role1_readwrite;
   CREATE ROLE db_prj_role2_readonly;

   ---Grant the DQL SELECT privilege and DML UPDATE, INSERT, and DELETE privileges on tables created by db_prj_owner to the role db_prj_role1_readwrite.
   ALTER DEFAULT PRIVILEGES FOR ROLE db_prj_owner GRANT ALL ON TABLES TO db_prj_role1_readwrite;

   ---Grant the DQL SELECT privilege and DML UPDATE, INSERT, and DELETE privileges on sequences created by db_prj_owner to the role db_prj_role1_readwrite.
   ALTER DEFAULT PRIVILEGES FOR ROLE db_prj_owner GRANT ALL ON SEQUENCES TO db_prj_role1_readwrite;

   ---Grant the DQL SELECT privilege on tables created by db_prj_owner to the role db_prj_role2_readonly.
   ALTER DEFAULT PRIVILEGES FOR ROLE db_prj_owner GRANT SELECT ON TABLES TO db_prj_role2_readonly;
   ```

2. Create users **db_prj_user_readwrite** and **db_prj_user_readonly**.

   DBAs use the privileged account **root** to perform the following operations.

   ```
   ---Grant the DQL SELECT privilege and DML UPDATE, INSERT, and DELETE privileges to the user db_prj_user_readwrite.
   CREATE USER db_prj_user_readwrite WITH LOGIN PASSWORD 'XXXXXXX';
   GRANT db_prj_role1_readwrite TO db_prj_user_readwrite;

   ---Grant the DQL SELECT privilege to the user db_prj_user_readonly.
   CREATE USER db_prj_user_readonly WITH LOGIN PASSWORD 'XXXXXXXX';
   GRANT db_prj_role2_readonly TO db_prj_user_readonly;
   ```

3. Create a schema and grant privileges on the schema to the project roles.

   DBAs use the privileged account **root** to perform the following operations.

   ```
   ---Specify db_prj_owner as the owner of the db_prj schema.
   CREATE SCHEMA db_prj  AUTHORIZATION db_prj_owner;

   ---Grant the privileges on this schema to the project roles.
   GRANT USAGE ON SCHEMA db_prj TO db_prj_role1_readwrite;
   GRANT USAGE ON SCHEMA db_prj TO db_prj_role2_readonly;
   ```


> **NOTE**
>
> The users **db_prj_user_readwrite** and **db_prj_user_readonly** inherit the privilege changes of their associated roles. You do not need to grant privileges to the users.

Scenarios

Scenario 1: Using the account **db_prj_owner** to perform DDL operations on tables in the **db_prj** schema

```
CREATE TABLE db_prj.table1(col1 bigserial primary key, col2 int);
DROP TABLE db_prj.table1;
```

Scenario 2: Using the user **db_prj_user_readwrite** or **db_prj_user_readonly** for workload development

The user that you use for workload development follows the principle of least privilege. Use the **db_prj_user_readonly** user if existing data does not need to be changed. Use the **db_prj_user_readwrite** user when you need to perform DML operations (INSERT, UPDATE, and DELETE) to change data. Data operations of different user accounts are isolated, enhancing security.

- Use **db_prj_user_readwrite** to add, delete, query, and modify data of tables in the **db_prj** schema.

  ```
  # Common DML and DQL operations are not affected.
  INSERT INTO db_prj.table1 (col2) VALUES(88),(99);
  SELECT * FROM db_prj.table1;

  ---The user db_prj_user_readwrite does not have the DDL CREATE, DROP, or ALTER privilege.
  CREATE TABLE db_prj.table2(id int);
  ERROR:  permission denied for schema db_prj
  LINE 1: create table db_prj.table2(id int);

  DROP TABLE db_prj.table1;
  ERROR:  must be owner of table test

  ALTER TABLE db_prj.table1 ADD col3 int;
  ERROR:  must be owner of table test

  CREATE INDEX idx_xxxx on db_prj.table1(col1);
  ERROR:  must be owner of table test
  ```

- Use **db_prj_user_readonly** to perform operations on tables in the **db_prj** schema.

  ```
  INSERT INTO db_prj.table1 (col2) VALUES(88),(99);
  ERROR:  permission denied for table table1

  SELECT id,name FROM db_prj.table1 limit 1;
   col1 | col2
  ----+-------
    1 | 88
  (1 row)
  ```

Scenario 3: Granting privileges of a project to a user belonging to another project

Assume that there is another project **db_prj1** and you need to grant the read privilege on tables of project **db_prj** to the user **db_prj1_user_readwrite** that belongs to project **db_prj1**. DBAs use the privileged account **root** to perform the following operation.

```
---Grant the privileges of user db_prj_role2_readonly to the user db_prj1_user_readwrite.
GRANT db_prj_role2_readonly TO db_prj1_user_readwrite;
```

Scenario 4: Creating a schema **db_prj1** and granting privileges on this schema to project roles

The users **db_prj1_user_readwrite**, **db_prj1_user_readonly**, and **db_prj1_user_readwrite** inherit the privilege changes of their associated roles. You do not need to grant privileges to these users. DBAs use the privileged account **root** to perform the following operations.

```
CREATE SCHEMA db_prj1 AUTHORIZATION db_prj_owner;

---Grant the access privilege on the schema to roles.
---Grant the DDL CREATE, DROP, and ALTER privileges on tables in schema db_prj1 to the account db_prj_owner.
GRANT USAGE ON SCHEMA db_prj1 TO db_prj_role1_readwrite;
GRANT USAGE ON SCHEMA db_prj1 TO db_prj_role2_readonly;
```

Querying Users and Roles

- Use a command-line tool to connect to your RDS for PostgreSQL instance. Run the **\du** command to query all users and roles. The following is an example.

  Figure 12-1 Running the **\du** command

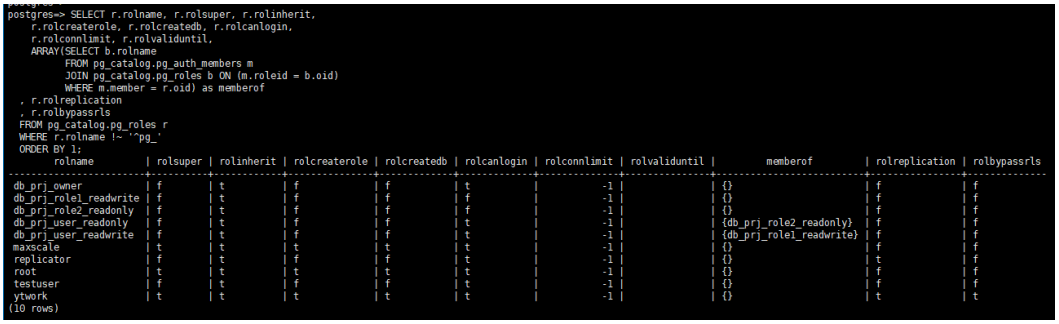
  ```
  postgres=> \du
                                     List of roles
           Role name          |                   Attributes                   |                          Member of
  ----------------------------+------------------------------------------------+-------------------------------------------------------------
   db_prj_owner               |                                                | {}
   db_prj_role1_readwrite     | Cannot login                                   | {}
   db_prj_role2_readonly      | Cannot login                                   | {}
   db_prj_user_readonly       |                                                | {db_prj_role2_readonly}
   db_prj_user_readwrite      |                                                | {db_prj_role1_readwrite}
   maxscale                   | Superuser, Create role, Create DB              | {}
   pg_checkpoint              | Cannot login                                   | {}
   pg_create_subscription     | Cannot login                                   | {}
   pg_database_owner          | Cannot login                                   | {}
   pg_execute_server_program  | Cannot login                                   | {}
   pg_monitor                 | Cannot login                                   | {pg_read_all_settings,pg_read_all_stats,pg_stat_scan_tables}
   pg_read_all_data           | Cannot login                                   | {}
   pg_read_all_settings       | Cannot login                                   | {}
   pg_read_all_stats          | Cannot login                                   | {}
   pg_read_server_files       | Cannot login                                   | {}
   pg_signal_backend          | Cannot login                                   | {}
   pg_stat_scan_tables        | Cannot login                                   | {}
   pg_use_reserved_connections | Cannot login                                   | {}
   pg_write_all_data          | Cannot login                                   | {}
   pg_write_server_files      | Cannot login                                   | {}
   replicator                 | Replication                                    | {}
   root                       | Superuser, Create role, Create DB              | {}
   testuser                   |                                                | {}
  ```

  The command output shows that **db_prj_role2_readonly, db_prj1_role1_readwrite** is displayed in the **Member of** column of the user **db_prj_user_readwrite**.

- Run SQL statements to query users and roles.

  ```
  SELECT r.rolname, r.rolsuper, r.rolinherit,
    r.rolcreaterole, r.rolcreatedb, r.rolcanlogin,
    r.rolconnlimit, r.rolvaliduntil,
    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
  , r.rolreplication
  , r.rolbypassrls
  FROM pg_catalog.pg_roles r
  WHERE r.rolname !~ '^pg_'
  ORDER BY 1;
  ```


Figure 12-2 Running SQL statements



```
postgres=> SELECT r.rolname, r.rolsuper, r.rolinherit,
    r.rolcreaterole, r.rolcreatedb, r.rolcanlogin,
    r.rolconnlimit, r.rolvaliduntil,
    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
  , r.rolreplication
  , r.rolbypassrls
  FROM pg_catalog.pg_roles r
  WHERE r.rolname !~ '^pg_'
  ORDER BY 1;
        rolname         | rolsuper | rolinherit | rolcreaterole | rolcreatedb | rolcanlogin | rolconnlimit | rolvaliduntil |        memberof         | rolreplication | rolbypassrls
------------------------+----------+------------+---------------+-------------+-------------+--------------+---------------+-------------------------+----------------+--------------
 db_prj_owner           | f        | t          | f             | f           | t           |           -1 |               | {}                      | f              | f
 db_prj_role1_readwrite | f        | t          | f             | f           | f           |           -1 |               | {}                      | f              | f
 db_prj_role2_readonly  | f        | t          | f             | f           | f           |           -1 |               | {}                      | f              | f
 db_prj_user_readonly   | f        | t          | f             | f           | t           |           -1 |               | {db_prj_role2_readonly} | f              | f
 db_prj_user_readwrite  | f        | t          | f             | f           | t           |           -1 |               | {db_prj_role1_readwrite} | f              | f
 maxscale               | t        | t          | t             | t           | t           |           -1 |               | {}                      | f              | f
 replicator             | f        | t          | f             | f           | t           |           -1 |               | {}                      | t              | f
 root                   | t        | t          | t             | t           | t           |           -1 |               | {}                      | f              | f
 testuser               | f        | t          | f             | f           | t           |           -1 |               | {}                      | f              | f
 ytwork                 | t        | t          | t             | t           | t           |           -1 |               | {}                      | t              | t
(10 rows)
```


13 Troubleshooting WAL Accumulation

Description

PostgreSQL uses Write-Ahead Logging (WAL) to ensure data durability and consistency. WAL logs record all changes to databases and play a key role in primary/standby replication, crash recovery, and logical replication.

You need to pay attention to the following key metrics:

- Transaction Logs Usage (WAL logs): rds040_transaction_logs_usage
- Inactive Logical Replication Slots: inactive_logical_replication_slot
- Oldest Replication Slot Lag (WAL logs accumulated due to replication slot problems): rds045_oldest_replication_slot_lag
- Transaction Logs Generation: rds044_transaction_logs_generations

Troubleshooting and Solution

- **Check whether the WAL log size is within its allowed range. If no, rectify the fault.**

  Check the **rds040_transaction_logs_usage** metric to see how much storage the WAL logs occupy. Check whether the WAL log size has significantly increased recently or whether WAL logs occupy a large proportion of the storage space.

  You can also run the SQL statement below to check the WAL log size. If there are too many WAL logs, perform the subsequent steps to locate the fault.

  ```
  select pg_size_pretty(sum(size)) from pg_ls_waldir();
  ```

- **Check the replication slot statuses and the size of logs that are not cleared in a timely manner.**

  Replication slots can block WAL recycling. If the value of **inactive_logical_replication_slot** is not **0** and the value of **rds045_oldest_replication_slot_lag** is large or even the same as that of **rds040_transaction_logs_usage**, replication slots are blocking WAL recycling.

  Alternatively, you can run the following SQL statement to query the slot statuses and WAL lag:

  ```
  select slot_name, active,
  pg_size_pretty(pg_wal_lsn_diff(b, a.restart_lsn)) as slot_latency
  from pg_replication_slots as a, pg_current_wal_lsn() as b;
  ```


If the query result shows that there is any slot whose **active** field is **f** and **slot_latency** field is large, inactive replication slots are blocking WAL recycling. WAL logs are accumulated and cannot be cleared.

Analyze whether this slot is still required. If not, run the following SQL statement to delete it:

```
select pg_drop_replication_slot('slot_name');
```

- **Check the WAL log retention parameters.**

  Check whether the **wal_keep_segments**, **wal_keep_size**, and **max_wal_size** parameters are properly set.

  ```
  select name, setting from pg_settings where name in ('wal_keep_segments', 'wal_keep_size', 'max_wal_size');
  ```

  – For RDS for PostgreSQL 12 or earlier versions, check the value of **wal_keep_segments**. For later versions, check the value of **wal_keep_size**.

  – The values of the WAL log retention parameters should be greater than 4 GB but less than 10% of the total storage. If they are too small, the primary instance may clear the WAL logs required by the standby instance, causing exceptions on the standby instance.

- **Check how busy write services are.**

  View the **rds044_transaction_logs_generations** metric to determine how busy write services are. This metric indicates the average size of transaction logs (WAL logs) generated per second.

  If the value of this metric is large (greater than 50 MB/s on average), there are a large number of write services. In this case, the database kernel reserves more WAL logs for recycling, and the storage usage of WAL logs increases. You are advised to scale up storage to ensure storage redundancy.

WAL Logs Generated While Backups Are Created

If the WAL log size increases during backup creation and returns to normal afterward, the WAL logs are generated faster than the backups.

You can adjust the backup time to avoid heavy data writes during backup creation or scale up storage to ensure enough cache space.

14 Updating, Deleting, or Inserting Data Records at a Time

This topic describes how to insert (BULK INSERT), update (BULK UPDATE), and delete (BULK DELETE) data records at a time. These operations significantly reduce interactions between your DB instance and applications, reducing system overhead and increasing the overall throughput.

Creating an Example Table Structure

```
CREATE TABLE product_inventory (
item_id INT PRIMARY KEY,
item_name VARCHAR(100),
stock_qty INT,
last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Inserting Data Records at a Time

- Method 1: Using the VALUES clause

  Insert multiple product records at a time.

  ```
  INSERT INTO product_inventory (item_id, item_name, stock_qty)
  VALUES
      (101, 'Smartwatch', 50),
      (102, 'Wireless headset', 120),
      (103, 'Bluetooth speaker', 75),
      (104, 'Tablet', 30);
  ```

- Method 2: Using SELECT to generate sequence data

  Generate 100 test data records at a time.

  ```
  INSERT INTO product_inventory (item_id, item_name, stock_qty)
  SELECT
      200 + n,
      'Test product-' || n,
      (random() * 100)::INT
  FROM generate_series(1, 100) AS n;
  ```

- Method 3: Using multiple INSERT statements wrapped in a transaction

  ```
  BEGIN;
  INSERT INTO product_inventory VALUES (301, 'Game controller', 40);
  INSERT INTO product_inventory VALUES (302, 'Mechanical keyboard', 60);
  INSERT INTO product_inventory VALUES (303, 'Gaming mouse', 55);
  COMMIT;
  ```

- Method 4: Using the COPY command

PostgreSQL provides COPY, a protocol for fast data import/export. It uses a simpler format and transfer mechanism. This makes data import/export much faster than using INSERT statements especially when there are large volumes of data to handle. The following describes its advantages.

- Reducing overhead: COPY uses the binary or text format to transfer data, avoiding the overhead caused by parsing SQL statements one by one.
- Improving performance: Using COPY to insert tens of thousands of data records is more than 10 times faster than using INSERT.
- Applicability across diverse situations: It works for various scenarios like initializing, migrating, or periodically importing data in batches. It can directly read data from CSV files, binary files, or program streams.

The following is an example of using COPY to import data from a CSV file:

```
COPY product_inventory FROM STDIN WITH (FORMAT csv);
401, "4K monitor", 25
402, "Curved monitor", 18
403, "Portable monitor", 32
\.
```

NOTE

The COPY APIs vary based on the language driver you use. For more information, see the following:

- **PostgreSQL JDBC Driver - API**
- **PostgreSQL 9.6.2 Documentation — COPY**

Updating Data Records at a Time

- Using temporary tables

  Create a temporary table to perform a join-based update.

  ```
  WITH update_data(item_id, new_qty) AS (
     VALUES
        (101, 45),
        (102, 110),
        (104, 28)
  )
  --Execute the batch update.
  UPDATE product_inventory p
  SET stock_qty = u.new_qty
  FROM update_data u
  WHERE p.item_id = u.item_id;
  ```

- Using the CASE clause

  Update the inventory data at a time based on different conditions.

  ```
  UPDATE product_inventory
  SET stock_qty = CASE
     WHEN item_id = 201 THEN 90
     WHEN item_id = 202 THEN 65
     WHEN item_id = 203 THEN 40
     ELSE stock_qty
  END
  WHERE item_id IN (201, 202, 203);
  ```

Deleting Data Records at a Time

- Using the IN clause

  Delete the product records with specified IDs.

  ```
  DELETE FROM product_inventory
  WHERE item_id IN (301, 302, 303, 304);
  ```

- Using sub-queries

  Delete the products whose in-stock quantity is less than 20.

  ```
  DELETE FROM product_inventory
  WHERE stock_qty < 20;
  ```

- Using TRUNCATE

  Clear all data in a table (this operation cannot be rolled back).

  ```
  TRUNCATE TABLE product_inventory;
  ```

Suggestions on Performance Optimization

- **Batch processing with a transaction**: Wrap operations in a single transaction.

  ```
  BEGIN;
  --Batch operation 1
  --Batch operation 2
  COMMIT;
  ```

- **Batch size control**: You are advised to process 1,000 to 5,000 records in each batch.

15 Using Event Triggers to Implement the DDL Recycle Bin, Firewalls, and Incremental Synchronization

Function Description

RDS for PostgreSQL allows you to use event triggers to implement features like the DDL recycle bin, firewalls, and incremental synchronization. Event triggers help you lower maintenance expenses. For strong database security needs, use PostgreSQL event triggers to implement the DDL recycle bin and firewalls to enhance data security.

This topic describes how to use PostgreSQL event triggers to implement the DDL recycle bin, firewalls, and incremental synchronization.

Table 15-1 Functions of event triggers

Function Type	Effect	Implementation
Pre-event defense	Blocking risky DDL operations, such as DROP TABLE, DROP INDEX, and DROP DATABASE	Use the ddl_command_start event trigger to block unauthorized operations.
Post-event backtracking	Restoring tables from the recycle bin after they are deleted by mistake	Use the ddl_command_end and sql_drop event triggers to record DDL operations.

DDL Recycle Bin

Use the event triggers, pg_get_ddl_command and pg_get_ddl_drop, to collect DDL statements and save them to the ddl_recycle.ddl_log table. The triggers help you track DDL operations performed on your databases.

1. Create a dedicated schema and a table.

```
CREATE SCHEMA ddl_recycle;
```



```
CREATE TABLE IF NOT EXISTS ddl_recycle.ddl_log (
    id SERIAL PRIMARY KEY,
    event_time TIMESTAMPTZ DEFAULT NOW(),
    username TEXT,
    database TEXT,
    client_addr INET,
    event TEXT,
    tag TEXT,
    object_type TEXT,
    schema_name TEXT,
    object_identity TEXT,
    command TEXT,
    is_dropped BOOLEAN DEFAULT FALSE
);
```

2. Create an event trigger function.

```
CREATE OR REPLACE FUNCTION ddl_recycle.log_ddl_command()
RETURNS event_trigger AS $$
DECLARE
    cmd TEXT;
    obj RECORD;
BEGIN
    SELECT query INTO cmd FROM pg_stat_activity WHERE pid = pg_backend_pid();

    IF TG_EVENT = 'ddl_command_end' THEN
        FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands() LOOP
            INSERT INTO ddl_recycle.ddl_log (
                username, database, client_addr, event, tag,
                object_type, schema_name, object_identity, command
            ) VALUES (
                current_user, current_database(), inet_client_addr(),
                TG_EVENT, TG_TAG, obj.object_type, obj.schema_name,
                obj.object_identity, cmd
            );
        END LOOP;
    ELSIF TG_EVENT = 'sql_drop' THEN
        FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects() LOOP
            --The original record is marked as deleted.
            UPDATE ddl_recycle.ddl_log
            SET is_dropped = TRUE
            WHERE object_identity = obj.object_identity AND is_dropped = FALSE;

            --Insert a new record to record the deletion operation.
            INSERT INTO ddl_recycle.ddl_log (
                username, database, client_addr, event, tag,
                object_type, schema_name, object_identity, command, is_dropped
            ) VALUES (
                current_user, current_database(), inet_client_addr(),
                TG_EVENT, TG_TAG, obj.object_type, obj.schema_name,
                obj.object_identity, cmd, TRUE
            );
        END LOOP;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

3. Register an event trigger.

```
CREATE EVENT TRIGGER ddl_recycle_trigger
ON ddl_command_end
EXECUTE FUNCTION ddl_recycle.log_ddl_command();

CREATE EVENT TRIGGER ddl_drop_trigger
ON sql_drop
EXECUTE FUNCTION ddl_recycle.log_ddl_command();
```

After the execution is complete, the DDL statements are recorded in the test_ddl.tb_ddl_command table.

4. Run a DDL statement and check whether changes can be recorded.

```
create table ddl_recycle.a(id int);
select * from ddl_recycle.ddl_log;
```


Figure 15-1 Checking the execution result

```
test=> drop table ddl_recycle.a;
DROP TABLE
test=> select * from ddl_recycle.ddl_log;
 id |          event_time           | username | database | client_addr |      event      |     tag      | object_type |
 schema_name | object_identity |               command               | is_dropped
----+-------------------------------+----------+----------+-------------+-----------------+--------------+-------------+
-------------+-----------------+-------------------------------------+------------
  1 | 2025-07-31 11:46:22.078908+08 | test     | test     |             | ddl_command_end | CREATE TABLE | table       |
 ddl_recycle | ddl_recycle.a   | create table ddl_recycle.a(id int); | f
(1 row)
```

DDL Firewall

You can create event triggers as needed and use the ddl_command_start event to block the execution of specific DDL statements.

1. Create a table containing firewall rules.

```
CREATE SCHEMA IF NOT EXISTS ddl_firewall;

CREATE TABLE IF NOT EXISTS ddl_firewall.rules (
    id SERIAL PRIMARY KEY,
    username TEXT,
    tag TEXT,
    pattern TEXT,
    action TEXT CHECK (action IN ('BLOCK', 'LOG')),
    created_at TIMESTAMPTZ DEFAULT NOW()
);
```

2. Create a firewall trigger function.

```
RETURNS event_trigger AS $$
DECLARE
    cmd TEXT;
    rule RECORD;
    is_blocked BOOLEAN := FALSE;
BEGIN
    SELECT query INTO cmd FROM pg_stat_activity WHERE pid = pg_backend_pid();

    FOR rule IN SELECT * FROM ddl_firewall.rules
                WHERE (username = current_user OR username = '*')
                  AND (tag = TG_TAG OR tag = '*')
                ORDER BY id
    LOOP
        IF cmd ~ rule.pattern THEN
            IF rule.action = 'BLOCK' THEN
                RAISE EXCEPTION 'DDL operation blocked by rule: %', rule.id;
            ELSIF rule.action = 'LOG' THEN
                INSERT INTO ddl_recycle.ddl_log (
                    username, database, client_addr, event, tag, command
                ) VALUES (
                    current_user, current_database(), inet_client_addr(),
                    'FIREWALL_LOG', TG_TAG, cmd
                );
            END IF;
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

3. Register an event trigger.

```
CREATE EVENT TRIGGER ddl_firewall_trigger
ON ddl_command_start
EXECUTE FUNCTION ddl_firewall.check_ddl();
```

4. Add firewall rules.

```
insert into ddl_firewall.rules values(1,'test', 'DROP TABLE', '', 'BLOCK');
```

5. When you try to delete the table as the user **test**, the deletion is blocked.

```
create table ddl_firewall.a(id int);
drop table ddl_firewall.a;
```


Figure 15-2 Checking the execution result

```
test=> create table ddl_firewall.a(id int);
CREATE TABLE
test=> drop table ddl_firewall.a;
ERROR:  DDL operation blocked by rule: 1
CONTEXT:  PL/pgSQL function ddl_firewall.check_ddl() line 16 at RAISE
```

DDL Incremental Synchronization

The publisher stores executed DDL statements in the ddl_recycle.ddl_log table. The subscriber can read the records to synchronize data.

1. Create a publication on the publisher.
   ```
   CREATE PUBLICATION my_ddl_publication FOR TABLE ONLY ddl_recycle.ddl_log;
   ```
2. Create the same table on the subscriber.
   ```
   CREATE SCHEMA ddl_recycle;
   CREATE TABLE IF NOT EXISTS ddl_recycle.ddl_log (
       id SERIAL PRIMARY KEY,
       event_time TIMESTAMPTZ DEFAULT NOW(),
       username TEXT,
       database TEXT,
       client_addr INET,
       event TEXT,
       tag TEXT,
       object_type TEXT,
       schema_name TEXT,
       object_identity TEXT,
       command TEXT,
       is_dropped BOOLEAN DEFAULT FALSE
   );
   ```
3. Create a subscription on the subscriber.
   ```
   CREATE SUBSCRIPTION my_ddl_subscriptin CONNECTION 'host=*** port=*** user=*** password=***
   dbname=**' PUBLICATION my_ddl_publication;
   ```
4. Create a trigger for the ddl_recycle.ddl_log table on the subscriber to implement incremental synchronization of DDL statements.

16 Creating Replication Slots to Enable CDC

Function Description

Open-source PostgreSQL uses replication slots to enable Change Data Capture (CDC). Replication slots are the core for logical replication. They can:

1. Ensure that WAL logs of the primary database are not cleared until all logs are consumed by secondary databases or subscribers.
2. Record subscribers' consumption positions (LSNs) to prevent data loss or repeated consumption.
3. Parse WAL logs into readable change data (such as INSERT, UPDATE, and DELETE operations) using logical decoding plugins (like test_decoding and wal2json) to implement CDC.

This topic describes how to enable CDC for an RDS for PostgreSQL instance.

Prerequisites

- You have purchased an ECS. For details, see **Purchasing an ECS in Custom Config Mode**. Ensure that the ECS is in the same region, VPC, and security group as your RDS for PostgreSQL instance.
- You have purchased an RDS for PostgreSQL instance and installed a PostgreSQL client on the ECS. For details, see **Buying a DB Instance and Connecting to It Using a PostgreSQL Client**.
- You have connected to the RDS for PostgreSQL instance through the ECS. For details, see **Connecting to a DB Instance from a Linux ECS over a Private Network**.

Precautions

- You can enable CDC and consume captured data only on the primary instance. Read replicas do not support this feature.
- RDS for PostgreSQL supports logical replication slot failover. A primary/standby switchover does not affect CDC. For more information, see **Failover Slot for Logical Subscriptions**.

- Before enabling CDC, modify RDS for PostgreSQL instance parameters and reboot the instance. To prevent impact on workloads, modify parameters during off-peak hours.

Enabling CDC

Step 1: Create a Test Database

The following uses the **testdb** database as an example.

Step 1 **Log in to the management console**.

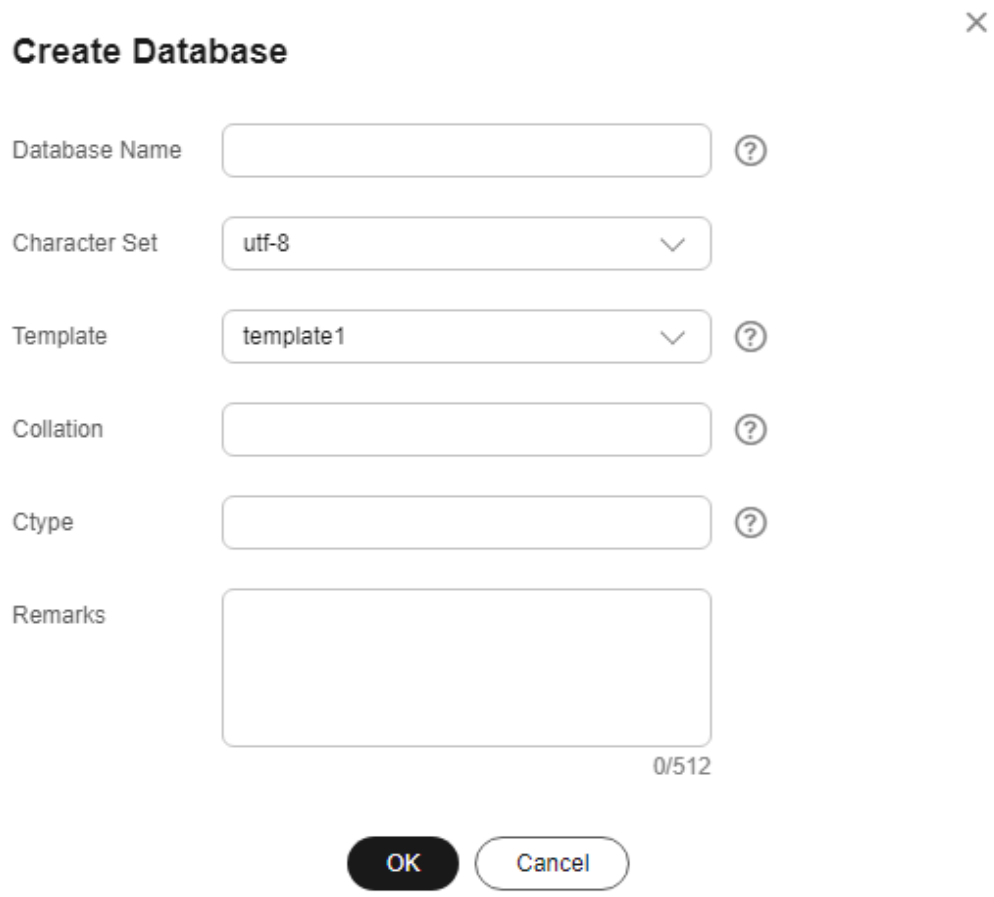
Step 2 Click  in the upper left corner and select a region.

Step 3 Click  in the upper left corner of the page and choose **Databases** > **Relational Database Service**.

Step 4 On the **Instances** page, click the instance name.

Step 5 On the **Databases** page, click **Create Database**. In the displayed dialog box, configure required parameters and click **OK**.

Figure 16-1 Creating a database



----End

Step 2: Create a Test Account and Configure Permissions

The account created in this example is for reference only. You can change it as required.

1. On the **Accounts** page, click **Create Account**.
2. In the displayed dialog box, enter the username, password, permissions, and remarks, and click **OK**.
3. Create a standard account, for example, **cdc_user**.
4. Connect to the RDS for PostgreSQL instance using the account **root**.
   ```
   psql -h <instance-address> -p 5432 -U root -d testdb
   ```
5. Assign the Replication role to the account **cdc_user** and query the result:
   ```
   ALTER USER cdc_user WITH REPLICATION;
   SELECT rolreplication FROM  pg_roles WHERE rolname='cdc_user';
   ```
   Query result:
   ```
   rolreplication
   ----------------
    t
   (1 row)
   ```
6. Grant permissions to the **cdc_user** account:


```
GRANT SELECT ON ALL TABLES IN SCHEMA PUBLIC to cdc_user;
```

Step 3: Modify RDS for PostgreSQL Instance Parameters

1. Query the instance's parameter settings:

```
SELECT name,
      setting,
      short_desc,
      source
FROM pg_settings
WHERE name ='wal_level';
```

   Query result:

```
name                  | setting |                              short_desc                              |      source
----------------------+---------+----------------------------------------------------------------------+-------------------
 wal_level            | replica | Sets the level of information written to the WAL.                    |
configuration file
(1 rows)
```

   wal_level specifies how much information is written to the WAL. It determines the WAL's purpose (such as crash recovery, physical replication, or logical replication). The following are three main values of **wal_level**, from least to most detailed:

   - **minimal**: The system records only the information required to recover from a crash. No replication is supported. (This value is almost not used in production environments.)
   - **replica** (default): Enough data needed for physical replication is written. This level supports physical replication (such as, primary-standby synchronization, where the standby instance replays WAL logs to remain synchronous with the primary instance) and allows a read replica to function as a hot standby.
   - **logical**: The system records the data logged at replica level and also adds the metadata (such as row-level change data and table structure information) necessary for logical replication. This setting supports logical replication and CDC.

2. On the **Instances** page, click the instance name.
3. In the navigation pane, choose **Parameters**.
4. Change the value of **wal_level** to **logical**.

   For details about how to modify instance parameters, see **Modifying Parameters of an RDS for PostgreSQL Instance**.

   To prevent impact on workloads, modify parameters during off-peak hours.
5. After changing the parameter value, reboot your RDS for PostgreSQL instance to apply it.

Step 4: Create a Logical Replication Slot

Instance parameters have been modified in **Step 3: Modify RDS for PostgreSQL Instance Parameters**, so perform the following operations after the RDS for PostgreSQL instance is rebooted and its status changes to **Available**.

1. Connect to the RDS for PostgreSQL instance using the account **root**.

```
psql -h <instance-address> -p 5432 -U root -d testdb
```

2. Create a replication slot named **cdc_replication_slot** with the decoding plugin **test_decoding**:


```
SELECT * FROM pg_create_logical_replication_slot(
    'cdc_replication_slot',
    'test_decoding'
);
```

After the execution, the replication slot name and initial LSN are displayed in the command output, as shown in the following:

```
    slot_name    |    lsn
 ----------------+-----------
 cdc_replication_slot | 0/16A00000
(1 row)
```

3. List all replication slots to verify that the replication slot is created:

```
SELECT slot_name, plugin, active FROM pg_replication_slots;
```

If **slot_name** is **cdc_replication_slot** and **plugin** is **test_decoding**, the replication slot is created.

```
slot_name             |     plugin     | slot_type | datoid | database | temporary | active | active_pid | xmin |
catalog_xmin | restart_lsn | confirmed_flush_lsn | wal_status | safe_wal_size  | two_phase
----------------------+---------------+-----------+--------+----------+-----------+--------+------------+------
+--------------+-------------+---------------------+------------+----------------+-----------
 cdc_replication_slot | test_decoding | logical   |  31721 | testdb   | f         | f      |            |      |
25523| 2/2500A1B0  | 2/25003882           | reserved   |                | f
(1 row)
```

Step 5: Create Test Data

Create test data to simulate a production environment.

1. Create a test table for generating change data.

```
CREATE TABLE public.tb_tests (
    id SERIAL PRIMARY KEY,
    product_name VARCHAR(50) NOT NULL,
    price NUMERIC(10,2) NOT NULL,
    create_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

2. Insert data.

```
INSERT INTO public.tb_tests (product_name, price) VALUES ('Laptop', 5999.99);
INSERT INTO public.tb_tests (product_name, price) VALUES ('Mechanical keyboard', 299.99);
```

3. Update data.

```
UPDATE public.tb_tests SET price = 5899.99 WHERE id = 1;
```

4. Delete data.

```
DELETE FROM public.tb_tests  WHERE id = 2;
```

Step 6: Read Data Using the Client

1. Connect to the RDS for PostgreSQL instance as the user **cdc_user**.

```
psql -h <instance-address> -p 5432 -U cdc_user -d testdb
```

2. Read data from the replication slot:

```
SELECT * FROM pg_logical_slot_peek_changes(
    'cdc_replication_slot', --Replication slot name
    NULL,              --Start LSN (NULL indicates that the data is read from the current position.)
    NULL,              --Maximum number of returned records (NULL indicates that the number is not
limited.)
   );
```

Example output:

```
lsn         |  xid  |                              data
------------+-------+-----------------------------------------------------------------------------
 0/16A01200 | 1234 | BEGIN 1234
 0/16A01200 | 1234 | table public.tb_tests: INSERT: id[integer]:1 product_name[character
varying]:'Laptop' price[numeric(10,2)]:5999.99 create_time[timestamp with time zone]:'2024-05-20
10:00:00+08'
 0/16A01350 | 1234 | table public.tb_tests: INSERT: id[integer]:2 product_name[character
```



```
varying]:'Mechanical keyboard' price[numeric(10,2)]:299.99 create_time[timestamp with time
zone]:'2024-05-20 10:01:00+08'
 0/16A014A0 | 1234 | COMMIT 1234
 0/16A015F0 | 1235 | BEGIN 1235
 0/16A015F0 | 1235 | table public.tb_tests: UPDATE: id[integer]:1 price[numeric(10,2)]:5899.99
(old:5999.99)
 0/16A016E0 | 1235 | COMMIT 1235
 0/16A017D0 | 1236 | BEGIN 1236
 0/16A017D0 | 1236 | table public.tb_tests: DELETE: id[integer]:2
 0/16A01860 | 1236 | COMMIT 1236
 (8 rows)
```

Step 7: Consume Captured Data Using the Client

On the client, run the following command to connect to the DB instance and consume data from the replication slot **cdc_replication_slot**:

```
pg_recvlogical \
-h <PostgreSQL-instance-address> \
  -p 5432 \
  -U cdc_user \
  -d testdb \
  --slot=cdc_replication_slot \
  --start \
  -f -
```

Specify the parameter values as required:

- -h: RDS for PostgreSQL instance address
- -p: port number (default: **5432**)
- -U: subscriber, which must have the REPLICATION privilege
- -d: database name
- --slot: replication slot name
- --start: the position where the system begins consuming data. This parameter can be omitted for the first consumption. By default, the first consumption starts from the start position.
- -f: output to the console. The value **-** indicates standard output.

Example output:

```
BEGIN 1234
table public.tb_tests: INSERT: id[integer]:1 product_name[character varying]:'Laptop'
price[numeric(10,2)]:5999.99 create_time[timestamp with time zone]:'2024-05-20 10:00:00+08'
table public.tb_tests: INSERT: id[integer]:2 product_name[character varying]:'Mechanical keyboard'
price[numeric(10,2)]:299.99 create_time[timestamp with time zone]:'2024-05-20 10:01:00+08'
COMMIT 1234
BEGIN 1235
table public.tb_tests: UPDATE: id[integer]:1 price[numeric(10,2)]:5899.99 (old:5999.99)
COMMIT 1235
BEGIN 1236
table public.tb_tests: DELETE: id[integer]:2
COMMIT 1236
```


NOTE

- After pg_recvlogical consumes data, the data is marked as processed and restart_lsn of the replication slot is updated to the latest position. pg_recvlogical and pg_logical_slot_peek_changes will not return these data (to prevent repeated consumption).
- The command remains connected. If there are new changes (for example, INSERT or UPDATE is executed again), new transaction data is generated in real time.
- To terminate the consumption, press **Ctrl+C**. The next execution will continue the consumption from the last position where it is terminated (depending on restart_lsn recorded by the replication slot).

Disabling CDC

RDS for PostgreSQL instances with CDC enabled require more space for storing WAL logs. If CDC is no longer needed, to prevent replication slots from running out of storage space, disable CDC by performing the following operations:

1. Connect to the RDS for PostgreSQL instance using the account **root**.
   ```
   psql -h <instance-address> -p 5432 -U root -d testdb
   ```
2. (Optional) Check for the replication slot.
   ```
   SELECT slot_name FROM pg_replication_slots WHERE slot_name = 'cdc_replication_slot';
   ```
3. Delete the replication slot. This operation cannot be undone. Exercise caution when performing this operation.
   ```
   SELECT pg_drop_replication_slot('cdc_replication_slot');
   ```
4. Verify that CDC is disabled.
   ```
   SELECT slot_name FROM pg_replication_slots;
   ```
   If the query result does not contain **cdc_replication_slot**, the feature has been disabled.

17 Read/Write Splitting with Pgpool

Function

Pgpool is a middleware designed for PostgreSQL. It is deployed between database servers and clients and transfers messages between the frontend and backend. Pgpool is transparent to servers and clients. It provides features such as connection pooling, replication, load balancing, and connection limiting to improve the performance, availability, and scalability of database clusters.

This topic describes how to use pgpool to implement read/write splitting for RDS for PostgreSQL DB instances and read replicas.

Prerequisites

- You have purchased an ECS that is in the same region, VPC, and security group as your RDS for PostgreSQL DB instance. For details, see **Purchasing an ECS in Custom Config Mode**.
- You have purchased an RDS for PostgreSQL DB instance and installed a PostgreSQL client on the ECS. For details, see **Buying a DB Instance and Connecting to It Using a PostgreSQL Client**.
- You have created a read replica for the RDS for PostgreSQL DB instance. For details, see **Creating a Read Replica**.
- You have connected to the RDS for PostgreSQL DB instance through the ECS. For details, see **Connecting to a DB Instance from a Linux ECS over a Private Network**.

Procedure

Step 1: Install Pgpool

This step uses an ECS using CentOS 7 and PostgreSQL 12 as an example. For other OS and database versions, adjust the parameter values in the following commands as needed.

Log in to the ECS and run the following commands to install PostgreSQL 12 and pgpool:

```
# Configure a yum repository in the environment.
sudo yum install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-
```



```
repo-latest.noarch.rpm
# Search for the postgresql package.
sudo yum search all postgresql
# Search for the pgpool package.
sudo yum search all pgpool
# Install PostgreSQL 12.
sudo yum install -y postgresql12-server
# Install pgpool 12.
sudo yum install -y pgpool-II-12-extensions
```

Step 2: Configure Pgpool

Use pgpool to implement load balancing for access. Authentication occurs between the client and pgpool. The client still needs to pass the authentication of PostgreSQL.

1. Query the pgpool installation path.

   ```
   rpm -qa | grep pgpool
   ```

   Figure 17-1 Execution results

   ```
   [root@ecs-          ~]# rpm -qa|grep pgpool
   pgpool-II-12-4.1.4-1.rhel7.x86_64
   pgpool-II-12-extensions-4.1.4-1.rhel7.x86_64
   ```

2. Run the SQL statements below on the RDS for PostgreSQL DB instance to create a user who is authorized to manage the DB instance.

   Create a user who is authorized to manage the DB instance.

   ```
   create role digoal login encrypted password 'xxxxxxx';
   create database digoal owner digoal;
   ```

   Create a user who is authorized to log in to **postgres** or a specified database and check the health heartbeats between pgpool and the read replica. With the parameters of pgpool properly configured, this user can check the WAL replay latency on the read replica.

   ```
   create role nobody login encrypted password 'xxxxxxx';
   ```

   xxxxxxx: password of the new database user

3. Modify the **pgpool.conf** file.

   ```
   cd /etc/pgpool-II-12/

   cp pgpool.conf.sample-stream pgpool.conf

   vi pgpool.conf
   ```

   - **/etc/pgpool-II-12/**: directory where pgpool is installed
   - **pgpool.conf.sample-stream**: default configuration file template provided by pgpool for streaming replication
   - **pgpool.conf**: pgpool configuration file

4. Press **i** to enter the edit mode. Modify the following items based on logs. You can choose to modify other configuration information according to the comments in the **pgpool.conf** file.

   ```
   listen_addresses = '*'

   backend_hostname0 = 'IP address of the primary instance'
   backend_port0 = 5432
   backend_flag0 = 'ALWAYS_MASTER'

   backend_hostname1 = 'IP address of the read replica'
   backend_port1 = 5432
   backend_flag1 = 'ALLOW_TO_FAILOVER'
   ```



```
enable_pool_hba = on

pid_file_name = '/var/run/pgpool-II-12/pgpool.pid'
```

- **listen_addresses**: IP addresses that pgpool listens on. In this example, specify all IP addresses.
- **backend_hostname**: IP address of the backend PostgreSQL database node to be connected
- **backend_port**: port of the backend PostgreSQL database node to be connected
- **backend_flag**: used by pgpool to control node behavior

5. Use MD5 authentication to configure the **pool_passwd** file.

```
pg_md5 --md5auth --username=digoal "xxxxxxx"
pg_md5 --md5auth --username=nobody "xxxxxxx"
```

The passwords of the **digoal** and **nobody** users are generated and automatically written to the **pool_passwd** file.

6. Check the automatically generated **pool_passwd** file.

```
cd /etc/pgpool-II-12

cat pool_passwd
```

pool_passwd: configuration file for pgpool to store encrypted passwords of PostgreSQL users

Figure 17-2 Execution results

```
[root@ecs-          pgpool-II-12]# cat pool_passwd
digoal:md5e8318d48f545394c4e67941aae8df253
nobody:md5ca14b3976cc05f036a450f0bd864b050
```

7. Configure the **pgpool_hba** file.

```
cd /etc/pgpool-II-12
cp pool_hba.conf.sample pool_hba.conf
vi pool_hba.conf

# Press i to enter the edit mode.
# Add the following content to the pool_hba.conf file:
host all all 0.0.0.0/0 md5
```

- **pool_hba.conf.sample**: configuration file template for client connection authentication rules
- **pool_hba.conf**: authentication rule file for client connections to pgpool

Figure 17-3 Execution results

```
# "local" is for Unix domain socket connections only
local   all         all                               trust
# IPv4 local connections:
host    all         all         127.0.0.1/32          trust
host    all         all         ::1/128               trust
host all all 0.0.0.0/0 md5
```

8. Configure the PCP password file.

This file is used to manage the users and passwords of pgpool, instead of database users and passwords.

```
cd /etc/pgpool-II-12
```



```
pg_md5 abc # The password abc is used as an example.
900150983cd24fb0d6963f7d28e17f72

cp pcp.conf.sample pcp.conf

vi pcp.conf

# Press i to enter the edit mode.
# Add the following content to the pcp.conf file:
USERID:MD5PASSWD
manage:900150983cd24fb0d6963f7d28e17f72 # The manage user is used to manage the PCP password file.
```

- **pcp.conf.sample**: template file used to generate the **pcp.conf** file
- **pcp.conf**: file used for authentication by the PCP interface. It defines the username and encrypted password for an administrator to remotely manage pgpool.

Step 3: Start Pgpool and Verify Read/Write Splitting

1. Start pgpool.

   ```
   cd /etc/pgpool-II-12

   pgpool -f ./pgpool.conf -a ./pool_hba.conf -F ./pcp.conf
   ```

2. Use pgpool to connect to the DB instance.

   ```
   psql -h 127.0.0.1 -p 9999 -U digoal -d postgres
   ```

   - **127.0.0.1**: the IP address specified by **listen_addresses** in the **pgpool.conf** file
   - **9999**: the default listening port configured in **pgpool.conf**

   Figure 17-4 Execution results

   ```
   [root@ecs-          pgpool-II-12]# psql -h 127.0.0.1 -p 9999 -U digoal -d postgres
   psql (12.22, server 16.8)
   WARNING: psql major version 12, server major version 16.
            Some psql features might not work.
   Type "help" for help.

   postgres=>
   ```

3. Check the pgpool cluster status.

   ```
   show pool_nodes;
   ```

   Figure 17-5 Execution results

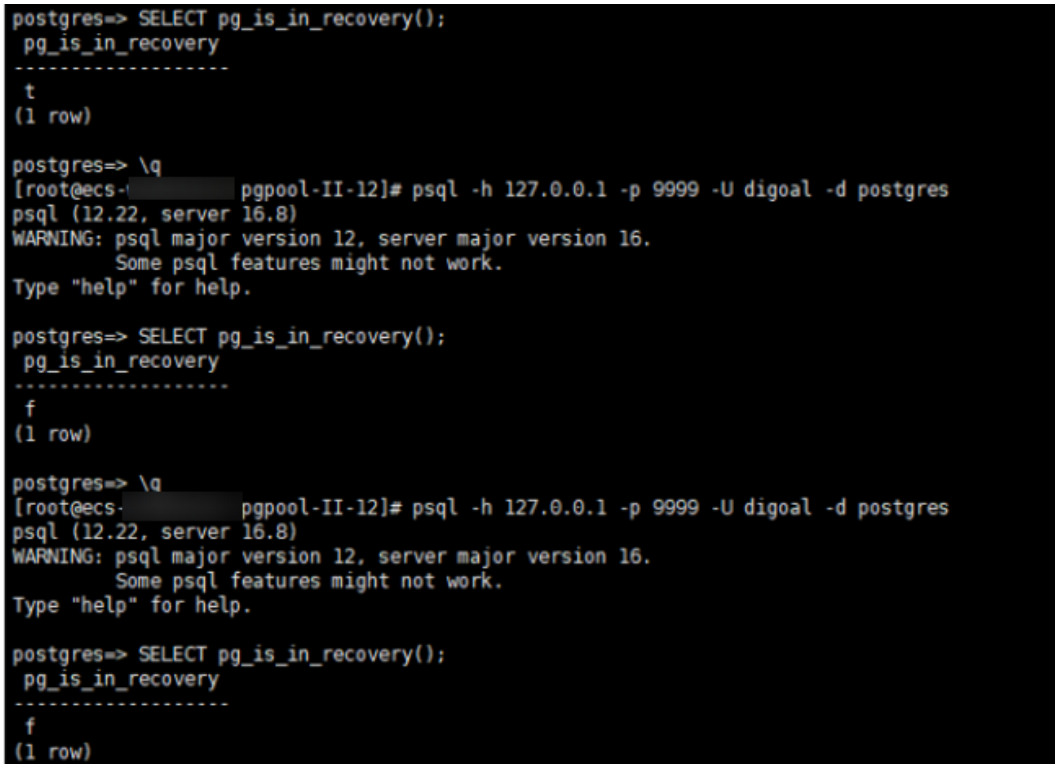
   ```
   postgres=> show pool_nodes;
    node_id |   hostname    | port | status | lb_weight |  role   | select_cnt | load_balance_node | replication_delay | replication_state | replication_sync_state | last_status_change
   ---------+---------------+------+--------+-----------+---------+------------+-------------------+-------------------+-------------------+------------------------+---------------------
    0       | 192.168.0.46  | 5432 | up     | 0.500000  | primary | 1          | false             | 0                 |                   |                        | 2025-07-09 11:27:57
    1       | 192.168.0.117 | 5432 | up     | 0.500000  | standby | 1          | true              | 0                 |                   |                        | 2025-07-09 11:27:57
   (2 rows)
   ```

4. Run the command below on the database, disconnect from and reconnect to the database, and query **pg_is_in_recovery()** again. If **t** and **f** are returned alternately, requests are sent to the primary instance and read replica alternately, indicating that read/write splitting is successful.

   ```
   SELECT pg_is_in_recovery();
   ```

   pg_is_in_recovery() is used to check whether the DB instance is in recovery mode. If **t** is returned, requests are sent to the read replica. If **f** is returned, requests are sent to the primary instance.

Figure 17-6 Execution results



FAQ

How Do I Stop Pgpool and Reload the Configuration for Pgpool?

You can run the **pgpool --help** command to obtain more information about the commands used in pgpool. Example:

```
cd /etc/pgpool-II-12

pgpool -f ./pgpool.conf -m fast stop
```

How Do I Configure the pgpool.conf File If There Are Multiple Read Replicas?

Add new configuration information according to the format of the **pgpool.conf** file. Example:

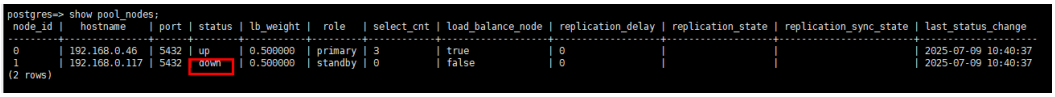
```
backend_hostname1 = 'xx.xx.xxx.xx'
backend_port1 = 5432
backend_weight1 = 1
backend_flag1 = 'ALLOW_TO_FAILOVER'
backend_application_name1 = 'server1'

backend_hostname2 = 'xx.xx.xx.xx'
backend_port2 = 5432
backend_weight2 = 1
backend_flag2 = 'ALLOW_TO_FAILOVER'
backend_application_name1 = 'server2'
```


What Can I Do If T or F Is Always Returned When I Query pg_is_in_recovery()?

If **t** is always returned, pgpool is connected only to the read replica. If **f** is always returned, pgpool is connected only to the primary instance. At the same time, an abnormal node is always displayed in the **show pool_nodes;** command output.

Figure 17-7 Execution results



You can clear the **pgpool_status** configuration file and restart pgpool. Example:

```
cd /etc/pgpool-II-12

# Stop the service.
pgpool -f ./pgpool.conf -m fast stop

# Delete the configuration file.
rm /tmp/pgpool_status

# Restart the service.
pgpool -f ./pgpool.conf -a ./pool_hba.conf -F ./pcp.conf
```

If the fault persists, you can use the pcp_attach_node tool to register the primary instance or read replica again. pcp_attach_node is a cluster management tool provided by pgpool. It can re-register nodes in a cluster with the cluster.

- Check the PCP connection user.

  ```
  cd /etc/pgpool-II-12

  cat pcp.conf
  ```


Figure 17-8 Execution results

```
[root@ecs-          pgpool-II-12]# cat pcp.conf
# PCP Client Authentication Configuration File
# ============================================
#
# This file contains user ID and his password for pgpool
# communication manager authentication.
#
# Note that users defined here do not need to be PostgreSQL
# users. These users are authorized ONLY for pgpool
# communication manager.
#
# File Format
# ===========
#
# List one UserID and password on a single line. They must
# be concatenated together using ':' (colon) between them.
# No spaces or tabs are allowed anywhere in the line.
#
# Example:
# postgres:e8a48653851e28c69d0506508fb27fc5
#
# Be aware that there will be no spaces or tabs at the
# beginning of the line! although the above example looks
# like so.
#
# Lines beginning with '#' (pound) are comments and will
# be ignored. Again, no spaces or tabs allowed before '#'.

USERID:MD5PASSWD
manage:32a9cfcd9edfd304ce34abed2ba8e3d3
```

- Use pcp_attach_node to register nodes again.

  ```
  pcp_attach_node -h local-IP-address -U pcp_user node_id
  ```

  Example:

  ```
  pcp_attach_node –h 127.0.0.1 –U manage 0
  ```


18 User Preference Recommendation Systems

Scenarios

Recommendation systems are widely used on Internet platforms and in traditional industries. They analyze user behavior, preferences, and context information to provide personalized content recommendations. Common scenarios include:

- E-commerce and retail: Products are recommended based on users' browsing and purchase history (for example, "You May Like").
- Streaming media and content platforms: Content (such as movies and short videos) is recommended based on watching records and ratings.
- Travel services: Destinations are recommended based on user preferences.
- App recommendation: Apps are recommended based on users' app download and usage habits.

This section uses movies as an example to describe how to design a recommendation system database.

Prerequisites

You have installed the **pg_trgm extension**.

Design and Implementation

The pg_trgm extension of PostgreSQL provides string similarity calculation based on trigrams. You can use this extension to build a simple user recommendation system.

1. Connect to your RDS for PostgreSQL instance.
2. Create a test table on the instance.

```
CREATE TABLE movies (
    id INT,                    -- Movie ID
    title VARCHAR(255),        -- Movie name
    description TEXT,          -- Movie description
    genres VARCHAR(255)[],     -- Movie genre
    year INTEGER               -- Movie year
);
```


3. Search for movies whose titles are similar to a specified movie title.

```
-- Create a GIN index to accelerate similarity queries.
CREATE INDEX movies_title_gin_idx ON movies USING gin(title gin_trgm_ops);

SELECT
    m2.id,
    m2.title,
    similarity(m1.title, m2.title) AS similarity_score
FROM
    movies m1,
    movies m2
WHERE
    m1.id = 123 AND                    -- Target movie ID
    m1.id != m2.id AND
    similarity(m1.title, m2.title) > 0.3  -- Similarity threshold
ORDER BY
    similarity_score DESC
LIMIT 10;
```

4. Search for movies based on the similarity of movie descriptions.

```
-- Create a GIN index for descriptions.
CREATE INDEX movies_description_gin_idx ON movies USING gin(description gin_trgm_ops);

SELECT
    m2.id,
    m2.title,
    similarity(m1.description, m2.description) AS similarity_score
FROM
    movies m1,
    movies m2
WHERE
    m1.id = 123 AND
    m1.id != m2.id AND
    similarity(m1.description, m2.description) > 0.1  -- The description similarity threshold can be set to
a small value.
ORDER BY
    similarity_score DESC
LIMIT 10;
```

5. Search for movies based on multiple features such as the title, description, and genre.

```
SELECT
    m2.id,
    m2.title,
    (
        0.5 * similarity(m1.title, m2.title) +
        0.3 * similarity(m1.description, m2.description) +
        0.2 * (SELECT COUNT(*) FROM unnest(m1.genres) AS g1
                JOIN unnest(m2.genres) AS g2 ON g1 = g2)::FLOAT /
              GREATEST(array_length(m1.genres, 1), array_length(m2.genres, 1))
    ) AS combined_similarity
FROM
    movies m1,
    movies m2
WHERE
    m1.id = 123 AND
    m1.id != m2.id
ORDER BY
    combined_similarity DESC
LIMIT 10;
```

6. Limit the search scope based on the user configuration. For example, run the following SQL statements to search for only movies of the same type:

```
SELECT m2.id, m2.title, similarity(m1.title, m2.title) AS similarity_score
FROM movies m1, movies m2
WHERE m1.id = 123 AND
      m1.id != m2.id AND
      m1.genres && m2.genres AND  -- Must have at least one same genre.
      similarity(m1.title, m2.title) > 0.3
```



```
ORDER BY similarity_score DESC
LIMIT 10;
```


19 Suggestions on RDS for PostgreSQL Metric Alarm Configurations

You can set alarm rules on Cloud Eye to customize the monitored objects and notification policies and keep track of the instance status. This topic describes how to configure RDS for PostgreSQL metric alarm rules.

Creating a Metric Alarm Rule

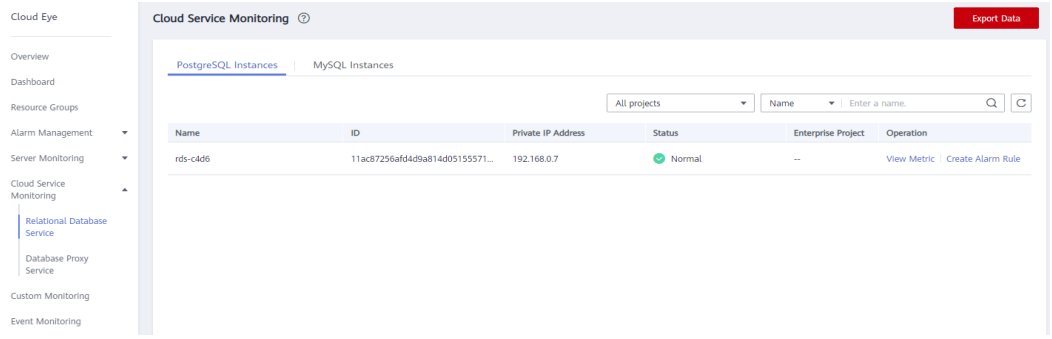
Step 1 Log in to the management console.

Step 2 Click  in the upper left corner and select a region and a project.

Step 3 Click **Service List**. Under **Management & Governance**, click **Cloud Eye**.

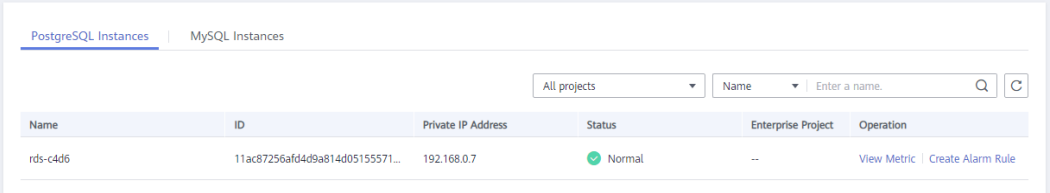
Step 4 In the navigation pane on the left, choose **Cloud Service Monitoring** > **Relational Database Service**.

Figure 19-1 Choosing a monitored object



Step 5 Locate the DB instance for which you want to create an alarm rule and click **Create Alarm Rule** in the **Operation** column.

Figure 19-2 Creating an alarm rule



Step 6 On the displayed page, set parameters as required.

Table 19-1 Alarm rule information

Parameter	Description
Name	Alarm rule name. The system generates a random name, which you can modify.
Description	Description about the rule.
Method	There are three options: **Associate template**, **Use existing template**, and **Configure manually**. **NOTE** If you select **Associate template**, after the associated template is modified, the policies contained in this alarm rule to be created will be modified accordingly. You are advised to select **Use existing template**. The existing templates already contain three common alarm metrics: CPU usage, memory usage, and storage space usage.
Template	Select the template to be used. You can select a default alarm template or create a custom template.
Alarm Policy	Policy for triggering an alarm. Whether to trigger an alarm depends on whether the metric data in consecutive periods reaches the threshold. For example, Cloud Eye triggers an alarm if the average CPU usage of the monitored object is 80% or more for three consecutive 5-minute periods. **NOTE** A maximum of 50 alarm policies can be added to an alarm rule. If any one of these alarm policies is met, an alarm is triggered.
Alarm Severity	The alarm severity can be **Critical**, **Major**, **Minor**, or **Informational**.

Figure 19-3 Configuring alarm notification

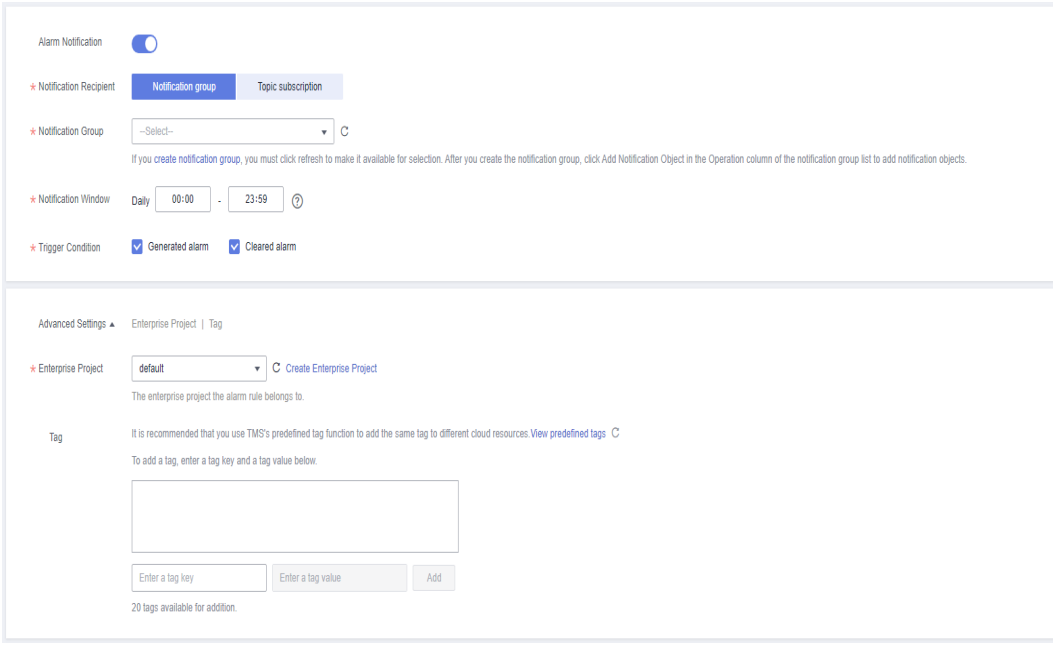


Table 19-2 Alarm notification

Parameter	Description
Alarm Notification	Whether to notify users when alarms are triggered. Notifications can be sent by email, text message, or HTTP/HTTPS message.
Notification Recipient	You can select a notification group or topic subscription as required.
Notification Group	Notification group the alarm notification is to be sent to.
Notification Object	Object the alarm notification is to be sent to. You can select the account contact or a topic. • The account contact is the mobile phone number and email address of the registered account. • A topic is used to publish messages and subscribe to notifications.
Notification Window	Cloud Eye sends notifications only within the notification window specified in the alarm rule. If **Notification Window** is set to **08:00-20:00**, Cloud Eye sends notifications only within 08:00-20:00.
Trigger Condition	Condition for triggering an alarm notification. You can select **Generated alarm** (when an alarm is generated), **Cleared alarm** (when an alarm is cleared), or both.
Enterprise Project	Enterprise project that the alarm rule belongs to. Only users with the enterprise project permissions can view and manage the alarm rule.

Parameter	Description
Tag	A tag is a key-value pair. Tags identify cloud resources so that you can easily categorize and search for your resources.

Step 7 Click **Create**. The alarm rule is created.

For details about how to create alarm rules, see **Creating an Alarm Rule** in the *Cloud Eye User Guide*.

----End

Metric Alarm Configuration Suggestions

Table 19-3 Suggestions on RDS for PostgreSQL metric alarm configurations

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
rds001_cpu_util	CPU Usage	CPU usage of the monitored object	Raw data > 80% for three consecutive periods	Major	1. Rectify the fault by referring to **Troubleshooting High CPU Usage**. 2. If the CPU usage remains high due to increased workloads, upgrade the instance specifications. For details, see **Changing a DB Instance Class**.

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
rds002_mem_util	Memory Usage	Memory usage of the monitored object	Raw data > 90% for three consecutive periods	Major	1. Rectify the fault by referring to **Troubleshooting High Memory Usage**. 2. If the memory usage remains high due to increased workloads, upgrade the instance specifications. For details, see **Changing a DB Instance Class**.
rds039_disk_util	Storage Space Usage	Storage space usage of the monitored object	Raw data > 80% for three consecutive periods	Major	1. Rectify the fault by referring to **Troubleshooting High Storage Space Usage**. 2. If the storage space usage remains high due to increased workloads, scale up the storage space. For details, see **Scaling Storage Space**.

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
rds045_oldest_replication_slot_lag	Oldest Replication Slot Lag	Lagging size of the most lagging replica in terms of WAL data received	Raw data > 20,480 MB for one period	Major	Rectify the fault by referring to **Troubleshooting High Oldest Replication Slot Lag or Replication Lag**.
rds046_replication_lag	Replication Lag	Replication lag	Raw data > 600s for three consecutive periods	Major	
rds083_conn_usage	Connection Usage	Percent of used PostgreSQL connections to the total number of connections	Raw data > 80% for three consecutive periods	Major	1. Evaluate the impact of increased connections on workloads and release unnecessary connections. For details, see **What Do I Do If There Are Too Many Database Connections?** 2. Set the maximum number of connections to an appropriate value. For details, see **What Is the Maximum Number of Connections to an RDS for PostgreSQL Instance?**

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
active_connections	Active Connections	Number of active database connections	Raw data > [vCPUs x 2] for one period	Major	Rectify the fault by referring to **Troubleshooting Abnormal Connections and Active Connections**
oldest_transaction_duration	Oldest Active Transaction Duration	Length of time since the start of the transaction that has been active longer than any other current transaction	Set the threshold as required. Reference value: Raw data > 7,200,000 ms for one period	Major	Rectify the fault by referring to **Troubleshooting Long-Running Transactions**.
oldest_transaction_duration_2pc	Oldest Two-Phase Commit Transaction Duration	Length of time since the start of the transaction that has been prepared for two-phase commit longer than any other current transaction	Set the threshold as required. Reference value: Raw data > 7,200,000 ms for one period	Major	

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
db_max_age	Maximum Database Age	Maximum age of the current database, which is the value of **max(age(datfrozenxid))** in the **pg_database** table	Raw data > 1,000,000,000 for one period	Major	Rectify the fault by referring to **Troubleshooting Database Age Increase Problem**.
slow_sql_three_second	Number of SQL Statements Executed for More Than 3s	Number of slow SQL statements whose execution time is longer than 3s This metric shows an instantaneous value at the collection time instead of an accumulated value within 1 minute.	Set the threshold as required. Reference value: Raw data > [vCPUs x 2] for one period	Major	Rectify the fault by referring to **Troubleshooting SQL Statements That Have Been Executed for 3s or 5s**.

Metric ID	Name	Metric Description	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
slow_sql_five_second	Number of SQL Statements Executed for More Than 5s	Number of slow SQL statements whose execution time is longer than 5s This metric shows an instantaneous value at the collection time instead of an accumulated value within 1 minute.	Set the threshold as required. Reference value: Raw data > [vCPUs x 2] for one period	Major	
inactive_logical_replication_slot	Inactive Logical Replication Slots	Number of inactive logical replication slots	Raw data > 1 for three consecutive periods	Major	Rectify the fault by referring to **Troubleshooting Inactive Logical Replication Slots**.

20 Security Best Practices

PostgreSQL has earned a reputation for reliability, stability, and data consistency, and has become the preferred choice as an open-source relational database for many enterprises. RDS for PostgreSQL is a cloud-based web service that is reliable, scalable, easy to manage, and immediately ready for use.

Make security configurations from the following dimensions to meet your service needs.

- **Configuring the Maximum Number of Connections to the Database**
- **Configuring the Timeout for Client Authentication**
- **Configuring SSL and Encryption Algorithm**
- **Configuring Password Encryption**
- **Configuring Appropriate pg_hba Rules**
- **Disabling the Backslash Quote**
- **Periodically Checking and Deleting Roles That Are No Longer Used**
- **Revoking All Permissions on the public Schema**
- **Setting a Proper Password Validity Period for a User Role**
- **Configuring the Log Level to Record SQL Statements That Cause Errors**
- **Configuring Least-Privilege Permissions for Database Accounts**
- **Enabling Data Backup**
- **Enabling Database Audit**
- **Avoiding Binding an EIP to Your RDS for PostgreSQL Instance**
- **Updating the Database Version to the Latest**
- **Configuring the Delay for Account Authentication Failures**

Configuring the Maximum Number of Connections to the Database

The **max_connections** parameter specifies the maximum concurrent connections allowed in a database. If the value of this parameter is large, the RDS for PostgreSQL database may request more System V shared memory or semaphore. As a result, the requested shared memory or semaphore may exceed the default value on the OS. Set **max_connections** based on service complexity. For details, see **Instance Usage Suggestions**.

Configuring the Timeout for Client Authentication

The **authentication_timeout** parameter specifies the maximum duration allowed to complete client authentication, in seconds. This parameter prevents clients from occupying a connection for a long time. The default value is 60s. If client authentication is not complete within the specified period, the connection is forcibly closed. Using this parameter can enhance the security of your RDS for PostgreSQL instance.

Configuring SSL and Encryption Algorithm

SSL is recommended for TCP/IP connections because SSL ensures that all communications between clients and servers are encrypted, preventing data leakage and tampering and ensuring data integrity. When configuring SSL, configure the TLS protocol and encryption algorithm on the server. TLSv1.2 and EECDH+ECDSA+AESGCM:EECDH+aRSA+AESGCM:EDH+aRSA+AESGCM:EDH+aDSS+AESGCM:!aNULL:!eNULL:!LOW:!3DES:!MD5:!EXP:!SRP:!RC4 are recommended. For details, see **SSL Connection**.

To configure the TLS protocol and encryption algorithm, use the parameters **ssl_min_protocol_version** and **ssl_ciphers**.

Configuring Password Encryption

Passwords must be encrypted. When you use **CREATE USER** or **ALTER ROLE** to change a password, the password is stored in a system catalog after being encrypted by default. **scram-sha-256** is recommended for password encryption. To change the password encryption algorithm, change the value of **password_encryption**.

The **MD5** option is used only for compatibility with earlier versions. New DB instances use **scram-sha-256** by default.

NOTICE

The modification of **password_encryption** takes effect only after the password is reset.

Configuring Appropriate pg_hba Rules

Appropriate pg_hba rules are crucial to ensure the security of RDS for PostgreSQL instances. You are advised to allow only necessary users and hosts to access your DB instance from specific IP addresses or subnets, and periodically review and update the pg_hba file to ensure that the rules meet your workload requirements. For details, see **Modifying pg_hba.conf**.

Disabling the Backslash Quote

The **backslash_quote** parameter specifies whether a single quotation mark (') in a string can be replaced by a backslash quote (\'). The preferred, SQL-standard way to represent a single quotation mark is by doubling it (''). If client-side code does escaping incorrectly then an SQL-injection attack is possible. You are advised to set

backslash_quote to **safe_encoding** to reject queries in which a single quotation mark appears to be escaped by a backslash, preventing SQL injection risks.

Periodically Checking and Deleting Roles That Are No Longer Used

Check whether all roles are mandatory. Every unknown role must be reviewed to ensure that it is used properly. If any role is no longer used, delete it. To query roles, run the following command:

SELECT rolname FROM pg_roles;

Revoking All Permissions on the public Schema

The **public** schema is the default schema. All users can access objects in it, including tables, functions, and views, which may cause security vulnerabilities. You can run the following command as user **root** to revoke the permissions:

revoke all on schema public from public;

Setting a Proper Password Validity Period for a User Role

When creating a role, you can use the **VALID UNTIL** keyword to specify when the password of the role becomes invalid. If this keyword is ignored, the password will be valid permanently. You are advised to change the password periodically, for example, every three months. To configure a password validity period, run the following command:

CREATE ROLE name WITH PASSWORD *'password'* **VALID UNTIL 'timestamp';**

To check whether a password validity period is configured, run the following command:

SELECT rolname,rolvaliduntil FROM pg_roles WHERE rolsuper = false AND rolvaliduntil IS NULL;

Configuring the Log Level to Record SQL Statements That Cause Errors

The **log_min_error_statement** parameter specifies which SQL statements that cause errors can be recorded in server logs. The SQL statements of the specified level or higher are recorded in logs. Valid values include **debug5**, **debug4**, **debug3**, **debug2**, **debug1**, **info**, **notice**, **warning**, **error**, **log**, **fatal**, and **panic**. The value of **log_min_error_statement** must be at least **error**. For details, see **Log Reporting**.

Configuring Least-Privilege Permissions for Database Accounts

RDS for PostgreSQL allows you to grant role-based permissions to a database account for data and command access. You are advised to create **database accounts** and configure least-privilege permissions for the accounts. If any account permission does not meet the role requirements, update the account permission or **delete** the account. RDS for PostgreSQL has some **built-in accounts**, which are used to provide background O&M services for DB instances and cannot be used or deleted by users.

Enabling Data Backup

When you create an RDS DB instance, an automated backup policy is enabled by default with the retention period set to seven days. You can change the backup retention period as required. RDS for PostgreSQL DB instances support **automated backups** and **manual backups**. You can periodically back up your instance. If the instance fails or data is damaged, **restore it using backups** to ensure data reliability. For details, see **Data Backups**.

Enabling Database Audit

By using the PostgreSQL Audit extension (pgAudit) with your RDS for PostgreSQL instance, you can capture detailed records that auditors usually need to meet compliance regulations. For example, you can use pgAudit to track changes made to specific databases and tables, as well as record users who make such changes and many other details. pgAudit is disabled by default. Enable it as required. For details, see **Using pgAudit**.

Avoiding Binding an EIP to Your RDS for PostgreSQL Instance

Do not deploy your instance on the Internet or in a demilitarized zone (DMZ). Instead, deploy it on a Huawei Cloud private network and use routers or firewalls to control access to your instance. Do not bind an EIP to your instance to prohibit unauthorized access and DDoS attacks from the Internet. If you have bound an EIP to your instance, you are advised to unbind it. If you do need an EIP, **configure security group rules** to restrict the source IP addresses that can access your instance.

Updating the Database Version to the Latest

For PostgreSQL versions that are no longer maintained by the PostgreSQL community, the RDS for PostgreSQL **product lifecycle** will be released accordingly. Using an earlier version may pose security risks. Running the software of the latest version can protect the system from certain attacks. You can upgrade **the minor version** or **the major version** of your DB instance as required.

Configuring the Delay for Account Authentication Failures

By default, RDS for PostgreSQL instances have a built-in auth_delay extension. auth_delay causes the server to stop for a short period of time before an authentication failure message is returned, making it more difficult to crack the database password. To configure the delay for account authentication failures, change the value of the **auth_delay.milliseconds** parameter (which indicates the number of milliseconds to wait before reporting an authentication failure) by referring to **Modifying Parameters of an RDS for PostgreSQL Instance**. The default value of this parameter is **3000**.

