

Set Up Highly-available PostgreSQL Replication Cluster on Ubuntu 20.04 Server

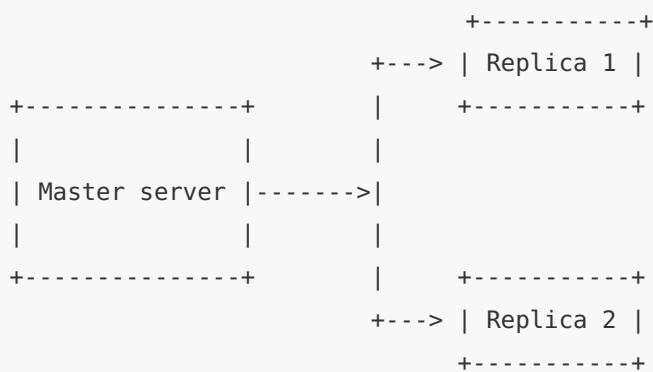
Learn how to set up a highly-available PostgreSQL replication cluster on Ubuntu 20.04 Server to ensure data redundancy and minimize downtime for your database.

Contents

01	Introduction	3
02	Prerequisites	3
03	Understanding Different Replication Models Supported by PostgreSQL	4
04	1. Configure the Master Node	6
05	2. Configure the Replica Node	9
06	3. Test the PostgreSQL Replication	11
07	Conclusion	13

Introduction

To ensure high availability, reliability, and fault-tolerance, the PostgreSQL database server supports replication. This is a process of copying data in real-time across multiple servers to create recovery backups. In this kind of setup, the primary node is referred to as a **master** and the servers receiving the replicated data are known as **replicas** or **standby** nodes as illustrated below.



In most cases, you'll set up PostgreSQL standby replicas to ensure data redundancy and prevent single-point-of-failure(SPOF). In the event that a master server fails, you can promote any of the replicas to become the new master. By design, PostgreSQL allows you to run replicas in **hot standby** mode. This is the ability of the standby nodes to accept connections and read-only operations. This allows you to share the burden of read-intensive operations across your entire cluster.

This guide takes you through the process of configuring a PostgreSQL replication cluster on your Ubuntu 20.04 server.

Prerequisites

To follow along with this tutorial, ensure you have a set of two Ubuntu 20.04 servers in the same data center configured with [Vultr private networking](#). This tutorial uses the following private IP addresses for the two servers.

- **server-1:** Master 10.106.0.1

- **server-2:** Replica 1 10.106.0.2

Each of the above servers should have:

- A [non-root user with sudo privileges](#).
- A [PostgreSQL server installed and configured with a password](#).

Understanding Different Replication Models Supported by PostgreSQL

In PostgreSQL, there are many different approaches that you can use to replicate data across multiple servers. Each method comes with its own pros and cons. However, since it is not feasible to run the PostgreSQL database in a single-server mode for mission-critical applications, you should weigh each solution and choose the best approach depending on your use case.

Replication Methods

- **Physical replication.** This is a file/disk-based replication that occurs at the hardware level. In this approach, any changes made to the master server's data files are mirrored to the other replicas' system files. The changes in the standby nodes must be done in the same order they arrive to ensure consistency across the database cluster. Physical replication is simple to implement, well-tested, and efficient since it does not require special handling. However, this approach is not suitable for multi-master replication and it is not bandwidth-friendly.
- **Logical replication.** This involves copying database objects in a row-based model only for committed transactions. Since this method only sends incremental data in a selective manner, it is very efficient in terms of bandwidth costs. It is also suitable for multi-master replications. The most common approach to achieve logical replication in PostgreSQL is to use the streaming replication model. This approach uses the PostgreSQL

transaction log (WAL) files to replicate data in a cluster. You can run the PostgreSQL streaming replication into modes as explained below.

- **Synchronous mode:** In this mode, the master server must wait for the first available replica server to receive and persist the transaction log file before reporting a successful `COMMIT` operation. This is useful in high-availability setups although there might be a slight delay before a transaction is committed across the replica.
- **Asynchronous mode:** The master server doesn't have to wait for an acknowledgment from the replica server(s) before reporting a successful `COMMIT` operation. This approach is faster but if the master server fails before the data is replicated to the standby nodes, this may result in data loss.

While there is no single replication method that works for every use case, here are some helpful tips that you should consider when choosing the right method.

- Use either `physical` replication or `logical` **asynchronous** replication for backups and disaster recovery setups.
- Use `logical` asynchronous replication if you want to take the read-only load off from the primary server. For instance, to run data analytics from the hot standby nodes.
- Use `logical` **synchronous** replication for high-availability clusters and in circumstances where you want **zero data loss** but at a **reduced** performance.
- Use `logical` **asynchronous** replication for high-availability clusters that require **better performance** but can tolerate some **low data loss**.

In this guide, you'll set up a `logical` **asynchronous** replication using the PostgreSQL transaction log (WAL) method that implements the streaming replication model.

1. Configure the Master Node

SSH to the master server(10.106.0.1) and follow the steps below to make configuration changes.

1. Log in to the PostgreSQL database server as `postgres` user. This is the default super-user account.

```
$ sudo -u postgres psql
```

2. Enter the password for the `postgres` user and press Enter to proceed. Next, issue the `CREATE ROLE` command below to set up a dedicated `repl_user` account with `REPLICATION` privileges. Replace `EXAMPLE_PASSWORD` with a strong value. Later, your replica node will use the credentials of the `repl_user` to connect to the master node to fetch replication data.

```
postgres-# CREATE ROLE repl_user WITH REPLICATION LOGIN PASSWORD  
'EXAMPLE_PASSWORD';
```

Output.

```
CREATE ROLE
```

3. Log out from the PostgreSQL database.

```
postgres-# \q
```

4. Next, use `nano` to open the default PostgreSQL configuration file.

```
$ sudo nano /etc/postgresql/12/main/postgresql.conf
```

5. With the above file open, locate the `listen_addresses` directive. This setting allows you to specify the interface under which the server listens for connections. For this setup, you want the database server to listen on the private IP address interface.

```
...  
  
#listen_addresses = 'localhost' ... # what IP address(es) to listen on;  
  
...
```

6. Uncomment the line above by removing the pound `#` symbol at the beginning. Then replace `localhost` with your master server's private IP address(For example, `10.106.0.1`).

```
listen_addresses = '10.106.0.1'
```

7. Next locate `wal_level`. This setting determines how much information is written to the Write Ahead Log(WAL) file.

```
...  
  
#wal_level = replica ... # minimal, replica, or logical  
  
...
```

8. Uncomment the line and change the value from `replica` to `logical`. This enables PostgreSQL streaming replication.

```
...  
  
wal_level = logical  
  
...
```

9. Next, find the `wal_log_hints` directive. By default, this value is `off`.

```
...  
  
#wal_log_hints = off ... # also do full page writes of non-critical updates  
  
...
```

10. Enable the `wal_log_hints` setting by removing the `#` symbol and changing its value to `on`. This allows the PostgreSQL server to write the entire content of each disk page to the WAL file. This helps in the recovery process in case the standby node goes out of sync with the master server.

```
...  
  
wal_log_hints = on  
  
...
```

11. Save and close the `/etc/postgresql/12/main/postgresql.conf` file when you're through with editing.
12. Next, you'll make a few changes to the `/etc/postgresql/12/main/pg_hba.conf` file still under the master node. In this file, you will include the IP address of the replica node so that it can connect to the master node. Open the `/etc/postgresql/12/main/pg_hba.conf` file using `nano` for editing purposes.

```
$ sudo nano /etc/postgresql/12/main/pg_hba.conf
```

13. Next, append the entries below at the bottom of the file to allow your replica node (`10.106.0.2`) to connect to the master under the `repl_user` account.

```
host      replication      repl_user      10.106.0.2/32      md5
```

14. Save and close the `/etc/postgresql/12/main/pg_hba.conf` file. Then, restart the PostgreSQL server on the master node to apply the new changes.

```
$ sudo systemctl restart postgresql
```

15. You've now set up your master server and you can now connect a standby server to start replicating data in real-time.

2. Configure the Replica Node

The replica server(`10.106.0.2`) needs to start somewhere before it can start replicating data from the master. The best way to initialize or bootstrap the replica server is by copying over the master server's data files using the `pg_basebackup` utility. SSH to the **replica server** and follow the process below to create a base copy of the master server's data.

1. Ensure the PostgreSQL server is not running on the replica server by stopping the `postgresql` service.

```
$ sudo systemctl stop postgresql
```

2. Still on the replica server, use the Linux `rm` command to remove all the files in the PostgreSQL data directory `/var/lib/postgresql/12/main/`.

```
$ sudo rm -rv /var/lib/postgresql/12/main/
```

3. Next, run the `pg_basebackup` on the replica server to copy data from the master server using the following options.

```
$ sudo pg_basebackup -h 10.106.0.1 -U repl_user -X stream -C -S replica_1 -v -R -W -D /var/lib/postgresql/12/main/
```

4. The `pg_basebackup` options explained:

- **-h:** You're using this option to specify the host. In this case, this is the private IP address of the master node(`10.106.0.1`) where you want to fetch the backup data from.
- **-U:** This option allows you to specify the replication user account(`repl_user`) which you created on the master node.
- **-D:** This option allows you to include the directory under which you want to export the backup files. In this case, you're placing the data under the `/var/lib/postgresql/12/main/` in the replica node.

- **-X:** This option together with a value of `stream` allows you to instruct the `pg_basebackup` utility to stream and include the WAL files in the backup.
- **-C:** This allows you to create a replication slot before starting the backup. You've named the slot `replica_1` by including it after the `-s` option which specifies the replication slot name.
- **-v:** This enables verbose mode in order to output the progress of the backup process.
- **-R:** This allows you to create a recovery configuration file and a master node connection settings file under the data directory. You'll find these files named `standby.signal` and `postgresql.auto.conf`.
- **-W:** This option prompts you to enter a password for the `repl_user` user.

5. Once you've pulled the database files from the master server using the `pg_basebackup` utility, you should get the following output.

```
pg_basebackup: initiating base backup, waiting for checkpoint to complete
pg_basebackup: checkpoint completed
pg_basebackup: write-ahead log start point: 0/2000028 on timeline 1
pg_basebackup: starting background WAL receiver
pg_basebackup: created replication slot "replica_1"
pg_basebackup: write-ahead log end point: 0/2000100
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: syncing data to disk ...
pg_basebackup: base backup completed
```

6. Next, run the following command on your replica node to ensure, the data files are correctly owned by the `postgres` user.

```
$ sudo chown postgres -R /var/lib/postgresql/12/main/
```

7. You now have the correct data files in your replica node, you can now bring it up as a hot standby node.

```
$ sudo systemctl start postgresql
```

8. In the next step, you'll test the replication settings.

3. Test the PostgreSQL Replication

In this step, you'll connect to the PostgreSQL database server on the master node, create a database, a table, and insert some records. Then, you'll connect to the replica node to check whether the database will be replicated. Use different SSH sessions for the master and replica in this step to avoid confusion.

1. Log in to the PostgreSQL server on the master node.

```
$ sudo -u postgres psql
```

2. Enter your `postgres` password and press Enter to proceed. Next, create a `test_db` by running the following command.

```
postgres=# CREATE DATABASE test_db;
```

3. Then, switch to the new database.

```
postgres=# \c test_db;
```

Output.

```
You are now connected to database "test_db" as user "postgres".
```

4. Create a new `products` table under the `test_db` database.

```
test_db=# CREATE TABLE products (  
        product_id SERIAL PRIMARY KEY,  
        product_name VARCHAR (50)  
    );
```

Output.

```
CREATE TABLE
```

5. Next, `INSERT` three records in the `products` table.

```
test_db=# INSERT INTO products(product_name) VALUES ('LEATHER JACKET');
          INSERT INTO products(product_name) VALUES ('WINTER HOODIE');
          INSERT INTO products(product_name) VALUES ('BROWN WALLET');
```

Output.

```
INSERT 0 1
...
```

6. Now in a new SSH terminal window, connect to the PostgreSQL server on the replica node as `postgres`.

```
$ sudo -u postgres psql
```

7. Enter the password for the `postgres` user and press Enter to proceed. Next, attempt switching to the `test_db`.

```
postgres=# \c test_db;
```

8. You'll get the following confirmation message.

```
You are now connected to database "test_db" as user "postgres".
```

9. Still on the replica node, query the `products` table.

```
test_db=# SELECT
           product_id,
           product_name
         FROM products;
```

10. You should now get the records that you inserted in the `products` table via the master node.

```
product_id | product_name
-----+-----
          1 | LEATHER JACKET
          2 | WINTER HOODIE
          3 | BROWN WALLET
(3 rows)
```

11. Your replica node is running in **hot standby mode** and can only accept **read-only** operations. You may attempt to submit the following `INSERT` statement in the replica node to check this behavior.

```
test_db=# INSERT INTO products(product_name) VALUES ('RED TSHIRT');
```

12. Since the replica node can not accept write operations, you should get the following error. However, the same command will succeed in the master node.

```
ERROR:  cannot execute INSERT in a read-only transaction
```

13. Your PostgreSQL replication setup is now working as expected.

Conclusion

In this guide, you've learned various replication methods supported by the PostgreSQL database server together with their pros, cons, and use-cases. Towards the end of the guide, you've set up a replication cluster using the PostgreSQL stream processing model and you're able to replicate data across two servers. Use this guide to implement real-time PostgreSQL backups, disaster recovery clusters, and high-availability setups.



VULTR

