AlloyDB Omni Configuration Guide

Table of contents

Table of contents 1
Configure AlloyDB Omni 3
- Enable AlloyDB Omni to automatically start 3
- Enable extensions on AlloyDB Omni 4

Configure Backups 5
- Overview 5
- Configure pgBackRest 5
  - Before you begin 5
    - File system paths 5
- Configure pgBackRest with local backups 6
  - Configure the database for continuous backups 10
- Configure backups for your clusters 12

Back up and restore data 12
- Before you begin 12
  - Requirements 13
- Prepare your environment 14
- Verify your source backups 14
- Prepare your target server 15
- Restore on the target server 15
- Verify the data restore 19

Advanced backup and recovery 20
- Before you begin 20
- Recovery scenarios 20
  - Point-in-time recovery (PITR) 22
  - Recover specific databases 27
  - Recover to a restore point 31
  - Recover to a specific log sequence number 35

High Availability and DR 39
- What is database resilience? 39
- How high availability works 40
- High availability with Patroni, etcd and HAProxy 41
  - Before you begin 41
  - Installation 42

Deploy the solution 43
Install a client on a machine in the same network 43
Test your high availability setup 45
Test your Patroni setup 45
Test your HAProxy setup 47
Test the automatic failover operation 48
Fallback considerations 51
Security and Compliance 52
Manage AlloyDB Omni database roles 52
About user roles in AlloyDB Omni 52
AlloyDB's predefined PostgreSQL roles 52
Data Migration 53
PostgreSQL to AlloyDB Omni 53
Oracle to AlloyDB Omni 53
Observability 54
Observability Scripts 54
Observability Tools 58
Using Grafana, Prometheus, and Postgres Exporter 58
Installing Postgres Exporter 58
Installing Prometheus 60
Installing Grafana 62
Standard Addresses for Postgres Exporter, Prometheus, and Grafana 63
Load a Dashboard to Grafana 63
Create a data source 64
Create a dashboard 66
Perfsnap 70
Recommended extensions for observability 70
Columnar engine observability 70
General columnar engine observability scripts: 71
Index advisor 73
Manage your AlloyDB Omni configuration 73
Default extensions to use 73
Log location 73
Add extensions to AlloyDB Omni 74

Configure AlloyDB Omni

Enable AlloyDB Omni to automatically start

If you are using Docker, AlloyDB Omni can be configured to automatically restart by providing Docker with a restart policy. Add `--restart RESTART_POLICY` to the `docker run` command, or update a running container's configuration:

Unset

```
docker update --restart <RESTART_POLICY> <CONTAINER_NAME>
```

Replace the following:

<RESTART_POLICY> Docker supports four restart policy options:

- `no`: Do not restart the container (default)
- `on-failure[:max-retries]`: Restart the container if it exits due to an error. The number of retries can be limited with the `max-retries` option.
- `always`: Always restart the container if it stops.
- `unless-stopped`: Similar to always, except that the docker daemon doesn't restart the container if it was shut down when the daemon restarts.

<CONTAINER_NAME>: The name to assign this new AlloyDB Omni container in your host machine's container registry, for example, `my-omni-1`

Podman does not provide a restart policy option, but containers can be configured to restart using `systemd`. Follow these instructions to configure a `systemd` service to run AlloyDB Omni:

1. Generate a `systemd` configuration file for the container. The command below generates a configuration file in the working directory with a name of container-$<CONTAINER_NAME>.service

Unset

```
podman generate systemd --new --files --name <CONTAINER_NAME>
```

Replace the following:

<CONTAINER_NAME>: The name to assign this new AlloyDB Omni container in your host machine's container registry, for example, `my-omni-1`

2. Copy the configuration file to the `/etc/systemd/system` directory and reload the services managed by `systemd`:

```
Unset
sudo cp -Z SYSTEMD_CONFIG_FILE /etc/systemd/system
sudo systemctl daemon-reload
sudo systemctl enable container-$<CONTAINER_NAME>.service
```

3. Start, stop, and check the status of the service using the commands below:

```
Unset
systemctl start container-$<CONTAINER_NAME>.service
systemctl stop container-$<CONTAINER_NAME>.service
systemctl status container-$<CONTAINER_NAME>.service
```

The `systemd` subsystem doesn't show an AlloyDB Omni instance that was manually started with `podman` command as running. The instance must be started using `systemd` in order to be managed by `systemd`.

Enable extensions on AlloyDB Omni

The list of extensions available in AlloyDB Omni is available in [Support database extensions](). Although PostGIS and Orafce are not included with AlloyDB Omni, they can both be installed by following instructions:

- [Install PostGIS for AlloyDB Omni]()
- [Install Orafce for AlloyDB Omni]()

Installed extensions are enabled using standard PostgreSQL `CREATE EXTENSION` statements as detailed in [Enable an extension]().

Configure Backups

Overview

pgBackRest is the recommended backup manager for AlloyDB Omni. Unlike the native PostgreSQL utilities pg_dump and pg_dumpall, which extracts database data logically, pgBackRest performs physical database cluster backups.

pgBackRest can do the following:

- Perform full, incremental, or differential backups
- Write to local disk, remote disk, or cloud storage destinations
- Parallelize and encrypt backups
- Compression and checksums (done in-stream) options
- Automatic management and expiration of backups

The AlloyDB Omni Docker container includes the pgBackRest utility and therefore, pgBackRest can be used to:

- Perform physical backups and restorations of your AlloyDB Omni database clusters.
- Build AlloyDB clone clusters–either to a current or specific point-in-time.
- Perform selective restores of only specific databases from within your AlloyDB Omni database cluster.

Configure pgBackRest

Use the information in this section to configure pgBackRest.

Before you begin

Before configuring AlloyDB Omni to work with pgBackRest, you need to have AlloyDB Omni installed and running on a server that you control.

File system paths

When using pgBackRest with AlloyDB Omni, refer to file system paths from the container's perspective.
The pgBackRest software included with AlloyDB Omni runs in the same Docker container as AlloyDB Omni. Because of this, all of the file system paths that you provide pgBackRest through its configuration file or as command line arguments are locations on the container's file system, and not your host machine's file system.

Many of the commands and examples on this page refer to your data directory as `/var/lib/postgresql/data`, regardless of the location of your data directory on your host system. This is because AlloyDB Omni mounts your data directory to `/var/lib/postgresql/data` on its containerized file system. As a result, you can use the data directory as a location to store pgBackRest configuration and repositories without further setup.

Note: You can replace `/var/lib/postgresql/data` with the path to the PGDATA directory where your database cluster is stored.

If you want to configure the containerized pgBackRest to read from or write to directories on your host machine's file system outside of your AlloyDB Omni data directory, then you need to make these directories available to the container.

Configure pgBackRest with local backups

Before running pgBackRest, configuration is required. The guide uses the following as an example:

Create the user name postgres to run the Omni container:

Unset
```
sudo useradd --uid 2345 --user-group --no-create-home postgres
```

Configure the data and backup directories:

Unset
```
mkdir alloydb-data
mkdir alloydb-backup

sudo chown -R postgres:postgres alloydb-data/
sudo chmod 770 alloydb-data

sudo chown -R postgres:postgres alloydb-backup/
sudo chmod 770 alloydb-backup
```

Start the docker container:

```
Unset
sudo docker run --detach \
  --name pg-service \
       -e POSTGRES_PASSWORD=${YOUR_PASSWORD} \
       -e PGDATA=/var/lib/postgresql/data/pgdata \
       -v "$PWD/alloydb-data":/var/lib/postgresql/data \
       -v "$PWD/alloydb-backup":/var/lib/postgresql/backup \
       -p 5432:5432 \
       -u 2345 \
google/alloydbomni
```

pgBackRest creates subdirectories for each backup taken and includes a plain-text manifest file.

pgBackRest uses the term *stanza* to refer to the configuration for a PostgreSQL database cluster. The stanza name is arbitrary and doesn't need to match the host server, PostgreSQL cluster, or database name. The pgBackRest documentation suggests naming the stanza after the cluster's function. For this example, we will use the stanza name"**omni**. You can adjust the stanza name to suit your environment.

The *repository* is where backups are written. pgBackRest supports writing to more than one repository in a given stanza. Most configuration parameters related to repositories are indexed with a numeric value, for example, **repo1-**. Parameters related to PostgreSQL clusters are also, independently, indexed, for example, **pg1-**.

pgBackRest leverages a configuration file, called **pgbackrest.conf**, to hold global and stanza-specific parameters.

Build and initialize a configuration file for backing up your AlloyDB Omni cluster using the following commands from the AlloyDB Omni host server:

```
Unset

echo -e "
[global]

# Paths (all mandatory):
repo1-path=/var/lib/postgresql/backup/backups
spool-path=/var/lib/postgresql/backup
```

```
lock-path=/var/lib/postgresql/backup

# Retention details:
repo1-retention-full=3
repo1-retention-full-type=count
repo1-retention-diff=16

# Force a checkpoint to start backup immediately:
start-fast=y

# Logging parameters:
log-path=/var/lib/postgresql/backup/backups
log-level-console=info
log-level-file=info

# Recommended ZSTD compression:
compress-type=zst

# Other performance parameters:
archive-async=y
archive-push-queue-max=1024MB
archive-get-queue-max=256MB
archive-missing-retry=y

[global:archive-push]
process-max=2

[global:archive-get]
process-max=2

[omni]
pg1-user=postgres
pg1-socket-path=/tmp
pg1-path=/var/lib/postgresql/data/pgdata
" | sudo -u postgres tee ~/alloydb-backup/pgbackrest.conf

#  Verify:
sudo ls -l ~/alloydb-backup/pgbackrest.conf
sudo cat ~/alloydb-backup/pgbackrest.conf
```

Some parameters are mandatory, but can be adjusted to meet your specific requirements if needed, such as the following:

- repo1-path: the directory location where the backups are written to–a location visible to both the host server and the Docker container is recommended. The default is **/var/lib/pgbackrest** which is visible only inside of the container.
- log-path: if you want to write log files to a separate location–not intermixed with the backups themselves–adjust this parameter. The default is **/var/log/pgbackrest** which is visible only inside of the container.
- repo1-retention-full: the number of full backups to retain. The default (unset) causes a warning.
- repo1-retention-full-type: whether the retention is measured by count or time period (days).
- repo1-retention-diff: the number of differential backups to retain.

Other, non-critical but recommended parameter settings that are compatible with AlloyDB Omni in the configuration file include the following:

- log-level-console: the level of logging written to the screen (STDOUT) when running pgBackup commands. You can adjust this to meet your needs in the configuration file, or override this value with the **--log-level-console** command line argument. The default is **warn**.
- start-fast: forces a checkpoint to start the backups quickly. The default is **n**.
- archive-async: push WAL segment files asynchronously for performance. The default is **n**.
- process-max: the maximum number of processes to use for compression and transfer. Typically set to `max_cpu/4` on a primary or `max_cpu/2` on a standby cluster. The default is **1**.
- compress-type: compression algorithm to use. The default is **gz**.

These parameters can also be adjusted to meet your specific needs.

To change any parameters, edit the configuration file from the AlloyDB Omni host server:

```
Unset
sudo vi ~/alloydb-backup/pgbackrest.conf
```

Many other pgBackRest configuration parameters exist and can be adjusted. This documentation covers only the parameters mandatory for the default AlloyDB Omni configuration and some recommended parameter settings. Refer to the pgBackRest Configuration Reference online documentation for the full list of configuration parameters and adjust to meet your specific needs.

After configuring pgBackRest, the target repositories where backups are written to must be initialized by creating the stanza–which uses the parameters as set in the configuration file.

Note: Since, within the Docker container, the pgBackRest configuration file is not in the default location, the file's location is provided as a command line argument.

Create the stanza using the **stanza-create** command:

```
Unset
docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni stanza-create
```

Sample output:

```
Unset
$ docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni stanza-create

2024-06-14 20:22:33.109 P00   INFO: stanza-create command begin 2.48:
--config-path=/var/lib/postgresql/backup --exec-id=218-6c78c96f
--lock-path=/var/lib/postgresql/backup --log-level-console=info
--log-level-file=info --log-path=/var/lib/postgresql/backup
--pg1-path=/var/lib/postgresql/data/pgdata --pg1-socket-path=/tmp
--pg1-user=postgres --repo1-path=/var/lib/postgresql/backup --stanza=omni
2024-06-14 20:22:33.714 P00   INFO: stanza-create for stanza 'omni' on repo1
2024-06-14 20:22:33.723 P00   INFO: stanza-create command end: completed
successfully (615ms)
```

Configure the database for continuous backups

To enable online, physical backups, some fundamental PostgreSQL parameters must be configured in your AlloyDB Omni cluster.

Specifically:

- `archive_command='pgbackrest --config-path=<location> --stanza=<name> archive-push %p'`
- `archive_mode=on`
- `max_wal_senders=10`
- `wal_level='replica'` (or `'logical'`)

Assuming a default AlloyDB Omni installation, only the **archive_command** and **archive_mode** parameters must be added. If you have adjusted the **max_wal_senders** or **wal_level** parameter yourself, you may need to also update those, or revert them back to the AlloyDB Omni defaults.

The PostgreSQL parameters can be adjusted using:

```
Unset
docker exec pg-service psql -h localhost -U postgres \
  -c "ALTER SYSTEM SET archive_command='pgbackrest
--config-path=/var/lib/postgresql/backup --stanza=omni archive-push %p';" \
  -c "ALTER SYSTEM SET archive_mode=on;"
```

After changing the host-based authentication, restart your AlloyDB Omni cluster:

```
Unset
docker restart pg-service
```

After restarting, confirm that the necessary parameters are all set appropriately using:

```
Unset
docker exec pg-service psql -h localhost -U postgres -c "
SELECT name, setting
  FROM pg_catalog.pg_settings
 WHERE name IN ('archive_command',
                'archive_mode',
                'max_wal_senders',
                'wal_level')
 ORDER BY name;
"
```

Sample output:

```
Unset
$ docker exec pg-service psql -h localhost -U postgres -c "
> SELECT name, setting
>   FROM pg_catalog.pg_settings
>  WHERE name IN ('archive_command',
>                 'archive_mode',
>                 'max_wal_senders',
```

```
>                  'wal_level')
>   ORDER BY name;
> "
      name       |                                      setting
-----------------+-------------------------------------------------------------
----------------------
 archive_command | pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni archive-push %p
 archive_mode    | on
 max_wal_senders | 10
 wal_level       | replica
(4 rows)
```

At this point, your AlloyDB Omni cluster is ready to be used with pgBackRest.

Configure backups for your clusters

For information on setting up pgbackrest, see Set up pgbackrest.

Back up and restore data

Before you begin

Before following these steps, ensure that you have the following:

- A new server with AlloyDB Omni installed. This server is referred to as the *target*. A *clone* is created by restoring an existing AlloyDB Omni database cluster onto the target server.
- pgBackRest configured against the AlloyDB Omni database cluster on the target server.
- Sufficient disk space on the target server to hold the restored AlloyDB Omni database cluster and the associated backups.
- Access to your primary AlloyDB Omni database cluster and existing pgBackRest backups, which is referred to as the *source*.
- An established and secured network between the servers.

Requirements

Ensure that you have the same major version of PostgreSQL and pgBackRest installed on both your source and target servers. Google might occasionally update the AlloyDB Omni version of either, or both, in the latest Docker image provided in the Google Container Registry (GCR). If your target server was provisioned at a later date than the source server or the other way around, it is possible that their versions differ.

Check on your version of PostgreSQL:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT version();"
```

Check on the version of pgBackRest included in the AlloyDB Omni Docker container:

```
Unset
docker exec CONTAINER_NAME pgbackrest version
```

Sample outputs:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
version();"
                                         version
--------------------------------------------------------------------------------
----------
 PostgreSQL 15.5 on x86_64-pc-linux-gnu, compiled by Debian clang version
12.0.1, 64-bit
(1 row)
```

If your target server has a different version of PostgreSQL or pgBackRest or both then you need to provision a new target server with a matching version. If this is not possible, then you need to use an alternative method, such as the PostgreSQL included pg_dump or pg_dumpall utilities to copy your databases across versions. For more information, see [Export a DMP file](#).

Prepare your environment

AlloyDB Omni pgBackRest backup files are owned by the postgres OS user and must be copied from the source server to the target. Therefore, ssh connectivity between the two servers, usually as the postgres user, must be established.

Assuming that a passwordless ssh login has been established, verify if a passwordless `ssh` login has been established using a simple test command from your AlloyDB Omni target host such as:

```
Unset
sudo -u postgres ssh postgres@${SOURCE_SERVER_IP} whoami
```

Verify your source backups

If existing backups of your AlloyDB Omni source database cluster are already available, taking a new backup is not necessary. pgBackRest restores the most recent backup and applies all available WAL segment file backups to make the cloned cluster on the target server as current as possible.

If needed, you can make a new full backup of your source AlloyDB Omni cluster:

```
Unset
docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni --type=full backup
```

List and verify the backups:

```
Unset
docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni info
```

To ensure that the most recent transactions are included, we recommend performing a WAL segment file log switch:

```
Unset
docker exec pg-service psql -h localhost -U postgres -c "SELECT
pg_switch_wal();"
```

Prepare your target server

Copy the pgBackRest configuration file from your source server:

```
Unset
sudo -u postgres scp
${SOURCE_SERVER_IP}:/home/$USER/alloydb-backup/pgbackrest.conf
/home/$USER/alloydb-backup/
```

If not already installed, install rsync using your normal processes for installing software from your package manager. Copy the pgBackRest repository (directory) and its contents from the source server–using the Linux rsync utility is recommended:

```
Unset
sudo -u postgres rsync -avzhrP
${SOURCE_SERVER_IP}:/home/$USER/alloydb-backup/backups
/home/$USER/alloydb-backup/
```

Restore on the target server

Before restoring, list the current databases in your target AlloyDB Omni cluster–this should show the default list of databases without any of your application data.

Warning: If you see application data, stop and check that you are working against the correct system.

The AlloyDB Omni Docker container, and hence PostgreSQL cluster, must be up and running on the target server at this point. List the databases:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "\l"
```

Sample output showing a default (initialized) AlloyDB Omni cluster:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "\l"
```

```
                                                              List of databases
      Name       |    Owner     | Encoding | Collate | Ctype | ICU Locale |
Locale Provider |       Access privileges
-----------------+--------------+----------+---------+-------+------------+----
-------------+-------------------------------
 alloydbadmin    | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
|
 alloydbmetadata | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
| alloydbadmin=CTc/alloydbadmin +
                 |              |          |         |       |            |
| alloydbmetadata=c/alloydbadmin
 postgres        | postgres     | UTF8     | C       | C     | und-x-icu  | icu
|
 template0       | postgres     | UTF8     | C       | C     | und-x-icu  | icu
| =c/postgres                   +
                 |              |          |         |       |            |
| postgres=CTc/postgres
 template1       | postgres     | UTF8     | C       | C     | und-x-icu  | icu
| =c/postgres                   +
                 |              |          |         |       |            |
| postgres=CTc/postgres
(5 rows)
```

Use pgBackRest to restore the database into a new location called `data-RESTORED`:

Unset

```
docker exec CONTAINER_NAME pgbackrest --config-path=/var/lib/postgresql/backup
--pg1-path=/var/lib/postgresql/data/data-RESTORED --stanza=omni restore
```

Sample output:

Unset

```
$ docker exec CONTAINER_NAME pgbackrest
--config-path=/var/lib/postgresql/backup
--pg1-path=/var/lib/postgresql/data/data-RESTORED --stanza=omni restore
```

```
2024-06-14 20:53:09.641 P00   INFO: restore command begin 2.48:
--config-path=/var/lib/postgresql/backup --exec-id=270-28543cdb
--lock-path=/var/lib/postgresql/data --log-level-console=info
--log-level-file=info --log-path=/var/lib/postgresql/data
--pg1-path=/var/lib/postgresql/data/data-RESTORED
--repo1-path=/var/lib/postgresql/data --spool-path=/var/lib/postgresql/data
--stanza=omni
2024-06-14 20:53:09.651 P00   INFO: repo1: restore backup set
20240614-202802F_20240614-202846D, recovery will start at 2024-06-14 20:28:46
2024-06-14 20:53:09.651 P00   INFO: remap data directory to
'/var/lib/postgresql/data/data-RESTORED'
WARN: unknown user in backup manifest mapped to current user
2024-06-14 20:53:13.192 P00   INFO: write updated
/var/lib/postgresql/data/data-RESTORED/postgresql.auto.conf
2024-06-14 20:53:13.196 P00   INFO: restore global/pg_control (performed last
to ensure aborted restores cannot be started)
2024-06-14 20:53:13.197 P00   INFO: restore size = 69MB, file total = 2194
2024-06-14 20:53:13.197 P00   INFO: restore command end: completed successfully
(3557ms)
```

Stop the target AlloyDB Omni database server:

Unset
```
docker stop pg-service
```

We recommend archiving the existing data directory by renaming it instead of removing it as a precautionary measure. Archive the existing data directory by renaming it, then move the restored data into the original location:

Unset
```
sudo mv ~/alloydb-data/pgdata ~/alloydb-data/pgdata-OLD

sudo mv ~/alloydb-data/data-RESTORED ~/alloydb-data/pgdata
```

The pgBackRest `restore` operation adds recovery parameters to the `postgresql.auto.conf` file and creates a `recovery.signal` file.

These parameters reference the restored directory name. Now that the restored data is back into a directory called `pgdata`, the `postgresql.auto.conf` file must be updated accordingly:

```
Unset
sudo sed -i 's|data-RESTORED|pgdata|'
~/alloydb-data/pgdata/postgresql.auto.conf
```

Restart the AlloyDB cluster. PostgreSQL will recover the database files and apply a redo of the logs for recovery automatically.

```
Unset
docker restart CONTAINER_NAME
```

Check for confirmation that the recovery is complete.

```
Unset
sudo docker logs CONTAINER_NAME |& grep -A4 'archive recovery complete'
```

Sample output:

```
Unset
$ sudo docker logs CONTAINER_NAME |& grep -A4 'archive recovery complete'

2024-06-14 21:59:36.628 UTC [16] LOG:  [xlog.c:6325]  archive recovery complete
2024-06-14 21:59:36.628 UTC [16] LOG:  [xlog.c:6442]  Setting InRecovery=false
- PG ready for connections
2024-06-14 21:59:36.629 UTC [14] LOG:  [xlog.c:7125]  checkpoint starting:
end-of-recovery immediate wait
2024-06-14 21:59:36.808 UTC [14] LOG:  [xlog.c:7278]  checkpoint complete:
wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.171
s, sync=0.002 s, total=0.180 s; sync files=2, longest=0.001 s, average=0.001 s;
distance=16384 kB, estimate=16384 kB
2024-06-14 21:59:36.809 UTC [16] LOG:  [xlog.c:6600]  StartupXLOG Finished
```

Verify the data restore

After restoring your AlloyDB Omni cluster, verify that you see the expected data in your target cluster. For example, check that your application databases are now present:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "\l"
```

Sample output showing that the db1 database is present:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "\l"
                                                            List of databases
      Name       |    Owner     | Encoding | Collate | Ctype | ICU Locale |
Locale Provider |       Access privileges
-----------------+--------------+----------+---------+-------+------------+----
-------------+-------------------------------
 alloydbadmin    | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
|
 alloydbmetadata | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
| alloydbadmin=CTc/alloydbadmin +
                 |              |          |         |       |            |
| alloydbmetadata=c/alloydbadmin
 db1             | postgres     | UTF8     | C       | C     | und-x-icu  | icu
|
 postgres        | postgres     | UTF8     | C       | C     | und-x-icu  | icu
|
 template0       | postgres     | UTF8     | C       | C     | und-x-icu  | icu
| =c/postgres                   +
                 |              |          |         |       |            |
| postgres=CTc/postgres
 template1       | postgres     | UTF8     | C       | C     | und-x-icu  | icu
| =c/postgres                   +
                 |              |          |         |       |            |
| postgres=CTc/postgres
(6 rows)
```

If needed, check the command entered and remove the archived datafiles:

```
Unset

sudo rm -rf ~/alloydb-data/pgdata-OLD
```

If required, start performing, and optionally schedule, pgBackRest backups of your restored target AlloyDB Omni cluster. The stanza does not need to be re-created–pgBackRest can start working against the recovered database and the copied stanza right away.

Advanced backup and recovery

Before you begin

You can familiarize yourself with the basic process for performing recoveries of your AlloyDB Omni database cluster to a secondary server using pgBackRest with the Cloning your AlloyDB Omni database cluster guide. Advanced recovery scenarios are all based on the same general steps, and bring in the same AlloyDB Omni specific recovery procedure differences and uniqueness, as the normal cloning scenario.

Ensure that you have the following:

- A new server with AlloyDB Omni installed. This server is referred to as the *target*.
- pgBackRest configured against your new AlloyDB Omni database cluster on the target server. For more information, see Configuring backups for use with AlloyDB Omni.
- Sufficient disk space on the target server to hold the restored AlloyDB Omni database cluster and the associated backups.
- Access to your primary AlloyDB Omni database cluster and existing pgBackRest backups. The AlloyDB Omni database cluster will be referred to as the “source”.
- An established and secured network between the servers.
- The ability to copy backup files and directories from the source system to the target. For example, passwordless ssh connectivity for the postgres operating system user.

Recovery scenarios

This section describes the process for performing more advanced, and less typical recovery scenarios of your AlloyDB database cluster using pgBackRest.

Specifically:

- Performing a point-in-time recovery (PITR).
- Restoration of specific databases only.

- Recovering to a manually-created *restore point*.
- Recovering to a specific LSN (Log Sequence Number).

Before beginning any recovery, ensure that you have at least one backup of your source AlloyDB Omni database cluster by using pgBackRest with the info command:

```
Unset
docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni info
```

Sample output:

```
Unset
$ docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni info

stanza: omni
    status: ok
    cipher: none
    db (current)
        wal archive min/max (15):
000000010000000000000002/000000010000000000000005
        full backup: 20240614-220839F
            timestamp start/stop: 2024-06-14 22:08:39+00 / 2024-06-14
22:08:43+00
            wal start/stop: 000000010000000000000005 / 000000010000000000000005
            database size: 70.2MB, database backup size: 70.2MB
            repo1: backup set size: 6.5MB, backup size: 6.5MB
```

```
Unset
This document will describe the process for performing more advanced, and less typical
recovery scenarios of your AlloyDB database cluster using pgBackRest.

Specifically:
    Performing a point-in-time recovery (PITR).
    Restoration of specific databases only.
    Recovering to a manually-created "restore point".
    Recovering to a specific LSN (Log Sequence Number).
```

Before beginning any recovery, ensure that you have at least one backup of your source AlloyDB Omni database cluster by using pgBackRest with the info command:

```
docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni info
```

```
Sample output:
```

```
$ docker exec pg-service pgbackrest --config-path=/var/lib/postgresql/backup
--stanza=omni info

stanza: omni
    status: ok
    cipher: none
    db (current)
        wal archive min/max (15):
000000010000000000000002/000000010000000000000005
        full backup: 20240614-220839F
            timestamp start/stop: 2024-06-14 22:08:39+00 / 2024-06-14
22:08:43+00
            wal start/stop: 000000010000000000000005 / 000000010000000000000005
            database size: 70.2MB, database backup size: 70.2MB
            repo1: backup set size: 6.5MB, backup size: 6.5MB
```

Point-in-time recovery (PITR)

A common scenario involves restoring a database onto a secondary or recovery server to a specific point in time. For example, to view or retrieve data from before a user-introduced issue, data corruption, or data loss.

Restoring your source AlloyDB Omni database cluster to a specific point in time on the target server is straightforward when using pgBackRest and simply involves including the **--type** and **--target** arguments.

For example, suppose we leverage the guestbook sample database. In that case, we can insert some rows in the AlloyDB Omni source database and then simulate a failure in the form of an accidental table truncation:

```
Unset
CREATE DATABASE guestbook;
\c guestbook

--  Create the sample table:
CREATE TABLE entries (guestName VARCHAR(255), content VARCHAR(255),
                        entryID SERIAL PRIMARY KEY);

--  Insert some test data:
INSERT INTO entries (guestName, content) values ('first
guest',transaction_timestamp());
SELECT pg_sleep(floor(random()*10)::int);

INSERT INTO entries (guestName, content) values ('second
guest',transaction_timestamp());
SELECT pg_sleep(floor(random()*10)::int);

INSERT INTO entries (guestName, content) values ('third
guest',transaction_timestamp());
--Last sleep to ensure there is some time between the last insert and the
truncate
SELECT pg_sleep(floor(random()*10)::int);

--  Verify the test data:
SELECT * FROM entries;

--  Introduce a failure
TRUNCATE TABLE entries;
SELECT transaction_timestamp();

SELECT * FROM entries;
```

Sample output:

```
Unset
postgres=# \c guestbook
You are now connected to database "guestbook" as user "postgres".
guestbook=#
guestbook=# --  Create the sample table:
guestbook=# CREATE TABLE entries (guestName VARCHAR(255), content VARCHAR(255),
guestbook(#                         entryID SERIAL PRIMARY KEY);
CREATE TABLE
```

```
guestbook=#
guestbook=# --  Insert some test data:
guestbook=# INSERT INTO entries (guestName, content) values ('first
guest',transaction_timestamp());
INSERT 0 1
guestbook=# SELECT pg_sleep(floor(random()*10)::int);
 pg_sleep
----------

(1 row)
guestbook=#
guestbook=# INSERT INTO entries (guestName, content) values ('second
guest',transaction_timestamp());
INSERT 0 1
guestbook=# SELECT pg_sleep(floor(random()*10)::int);
 pg_sleep
----------

(1 row)
guestbook=#
guestbook=# INSERT INTO entries (guestName, content) values ('third
guest',transaction_timestamp());
INSERT 0 1
guestbook=# --Last sleep to ensure there is some time between the last insert
and the truncate
guestbook=# SELECT pg_sleep(floor(random()*10)::int);
 pg_sleep
----------

(1 row)
guestbook=#
guestbook=# --  Verify the test data:
guestbook=# SELECT * FROM entries;
  guestname   |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:31:46.900483+00 |       1
 second guest | 2024-06-03 22:31:46.902606+00 |       2
 third guest  | 2024-06-03 22:31:53.938599+00 |       3
(3 rows)
guestbook=#
guestbook=# --  Introduce a failure
guestbook=# TRUNCATE TABLE entries;
TRUNCATE TABLE
guestbook=# SELECT transaction_timestamp();
```

```
     transaction_timestamp
-------------------------------
 2024-06-03 22:31:59.961435+00
(1 row)
guestbook=#
guestbook=# SELECT * FROM entries;
 guestname | content | entryid
-----------+---------+---------
(0 rows)
```

In this example, the timestamps show that the time of truncation of the table is somewhere between 22:31:54 and 22:31:59, so we are choosing 22:31:55. In a real world scenario things might not be as clear, so you might need to look at more data and log files to determine the correct date and time to use.

To ensure that all transactions are pushed to the pgBackRest backup repository, perform a log switch against your source AlloyDB Omni database cluster::

Unset

```
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_switch_wal();"
```

On your target server, follow the instructions in Cloning your AlloyDB Omni database cluster:

1. Prepare your target AlloyDB Omni database cluster.
2. Copy your pgBackRest configuration file from the source system to the target system.
3. Copy your pgBackRest backup repository (directory) from the source system to the target.

Stop before executing the pgBackRest restore command.

Adjust the restore command to include the `--type` and `--target` options and include the desired restore date and time, for example:

Unset

```
docker exec CONTAINER_NAME pgbackrest \
  --config-path=/var/lib/postgresql/backup \
  --pg1-path=/var/lib/postgresql/data/data-RESTORED \
  --stanza=omni \
```

```
--type=time \
--target="2024-06-03 22:31:55" \
restore
```

Complete the remaining steps from Cloning your AlloyDB Omni database cluster, including:

1. Stopping the target AlloyDB Omni database cluster.
2. Renaming data directories on the target server.
3. Updating the `postgresql.auto.conf` file on the target server. The pgBackRest automatically adds the `recovery_target_time` parameter to that file; there is no need to modify or remove this entry.
4. Re-starting the target AlloyDB Omni database cluster.

Check that the recovery completed as expected in the PostgreSQL log file for your AlloyDB Omni cluster:

Unset
```
sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping before'
```

Sample output:

Unset
```
$ sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping before'

2024-06-03 22:40:06.264 UTC [16] LOG:  [xlogrecovery.c:3178]  recovery stopping
before commit of transaction 912, time 2024-06-03 22:31:59.960505+00
2024-06-03 22:40:06.264 UTC [16] LOG:  [xlogrecovery.c:3411]  pausing at the
end of recovery
2024-06-03 22:40:06.264 UTC [16] HINT:  Execute pg_wal_replay_resume() to
promote.
2024-06-03 22:40:16.485 UTC [17] LOG:  [g_memory.c:48]  Memory worker finished
processing successfully
```

At this point, you can open your database in read-write mode and verify that the data was restored to the correct time:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"
```

Sample output:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

 pg_wal_replay_resume
----------------------

(1 row)

$ docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"

  guestname   |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:31:46.900483+00 |       1
 second guest | 2024-06-03 22:31:46.902606+00 |       2
 third guest  | 2024-06-03 22:31:53.938599+00 |       3
(3 rows)
```

Recover specific databases

When you just want to extract a specific table or set of tables, you don't need to restore the entire database cluster. Restoring the entire database cluster can use up excessive amounts of disk space on the recovery server and take more time to recover the lost data. When extracting specific tables or sets of tables, we recommend just restoring a specific database from the cluster instead.

For example, your AlloyDB Omni source database cluster may have several databases but you may only need to restore the `guestbook` database:

```
Unset
$ docker exec -it CONTAINER_NAME psql -h localhost -U postgres \
 -c "SELECT datname name, pg_database_size(datname) size FROM pg_database;"

      name       |   size
-----------------+----------
 postgres        | 14718639
 alloydbadmin    | 14169775
 template1       |  9273859
 template0       |  9273859
 db1             | 10548911
 alloydbmetadata | 13227695
 guestbook       | 10434223
(7 rows)
```

In this example, `guestbook` is comparatively small so excluding the other databases from the restore operation saves time and disk space.

As the online pgBackRest Command Reference states:

> "*... Databases not specifically included will be restored as sparse, zeroed files to save space but still allow PostgreSQL to perform recovery. After recovery, the databases that were not included will not be accessible but can be removed with the drop database command.*
>
> *NOTE: built-in databases (template0, template1, and postgres) are always restored unless specifically excluded.*
>
> *The `--db-include` option can be passed multiple times to specify more than one database to include.*"

To restore only a subset of databases into your *target* AlloyDB Omni database cluster, use `--db-include` option.

On your *target* AlloyDB Omni database cluster, follow all of the steps in Cloning your AlloyDB Omni database cluster, but stop before running the pgBackRest `restore` command. Adjust the `restore` command to include the `--db-include` option for the `alloydbadmin` database and any other databases you wish to restore.

Warning: The internal `alloydbadmin` database must always be restored as it has required metadata to make the database work properly.

For example:

```
Unset
docker exec pg-service pgbackrest \
  --config-path=/var/lib/postgresql/backup \
  --pg1-path=/var/lib/postgresql/data/data-RESTORED \
  --stanza=omni \
  --db-include=alloydbadmin \
  --db-include=guestbook \
  restore
```

Continue with the remaining recovery steps as documented.

Errors for the non-recovered databases in the PostgreSQL log are expected. Multiple lines of errors such as the following are expected:

```
Unset
2024-06-03 23:24:27.749 UTC [72] FATAL:  [relmapper.c:841]  relation mapping
file "base/16719/pg_filenode.map" contains invalid data
```

However, the restore should complete regardless.

Listing the databases shows that the PostgreSQL catalog still has records for all databases, including those that were not recovered:

```
Unset
$ docker exec -it CONTAINER_NAME psql -h localhost -U postgres -c "\l"
                                                      List of databases
      Name       |    Owner     | Encoding | Collate | Ctype | ICU Locale |
Locale Provider |       Access privileges
-----------------+--------------+----------+---------+-------+------------+----
-------------+-------------------------------
 alloydbadmin    | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
|
 alloydbmetadata | alloydbadmin | UTF8     | C       | C     | und-x-icu  | icu
| alloydbadmin=CTc/alloydbadmin +
                 |              |          |         |       |            |
| alloydbmetadata=c/alloydbadmin
 db1             | postgres     | UTF8     | C       | C     | und-x-icu  | icu
|
```

```
 guestbook       | postgres     | UTF8     | C        | C      | und-x-icu  | icu
|
 postgres        | postgres     | UTF8     | C        | C      | und-x-icu  | icu
|
 template0       | postgres     | UTF8     | C        | C      | und-x-icu  | icu
| =c/postgres                  +
                 |              |          |          |        |            |
| postgres=CTc/postgres
 template1       | postgres     | UTF8     | C        | C      | und-x-icu  | icu
| =c/postgres                  +
                 |              |          |          |        |            |
| postgres=CTc/postgres
(7 rows)
```

If you try to use any of the databases that were not actually recovered, an expected error is produced:

```
Unset
$ docker exec -it CONTAINER_NAME psql -h localhost -U postgres -d db1
psql: error: connection to server at "localhost" (::1), port 5432 failed:
FATAL:  relation mapping file "base/29305/pg_filenode.map" contains invalid
data
```

Drop the non-recovered databases and clean the PostgreSQL catalog, for example:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "DROP DATABASE
db1;"
DROP DATABASE
```

Repeat the dropping of non-recovered databases and cleaning the PostgreSQL catalog for all other non-recovered databases.

The reverse of the restore strategy, which uses the `--db-include` option, is possible using the `--db-exclude` option. Use whichever option is more applicable for your recovery scenario, while ensuring that the `alloydbadmin` is restored.

Recover to a restore point

Sometimes backups are made before significant database changes–such as major application changes or upgrades for example–or after an upgrade. You may manually create the restore point before major changes for reliability.

pgBackRest implicitly creates and uses PostgreSQL restore points. Restore points are created using the [pg_create_restore_point] function to create a marker in the WAL stream.

Sometimes administrators or utilities create their own restore points explicitly and without creating a pgBackRest backup.

pgBackRest supports restoring to an explicitly created restore point but does require you to specify the base backup to use. If you do not specify a base backup to use, the latest backup is used; which might be from a point in time after the desired restore point.

To test the recovery to a restore point, in your source AlloyDB Omni database cluster, create some new data prior to creating a restore point marker:

Unset

```
docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"INSERT INTO entries (guestName, content) values ('fourth guest','PRIOR TO APP
UPGRADE');"
```

Manually create a restore point:

Unset

```
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_create_restore_point('BEFORE_APPLICATION_UPDATE');"
```

Sample output:

Unset

```
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_create_restore_point('BEFORE_APPLICATION_UPDATE');"

 pg_create_restore_point
-------------------------
 0/F0004A0
```

```
(1 row)
```

To check that the restore point was successfully created, search the PostgreSQL log file:

```
Unset
$ sudo docker logs CONTAINER_NAME |& grep 'restore point'

2024-06-03 20:12:18.029 UTC [55] LOG:  [xlog.c:8642]  restore point "pgBackRest
Archive Check" created at 0/384B088
2024-06-03 20:18:47.633 UTC [143] LOG:  [xlog.c:8642]  restore point
"pgBackRest Archive Check" created at 0/6000140
2024-06-03 23:31:56.288 UTC [1144] LOG:  [xlog.c:8642]  restore point
"BEFORE_APPLICATION_UPDATE" created at 0/F0004A0
```

Create some post-restore-point data:

```
Unset
docker exec pCONTAINER_NAME psql -h localhost -U postgres -d guestbook \
  -c "INSERT INTO entries (guestName, content) values ('fifth guest','AFTER
FAILED APP UPGRADE');" \
  -c "SELECT pg_switch_wal();" \
  -c "SELECT * FROM entries;"
```

Sample output:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook \
>   -c "INSERT INTO entries (guestName, content) values ('fifth guest','AFTER
FAILED APP UPGRADE');" \
>   -c "SELECT pg_switch_wal();" \
>   -c "SELECT * FROM entries;"

INSERT 0 1
 pg_switch_wal
---------------
 0/F0005C0
(1 row)
```

```
 guestname    |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:48:14.988387+00 |       4
 second guest | 2024-06-03 22:48:21.762659+00 |       5
 third guest  | 2024-06-03 22:48:27.48224+00  |       6
 fourth guest | PRIOR TO APP UPGRADE          |       7
 fifth guest  | AFTER FAILED APP UPGRADE      |       8
(5 rows)
```

You can then restore your target AlloyDB Omni database cluster to the specific restore point using the `--type` and `--target` options.

IMPORTANT: This combination of options usually tries to use the latest full backup as the recovery starting point which might be from a later point in time. It is usually necessary to also specify the backup set using the `--set` option; use the most recent full backup from before your restore point.

Obtain the backup set name from your source AlloyDB Omni cluster using the `info` command. For example:

```
Unset
$ docker exec CONTAINER_NAME pgbackrest
--config-path=/var/lib/postgresql/backup
 --stanza=omni info
stanza: omni
    status: ok
    cipher: none
    db (current)
        wal archive min/max (15):
000000010000000000000003/00000001000000000000000A
        full backup: 20240603-201827F
            timestamp start/stop: 2024-06-03 20:18:27+00 / 2024-06-03
20:18:32+00
            wal start/stop: 000000010000000000000005 / 000000010000000000000005
            database size: 68.1MB, database backup size: 68.1MB
            repo1: backup set size: 6.5MB, backup size: 6.5MB
```

Follow all of the other steps from Cloning your AlloyDB Omni database cluster, but stop before running the pgBackRest `restore` command.

In your target AlloyDB Omni environment, adjust the `restore` command to include the `--set` option and the desired backup set name. Also add the `--type` and `--target` options:

```
Unset
docker exec CONTAINER_NAME pgbackrest \
  --config-path=/var/lib/postgresql/backup \
  --pg1-path=/var/lib/postgresql/data/data-RESTORED \
  --stanza=omni \
  --set 20240603-201827F \
  --type=name \
  --target=BEFORE_APPLICATION_UPDATE \
  restore
```

Continue with the remaining restoration steps as documented.

After restarting the AlloyDB Omni cluster, check that the recovery completed in the PostgreSQL log file for your AlloyDB Omni cluster:

```
Unset
sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping'
```

Sample output:

```
Unset
$ sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping'

2024-06-04 18:14:20.056 UTC [16] LOG:  [xlogrecovery.c:3238]  recovery stopping
at restore point "BEFORE_APPLICATION_UPDATE", time 2024-06-03 23:31:56.28848+00
2024-06-04 18:14:20.056 UTC [16] LOG:  [xlogrecovery.c:3411]  pausing at the
end of recovery
2024-06-04 18:14:20.056 UTC [16] HINT:  Execute pg_wal_replay_resume() to
promote.
2024-06-04 18:14:30.643 UTC [17] LOG:  [g_memory.c:48]  Memory worker finished
processing successfully
```

You can promote your database and verify that the data was restored to the correct time:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"
```

Sample output:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

 pg_wal_replay_resume
----------------------

(1 row)

$ docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"

  guestname   |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:48:14.988387+00 |       4
 second guest | 2024-06-03 22:48:21.762659+00 |       5
 third guest  | 2024-06-03 22:48:27.48224+00  |       6
 fourth guest | PRIOR TO APP UPGRADE          |       7
(4 rows)
```

Recover to a specific log sequence number

Sometimes a restoration to a specific LSN (log sequence number) is necessary. For example, an external system, a log file, or some other process might indicate the LSN when a failure occurs. The actual LSN number might be specified instead of the desired restore date, time, or backup set name.

Restoring your AlloyDB Omni database cluster to a specific LSN is almost identical to the process of restoring to a calendar date and time, but instead uses the LSN value.

To test, determine the current WAL LSN value from your source AlloyDB Omni database cluster:

```
Unset
SELECT pg_current_wal_lsn();
```

Perform a transaction to simulate corrupting some table data:

```
Unset
docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook \
  -c "SELECT * FROM entries;" \
  -c "SELECT pg_current_wal_lsn();" \
  -c "UPDATE entries SET content='A';" \
  -c "SELECT pg_switch_wal();" \
  -c "SELECT * FROM entries;"
```

Sample output:

```
Unset
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook \
  -c "SELECT * FROM entries;" \
  -c "SELECT pg_current_wal_lsn();" \
  -c "UPDATE entries SET content='A';" \
  -c "SELECT pg_switch_wal();" \
  -c "SELECT * FROM entries;"
  guestname   |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:48:14.988387+00 |       4
 second guest | 2024-06-03 22:48:21.762659+00 |       5
 third guest  | 2024-06-03 22:48:27.48224+00  |       6
 fourth guest | PRIOR TO APP UPGRADE          |       7
 fifth guest  | AFTER FAILED APP UPGRADE      |       8
(5 rows)
 pg_current_wal_lsn
--------------------
 0/15000148
(1 row)
UPDATE 5
 pg_switch_wal
---------------
 0/150004B0
(1 row)
  guestname   | content | entryid
```

```
--------------+---------+---------
 first guest  | A       |       4
 second guest | A       |       5
 third guest  | A       |       6
 fourth guest | A       |       7
 fifth guest  | A       |       8
(5 rows)
```

On your target AlloyDB Omni database cluster, follow all of the other steps from Cloning your AlloyDB Omni database cluster, but stop before running the pgBackRest `restore` command. Adjust the command to include `--type=lsn` and `--target=<LSN value>`.

For example:

Unset

```
docker exec CONTAINER_NAME pgbackrest \
  --config-path=/var/lib/postgresql/backup \
  --pg1-path=/var/lib/postgresql/data/data-RESTORED \
  --stanza=omni \
  --type=lsn \
  --target=0/15000148 \
  restore
```

Then continue with the remaining recovery steps as documented.

After restarting the AlloyDB Omni cluster, check that the recovery completed in the PostgreSQL log file for your AlloyDB Omni cluster:

Unset

```
sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping'
```

Sample output:

Unset

```
$ sudo docker logs CONTAINER_NAME |& grep -A4 'recovery stopping'
```

```
2024-06-04 18:38:12.804 UTC [16] LOG:  [xlogrecovery.c:3255]  recovery stopping
after WAL location (LSN) "0/15000148"
2024-06-04 18:38:12.805 UTC [16] LOG:  [xlogrecovery.c:3411]  pausing at the
end of recovery
2024-06-04 18:38:12.805 UTC [16] HINT:  Execute pg_wal_replay_resume() to
promote.
2024-06-04 18:38:23.496 UTC [17] LOG:  [g_memory.c:48]  Memory worker finished
processing successfully
```

You can promote your database and verify that the data was restored to the correct time:

Unset
```
docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"
```

Sample output:

Unset
```
$ docker exec CONTAINER_NAME psql -h localhost -U postgres -c "SELECT
pg_wal_replay_resume();"

 pg_wal_replay_resume
----------------------

(1 row)

$ docker exec CONTAINER_NAME psql -h localhost -U postgres -d guestbook -c
"SELECT * FROM entries;"

  guestname   |            content            | entryid
--------------+-------------------------------+---------
 first guest  | 2024-06-03 22:48:14.988387+00 |       4
 second guest | 2024-06-03 22:48:21.762659+00 |       5
 third guest  | 2024-06-03 22:48:27.48224+00  |       6
 fourth guest | PRIOR TO APP UPGRADE          |       7
 fifth guest  | AFTER FAILED APP UPGRADE      |       8
```

```
(5 rows)
```

High Availability and DR

What is database resilience?

Customers think of database resilience in terms of availability, time to restore service, and data loss. Availability is usually measured in terms of uptime and expressed as the percentage of time the database is available. For example, to achieve 99.99% availability, the database can't be down for more than 52.6 minutes of downtime per year, or 4.38 minutes per month. The time to restore service after an outage is called recovery time objective, or RTO. The amount of acceptable data loss due to an outage is called recovery point objective, or RPO, and is expressed as the amount of time for which transactions are lost.

Customers often set an availability target, or service level objective (SLO), together with targets for RTO and RPO. For example, for a given workload, the customer might set the SLO to 99.99%, and also require a RPO of 0–no data loss on any failure–and a RTO of 30 seconds. For another workload, they might set the SLO to 99.9%, the RPO to 5 minutes, and the RTO to 10 minutes.

You can implement database resilience with database backups. AlloyDB Omni supports backups using pgbackrest and also archives the database WAL (write ahead log) files to minimize data loss. With this approach, if the primary database goes down, it can be restored from a backup with an RPO of minutes, and a RTO of minutes to hours, depending on the size of the database.

For stricter RPO and RTO requirements, you can set up AlloyDB Omni in a high availability configuration using Patroni. In this architecture, there is a primary database and two standby or replica databases. You can configure AlloyDB Omni to use standard PostgreSQL streaming replication to ensure each transaction that is committed on the primary is synchronously replicated to both standby databases. This provides a RPO of zero, and a RTO of less than sixty seconds for most failure scenarios.

Synchronous replication can impact response time for transactions, and some customers choose to risk a small amount of data loss, for example a RPO above zero, in exchange for lower transactional latency, by implementing high availability with asynchronous replication instead of synchronous. Due to the potential impact of synchronous replication on transaction latency, high availability architectures are almost always implemented within a single data center, or between data centers that are close together (tens of km apart / <10 milliseconds of latency apart).

For disaster recovery, which is protection against the loss of a data center or a region where there are multiple data centers close together, AlloyDB Omni can be configured with asynchronous streaming replication from the primary region to a secondary region, typically hundreds or thousands of km apart, or 10’s to 100’s of milliseconds apart. In this configuration, the primary region is configured with synchronous streaming replication between the primary and standby databases within the region, and asynchronous streaming replication is configured from the primary region to one or more secondary regions. AlloyDB Omni can be configured in the secondary region with multiple database instances to ensure that it is protected immediately after a failover from the primary region.

The next section focuses on setting up a high availability solution for AlloyDB Omni using the Patroni, etcd, and HAProxy open source tools. This architecture can be extended across regions or geographically separated data centers to implement disaster recovery.

How high availability works

The specific techniques and tools used to implement high availability for databases can vary depending on the database management system. The following are some of the techniques and tools usually involved in implementing high availability for databases, which can vary depending on the database management system:

- **Redundancy**: Replicating your database across multiple servers or geographical regions provides failover options if a primary instance goes down.
- **Automated Failover**: Mechanism to detect failures and seamlessly switch to a healthy replica, minimizing downtime. Queries are routed so that application requests reach the new primary node.
- **Data Continuity**: Safeguards are implemented to protect data integrity during failures. This includes replication techniques and data consistency checks.
- **Clustering**: Clustering involves grouping multiple database servers to work together as a single system. In this way, all nodes in the cluster are active and handle requests which provides load balancing and redundancy.
- **Fallback**: Methods to fall back to the original architecture using pre-failover primary and replica nodes in their original capacities.
- **Load Balancing**: Distributing database requests across multiple instances improves performance and handles increased traffic.
- **Monitoring and Alerts:** Monitoring tools detect issues like server failure, high latency, resource exhaustion and trigger alerts, or automatic failover procedures.
- **Backup and Restore:** Backups can be used to restore databases to a previous state in case of data corruption or catastrophic failure.
- **Connection pooling (optional)**: Optimizes the performance and scalability of applications that interact with your databases.

High availability with Patroni, etcd and HAProxy

Patroni is an open-source cluster management tool for PostgreSQL databases designed to manage and automate high availability for PostgreSQL clusters. Patroni uses various distributed consensus systems like etcd, Consul, or Zookeeper to coordinate and manage the cluster state. Some key features and components of Patroni include high availability with automatic failover, leader election, replication, and recovery. Patroni is co-located with PostgreSQL server instances as it directly manages and monitors their health and performs necessary operations such as failovers and replication to maintain database cluster high availability and reliability.

Patroni uses a distributed consensus system to store metadata and manage the cluster. In this guide we use a distributed configuration store (DCS) called etcd. One of the uses of etcd is to store and retrieve distributed systems information such as configuration, health, and current status, ensuring consistent configuration across all nodes.

HAProxy (High Availability Proxy) is an open-source software used for load balancing and proxying TCP and HTTP-based applications, used to improve the performance and reliability of web applications by distributing incoming requests across multiple servers. HAProxy offers load balancing by distributing network traffic across multiple servers. HAProxy also maintains the health state of the backend servers it connects to by performing health checks. If a server fails a health check, HAProxy stops sending traffic to it until it passes the health checks again.

Before you begin

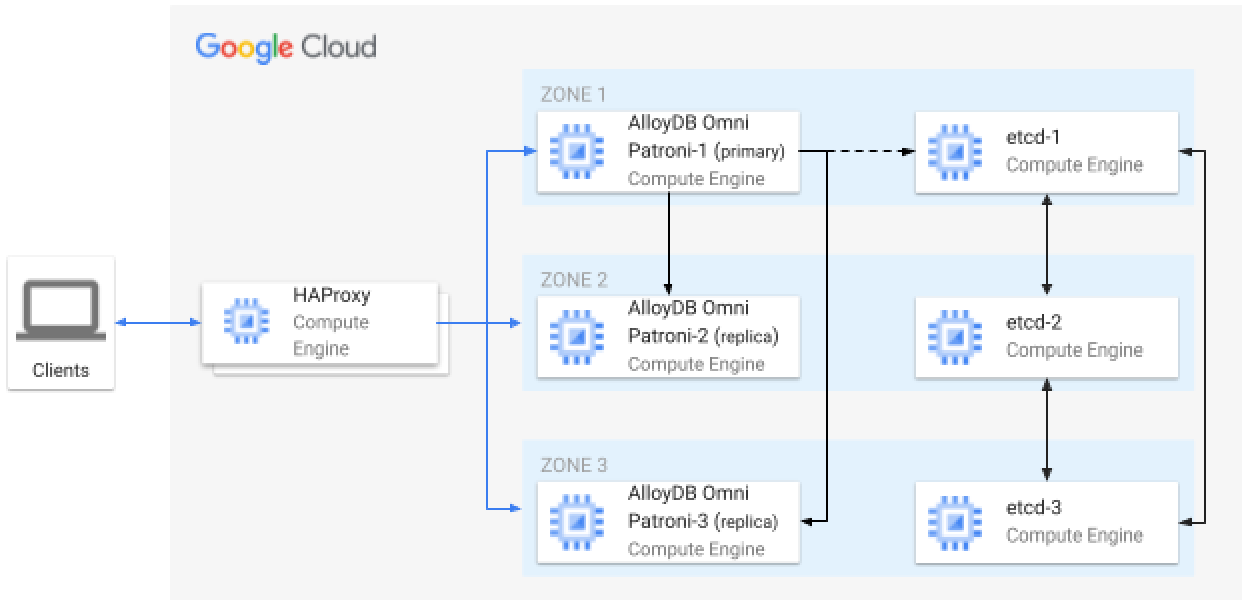
1. Create a Google Cloud project.
2. Make sure that billing is enabled for your Google Cloud project
3. Open [Cloud Shell](https://cloud.google.com/shell) in the Cloud Console.
4. In Cloud Shell, clone the source repository and go to the directory for this tutorial:

   ```
   git clone
   https://github.com/GoogleCloudPlatform/cloud-solutions.git
   ```

Installation

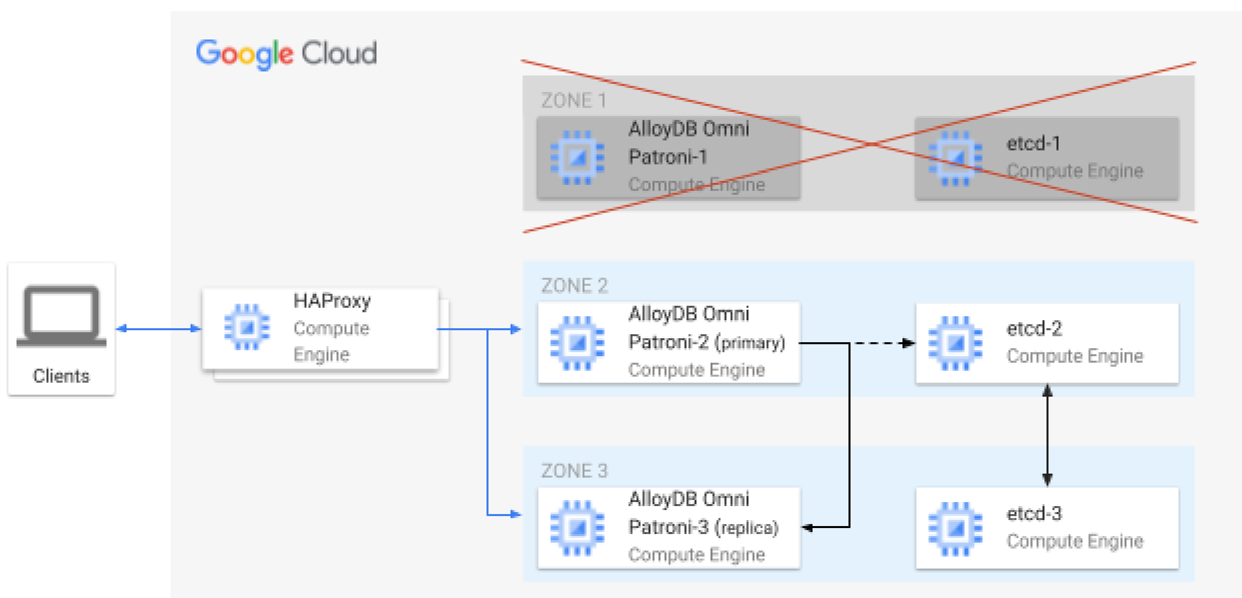
In this guide we deploy a three node Patroni cluster with AlloyDB Omni and a three node cluster etcd as the configuration store. In the front of the cluster, we use HAProxy in a managed instance group for the floating IP address so that the failover is transparent to clients.

The initial configuration of the cluster:



The configuration after a zone outage and a failover:

.



If the number of clients that connect to the database becomes an issue and you have performance issues due to the high number of simultaneous database connections, you might add an additional connection pooling component like PgBouncer.

Deploy the solution

1. In Cloud Shell, go to the terraform directory of this tutorial:

   ```
   cd
   cloud-solutions/projects/alloydbomni-ha-patroni-etcd/terrafo
   rm
   ```

2. Open the `terraform.tfvars` file. Set values for your project ID, region, zones and some settings for your Patroni cluster like cluster name and postgres superuser and replication usernames and passwords.

3. Run the Terraform script to create all resources.
   The Terraform script creates and configures:
   - Three nodes for your etcd cluster
   - Three nodes for your Patroni cluster
   - One node for HAProxy

   ```
   terraform init && terraform apply
   ```

Install a client on a machine in the same network

You can install a client on a machine that has network connectivity to your database instances. To do that, open terraform.tfvars file and make sure that the value of `provision_monitoring_machine` is set to true:

```
provision_monitoring_machine = true
```

In this example, we use [pgAdmin](https://www.pgadmin.org/).

Synchronous and asynchronous replication considerations

In a Patroni-managed PostgreSQL cluster, replication can be configured in both synchronous and asynchronous modes. By default, Patroni uses asynchronous streaming replication. Although each replication type offers distinct advantages and trade-offs, some business use cases might require synchronous replication.

Asynchronous replication allows transactions to be committed on the primary without waiting for acknowledgments from standbys. The primary sends write-ahead log (WAL) records to standbys, which apply them asynchronously. This asynchronous approach reduces write latency and improves performance, but comes with the risk of data loss if the primary fails before the standby has caught up. Standbys might be behind the primary, leading to potential inconsistencies during failover.

Synchronous replication in PostgreSQL ensures data consistency by waiting for transactions to be written to both the primary and at least one synchronous standby before committing. Synchronous replication guarantees that data is not lost in the event of a primary failure, providing strong data durability and consistency. The primary waits for acknowledgments from the synchronous standby, which can lead to higher latency and potentially lower throughput due to the added round-trip time. This can reduce overall system throughput, especially under high load.

The choice between synchronous and asynchronous replication in a Patroni cluster depends on the specific requirements for data durability, consistency, and performance. Synchronous replication is preferable in scenarios where data integrity and minimal data loss are critical, while asynchronous replication suits environments where performance and lower latency are prioritized. You can configure a mixed solution that involves having a three node cluster with a synchronous standby in the same region but a different nearby zone or data center, and a second asynchronous standby in a different region or a more distant data center to protect against potential regional outages.

To make Patroni use only synchronous replication in your three nodes cluster, add configuration items like `synchronous_mode`, `synchronous_node_count`, `synchronous_commit` and `synchronous_standby_names` in the bootstrap section in your Patroni configuration files. The bootstrap section of your yml configuration files looks similar to the following:

```
bootstrap:
 dcs:
  ttl: 30
  loop_wait: 10
  retry_timeout: 10
  maximum_lag_on_failover: 1048576
  synchronous_mode: true
  synchronous_node_count: 2
  postgresql:
   use_pg_rewind: true
   use_slots: true
   parameters:
    hot_standby: "on"
    wal_keep_segments: 20
    max_wal_senders: 8
    max_replication_slots: 8
    synchronous_commit: remote_apply
    synchronous_standby_names: '*'
```

When `synchronous_mode` is turned on, Patroni uses synchronous replication between its primary and the other replicas. The parameter `synchronous_node_count` is used by Patroni to manage the number of synchronous standby databases. Patroni manages precise number of synchronous standby databases based on `parameter synchronous_node_count` and adjusts the state in the configuration store and in the `synchronous_standby_names` as members join and leave. For more information about synchronous replication, see the [Replication modes] section in Patroni's documentation.

Test your high availability setup

Ensuring the reliability and quality of your high availability Patroni setup is crucial for maintaining continuous database operations and minimizing downtime. This section provides a comprehensive guide to testing your Patroni cluster, covering various failure scenarios, replication consistency, and failover mechanisms. Follow the sections below to validate the integrity and performance of your high availability Patroni configuration.

Test your Patroni setup

Connect to any of your patroni instances (patroni1, patroni2 or patroni3) and navigate to the alloydb omni patroni folder:

```
cd /alloydbomni-patroni/
```

Run the docker compose logs command to inspect the patroni logs

```
docker-compose logs alloydbomni-patroni
```

The last entries should reflect information about the patroni node. You should see something similar to the following:

```
alloydb-patroni        | 2024-06-12 15:10:29,020 INFO: no action. I am
(patroni1), the leader with the lock
alloydb-patroni        | 2024-06-12 15:10:39,010 INFO: no action. I am
(patroni1), the leader with the lock
alloydb-patroni        | 2024-06-12 15:10:49,007 INFO: no action. I am
(patroni1), the leader with the lock
```

Connect to any instance running linux that has network connectivity to your primary patroni instance (patroni1) and get information about the patroni-1 instance:

```
curl -s http://patroni1:8008/patroni | jq .
```

You should see something similar to the following displayed:


```
{
  "state": "running",
  "postmaster_start_time": "2024-05-16 14:12:30.031673+00:00",
  "role": "master",
  "server_version": 150005,
  "xlog": {
    "location": 83886408
  },
  "timeline": 1,
  "replication": [
    {
      "usename": "alloydbreplica",
      "application_name": "patroni2",
      "client_addr": "10.172.0.40",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    },
    {
      "usename": "alloydbreplica",
      "application_name": "patroni3",
      "client_addr": "10.172.0.41",
      "state": "streaming",
      "sync_state": "async",
      "sync_priority": 0
    }
  ],
  "dcs_last_seen": 1715870011,
  "database_system_identifier": "7369600155531440151",
  "patroni": {
    "version": "3.3.0",
    "scope": "my-patroni-cluster",
    "name": "patroni1"
  }
}
```


Note: You might need to install the jq tool by running `sudo apt-get install jq -y`

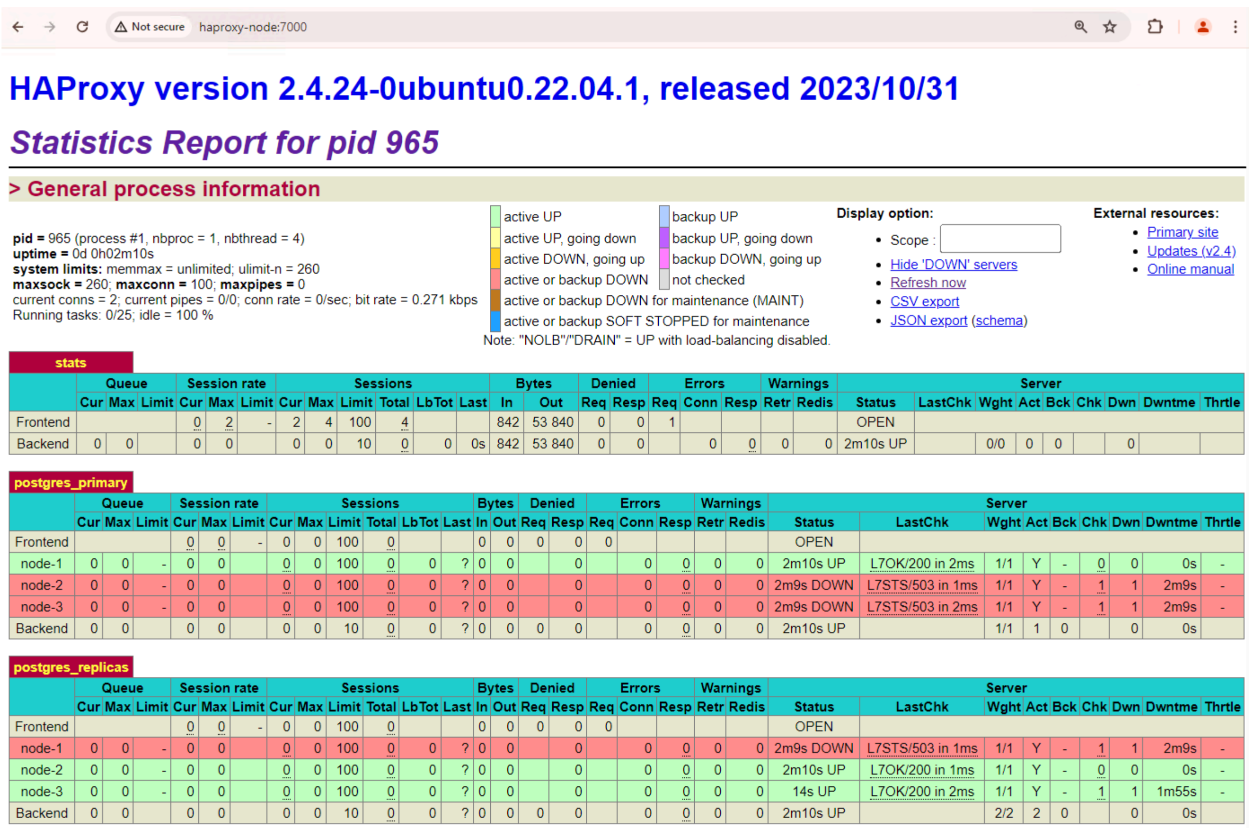
Calling the Patroni HTTP API endpoint on a Patroni node exposes various details about the state and configuration of that particular PostgreSQL instance managed by Patroni, including cluster state information, timeline, WAL information, and health checks indicating whether the nodes and cluster are up and running correctly.

Test your HAProxy setup

On a machine with a browser and network connectivity to your HAProxy node, go to the following address:

http://haproxy:7000

You should see something resembling the below screenshot:



HAProxy version 2.4.24-0ubuntu0.22.04.1, released 2023/10/31

Statistics Report for pid 965

> General process information

pid = 965 (process #1, nbproc = 1, nbthread = 4)
uptime = 0d 0h02m10s
system limits: memmax = unlimited; ulimit-n = 260
maxsock = 260; **maxconn =** 100; **maxpipes =** 0
current conns = 2; current pipes = 0/0; conn rate = 0/sec; bit rate = 0.271 kbps
Running tasks: 0/25; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:

- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:

- Primary site
- Updates (v2.4)
- Online manual

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	2	-	2	4	100	4			842	53 840	0	0	1					OPEN								
Backend	0	0		0	0		0	0	10	0	0	0s	842	53 840	0	0		0	0	0	0	2m10s UP		0/0	0	0		0		

postgres_primary

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	100	0			0	0	0	0	0					OPEN								
node-1	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m10s UP	L7OK/200 in 2ms	1/1	Y	-	0	0	0s	-
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m9s DOWN	L7STS/503 in 1ms	1/1	Y	-	1	1	2m9s	-
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m9s DOWN	L7STS/503 in 2ms	1/1	Y	-	1	1	2m9s	-
Backend	0	0		0	0		0	0	10	0	0	?	0	0	0	0		0	0	0	0	2m10s UP		1/1	1	0		0	0s	

postgres_replicas

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	100	0			0	0	0	0	0					OPEN								
node-1	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m9s DOWN	L7STS/503 in 1ms	1/1	Y	-	1	1	2m9s	-
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m10s UP	L7OK/200 in 1ms	1/1	Y	-	0	0	0s	-
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	14s UP	L7OK/200 in 2ms	1/1	Y	-	1	1	1m55s	-
Backend	0	0		0	0		0	0	10	0	0	?	0	0	0	0		0	0	0	0	2m10s UP		2/2	2	0		0	0s	

In the HAProxy dashboard you can see the health status and latency of your primary Patroni, patroni1, and of the two replicas, patroni2 and patroni3.

If you connect to the HAProxy server from your pgAdmin client, you can perform some queries to check the replication stats in your cluster.

Connect to your HAProxy-node and run the following query:

```
SELECT
     pid, usename, application_name, client_addr, state,
sync_state
FROM
     pg_stat_replication;
```

You should see something similar to the following:

	pid integer	usename name	application_name text	client_addr inet	state text	sync_state text
1	134	alloydbreplica	patroni-n-2	10.172.0.40	streaming	sync
2	600	alloydbreplica	patroni-n-3	10.172.0.41	streaming	sync

Test the automatic failover operation

In this section, in your three node cluster, we simulate an outage on the primary node by stopping the attached-running Patroni container. You can either stop the Patroni service on the primary node to simulate an outage or enforce some firewall rules to stop communication to that node.

1. Navigate to the alloydb omni patroni folder:

   ```
   cd /alloydbomni-patroni/
   ```

2. Run docker compose down command to stop the running container:

   ```
   docker compose down
   ```

   You should see something similar to this:

   ```
   root@patroni1:/alloydbomni-patroni# docker compose down
   [+] Running 2/2
    ✔ Container alloydb-patroni           Removed
    ✔ Network alloydbomni-patroni_default  Removed
   ```

3. Refresh the HAProxy dashboard and see how failover takes place:

Not secure haproxy-node:7000

HAProxy version 2.4.24-0ubuntu0.22.04.1, released 2023/10/31

Statistics Report for pid 965

> General process information

pid = 965 (process #1, nbproc = 1, nbthread = 4)
uptime = 0d 0h16m02s
system limits: memmax = unlimited; ulimit-n = 260
maxsock = 260; **maxconn =** 100; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 0/24; idle = 100 %

- active UP
- active UP, going down
- active DOWN, going up
- active or backup DOWN
- active or backup DOWN for maintenance (MAINT)
- active or backup SOFT STOPPED for maintenance
- backup UP
- backup UP, going down
- backup DOWN, going up
- not checked

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:

- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:

- Primary site
- Updates (v2.4)
- Online manual

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	2	-	1	4	100	7			1 684	108 198	0	0	3					OPEN								
Backend	0	0		0	0		0	0	10	0	0	0s	1 684	108 198	0	0		0	0	0	0	16m2s UP		0/0	0	0		0		

postgres_primary

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	T
Frontend				0	2	-	0	3	100	3			16 489	18 055	0	0	0					OPEN								
node-1	0	0	-	0	2		0	2	100	3	3	9m55s	16 489	18 055		0		0	0	0	0	55s DOWN	L4CON in 0ms	1/1	Y	-	3	1	55s	
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	16m1s DOWN	L7STS/503 in 1ms	1/1	Y	-	1	1	16m1s	
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	53s UP	L7OK/200 in 2ms	1/1	Y	-	1	1	15m8s	
Backend	0	0		0	2		0	2	10	3	3	9m55s	16 489	18 055	0	0		0	0	0	0	53s UP		1/1	1	0		1	2s	

postgres_replicas

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	100	0			0	0	0	0	0					OPEN								
node-1	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	16m1s DOWN	L4CON in 0ms	1/1	Y	-	1	1	16m1s	-
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	16m2s UP	L7OK/200 in 1ms	1/1	Y	-	0	0	0s	-
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	54s DOWN	L7STS/503 in 2ms	1/1	Y	-	7	3	3m29s	-
Backend	0	0		0	0		0	0	10	0	0	?	0	0	0	0		0	0	0	0	16m2s UP		1/1	1	0		0	0s	

The patroni3 instance became the new primary, and patroni2 is the only remaining replica. The previous primary, patroni1, is down and Layer 4 checks fail for it.

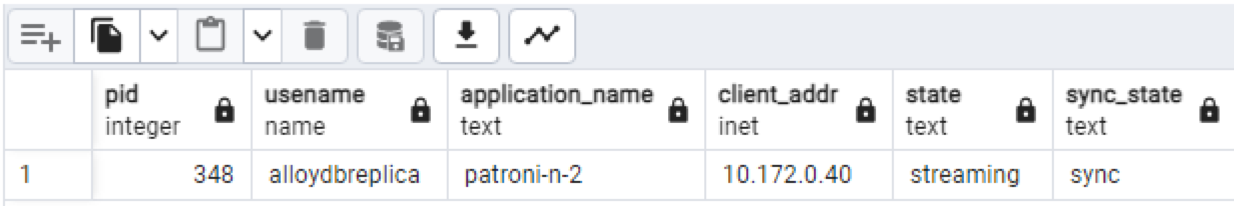
Patroni performs and manages the failover through a combination of monitoring, consensus, and automated orchestration. As soon as the primary node fails to renew its lease within a specified timeout, or if it reports a failure, the other nodes in the cluster recognize this condition through the consensus system. The remaining nodes coordinate to select the most suitable replica to promote as the new primary. Once a candidate replica is selected, Patroni promotes this node to primary by applying the necessary changes, such as updating the PostgreSQL configuration and replaying any outstanding WAL records. Then, the new primary node updates the consensus system with its status and the other replicas reconfigure themselves to follow the new primary, including switching their replication source and potentially catching up with any new transactions. HAProxy detects the new primary and redirects client connections accordingly, ensuring minimal disruption.

If you have access to a pgAdmin client, connect to your HAProxy-node and check the replication stats in your cluster after failover:

```
    SELECT pid, usename, application_name, client_addr, state,
sync_state
    FROM
```

```
pg_stat_replication;
```

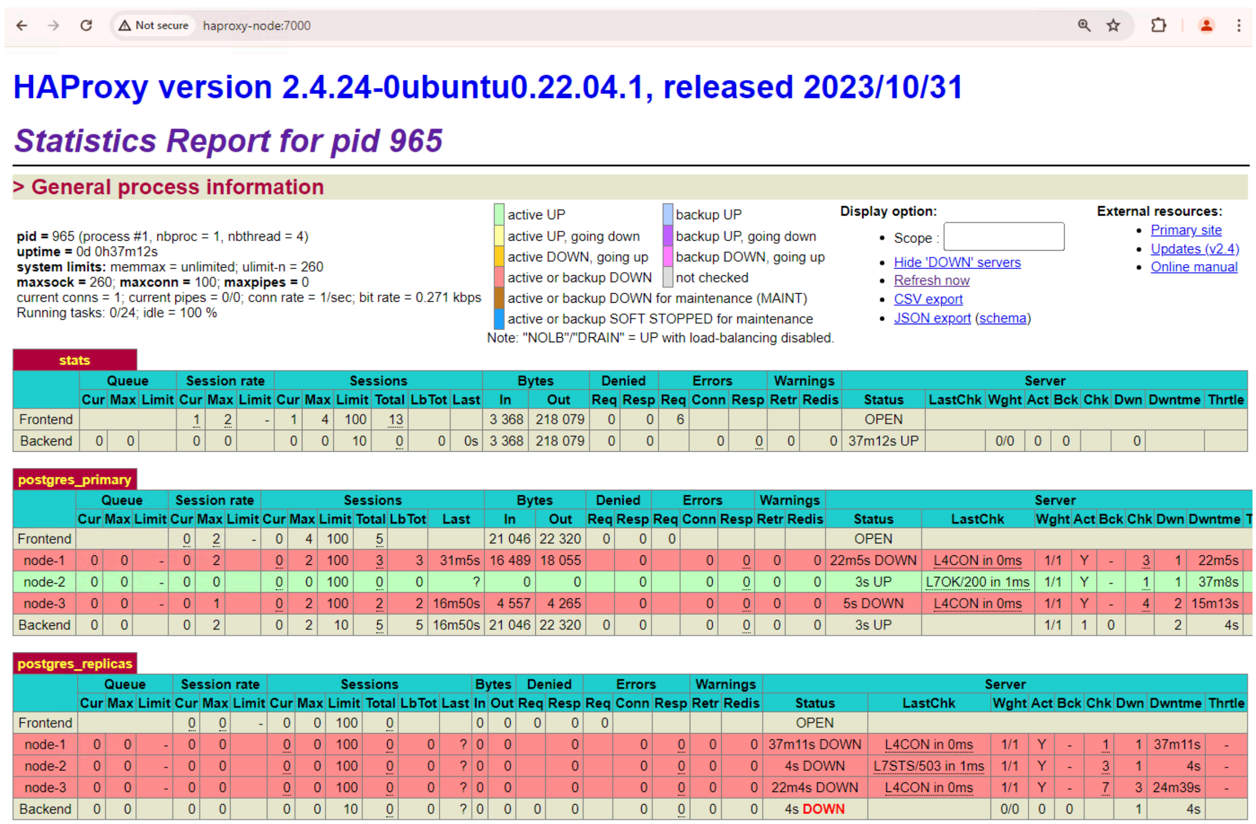
You should see something similar:



	pid integer	usename name	application_name text	client_addr inet	state text	sync_state text
1	348	alloydbreplica	patroni-n-2	10.172.0.40	streaming	sync

patroni2 is now the only replica remaining and following the new primary patroni3.

Your three node cluster can survive one more outage. If you stop the current primary node (patroni3), another failover takes place:



HAProxy version 2.4.24-0ubuntu0.22.04.1, released 2023/10/31

Statistics Report for pid 965

> General process information

pid = 965 (process #1, nbproc = 1, nbthread = 4)
uptime = 0d 0h37m12s
system limits: memmax = unlimited; ulimit-n = 260
maxsock = 260; **maxconn =** 100; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 0/24; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:

- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:

- Primary site
- Updates (v2.4)
- Online manual

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	2	-	1	4	100	13			3 368	218 079	0	0	6					OPEN								
Backend	0	0		0	0		0	0	10	0	0	0s	3 368	218 079	0	0		0	0	0	0	37m12s UP		0/0	0	0		0		

postgres_primary

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	T
Frontend				0	2	-	0	4	100	5			21 046	22 320	0	0	0					OPEN								
node-1	0	0	-	0	2		0	2	100	3	3	31m5s	16 489	18 055		0		0	0	0	0	22m5s DOWN	L4CON in 0ms	1/1	Y	-	3	1	22m5s	
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	3s UP	L7OK/200 in 1ms	1/1	Y	-	1	1	37m8s	
node-3	0	0	-	0	1		0	2	100	2	2	16m50s	4 557	4 265		0		0	0	0	0	5s DOWN	L4CON in 0ms	1/1	Y	-	4	2	15m13s	
Backend	0	0		0	2		0	2	10	5	5	16m50s	21 046	22 320	0	0		0	0	0	0	3s UP		1/1	1	0		2	4s	

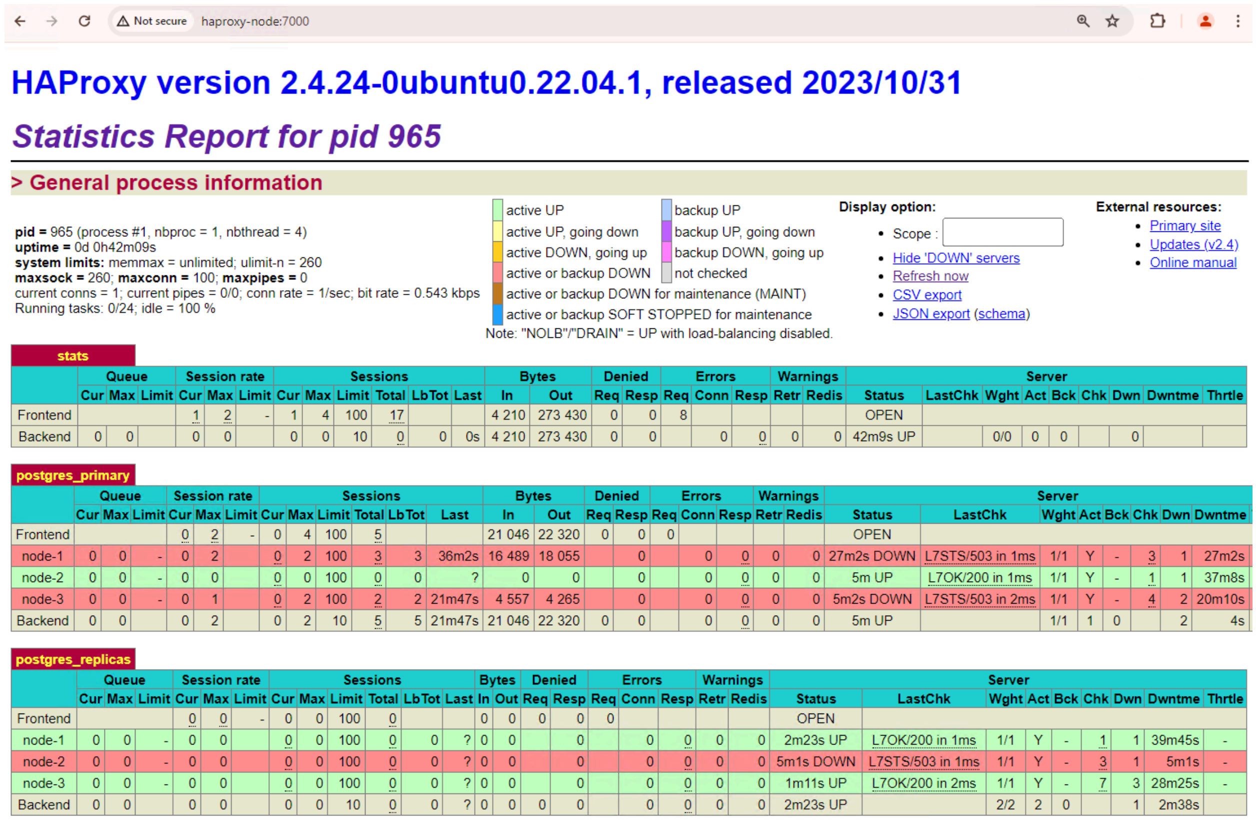
postgres_replicas

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	100	0			0	0	0	0	0					OPEN								
node-1	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	37m11s DOWN	L4CON in 0ms	1/1	Y	-	1	1	37m11s	-
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	4s DOWN	L7STS/503 in 1ms	1/1	Y	-	3	1	4s	-
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	22m4s DOWN	L4CON in 0ms	1/1	Y	-	7	3	24m39s	-
Backend	0	0		0	0		0	0	10	0	0	?	0	0	0	0		0	0	0	0	4s DOWN		0/0	0	0		1	4s	

Fallback considerations

Fallback is the process to reinstate the former source node after a failover has occurred. Automatic fallback is generally not recommended in a high availability database cluster because of several critical concerns, like incomplete recovery, risk of split-brain scenarios, and replication lag.

In your Patroni cluster, if you bring up the two nodes that you simulated an outage with, they will rejoin the cluster as standby replicas:



Not secure haproxy-node:7000

HAProxy version 2.4.24-0ubuntu0.22.04.1, released 2023/10/31

Statistics Report for pid 965

> General process information

pid = 965 (process #1, nbproc = 1, nbthread = 4)
uptime = 0d 0h42m09s
system limits: memmax = unlimited; ulimit-n = 260
maxsock = 260; **maxconn =** 100; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.543 kbps
Running tasks: 0/24; idle = 100 %

- active UP
- active UP, going down
- active DOWN, going up
- active or backup DOWN
- active or backup DOWN for maintenance (MAINT)
- active or backup SOFT STOPPED for maintenance
- backup UP
- backup UP, going down
- backup DOWN, going up
- not checked

Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:

- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:

- Primary site
- Updates (v2.4)
- Online manual

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	2	-	1	4	100	17			4 210	273 430	0	0	8					OPEN								
Backend	0	0		0	0		0	0	10	0	0	0s	4 210	273 430	0	0		0	0	0	0	42m9s UP		0/0	0	0		0		

postgres_primary

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server							
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme
Frontend				0	2	-	0	4	100	5			21 046	22 320	0	0	0					OPEN							
node-1	0	0	-	0	2		0	2	100	3	3	36m2s	16 489	18 055		0		0	0	0	0	27m2s DOWN	L7STS/503 in 1ms	1/1	Y	-	3	1	27m2s
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	5m UP	L7OK/200 in 1ms	1/1	Y	-	1	1	37m8s
node-3	0	0	-	0	1		0	2	100	2	2	21m47s	4 557	4 265		0		0	0	0	0	5m2s DOWN	L7STS/503 in 2ms	1/1	Y	-	4	2	20m10s
Backend	0	0		0	2		0	2	10	5	5	21m47s	21 046	22 320	0	0		0	0	0	0	5m UP		1/1	1	0		2	4s

postgres_replicas

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	100	0			0	0	0	0	0					OPEN								
node-1	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	2m23s UP	L7OK/200 in 1ms	1/1	Y	-	1	1	39m45s	-
node-2	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	5m1s DOWN	L7STS/503 in 1ms	1/1	Y	-	3	1	5m1s	-
node-3	0	0	-	0	0		0	0	100	0	0	?	0	0		0		0	0	0	0	1m11s UP	L7OK/200 in 2ms	1/1	Y	-	7	3	28m25s	-
Backend	0	0		0	0		0	0	10	0	0	?	0	0	0	0		0	0	0	0	2m23s UP		2/2	2	0		1	2m38s	

Now patroni1 and patroni3 are replicating from the current primary patroni2.

	pid integer	usename name	application_name text	client_addr inet	state text	sync_state text
1	1777	alloydbreplica	patroni-n-1	10.172.0.37	streaming	sync
2	1962	alloydbreplica	patroni-n-3	10.172.0.41	streaming	sync

If you want to manually fall back to your initial primary, you can do that by using the [patronictl](#) command-line interface. By opting for manual fallback, you can ensure a more reliable, consistent, and thoroughly verified recovery process, maintaining the integrity and availability of your database systems.

Security and Compliance

Manage AlloyDB Omni database roles

About user roles in AlloyDB Omni

An AlloyDB database uses [the standard PostgreSQL concept of *roles*]. A role can act as a database user, a group of users, or both.

A *user role* has the LOGIN privilege that lets users log into the system. A *group role* has member roles with various privileges, which you can grant to or revoke from all members at once

AlloyDB's predefined PostgreSQL roles

PostgreSQL has a set of predefined roles with various privileges. AlloyDB Omni adds several user and group roles to this set of PostgreSQL's predefined roles.

The following table lists the PostgreSQL roles that AlloyDB predefines:

Role name	Privileges
alloydbadmin	SUPERUSER, CREATEROLE, CREATEDB, REPLICATION, BYPASSRLS
alloydbagent	-
alloydbexport	-
alloydbiamgroupuser	-
alloydbiamuser	-
alloydbimportexport	-
alloydbmetadata	LOGIN
alloydbobservability	-
alloydbreplica	-
alloydbsuperuser	-
postgres	SUPERUSER, CREATEROLE, CREATEDB, BYPASSRLS, and LOGIN

The `alloydbsuperuser` role is a predefined role to initially set up the database system and perform other superuser tasks. This role has the following privileges:

- Create extensions that require superuser privileges

- Create event triggers
- Create replication users
- Create replication publications and subscriptions

The other users are simply reserved names that are unused.

Omni users have the option to modify these predefined PostgreSQL roles if needed by following the Postgres documentation.

Data Migration

The most appropriate approach for migrating from any source database to AlloyDB Omni depends on the nature of the source system and the downtime available to switch from the source environment to the destination AlloyDB Omni environment.

PostgreSQL to AlloyDB Omni

In the simplest migration case, sufficient downtime is available to move the source database to the destination using pg_dump and pg_restore. Migrations that can be completed within the available downtime are simpler than ones that cannot because migrations that cannot require multiple transfers of data, which might involve multiple tools and data movement methods.

As the volume of data and other forms of complexity increase, use of pgloader might become more appropriate.

For migrations where downtime must be minimized, you can use PostgreSQL logical replication for the requirement for some form of replication from source to destination.

Oracle to AlloyDB Omni

In the simplest migration case, sufficient downtime is available to move the volume of data in Oracle and all Oracle resident application logic can be converted to PostgreSQL. In this situation the open source tool Ora2Pg is recommended for schema conversion and data movement. Ora2Pg may also be appropriate for the code conversion.

With the increase in complexity of application logic in the source Oracle system, volume of data to migrate, and rate of new data creation, it's likely that other tools might be required to migrate the application logic and to switch from source to destination in an available downtime window.

The Database Migration Service can be used for code and schema conversion, while the data migration should be done by a different tool, Equalum, as Database Migration Service doesn't support AlloyDB Omni. Database Migration Service can be used for code and schema conversion with another tool such as Ora2Pg, and, where required, a CDC (change data capture) tool.

Observability

Since AlloyDB Omni is an edge installation, generally the same techniques used to observe other edge type installations of PostgreSQL apply.

Observability Scripts

To tell how your AlloyDB Omni database is performing you can either use scripts to query the system tables or use observability tools like the Prometheus Exporter for Postgres which is detailed in the observability tools section. If you want to utilize scripts, the following script is a good starting point for understanding how your installation is performing:

```
/* To determine the state of connected processes and any current wait events */
SELECT
    pid,
    datname,
    age(backend_xid) AS age_in_xids,
    now() - xact_start AS xact_age,
    now() - query_start AS query_age,
    state,
    wait_event_type,
    wait_event,
    query_id,
    query
FROM
    pg_stat_activity
WHERE
    state != 'idle'
    AND pid <> pg_backend_pid()
ORDER BY
    4 DESC
LIMIT 10;

/* Large tables size with #of seq / index scan */
SELECT
    oid,
    oid::regclass table_name,
```

```
    pg_size_pretty(pg_relation_size(oid)),
    relpages,
    s.seq_scan,
    s.idx_scan
FROM
    pg_class,
    pg_stat_user_tables s
WHERE
    s.relid = oid
    AND oid > 16383
    AND relpages > 100
    AND relkind = 'r'
ORDER BY
    relpages DESC
LIMIT 20;

/* Top Sequential scans: */

SELECT
    relid,
    relname,
    seq_scan,
    pg_size_pretty(pg_relation_size(relid))
FROM
    pg_stat_user_tables
ORDER BY
    seq_scan DESC
LIMIT 15;


/* Top Index scans: */

SELECT
    relid,
    relid::regclass table_name,
    idx_scan,
    pg_size_pretty(pg_relation_size(relid))
FROM
    pg_stat_user_tables
WHERE
    idx_scan > 10
ORDER BY
    idx_scan DESC
LIMIT 15;
```

```
/* Check on Vacuum Progress */
SELECT
  p.pid,
  now() - a.xact_start AS duration,
  coalesce(wait_event_type ||'.'|| wait_event, 'f') AS waiting,
  CASE
    WHEN a.query ~*'^autovacuum.*to prevent wraparound' THEN 'wraparound'
    WHEN a.query ~*'^vacuum' THEN 'user'
  ELSE
    'regular'
  END AS mode,
  p.datname AS database,
  p.relid::regclass AS table,
  p.phase,
  pg_size_pretty(p.heap_blks_total * current_setting('block_size')::int) AS table_size,
  pg_size_pretty(pg_total_relation_size(relid)) AS total_size,
  pg_size_pretty(p.heap_blks_scanned * current_setting('block_size')::int) AS scanned,
  pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS vacuumed,
  round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scanned_pct,
  round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total, 1) AS vacuumed_pct,
  p.index_vacuum_count,
  round(100.0 * p.num_dead_tuples / p.max_dead_tuples,1) AS dead_pct
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a using (pid)
ORDER BY now() - a.xact_start DESC;

/* Is a query running in parallel? */
SELECT
    query,
    leader_pid,
    array_agg(pid) FILTER (WHERE leader_pid != pid) AS members
FROM
    pg_stat_activity
WHERE
    leader_pid IS NOT NULL
GROUP BY
    query,
    leader_pid;

/* Blocking Lock SQL */
SELECT blocked_locks.pid     AS blocked_pid,
       blocked_activity.usename  AS blocked_user,
       blocking_locks.pid     AS blocking_pid,
       blocking_activity.usename AS blocking_user,
       blocked_activity.query    AS blocked_statement,
       blocked_activity.wait_event AS blocked_wait_event,
```

```
       blocking_activity.wait_event AS blocking_wait_event,
       blocking_activity.query   AS current_statement_in_blocking_process
 FROM  pg_catalog.pg_locks         blocked_locks
  JOIN pg_catalog.pg_stat_activity blocked_activity  ON blocked_activity.pid =
blocked_locks.pid
  JOIN pg_catalog.pg_locks         blocking_locks
      ON blocking_locks.locktype = blocked_locks.locktype
      AND blocking_locks.database IS NOT DISTINCT FROM blocked_locks.database
      AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
      AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
      AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
      AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
      AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
      AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
      AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
      AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
      AND blocking_locks.pid != blocked_locks.pid
  JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid =
blocking_locks.pid
 WHERE NOT blocked_locks.granted;

/* Check for the 10 Longest Running Transactions */
SELECT
    pid,
    age(backend_xid) AS age_in_xids,
    now() - xact_start AS xact_age,
    now() - query_start AS query_age,
    state,
    query
FROM
    pg_stat_activity
WHERE
    state != 'idle'
ORDER BY
    2 DESC
LIMIT 10;
```

To determine if your `work_mem / temp_buffers` are sized correctly for your needs, the `postgres.log` or `pg_stat_database`. Using `pg_stat_database`, execute the following query and if there is any growth in `temp_files` or `temp_bytes` between executions, then tuning is likely necessary for either `work_mem` or `temp_buffers`.

```
SELECT
    datname,
```

```
    temp_files,
    temp_bytes
FROM
    pg_stat_database;
```

From within the `postgres.log` file if temp files were being used, the following line is present:

```
LOG:  [fd.c:1772]  temporary file: path "base/pgsql_tmp/pgsql_tmp4640.1", size 139264
```

It is important to realize that the goal is to minimize the creation of temporary files, not completely prevent them from happening. This is because setting both `work_mem` and `temp_buffers` is a balance between available memory on the host and the number of connections that require the memory. Setting these parameters correctly required understanding about each individual workload.

Observability Tools

To get a complete view of the system, we recommend products like Datadog which are already integrated with PostgreSQL and track a large number of data points. Postgres Exporter, Prometheus, and Grafana can also be used to construct your own dashboards.

Using Grafana, Prometheus, and Postgres Exporter

Prometheus is a standard logging format that dashboarding tools like Grafana can ingest to create trending graphs and subsequent alerting mechanisms.

Installing Postgres Exporter

Open source [Postgres Exporter] is a standard mechanism to export observability queries into a format that Prometheus can read. The exporter comes with many standard queries already built in, however you can add additional queries and rules depending on your needs. Additional security options such as SSL and user authentication options can be changed to fit the installation needs. For this example the basic options are used.

To install the exporter, prepare the software location and copy the binary to a suitable location:

```
/* Create a software staging area */
sudo mkdir /opt/postgres_exporter
sudo chown your linux user:your linux user /opt/postgres_exporter
```

```
cd /opt/postgres_exporter
wget
https://github.com/prometheus-community/postgres_exporter/releases/download/v0.15.0/postgres_ex
porter-0.15.0.linux-amd64.tar.gz

tar -xzvf postgres_exporter-0.15.0.linux-amd64.tar.gz

cd postgres_exporter-0.15.0.linux-amd64
sudo cp postgres_exporter /usr/local/bin
```

Create an appropriate ENV file for the exporter:

```
cd /opt/postgres_exporter
sudo vi postgres_exporter.env

# Inside the postgres_exporter.env put the following:
# to Monitor one single database
DATA_SOURCE_NAME="postgresql://username:password@postgres_ip_address:port/database-name
?sslmode=disable"

# or you can use the following to monitor all the databases available on localhost
DATA_SOURCE_NAME="postgresql://username:password@postgres_ip_address:port
/?sslmode=disable"
```

Create a `system.d` service so that the exporter will survive the reboot:

```
/* Add the contents to the following file: /etc/systemd/system/postgres_exporter.service */
[Unit]
Description=Prometheus exporter for Postgresql
Wants=network-online.target
After=network-online.target
[Service]
User=postgres
Group=postgres
WorkingDirectory=/opt/postgres_exporter
EnvironmentFile=/opt/postgres_exporter/postgres_exporter.env
ExecStart=/usr/local/bin/postgres_exporter --web.listen-address=:postgres_exporter_port
--web.telemetry-path=/metrics
Restart=always
[Install]
WantedBy=multi-user.target
```

Reload and start the Postgres Exporter Service:

```
/* Reload Systemd */
sudo systemctl daemon-reload

/* Enable and Start Service */
sudo systemctl start postgres_exporter
sudo systemctl enable postgres_exporter
sudo systemctl status postgres_exporter
```

Installing Prometheus

Prometheus is required to query the exporter and return the observability data into a readable format.

```
/* Create prometheus user */
sudo groupadd --system prometheus
sudo useradd -s /sbin/nologin --system -g prometheus prometheus

/* Create Directories for Prometheus */
sudo mkdir /etc/prometheus
sudo mkdir /var/lib/prometheus

/* Download the latest Prometheus */
wget
https://github.com/prometheus/prometheus/releases/download/v2.52.0/prometheus-2.52.0.linux-amd6
4.tar.gz

/* Untar and set ownership to Prometheus User */
sudo tar xvf prometheus*.tar.gz
cd prometheus*/
sudo mv prometheus /usr/local/bin
sudo mv promtool /usr/local/bin
sudo chown prometheus:prometheus /usr/local/bin/prometheus
sudo chown prometheus:prometheus /usr/local/bin/promtool

/* Move the Configuration Files & Set Owner */
sudo mv consoles /etc/prometheus
sudo mv console_libraries /etc/prometheus
sudo mv prometheus.yml /etc/prometheus
sudo chown prometheus:prometheus /etc/prometheus
sudo chown prometheus:prometheus /etc/prometheus/*
sudo chown -R prometheus:prometheus /etc/prometheus/consoles
sudo chown -R prometheus:prometheus /etc/prometheus/console_libraries
```

```
sudo chown -R prometheus:prometheus /var/lib/prometheus
```

Create the Prometheus configuration files:

```
/* Edit the prometheus parameter file */
sudo vi /etc/prometheus/prometheus.yml

/* Basic Scrape Config */
global:
  scrape_interval: 15s

scrape_configs:
- job_name: postgres
  static_configs:
  - targets: ['postgres_exporter_machine_IP_address:9187']
```

Create a `system.d` service so that Prometheus will survive reboot:

```
/* Add the contents to the following file: /etc/systemd/system/prometheus.service */
[Unit]
Description=Prometheus
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
    --config.file /etc/prometheus/prometheus.yml \
    --storage.tsdb.path /var/lib/prometheus/ \
    --web.console.templates=/etc/prometheus/consoles \
    --web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target


/* Reload Systemd */
sudo systemctl daemon-reload

/* Start Prometheus Service */
sudo systemctl enable prometheus
```

```
sudo systemctl start prometheus
sudo systemctl status prometheus
```

Reload and start the Prometheus service:

```
/* Reload Systemd */
sudo systemctl daemon-reload

/* Start Prometheus Service */
sudo systemctl enable prometheus
sudo systemctl start prometheus
sudo systemctl status prometheus
```

Installing Grafana

Grafana is a dashboarding tool that exposes Prometheus metrics to an end user through a dashboard. Multiple standard dashboards are available for the Postgres Exporter and this observability example leverages those available dashboards. Grafana is available through normal `apt` and `yum` repositories and we leverage those to install this product.

Install on Ubuntu or Debian:

```
/* Install from apt for Ubuntu */
sudo apt-get update
sudo apt-get install grafana
```

Install on RHEL, CentOS, or Rocky Linux:

```
/* Import the GPG Key */
wget -q -O gpg.key https://rpm.grafana.com/gpg.key
sudo rpm --import gpg.key

/* Create /etc/yum.repos.d/grafana.repo with the following content: */
[grafana]
name=grafana
baseurl=https://rpm.grafana.com
repo_gpgcheck=1
enabled=1
gpgcheck=1
gpgkey=https://rpm.grafana.com/gpg.key
sslverify=1
```

```
sslcacert=/etc/pki/tls/certs/ca-bundle.crt

/* Install Open Source Grafana */
sudo dnf install grafana
```

Reload and start the service:

```
/* Reload Systemd */
sudo systemctl daemon-reload

/* Start Grafana Service */
sudo systemctl enable grafana-server
sudo systemctl start grafana-server
sudo systemctl status grafana-server
```

Standard Addresses for Postgres Exporter, Prometheus, and Grafana

```
/* Prometheus Address */
http://prometheus-host-ip:9090

/* Postgres Exporter Address */
http://postgres_exporter-host-ip:9187/metrics

/* Grafana Address */
http://grafana-host-ip:3000
```

Load a Dashboard to Grafana

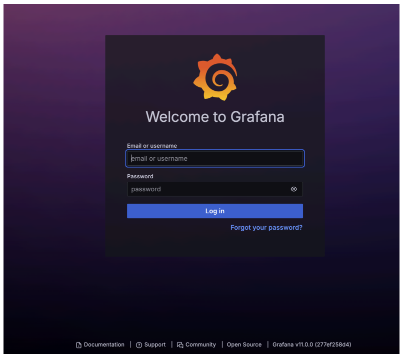
You can find general instructions on how to configure and operate open source Grafana on the Set up Grafana page.

While there are many public dashboards available, we use the following dashboard:

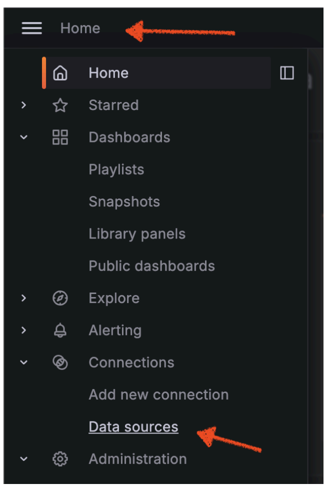
https://grafana.com/grafana/dashboards/13494-postgresql-statistics/

Create a data source

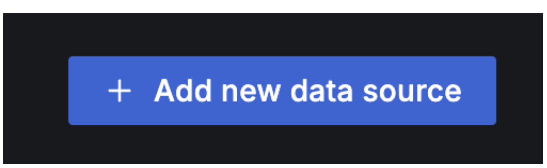
1. Navigate and log into the Grafana console using the Grafana address above. Both the default username and password are `admin`. Change this password.

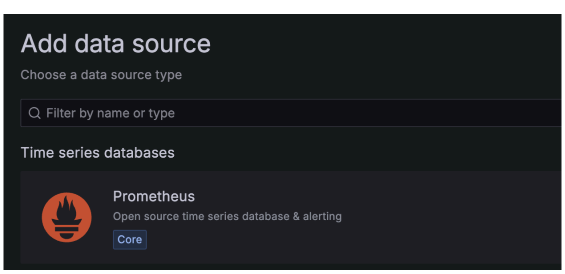


2. If the prometheus datasource has not yet been set up, go to **Home** > **Data sources**

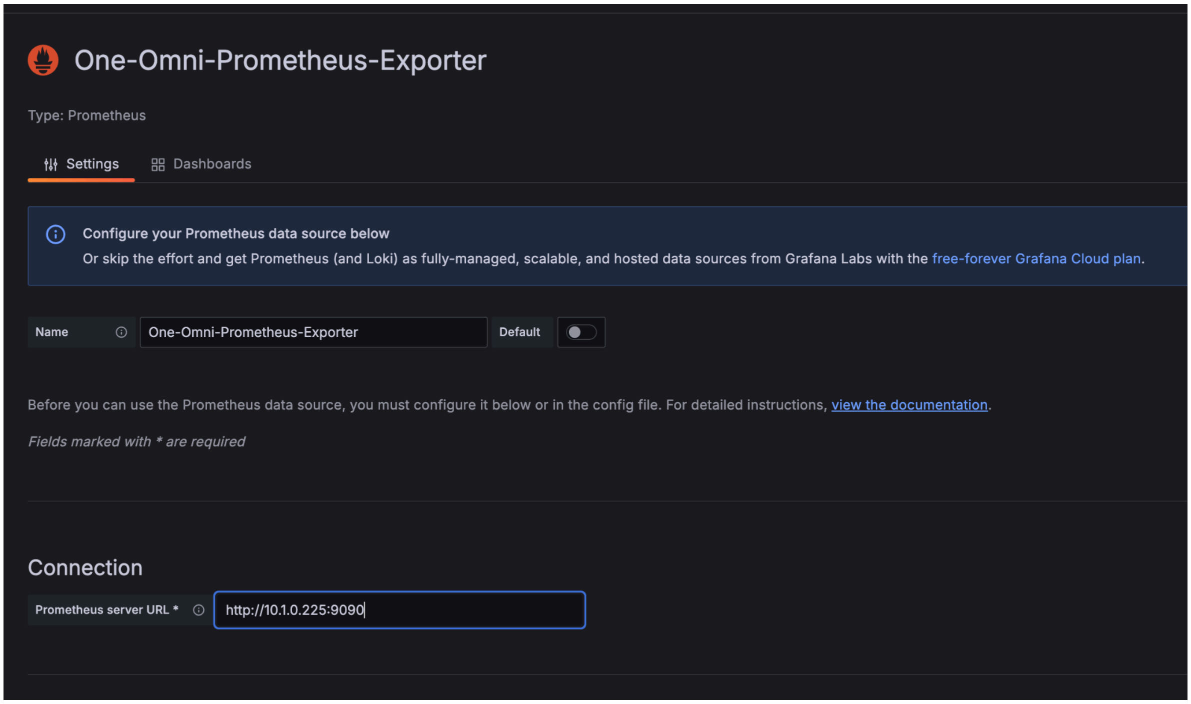


3. Click **Add new datasource** and select **Prometheus**.





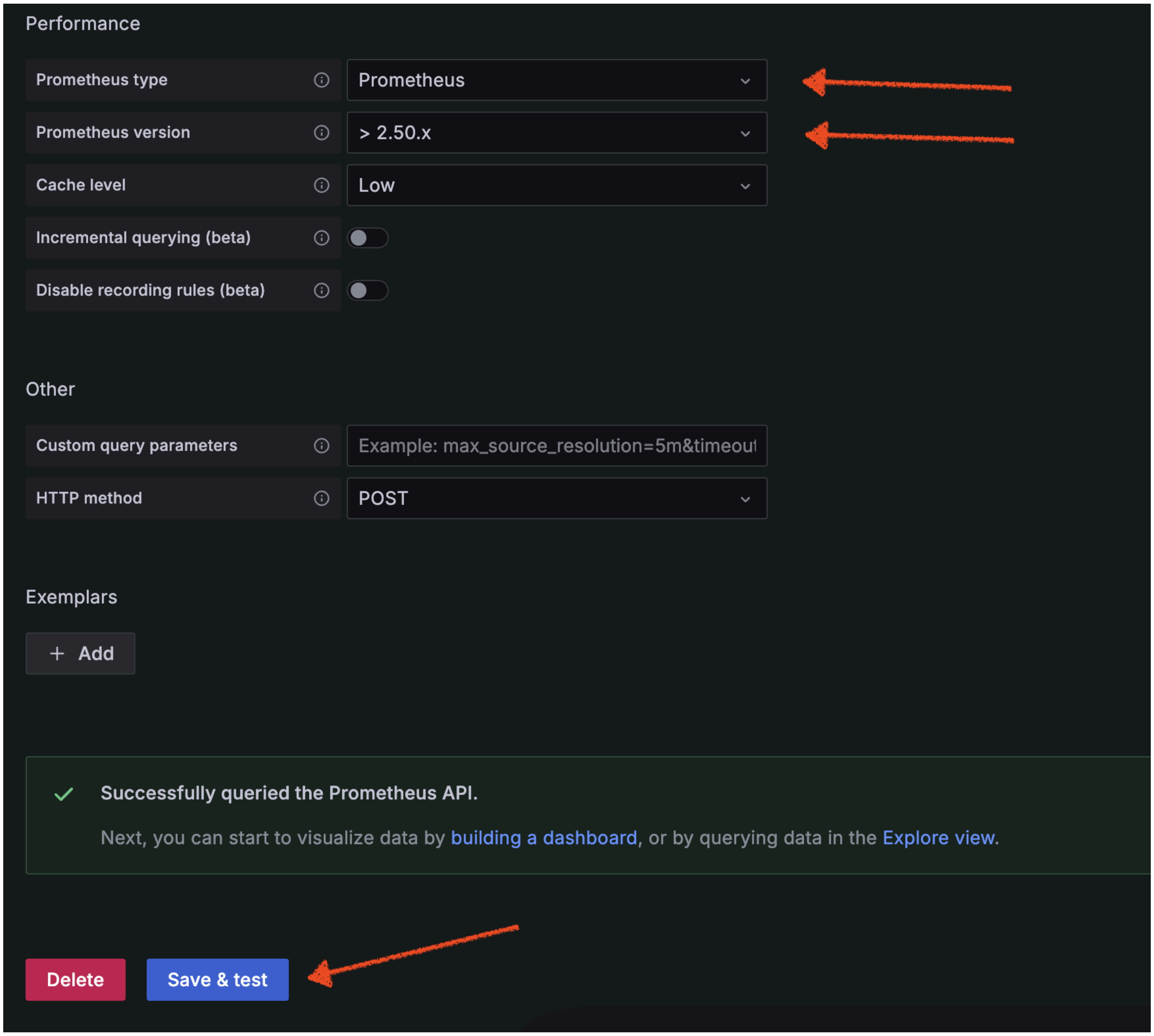
4. Enter the ip address and port of the prometheus server created in the previous step into the **Prometheus server URL** field.



For a basic configuration, leave everything as the defaults except for:

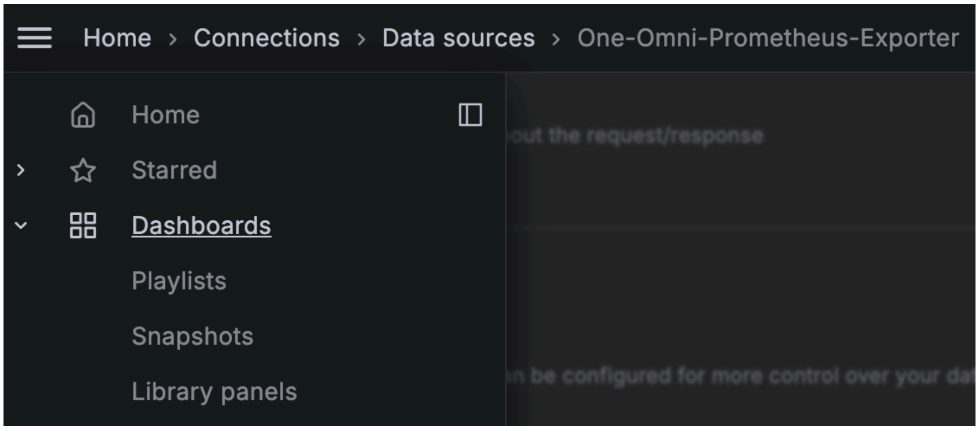
- **Prometheus type**: Select **Prometheus**
- **Prometheus version**: Select **> 2.5.x**

5. Click **Save & test**.

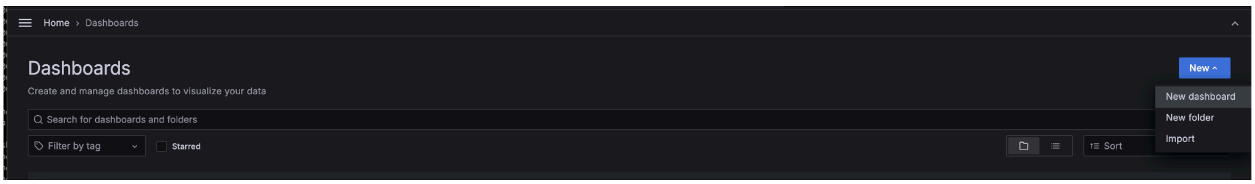


Create a dashboard

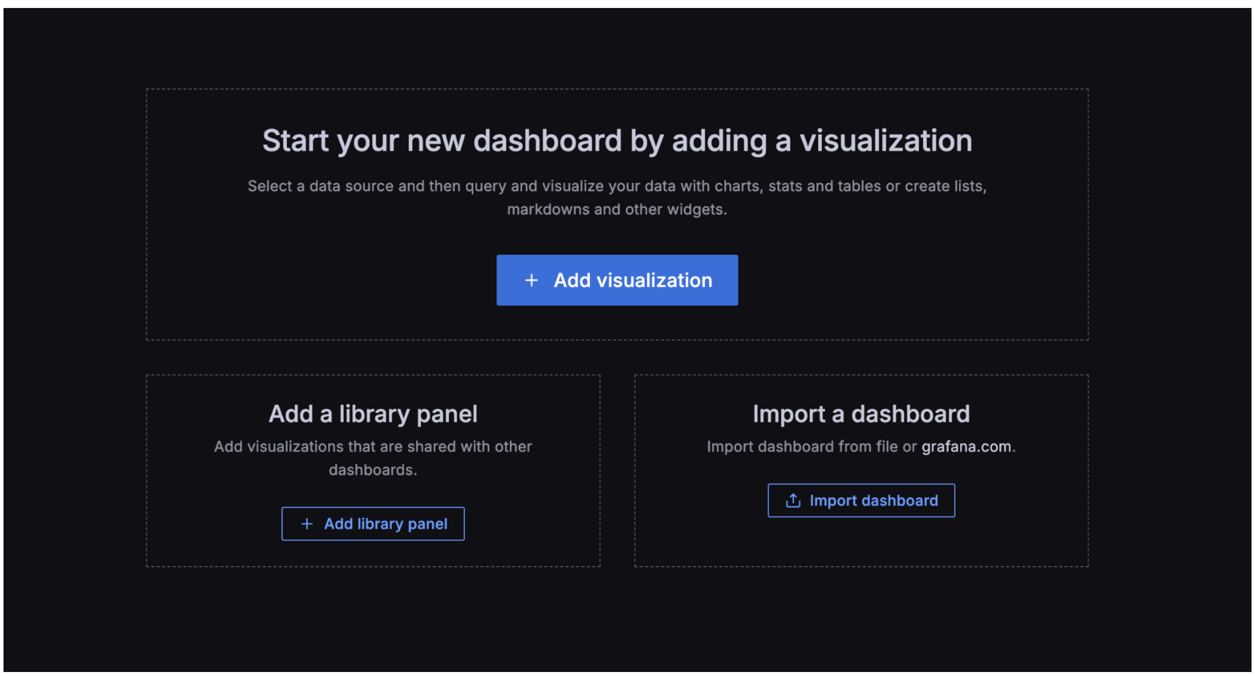
1. Go to **Home** > **Dashboards**.



2. Click **New**, and select **New dashboard**.

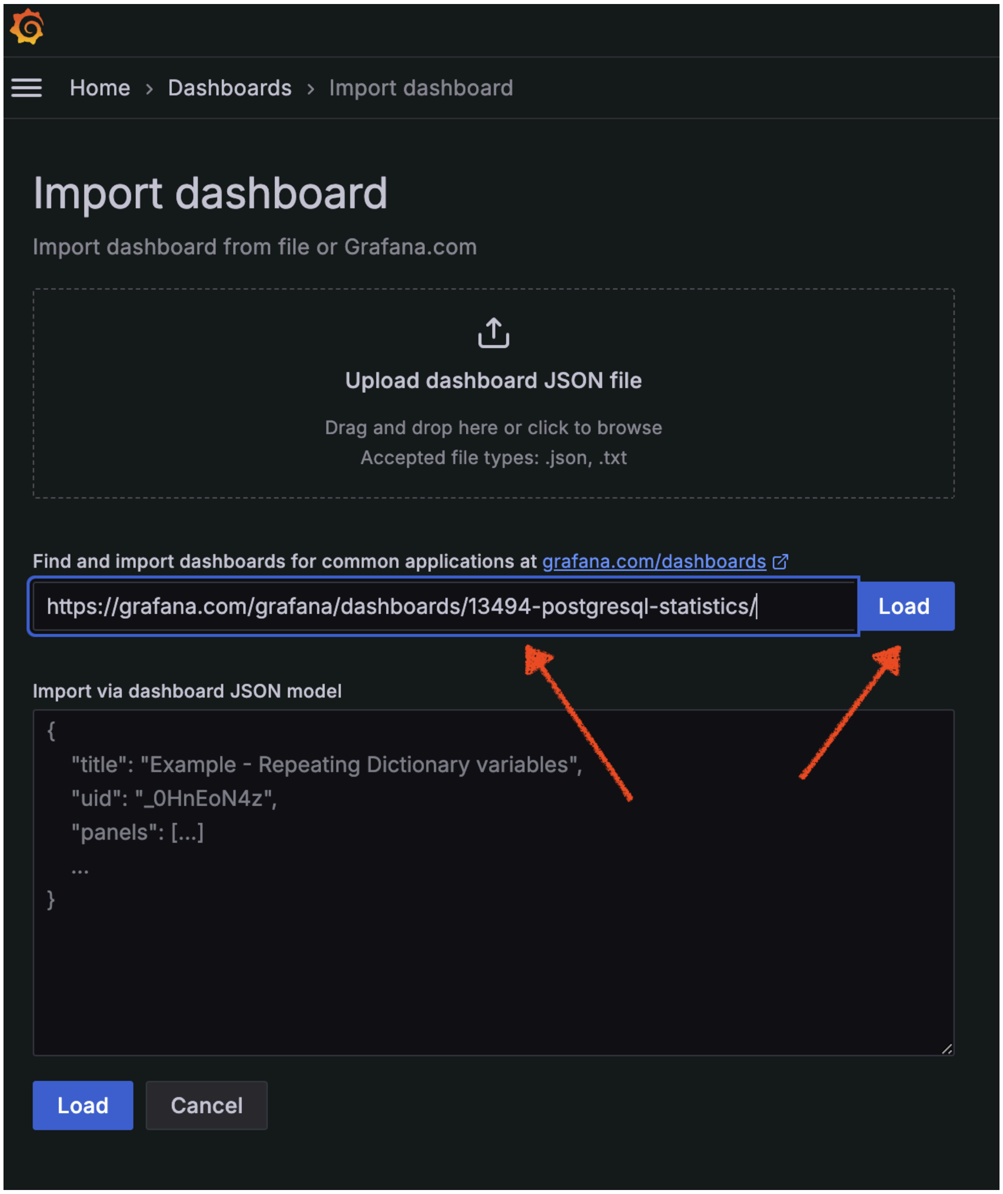


3. Click **Import dashboard**.

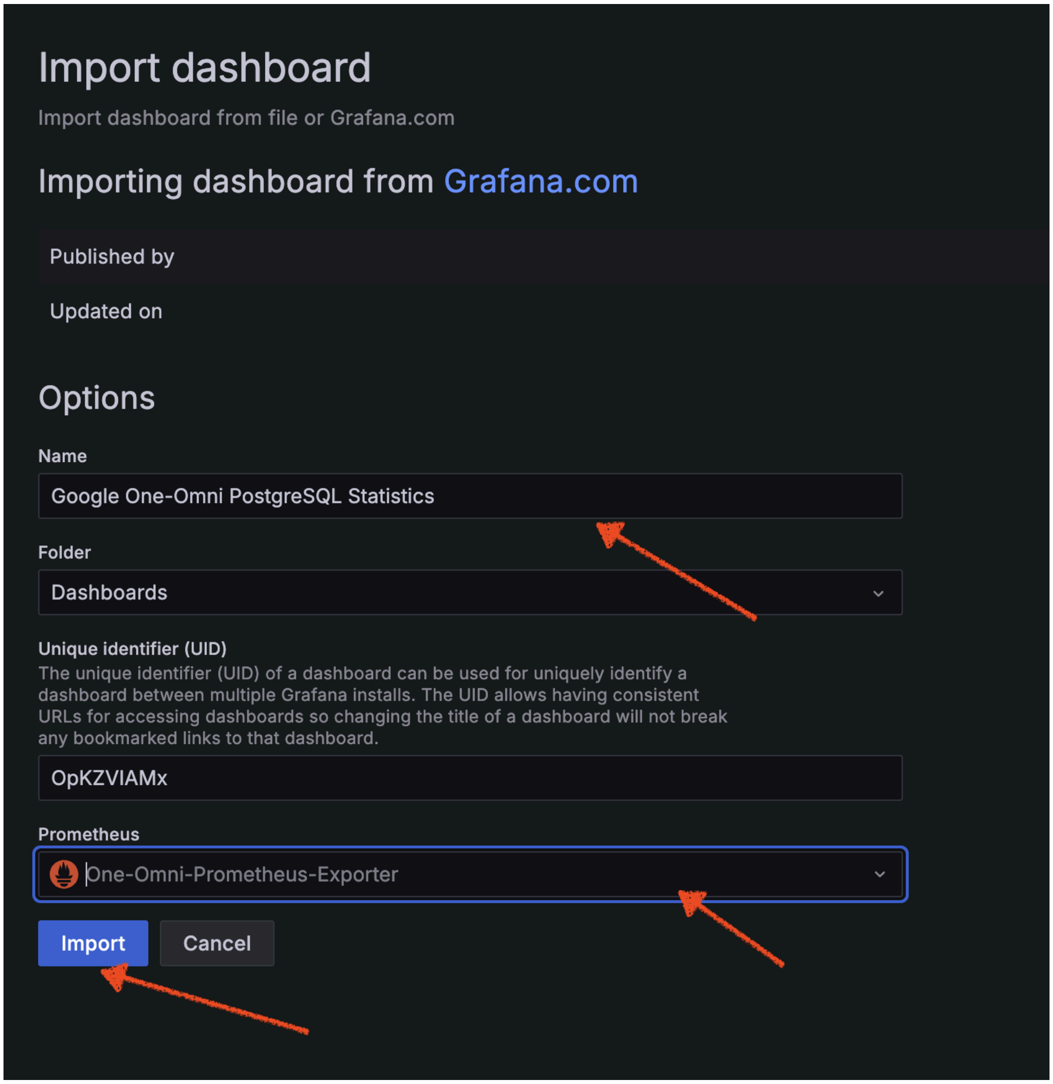


4. Import the dashboard using the following URL: https://grafana.com/grafana/dashboards/13494-postgresql-statist

5. Click **Load**.



6. Update the name of the dashboard
7. Enter the data source into the **Prometheus** field, and click **Import**.


Import dashboard
Import dashboard from file or Grafana.com
Importing dashboard from Grafana.com
Published by
Updated on
Options
Name
Google One-Omni PostgreSQL Statistics
Folder
Dashboards
Unique identifier (UID)
The unique identifier (UID) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The UID allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.
OpKZVIAMx
Prometheus
One-Omni-Prometheus-Exporter
Import
Cancel

After the import is complete, the following monitoring dashboard is available:

Perfsnap

Perfsnap is a tool within AlloyDB Omni which can be used to snapshot two time periods in order to obtain detailed observability information during the the two time periods. This can be especially helpful when lots of different data is needed to diagnose a specific time period. Proceed to the following [Perfsnap](#) section for additional details.

Recommended extensions for observability

Any extension can be added to AlloyDB Omni using the [instructions](#) in this guide.

In terms of observability, the following extensions are recommended to be installed:

- pg_stat_statements (included)
- pgSentinel (https://github.com/pgsentinel/pgsentinel) (not included)

Columnar engine observability

The columnar engine is best observed using standard scripts which can be executed from the psql command line or integrated into the dashboard of your choice. Standard observability scripts are detailed as follows:

General columnar engine observability scripts:

```
/* determine columnar engine settings */
SELECT
    name,
    setting,
    boot_val,
    reset_val
FROM
    pg_settings
WHERE
    name LIKE '%google_columnar_engine%'
ORDER BY
    1;

/* To view the list of recommended column detail: */
SELECT
      crc.schema_name AS schema_name,
      crc.relation_name AS table_name,
      pi.inhparent::regclass,
      crc.column_name,
      crc.column_format,
      crc.compression_level,
      crc.estimated_size_in_bytes
FROM g_columnar_recommended_columns crc
JOIN pg_stat_all_tables ps
      ON ps.schemaname::text = crc.schema_name
            AND ps.relname::text = crc.relation_name
JOIN pg_class pc
      ON ps.relid = pc.oid
LEFT JOIN pg_catalog.pg_inherits pi
      ON ps.relid = pi.inhrelid
ORDER BY 1,2,4 NULLS LAST;

/* List of items in the column store */
SELECT
    database_name,
    schema_name,
    relation_name,
    column_name,
    size_in_bytes,
    last_accessed_time
FROM
    g_columnar_columns;

SELECT
```

```
    *
FROM
    g_columnar_relations
ORDER BY
    relation_name;

/* To see current status of items in columnstore */
SELECT
    schema_name,
    relation_name,
    status,
    swap_status,
    sum(end_block - start_block) ttl_block,
    sum(invalid_block_count) invalid_block,
    round(100 * sum(invalid_block_count) / sum(end_block - start_block), 1) AS
invalid_block_perc,
    pg_size_pretty(sum(size)) ttl_size,
    pg_size_pretty(sum(cached_size_bytes)) ttl_cached_size
FROM
    g_columnar_units
WHERE
    g_columnar_units.database_name = current_database()
GROUP BY
    schema_name,
    relation_name,
    status,
    swap_status;

/* Check utilization of columnar memory */
select memory_name ,
       memory_total/1024/1024 memory_total_MB,
       memory_available/1024/1024 memory_available_MB ,
       memory_available_percentage
from g_columnar_memory_usage;


/* To see Columnar engine column Swap-out */
SELECT
    pg_size_pretty(memory_total) AS cc_allocated,
    pg_size_pretty(memory_total - memory_available) AS cc_consumed,
    pg_size_pretty(memory_available) cc_available,
    google_columnar_engine_local_storage_used () AS cc_local_storage_used_mb,
    google_columnar_engine_local_storage_available () AS cc_local_storage_avail_mb,
    CASE WHEN google_columnar_engine_local_storage_used () IS NOT NULL THEN
        'Swapped-out Column(s)'
    ELSE
```

```
        NULL
    END AS "SwapOut",
    (
        SELECT
            CONCAT_WS('-', STRING_AGG(DISTINCT g_columnar_units.relation_name, '/'), STATUS,
swap_status)
        FROM
            g_columnar_units
        GROUP BY
            status,
            swap_status) AS current_obj
FROM
    g_columnar_memory_usage
WHERE
    memory_name = 'main_pool';
```

Index advisor

An index advisor is included with AlloyDB Omni. This advisor keeps track of the queries being executed and suggests indexes that the query might benefit from. You can find more information about the index advisor on the Index advisor overview page.

Manage your AlloyDB Omni configuration

Default extensions to use

By design, a minimal set of extensions are loaded to One-Omni. For maximum observability, create the following extension in each user database:

```
CREATE EXTENSION pg_stat_statements;
```

Log location

The logs for the container, including the postgres log, can be viewed by executing the following command “docker logs [omni container name]”

Add extensions to AlloyDB Omni

Should your installation require additional extensions, they can be added to AlloyDB Omni. The base extensions included with the product are listed here:

https://cloud.google.com/alloydb/docs/reference/extensions

Each extension requires different steps to download / install and the existing process will not work with One-Omni.