PostgreSQL AQMS backup and restore

From PNSNWikiDocs

This page deals with the specifics of database backup on PNSN PostgreSQL AQMS systems. A general overview of PostgreSQL database backup can be found at PostgreSQL#backup/transfer of database.

Contents

backup overview

PostgreSQL have two mechanisms for backup that are relevant to us. We will follow the following multiply-redundant backup strategy:

- pp streaming replication - PostgreSQL streaming replication is used to continuously replicate the state of the 'master' post-processing database on pp1 or pp2 to the 'shadow' instance on the other pp machine. The 'shadow' instance is then becomes a hot-standby server with read-only access to its database.
- database dump - at a fixed interval (weekely?), the current state of the database is saved to a dump file and migrated to long-term storage.
- WAL shipping (AKA Point In Time Recovery or PITR) - logical Write-Ahead-Logs (WAL) are continuously written on the master system and streamed to the standby system. These can be either discarded or saved to allow spinning

up a database to the current state from a prior full database dump. This can be accomplished by periodically migrating WAL files from storage on the master to off-server storage. Older WAL files are cleaned when the next full database dump makes them superfluous.

overview on generating point-in-time backups

pg_dump/pg_dumpall

It is possible (and recommended by some) to do the database dump step with pg_dump or pg_dumpall, but the resulting file will not be a valid starting point for a WAL update. This is because the dumps do not contain information to allow the WAL files to sync to the state of the dump. This is called an 'exclusive' method, as database writes during the backup can put the backup in an inconsistent state. For AQMS we sill implement a more complicated 'non-exclusive' solution that will (in theory) allow restoration of current state from backup at any point.

pg_base_backup

The recommended way to do backups is to use the 'non-exclusive' pg_basebackup method. This can be done either using the low-level api's directly, or using the pg_basebackup shell command, which is a wrapper for the api's. pg_basebackup makes a binary copy of the database cluster files, while making sure the system is put in and out of backup mode automatically. Backups are always taken of the entire database cluster; it is not possible to back up individual databases or database objects. For individual database backups, a tool such as pg_dump must be used.

Both base backup strategies rely on replication slots. They can operate as a 'pull' operation from one host to another, but for now we will execute on the database host machine. Both require a pg_hba entry of the form:

```
local   replication,repmgr     repadmin                                 trust
host    replication,repmgr     repadmin        127.0.0.1/32             trust
host    replication,repmgr     repadmin        172.25.16.28/32          trust
host    replication,repmgr     repadmin        172.25.16.29/32          trust
```

pg_basebackup shell command

(the easy way)

See https://www.postgresql.org/docs/10/static/app-pgbasebackup.html

This is simple one-stop operation and can be run from a second connected machine using the replication infrastructure and user. This can save files as either

regular files or tar archives.

via low-level api's

(the hard way..but more flexible)

It is possible to duplicate the functionality of the shell command with even more flexibility using the underlying api calls. We won't use this, but it does give insight into what is going on under the hood.

From https://www.postgresql.org/docs/10/static/continuous-archiving.html:

1. Ensure that WAL archiving is enabled and working.
2. Connect to the server (it does not matter which database) as a user with rights to run pg_start_backup (superuser, or a user who has been granted EXECUTE on the function) and issue the command: 'SELECT pg_start_backup('label', false, false);' where label is any string you want to use to uniquely identify this backup operation.
    - The connection calling pg_start_backup must be maintained until the end of the backup, or the backup will be automatically aborted.
    - By default, pg_start_backup can take a long time to finish. This is because it performs a checkpoint, and the I/O required for the checkpoint will be spread out over a significant period of time, by default half your inter-checkpoint interval (see the configuration parameter checkpoint_completion_target). This is usually what you want, because it minimizes the impact on query processing. If you want to start the backup as soon as possible, change the second parameter to true, which will issue an immediate checkpoint using as much I/O as available.
    - The third parameter being false tells pg_start_backup to initiate a non-exclusive base backup.
3. Perform the backup, using any convenient file-system-backup tool such as tar or cpio (not pg_dump or pg_dumpall). It is neither necessary nor desirable to stop normal operation of the database while you do this. Backup will proceed to update state until the following command.
4. In the same connection as before, issue the command: 'SELECT * FROM pg_stop_backup(false, true);' This terminates backup mode.
    - On a primary, it also performs an automatic switch to the next WAL segment.
    - On a standby, it is not possible to automatically switch WAL segments, so you may wish to run pg_switch_wal on the primary to perform a manual switch. The reason for the switch is to arrange for the last WAL segment file written during the backup interval to be ready to archive.
    - The pg_stop_backup will return one row with three values. The second of these fields should be written to a file named backup_label in the root

directory of the backup. The third field should be written to a file named tablespace_map unless the field is empty. These files are vital to the backup working, and must be written without modification.

Once the WAL segment files active during the backup are archived, you are done. The file identified by pg_stop_backup's first return value is the last segment that is required to form a complete set of backup files. On a primary, if archive_mode is enabled and the wait_for_archive parameter is true, pg_stop_backup does not return until the last segment has been archived. On a standby, archive_mode must be always in order for pg_stop_backup to wait. Archiving of these files happens automatically since you have already configured archive_command. In most cases this happens quickly, but you are advised to monitor your archive system to ensure there are no delays. If the archive process has fallen behind because of failures of the archive command, it will keep retrying until the archive succeeds and the backup is complete. If you wish to place a time limit on the execution of pg_stop_backup, set an appropriate statement_timeout value, but make note that if pg_stop_backup terminates because of this your backup may not be valid.

If the backup process monitors and ensures that all WAL segment files required for the backup are successfully archived then the wait_for_archive parameter (which defaults to true) can be set to false to have pg_stop_backup return as soon as the stop backup record is written to the WAL. By default, pg_stop_backup will wait until all WAL files have been archived, which can take some time. This option must be used with caution: if WAL archiving is not monitored correctly then the backup might not include all of the WAL files and will therefore be incomplete and not able to be restored.

WAL files

Wall files will be found in the pg_wal subdirectory of the PostgreSQL data directory (e.g. /var/lib/pgsql/10/data/pg_wal). They are maintained by postgres using settins from postgresql.conf. For backup purposes, the most important setting is 'wall_keep_segments'. This configures the number of wal files to keep on master AFTER transmission to standby server. By default this is zero, but we want something much larger.

For backup purposes, one must keep a copy of all WAL files from the last snapshot backup in order to be able to restore current state. This can be done by automatically copying files from pg_wal to an archival location. This is done by adding something like following to postgresql.conf:

```
archive_mode = on
archive_command = 'test ! -f /archive/dbbackup/wal/%f && cp %p /archive/dbbackup/wal/%f'
```

Postgres will replace %f with the target filename and %p with the full target file

path. More complicated options can be used, including dedicated scripts with lots of error checking. The important thing is that the command needs to return zero exit status if and only if it succeeds.

'archive_mode = on' will archive only on primary database. To archive on both primary and standby, set archive_mode to 'always'.

It is extra safe to keep wal files back to the next to last snapshot. Cleanup is very easy using the pg_archivecleanup command:

```
postgres>$ pg_archivecleanup -d /archive/dbbackup/wal/ 000000010000003700000010.00000020.backup
```

This can also be incorporated in recovery.conf on the standby instance:

```
archive_cleanup_command = 'pg_archivecleanup /archive/dbbackup/wal/ %r'
```

implementation

base backup

The software for a point-in-time backup is wrapped in the bash script /var/lib /pgsql/scripts/backup.sh. It writes output to /archive/dbbackup/yymmdd /base.tar.gz, which should be independently automatically synced to other backup servers. The first run took less than a minute, but expect longer when our database gets bigger.

Run by crontab on all servers with xx1-xx2 pairs running alternate weeks.

We are setting our backups to include wal files pulled after the base backup, so the result is sufficient to restore the state of the system at the end of the backup.

wal archiving

WAL files are archived automatically to the /archive/dbbackup/wal directory where they can be independently rsynced off-host. They are intended to be used to restore the state of a system from the point of the last base backup to some later state.

WAL files are only archived on the PP primary, not the shadow.

At first we will keep these archived wal files for two base-backup cycles to provide redundancy and allow point-in-time recovery farther back in time. That policy may change in the future.

restore

We assume you are starting with a fairly bare freshly installed machine and an existing point-in-time backup.

First install database software as described in PostgreSQL#server installation. Next install all the extras as described in PostgreSQL AQMS.

Copy tarred base backup and wall files to new machine. In our case we will put them in /var/lib/postgres/10/restore.

```
[root@localhost 10]# chown -R postgres:postgres /var/lib/pgsql/10/restore
[root@localhost 10]# sudox postgres
-bash-4.2$ cd 10/restore
-bash-4.2$ tar -xvzf base.tar.gz
```

This expands in place (no subdir creation) filling the restore directory with the state of the system at the time of the last base_backup.

This is a good time to make necessary changes to postgresql.conf, pg_hba.conf etc. For example, you may want to comment the "shared_preload_libraries = 'pglogical'" line in postgresql.conf if that extension is not installed.

Copy the contents into the currently empty /var/lib/pgsql/10/data directory. Copy allows you to keep the original if things don't work right.

If you have archived wal files from after the last base_backup was taken,

1. cd to the data directory
2. delete files in the unzipped pg_wal directory. These are relative to the last base_backup.
3. copy any unarchived WAL files you might have into pg_wal.
4. create a file recovery.conf with minimum contents like:

```
restore_command = 'cp /var/lib/pgsql/10/restore/wal/%f "%p"'
```

On startup, the recovery.conf file is read as all the relevant WAL files are loaded into the database, bringing you up to the latest state. You can specify other target states. See https://www.postgresql.org/docs/10/static/recovery-target-settings.html.

Now, all that remains, as root:

```
[root@vtahoma ~]# systemctl start postgresql-10
[root@vtahoma ~]# systemctl status postgresql-10
● postgresql-10.service - PostgreSQL 10 database server
```

```
  Loaded: loaded (/usr/lib/systemd/system/postgresql-10.service; disabled; vendor preset: disabled)
  Active: active (running) since Tue 2018-10-02 19:19:31 EDT; 6s ago
    Docs: https://www.postgresql.org/docs/10/static/
 Process: 11169 ExecStartPre=/usr/pgsql-10/bin/postgresql-10-check-db-dir ${PGDATA} (code=exited, status
Main PID: 11174 (postmaster)
  CGroup: /system.slice/postgresql-10.service
          ├─11174 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data/
          ├─11176 postgres: logger process
          ├─11178 postgres: checkpointer process
          ├─11179 postgres: writer process
          ├─11180 postgres: wal writer process
          ├─11181 postgres: autovacuum launcher process
          ├─11182 postgres: archiver process
          ├─11183 postgres: stats collector process
          └─11184 postgres: bgworker: logical replication launcher
```

Log file will show something like:

```
2018-10-02 16:19:30 PDT @: [11177] LOG:  database system was interrupted; last known up at 2018-09-30 04:
2018-10-02 16:19:31 PDT @: [11177] LOG:  recovered replication state of node 1 to 0/1BC29D78
2018-10-02 16:19:31 PDT @: [11177] LOG:  recovered replication state of node 3 to 0/4CB09408
2018-10-02 16:19:31 PDT @: [11177] LOG:  redo starts at 4/D7000028
2018-10-02 16:19:31 PDT @: [11177] LOG:  consistent recovery state reached at 4/D72A69B0
2018-10-02 16:19:31 PDT @: [11177] LOG:  redo done at 4/D72A69B0
2018-10-02 16:19:31 PDT @: [11177] LOG:  last completed transaction was at log time 2018-09-30 04:06:35.2
2018-10-02 16:19:31 PDT @: [11174] LOG:  database system is ready to accept connections
```

Works! When the recovery to the most recent state is complete, 'recovery.conf' is automatically renamed to 'recovery.done' so that it won't run on next restart.

Retrieved from "https://internal.pnsn.org/LOCAL/WikiDocs/index.php?title=PostgreSQL_AQMS_backup_and_restore&oldid=49275"
Categories: PostgreSQL | AQMS

- This page was last modified on 16 October 2018, at 11:38.
- This page has been accessed 283 times.