PGDAY BELGIUM 2019

# Streaming Replication, the basics


17 May 2019

Stefan Fercot

# Streaming Replication, the basics

PGDAY BELGIUM 2019

TITRE : Streaming Replication, the basics
SOUS-TITRE : PGDAY BELGIUM 2019

DATE: 17 May 2019

## WHO AM I?

- Stefan Fercot
- aka. pgstef
- PostgreSQL user since 2010
- involved in the community since 2016
- @dalibo since 2017

## DALIBO

- Services

| Support | Training | Advice |
|---|---|---|

- Based in France
- Contributing to PostgreSQL community

# INTRODUCTION

Streaming Replication can be used to setup a replicated PostgreSQL cluster, with one or more standby servers ready to take over operations if the primary server fails.

This feature relies on the Write Ahead Logs, aka WALs, and offers a lot of possibilities.

The talk will summarize for beginners what WALs are, how does the Streaming Replication works, give some advices and best practices.

---

## WRITE-AHEAD LOG (WAL)

- transactions written sequentially
  - COMMIT when data are flushed to disk
- WAL replay after a crash
  - make the database consistent

WAL is the mechanism that PostgreSQL uses to ensure that no committed changes are lost. Transactions are written sequentially to the WAL and a transaction is considered to be committed when those writes are flushed to disk. Afterwards, a background process writes the changes into the main database cluster files (also known as the heap). In the event of a crash, the WAL is replayed to make the database consistent.

https://www.postgresql.org/docs/current/wal-intro.html

---

## POSTGRESQL WAL

- REDO log only
  - no UNDO log (yet)
  - instant rollback

---

## STRUCTURE

- WAL is divided into WAL segments
  - each segment is a file in pg_wal directory

```
$ ls pg_wal/
000000010000000010000002E
000000010000000010000002F
000000010000000000000030
...
```

---

## FILENAMES

- 000000010000000010000002E
  - 00000001 : TLI
  - 000000010000002E : LSN
    - 00000001 : log id
    - 0000002E : segment number

---

## CHECKPOINTS

- flush all data pages to disk
- write a checkpoint record
- recycle / remove old WAL

---

## ARCHIVING

- old WAL segments are deleted / recycled after a checkpoint
- can also be archived with `archive_command`

Allows online backups and Point-in-Time Recovery

---

## REPLICATION

- apply WAL when generated on a standby server
  - using WAL archives (files)
  - or by **streaming** over a TCP connection

---

## STREAMING REPLICATION

- architecture/compile flag dependent
- whole cluster only
- read-only standby
- no built-in cluster management
- no (easy) fail-back

---

# SETUP

---

## WAL_LEVEL

```
wal_level = 'replica'
```

---

## MAX_WAL_SENDERS

`max_wal_senders=10` (default from v10)

https://www.postgresql.org/docs/current/runtime-config-replication.html#GUC-MAX-WAL-SENDERS

---

## AUTHENTICATION

- On primary
  - `CREATE ROLE replicator WITH LOGIN REPLICATION;`
  - ... and setup a password
  - in `pg_hba.conf`
    - `host replication replicator standby_ip/32 md5`

On standby, specify user/password in f.e. `.pgpass`.

If not done previously on primary:

- change `listen_addresses = '*'`
- allow firewall
  - `firewall-cmd --permanent --add-service=postgresql`
  - `firewall-cmd --reload`

---

## DATA INITIALIZATION

```
$ pg_basebackup -D /var/lib/pgsql/11/data \
    -h primary -U replicator -R -P
```

- before v10, add `-X stream`

- -D, –pgdata=DIRECTORY receive base backup into directory
- -h, –host=HOSTNAME database server host or socket directory
- -U, –username=NAME connect as specified database user
- -R, –write-recovery-conf write recovery.conf for replication
- -S, –slot=SLOTNAME replication slot to use
- -P, –progress show progress information

---

## RECOVERY.CONF

- standby_mode
- primary_conninfo
- recovery_target_timeline

https://www.postgresql.org/docs/current/standby-settings.html

https://www.postgresql.org/docs/current/recovery-target-settings.html

---

### STANDBY_MODE

- `standby_mode=on`
- continuous recovery by fetching new WAL segments
  - using restore_command
  - by connecting to the primary server

Specifies whether to start the PostgreSQL server as a standby. If this parameter is on, the server will not stop recovery when the end of archived WAL is reached, but will keep trying to continue recovery by fetching new WAL segments using restore_command and/or by connecting to the primary server as specified by the primary_conninfo setting.

---

### PRIMARY_CONNINFO

- `primary_conninfo = 'user=replicator host=primary'`
- connection string to the primary server

Specifies a connection string to be used for the standby server to connect to the primary.

---

### RECOVERY_TARGET_TIMELINE

- particular timeline for recovery
  - `latest` is useful in a standby server
- new timeline created after a recovery
  - to identify the series of WAL records generated afterwards

Specifies recovering into a given timeline. The default is to recover along the same timeline that was current when the base backup was taken. Setting this to latest recovers to the latest timeline found in the archive, which is useful in a standby server.

---

### POSTGRESQL 12 CHANGES

"Integrate recovery.conf into postgresql.conf" (2018-11-25)

- recovery.signal / standby.signal
- pg_basebackup **-R** append `postgresql.auto.conf`

https://pgstef.github.io/2018/11/26/postgresql12_preview_recovery_conf_disappears.html

---

## START

```
# systemctl start postgresql-11
```

---

## PROCESSES

On primary:

- `walsender replicator ... streaming 0/3BD48728`

On standby:

- `walreceiver   streaming 0/3BD48728`

## MONITORING

- lag
  - amount of WAL records generated in the primary
  - not yet received / applied on the standby
- `pg_current_wal_lsn()` on the primary
- `pg_last_wal_receive_lsn()`, `pg_last_wal_replay_lsn()` on the standby

---

## PG_STAT_REPLICATION

On primary:

```
usename          | replicator
application_name | walreceiver
state            | streaming
sent_lsn         | 0/3BD48728
write_lsn        | 0/3BD48728
flush_lsn        | 0/3BD48728
replay_lsn       | 0/3BD48728
sync_state       | async
...
```

You can retrieve a list of WAL sender processes via the pg_stat_replication view. Large differences between pg_current_wal_lsn and the view's sent_lsn field might indicate that the primary server is under heavy load, while differences between sent_lsn and pg_last_wal_receive_lsn on the standby might indicate network delay, or that the standby is under heavy load.

---

## PG_STAT_WAL_RECEIVER

On standby:

```
status               | streaming
received_lsn         | 0/3BD48728
received_tli         | 1
...
```

On a hot standby, the status of the WAL receiver process can be retrieved via the pg_stat_wal_receiver view. A large difference between pg_last_wal_replay_lsn and the view's received_lsn indicates that WAL is being received faster than it can be replayed.

---

# FAIL-OVER

---

## SPLIT-BRAIN

- if standby server becomes new primary
  - make sure the old primary is no longer the primary
- avoid situations where both systems think they are the primary
  - lead to confusion and ultimately data loss

Some scary stories:

- https://github.blog/2018-10-30-oct21-post-incident-analysis
- https://aphyr.com/posts/288-the-network-is-reliable

---

## CHECK-UP BEFORE CLEAN PROMOTE

On primary:

```
# systemctl stop postgresql-11
$ pg_controldata -D /var/lib/pgsql/11/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               shut down
Latest checkpoint's REDO location:    0/3BD487D0
```

On standby:

```
$ psql -c 'CHECKPOINT;'
$ pg_controldata -D /var/lib/pgsql/11/data/ \
| grep -E '(Database cluster state)|(REDO location)'
Database cluster state:               in archive recovery
Latest checkpoint's REDO location:    0/3BD487D0
```

---

## PROMOTE

- `pg_ctl promote [-D datadir] [-W] [-t seconds] [-s]`
- `trigger_file` in recovery.conf

https://www.postgresql.org/docs/current/app-pg-ctl.html

trigger_file specifies a trigger file whose presence ends recovery in the standby. More frequently nowadays, you will promote the standby using pg_ctl promote.

---

## LOGS AFTER PROMOTE

```
LOG:  received promote request
LOG:  redo done at 0/3BD487D0
LOG:  last completed transaction was at log time ...
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

---

## FAIL-BACK

- old primary as a standby
  - full copy of the new primary
  - pg_rewind
    - `--source-pgdata`
    - `--source-server`

---

## PG_REWIND

- rewinding a cluster until its divergence with another
- needs `wal_log_hints` or data checksums
- `--dry-run`

https://www.postgresql.org/docs/current/app-pgrewind.html

To enable checksums at initdb : `PGSETUP_INITDB_OPTIONS="--data-checksums"`.

---

## PG_REWIND (2)

```
$ pg_rewind -D /var/lib/pgsql/11/data/ \
  --source-server="user=postgres host=primary" -P
connected to server
servers diverged at WAL location 0/3BD48840 on timeline 1
rewinding from last common checkpoint at 0/3BD487D0 on timeline 1
reading source file list
reading target file list
reading WAL in target
need to copy 196 MB (total source directory size is 561 MB)
200806/200806 kB (100%) copied
creating backup label and updating control file
syncing target data directory
Done!
```

---

# TROUBLES

What if the connection between primary and standby fails?

---

## REPLICATION SLOTS

- primary does not remove WAL segments
  - until received by all standbys
- `pg_create_physical_replication_slot('slot_name');`
- `primary_slot_name`
- `max_replication_slots = 10` (default from v10)

https://www.postgresql.org/docs/current/warm-standby.html#STREAMING-REPLICATION-SLOTS

---

## LOG-SHIPPING

Don't prevent the removal of old WAL segments, use the archives!

- `restore_command`
- `archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'`

https://www.postgresql.org/docs/current/archive-recovery-settings.html

---

## PITR

Combine with PITR backups for easier fail-backs!

- online backups
- the standby use archives from the PITR repository
  - to catchup the primary
- faster standby creation through backup restore
  - or refresh an old one

---

## SYNCHRONOUS REPLICATION

- `synchronous_commit`
  - off
  - local
  - remote_write
  - on
  - remote_apply
- can be applied **by transaction**

https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-SYNCHRONOUS-COMMIT

---

## SYNCHRONOUS_STANDBY_NAMES

- Single (9.1)
  - `synchronous_standby_names = s1,s2,s3`
- First (9.6)
  - `synchronous_standby_names = 2(s1,s2,s3)`
- Quorum (10)
  - `synchronous_standby_names = ANY 2(s1,s2,s3)`

https://www.postgresql.org/docs/current/runtime-config-replication.html#GUC-SYNCHRONOUS-STANDBY-NAMES

---

## HOT STANDBY AND CONFLICTS

- `DROP TABLE` on primary…
  - cannot wait for the end of queries on standby
- on standby (`max_standby_archive_delay` and `max_standby_streaming_delay`)
  - delay application of WAL record
  - or cancel the conflicting query

An example of the problem situation is an administrator on the primary server running DROP TABLE on a table that is currently being queried on the standby server. Clearly the standby query cannot continue if the DROP TABLE is applied on the standby. If this situation occurred on the primary, the DROP TABLE would wait until the other query had finished. But when DROP TABLE is run on the primary, the primary doesn't have information about what queries are running on the standby, so it will not wait for any such standby queries. The WAL change records come through to the standby while the standby query is still running, causing a conflict. The standby server must either delay application of the WAL records (and everything after them, too) or else cancel the conflicting query so that the DROP TABLE can be applied.

---

## EARLY CLEANUP

- cleanup on the primary
  - according to MVCC rules
  - remove row versions still visible to a transaction on the standby
- `hot_standby_feedback`
  - or replication slots…

Similarly, hot_standby_feedback and vacuum_defer_cleanup_age provide protection against relevant rows being removed by vacuum, but the former provides no protection during any time period when the standby is not connected, and the latter often needs to be set to a high value to provide adequate protection. Replication slots overcome these disadvantages.

- https://www.postgresql.org/docs/devel/runtime-config-replication.html#GUC-HOT-STANDBY-FEEDBACK
- https://www.postgresql.org/docs/devel/runtime-config-replication.html#GUC-VACUUM-DEFER-CLEANUP-AGE

---

## UPDATES

- different minor release on primary and standby usually works
  - not advised!
- update the standby servers first

In general, log shipping between servers running different major PostgreSQL release levels is not possible. It is the policy of the PostgreSQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

# TOOLS

## AUTOMATED FAIL-OVER

- Patroni
- repmgr
- PAF

### PATRONI

- Python
- “template” for high-availability
  - with ZooKeeper, etcd, Consul or Kubernetes
- integrates with HAProxy

https://patroni.readthedocs.io/

### REPMGR

- fewer prerequisites
- easier for manual processing
  - repmgrd for automatic fail-over
  - witness to avoid split-brain
- no connection management

https://repmgr.org/

### PAF

- agent for Pacemaker/Corosync
  - linux HA
  - possible management of other services
- connection routing with virtual IP
- STONITH

https://clusterlabs.github.io/PAF/

---

## PITR

- pgBackRest, …
  - … but that's for another talk!

---

### PGBACKREST MAIN FEATURES

- custom protocol
  - local or remote operation (via SSH)
- multi-process
- full/differential/incremental backup
- backup rotation and archive expiration
- parallel, asynchronous WAL push and get
- Amazon S3 support
- encryption
- …

---

## LOGICAL REPLICATION

- reconstructs changes by row
- replicates row content, not SQL statements
- table-level partial / bi-directional replication
- data replication only
  - no schema
  - no sequences
- suitable for data distribution
  - but not for **HA** !

## CONCLUSION

- consolidated during 9.x versions
- out of the box in 10
  - wal_level
  - max_wal_senders
  - ...

# THANK YOU FOR YOUR ATTENTION!