Nordic PGDay

Point-in-time Recovery, target 2020

DALIBO
L'expertise PostgreSQL

24 March 2020

Stefan Fercot

Point-in-time Recovery, target 2020

Nordic PGDay

TITRE : Point-in-time Recovery, target 2020
SOUS-TITRE : Nordic PGDay

DATE: 24 March 2020

WHO AM I?

- Stefan Fercot
- aka. pgstef
- https://pgstef.github.io
- PostgreSQL user since 2010
- pgBackRest fan
- @dalibo since 2017

DALIBO

- Services

Support	Training	Advice

- Based in France
- Contributing to PostgreSQL community

INTRODUCTION

- What is WAL?
- Point-In-Time Recovery (PITR)
  - WAL archives
  - File-system-level backup
  - Restore
- PITR Tools

Your pg_dump takes forever? You want to save your data more frequently? Have you ever heard of Point-in-time recovery?

In his talk, we'll introduce what is called Point-in-time Recovery (aka "live backup").

We'll see how to achieve it step-by-step if you want to do manually: * archive_command / pg_recievewal; * pg_basebackup; * exclusive backup; * non-exclusive backup; * restore.

PostgreSQL 12 brought a significant change in this area with the removal of the recovery configuration file. We'll see more precisely the impact of this change.

We'll then mention some interesting backup (and restore) tools and give some key points to compare them (documentation, parallel execution, compression, incremental backups,...).

WHAT IS WAL?

- write-ahead log
  - transaction log (aka xlog)
- usually 16 MB (default)
  - `--wal-segsize` *initdb* parameter to change it
- pg_xlog (<= v9.6) -> pg_wal (v10+)
- designed to prevent data loss in most situations

https://fr.slideshare.net/PGDayAmsterdam/pgdayamsterdam-2018-devrim-gunduz-wal-everything-you-want-to-know

---

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

---

DATA MODIFICATIONS

- transactions modify data in `shared_buffers`
- checkpoints and background writer…
  - … push all dirty buffers to the storage

Remark: back-ends may also write data to the storage

DATA MODIFICATIONS (2)

POINT-IN-TIME RECOVERY (PITR)

- combine
  - file-system-level backup
  - continuous archiving of WAL files
- restore the file-system-level backup and replay archived WAL files

https://www.postgresql.org/docs/current/continuous-archiving.html

---

BENEFITS

- live backup
- less data-losses
- not mandatory to replay WAL entries all the way to the end

DRAWBACKS

- complete cluster backup…
  - … and restore
- big storage space (data + WAL archives)
- WAL clean-up blocked if archiving fails
- not as simple as `pg_dump`

WAL ARCHIVES

- 2 possibilities
  - archiver process
  - `pg_receivewal` (via *Streaming Replication*)

ARCHIVER PROCESS

- configuration (`postgresql.conf`)
  - `wal_level = replica`
  - `archive_mode = on` or `always`
  - `archive_command = '... some command ...'`
  - `archive_timeout = 0`
- don't forget to flush the file on disk!

PG_RECEIVEWAL

- archiving via *Streaming Replication*
- writes locally WAL files
- supposed to get data faster than the archiver process
- replication slot advised!

```
$ pg_receivewal --help
pg_receivewal receives PostgreSQL streaming write-ahead logs.

Usage:
  pg_receivewal [OPTION]...

Options:
  -D, --directory=DIR    receive write-ahead log files into this directory
  -E, --endpos=LSN       exit after receiving the specified LSN
      --if-not-exists    do not error if slot already exists when creating a slot
  -n, --no-loop          do not loop on connection lost
      --no-sync          do not wait for changes to be written safely to disk
  -s, --status-interval=SECS
                         time between status packets sent to server (default: 10)
  -S, --slot=SLOTNAME    replication slot to use
      --synchronous      flush write-ahead log immediately after writing
  -v, --verbose          output verbose messages
  -V, --version          output version information, then exit
  -Z, --compress=0-9     compress logs with given compression level
  -?, --help             show this help, then exit

Connection options:
  -d, --dbname=CONNSTR   connection string
  -h, --host=HOSTNAME    database server host or socket directory
  -p, --port=PORT        database server port number
  -U, --username=NAME    connect as specified database user
  -w, --no-password      never prompt for password
  -W, --password         force password prompt (should happen automatically)

Optional actions:
      --create-slot      create a new replication slot (for the slot's name see --slot
      --drop-slot        drop the replication slot (for the slot's name see --slot)

Report bugs to <pgsql-bugs@lists.postgresql.org>.
```

BENEFITS AND DRAWBACKS

- archiver process
  - easy to setup
  - maximum 1 WAL possible to lose
- `pg_receivewal`
  - more complex implementation
  - only the last transactions are lost

FILE-SYSTEM-LEVEL BACKUP

- pg_basebackup
- manual steps

PG_BASEBACKUP

- takes a file-system-level copy
  - using *Streaming Replication* connection(s)
- collects WAL archives during (or after) the copy
- no incremental backup

```
$ pg_basebackup --format=tar --wal-method=stream \
 --checkpoint=fast --progress -h HOSTNAME -U NAME \
 -D DIRECTORY
```

```
$ pg_basebackup --help
pg_basebackup takes a base backup of a running PostgreSQL server.

Usage:
  pg_basebackup [OPTION]...

Options controlling the output:
  -D, --pgdata=DIRECTORY receive base backup into directory
  -F, --format=p|t       output format (plain (default), tar)
  -r, --max-rate=RATE    maximum transfer rate to transfer data directory
                         (in kB/s, or use suffix "k" or "M")
  -R, --write-recovery-conf
                         write configuration for replication
  -T, --tablespace-mapping=OLDDIR=NEWDIR
                         relocate tablespace in OLDDIR to NEWDIR
      --waldir=WALDIR    location for the write-ahead log directory
  -X, --wal-method=none|fetch|stream
                         include required WAL files with specified method
  -z, --gzip             compress tar output
  -Z, --compress=0-9     compress tar output with given compression level

General options:
  -c, --checkpoint=fast|spread
                         set fast or spread checkpointing
  -C, --create-slot      create replication slot
  -l, --label=LABEL      set backup label
  -n, --no-clean         do not clean up after errors
  -N, --no-sync          do not wait for changes to be written safely to disk
  -P, --progress         show progress information
  -S, --slot=SLOTNAME    replication slot to use
  -v, --verbose          output verbose messages
  -V, --version          output version information, then exit
      --no-slot          prevent creation of temporary replication slot
      --no-verify-checksums
                         do not verify checksums
  -?, --help             show this help, then exit

Connection options:
  -d, --dbname=CONNSTR   connection string
  -h, --host=HOSTNAME    database server host or socket directory
  -p, --port=PORT        database server port number
  -s, --status-interval=INTERVAL
                         time between status packets sent to server (in seconds)
  -U, --username=NAME    connect as specified database user
  -w, --no-password      never prompt for password
  -W, --password         force password prompt (should happen automatically)

Report bugs to <pgsql-bugs@lists.postgresql.org>.
```

MANUAL STEPS

- `pg_start_backup()`
- manual file-system-level copy
- `pg_stop_backup()`

PG_START_BACKUP()

`SELECT pg_start_backup (`

- `label` : arbitrary user-defined text
- `fast` : immediate checkpoint?
- `exclusive` : exclusive mode?

`)`

EXCLUSIVE MODE

- easy to use but deprecated since 9.6
- `pg_start_backup()`
  - writes `backup_label`, `tablespace_map`
- works only on primary servers

NON-EXCLUSIVE MODE

- `pg_stop_backup()`
  - executed in the same connection as `pg_start_backup()`!
  - returns `backup_label` and `tablespace_map` content

When used in exclusive mode, `pg_start_backup()` writes a backup label file (`backup_label`) and, if there are any links in the `pg_tblspc/` directory, a tablespace map file (`tablespace_map`) into the data directory.

When used in non-exclusive mode, the contents of these files are instead returned by the `pg_stop_backup` function, and should be written to the backup by the caller.

If the server crashes during a backup, the exclusive mode may lead to some confusion by getting a message like:

```
HINT: If you are not restoring from a backup, try removing the file
"<path to $PGDATA goes here>/backup_label"
```

See this mail for more information about that.

Point-in-time Recovery, target 2020

DATA COPY

- copy data files while PostgreSQL is running
  - *PGDATA* directory
  - tablespaces
- inconsistency protection with WAL archives
- ignore
  - `postmaster.pid`, `postmaster.opts`, `pg_internal.init`
  - `log`, `pg_wal`, `pg_replslot`,...
- don't forget configuration files!

https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-LOWLEVEL-BASE-BACKUP-DATA

PG_STOP_BACKUP()

`SELECT * FROM pg_stop_backup (`

- exclusive
- wait_for_archive

`)`

- on primary server
  - automatic switch to the next WAL segment
- on standby server
  - consider using `pg_switch_wal()` on the primary...

SUMMARY

RESTORE

- recovery procedure is simple but…
  - must be followed carefully!

https://www.postgresql.org/docs/current/continuous-archiving.html#BACKUP-PITR-RECOVERY

RECOVERY STEPS (1/5)

- stop the server if it's running
- keep a temporary copy of your PGDATA / tablespaces
  - or at least the `pg_wal` directory
- remove the content of PGDATA / tablespaces directories

RECOVERY STEPS (2/5)

- restore database files from your file system backup
  - pay attention to ownership and permissions
  - verify tablespaces symbolic links
- remove content of `pg_wal` (if not already the case)
- copy unarchived WAL segment files

RECOVERY STEPS (3/5)

- configure the recovery…
  - before v12: `recovery.conf`
  - after: `postgresql.conf` + `recovery.signal`
- `restore_command = '... some command ...'`
- prevent ordinary connections in `pg_hba.conf` if needed

POSTGRESQL 12

Integrate recovery.conf into postgresql.conf

```
recovery.conf settings are now set in postgresql.conf (or other GUC
sources). Currently, all the affected settings are PGC_POSTMASTER;
this could be refined in the future case by case.

Recovery is now initiated by a file recovery.signal. Standby mode is
initiated by a file standby.signal. The standby_mode setting is
gone. If a recovery.conf file is found, an error is issued.

...

pg_basebackup -R now appends settings to postgresql.auto.conf and
creates a standby.signal file.
```

# 2dedf4d9a899b36d1a8ed29be5efbd1b31a8fe85

https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=2dedf4d9a899b36d1a8ed29be5efb

---

RECOVERY STEPS (4/5)

- recovery target:
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
  - `recovery_target_inclusive`
- timeline to follow:
  - `recovery_target_timeline`
- action once recovery target is reached?
  - `recovery_target_action`
  - `pg_wal_replay_resume`

https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-ARCHIVE-RECOVERY

RECOVERY STEPS (5/5)

- start the server
- watch the restore process
  - until consistent recovery state reached
- inspect your data

LSN

- log sequence number
  - position of the record in WAL file
  - provides uniqueness for each WAL record

```
=# SELECT pg_current_wal_lsn();
 pg_current_wal_lsn
--------------------
 2/3002020
(1 row)

=# SELECT pg_walfile_name(pg_current_wal_lsn());
     pg_walfile_name
--------------------------
 000000010000000200000003
(1 row)
```

TIMELINES

- archive recovery complete -> new timeline
  - part of WAL segment file names
  - to identify the series of WAL records generated after that recover
  - `.history` files
- `recovery_target_timeline`
  - default: `latest` (v12+) or `current` (< v12)

TIMELINES (2)

WAL FILENAME

- 000000010000000200000003
  - 00000001 : timeline
  - 00000002 : wal
  - 00000003 : segment
- hexadecimal
  - 000000010000000000000001
  - 0000000100000000000000FF
  - 000000010000000100000000
  - …

Since version 9.3, segment names are from 00000000 to 000000FF. Previously, to 000000FE.

PITR TOOLS

- tools make life easier
  - pgBackRest
  - pitrery
  - Barman
  - WAL-G
- providing
  - backup, restore, purge methods
  - archiving commands

PGBACKREST

- written in C (since version 2.21)
- custom protocol
  - local or remote operation (via SSH)
- full/differential/incremental backup
- parallel, asynchronous WAL push and get
- Amazon S3 support

https://pgbackrest.org

PITRERY

- set of Bash scripts
  - `archive_wal`
  - `pitrery`
  - `restore_wal`
- *push* mode (*SSH*)
- mono-server
- *tar* or *rsync* backup method

https://dalibo.github.io/pitrery

Point-in-time Recovery, target 2020

BARMAN

- written in Python
- remote backups (*pull* mode)
  - via *SSH*
  - or *Streaming Replication*
- handles multiple servers
- `pg_receivewal` & `pg_basebackup` support

https://www.pgbarman.org

Because Barman transparently makes use of `pg_basebackup`, features such as incremental backup, parallel backup, deduplication, and network compression are currently not available.

WAL-G

- written in Go
- based on WAL-E
- storage
  - Amazon S3
  - Google Cloud
  - Azure
  - local

https://github.com/wal-g/wal-g

WHAT IS A GOOD DATABASE BACKUP TOOL?

- usable
  - documentation & support
  - out-of-box automatization of various routines
- scalable
  - parallel execution
  - compression
  - incremental & differential backups
- reliable
  - Schrödinger's backup law
    - *The condition of any backup is unknown until a restore is attempted*

https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2098/slides/123/Advanced%20bac

---

WAL ARCHIVES

	archive_command	restore_command	`pg_receivewal`
pgBackRest	YES(+ archive-async)	YES(+ archive-async)	NO
pitrery	YES	YES	NO
Barman	YES	YES	YES
WAL-G	YES	YES(+ wal prefetch)	NO

ENCRYPTION

		method
pgBackRest	YES	aes-256-cbc
pitrery	NO	
Barman	NO	
WAL-G	YES	S3 server-side / libsodium

PARALLEL EXECUTION

	backup, restore	archiving	parameters
pgBackRest	YES	YES	process-max
pitrery	NO	NO	
Barman	YES rsync	NO	parallel_jobs
WAL-G	YES	YES	WALG_*_CONCURRENCY

COMPRESSION

	backups	archives	how?
pgBackRest	YES	YES	gzip
pitrery	YES tar	YES	gzip, pigz, bzip2,…
Barman	NO	YES	gzip, pigz, bzip2,…
WAL-G	YES	YES	lz4, lzma, brotli

NETWORK

	network compression	bandwidth limit
pgBackRest	YES	NO
pitrery	NO	YES rsync
Barman	YES rsync	YES rsync
WAL-G	NO	YES

INCREMENTAL BACKUPS

		how?
pgBackRest	YES	`--type=incr--type=diff`
pitrery	YES rsync	hardlinks
Barman	YES rsync	hardlinks
WAL-G	YES	WALG_DELTA_MAX_STEPSWALG_DELTA_ORIGIN

Point-in-time Recovery, target 2020

USEFUL RESOURCES

- Devrim Gündüz - WAL: Everything You Want to Know
- PostgreSQL docs - WAL introduction
- PostgreSQL docs - Continuous Archiving and PITR
- Anastasia Lubennikova - Advanced backup methods

CONCLUSION

- PITR is
  - reliable
  - fast[er than pg_dump]
  - continuous
- tools make life easier
  - choose wisely…
  - validate your backups!

QUESTIONS?