pgBackRest magic you'll wish you knew sooner

data egret

Your remote PostgreSQL DBA team

Stefan FERCOT

stefan.fercot@dataegret.com

SECURING YOUR POSTGRESQL DATABASE AVAILABILITY AND HIGH PERFORMANCE

- **Migration**
- Performance **audit**
- **Cloud Cost** management
- **Backup & restore**
- **Architecture** review
- **DataOps/CDC** projects
- **Consulting** for data science and analytics teams
- **PostgreSQL Courses** for DBA and Developers

Contributing sponsor to PostgreSQL

Open Alliance
For PostgreSQL Education

OSB Open Source Business ALLIANCE
Bundesverband Für digitale Souveränität e.V.
MITGLIED

EXPERTISE

Senior DBA with 10+ years of experience in PostgreSQL administration.

DEVELOPMENT

PostgreSQL Contributors involved in new PostgreSQL feature and extention development.

CUSTOMISATION

Flexible approach and dedicated team focused on success of your project.

COMMUNITY

Recognised as Significant Contributing Sponsor to PostgreSQL.

Stefan Fercot

- PostgreSQL Expert **@Data Egret**
  - PostgreSQL consulting, support, and training
- pgBackRest contributor and advocate
- Also known as pgstef
- https://pgstef.github.io


What is pgBackRest?

- Reliable backup and restore tool for PostgreSQL
- Supports local and remote operation (via SSH or TLS)
- Parallel and asynchronous operations
- Works with S3, Azure, GCS, NFS, and other storage backends
- Multiple compression methods (`gz` , `bz2` , `lz4` , `zst`)
- Supports client-side encryption (`aes-256-cbc`)

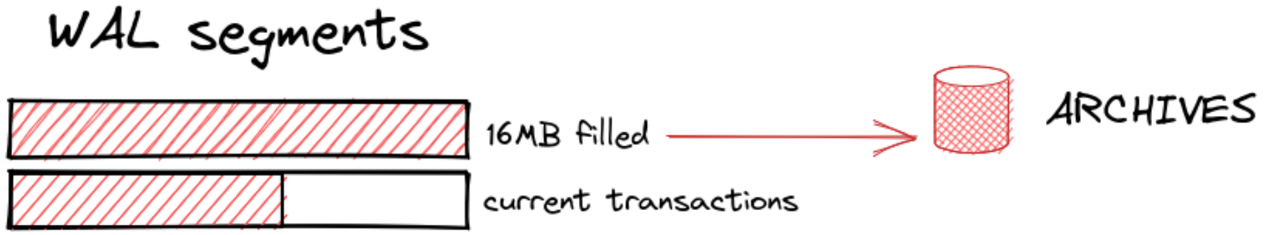
pgBackRest: faster WAL archiving, safer restores

- Archiving mechanics
- Restore scenarios

Archiving: the happy path and the traps

- `archive_command` vs `pg_receivewal` (not supported)
- How can we make archiving faster?
- What can go wrong?

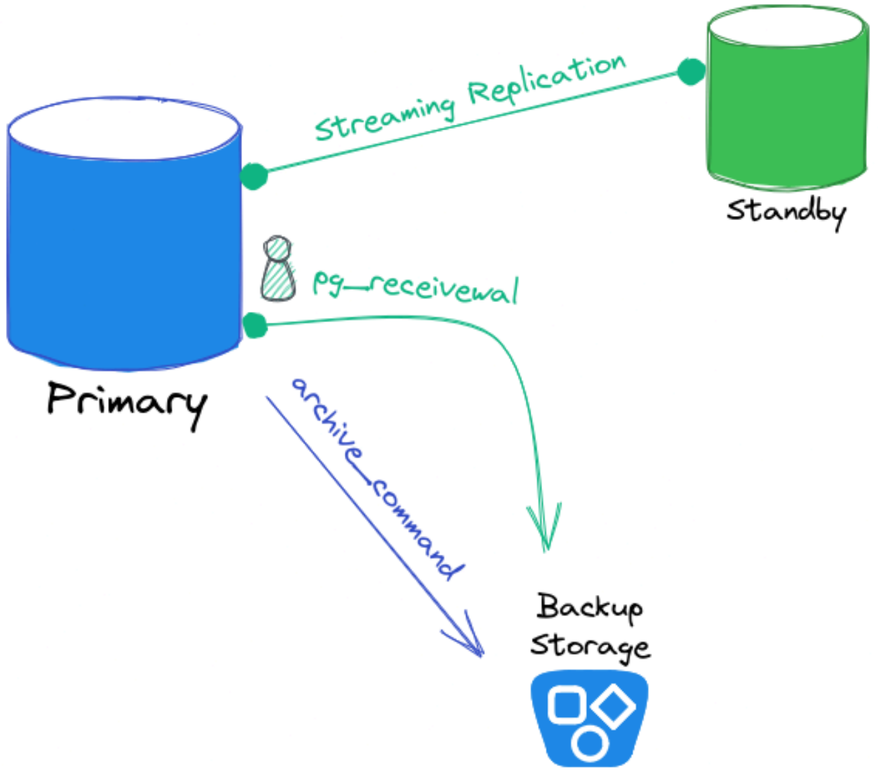
WAL archiving process



`.partial` WAL file

- Usually < 16MB
- `pg_receivewal` behaves like a standby without data files
- `.partial` is pushed by the standby server at promotion time

Example (1)


Streaming Replication
Standby
pg_receivewal
Primary
archive_command
Backup
Storage

Example (2)

```
archive_command = 'gzip < %p > /shared/archives/%f.gz'
```

```
pg_receivewal -D /shared/receivewal --compress=gzip -v
```

```
/usr/pgsql-18/bin/pgbench -i -s 65
```

Example (3)

```
$ ps -o pid,cmd fx
    PID CMD
   5461 /usr/pgsql-18/bin/postgres -D /var/lib/pgsql/18/data/
   ...
   5537  \_ postgres: walsender ... streaming 0/34E74D38
   5590  \_ postgres: walsender ... streaming 0/34E74D38
```

```
$ ls /shared/archives
...
000000010000000000000033.gz
```

```
$ ls /shared/receivewal
...
000000010000000000000033.gz
000000010000000000000034.gz.partial
```

Example (4)

```
$ psql -c "SELECT pg_promote();"
```

```
$ ls /shared/archives/
...
000000010000000000000034.partial.gz
00000002.history.gz
```

`pg_receivewal` still points to the old primary!

Timelines and `.history` files

A correct restore — PITR or not — always involves a timeline switch.

- When archive recovery completes or a standby is promoted -> new timeline
  - encoded in WAL segment file names
  - identifies the WAL record series generated after that recovery
  - recorded in `.history` files

Faster archiving?

- Compression types
- Compression level
- Async archiving

Compression types

- File compression types supported (`compress-type`):
  - `bz2` - bzip2
  - `gz` - gzip (default)
  - `lz4`
  - `zst` - Zstandard

Compression level

- When `compress-level` is not specified:
  - defaults to values based on `compress-type`
    - `bz2`: 9
    - `gz`: 6
    - `lz4`: 1
    - `zst`: 3

Async archiving

- With `archive-async=y`:
  - temporary data (acknowledgments) stored in `spool-path`
  - early archiving with `process-max` workers

Example (1) - initial state

```
SELECT application_name, sent_lsn, write_lsn, flush_lsn FROM pg_stat_replication;
 application_name |  sent_lsn  | write_lsn  | flush_lsn
------------------+------------+------------+------------
 walreceiver      | 0/34E74D38 | 0/34E74D38 | 0/34E74D38
 pg_receivewal    | 0/34E74D38 | 0/34E74D38 | 0/34000000
(2 rows)
```

```
SELECT last_archived_wal FROM pg_stat_archiver;
    last_archived_wal
--------------------------
 000000010000000000000033
(1 row)
```

Example (2) - archiving too slow

```
/usr/pgsql-18/bin/pgbench -n -P 1 -T 60 -j 2 -c 50
```

```
SELECT application_name, sent_lsn, write_lsn, flush_lsn FROM pg_stat_replication;
 application_name |  sent_lsn   | write_lsn   | flush_lsn
------------------+-------------+-------------+-------------
 walreceiver      | 0/67F3CE30 | 0/67F33830 | 0/67F0D558
 pg_receivewal    | 0/67F3CE30 | 0/62780B38 | 0/62000000
(2 rows)
```

```
SELECT last_archived_wal FROM pg_stat_archiver;
    last_archived_wal
--------------------------
 000000010000000000000048
(1 row)
```

Example (3) - final state

```
SELECT application_name, sent_lsn, write_lsn, flush_lsn FROM pg_stat_replication;
 application_name |  sent_lsn   | write_lsn   | flush_lsn
------------------+-------------+-------------+-------------
 walreceiver      | 0/A82E1770 | 0/A82E1770 | 0/A82E1770
 pg_receivewal    | 0/A82E1770 | 0/A82E1770 | 0/A8000000
(2 rows)
```

```
SELECT last_archived_wal FROM pg_stat_archiver;
    last_archived_wal
--------------------------
 000000010000000000000A7
(1 row)
```

Example (4) - improvements

```
process-max=2
archive-async=y
compress-type=zst
```

```
/usr/pgsql-18/bin/pgbench -n -P 1 -T 60 -j 2 -c 50
```

```
SELECT application_name, sent_lsn, write_lsn, flush_lsn FROM pg_stat_replication;
 application_name |  sent_lsn   | write_lsn   | flush_lsn
------------------+-------------+-------------+------------
 walreceiver      | 0/C9B5F610 | 0/C9B5F610 | 0/C9B5AE00
 pg_receivewal    | 0/C9B5F610 | 0/BE8D22E0 | 0/BE000000
(2 rows)
```

```
SELECT last_archived_wal FROM pg_stat_archiver;
    last_archived_wal
--------------------------
 000000010000000000000C7
(1 row)
```

Understanding logs

- `archive-push` console output goes into the PostgreSQL logs

```
P00   INFO: archive-push command begin 2.58.0: [pg_wal/0000000100000000000000C6] ...
P00   INFO: pushed WAL file '0000000100000000000000C6' to the archive asynchronously
P00   INFO: archive-push command end: completed successfully (639ms)
P00   INFO: archive-push command begin 2.58.0: [pg_wal/0000000100000000000000C7] ...
P00   INFO: pushed WAL file '0000000100000000000000C7' to the archive asynchronously
P00   INFO: archive-push command end: completed successfully (20ms)
```

- `demo-archive-push-async.log`

```
P00   INFO: archive-push:async command begin 2.58.0: [/var/lib/pgsql/18/data/pg_wal] ...
P00   INFO: push 2 WAL file(s) to archive: 0000000100000000000000C6...0000000100000000000000C7
P02 DETAIL: pushed WAL file '0000000100000000000000C7' to the archive
P01 DETAIL: pushed WAL file '0000000100000000000000C6' to the archive
P00   INFO: archive-push:async command end: completed successfully (554ms)
```

Async high-level implementation details

- Create the spool `out` path if it does not already exist
- Read `.ready` files from `archive_status`
- Remove `.ok` files that are not in the ready list
- Return `.ready` files that are not in the `.ok` list
- Create the parallel executor and run jobs
- On success:
  - log success
  - write the status file

What can go wrong with archiving?

> *Things can get worse… and they will!*

WAL segments piling up…

An error prevents PostgreSQL from removing/recycling WAL files!

```
$ ls data/pg_wal/archive_status |grep .ready
...
000000010000000020000001B.ready
000000010000000020000001C.ready
000000010000000020000001D.ready
000000010000000020000001E.ready
```

Archiving queue

- `archive-push-queue-max`
  - maximum size of the PostgreSQL archive queue
  - prevents WAL storage from filling up until PostgreSQL stops…
  - …but can create **missing archives**!
- **Monitor** archiving to ensure it keeps working

A very realistic failure chain

- NFS/S3/… hiccup -> archive queue fills
- `archive-push-queue-max` reached
- PostgreSQL continues -> WAL is recycled
- Backup exists…
- …but PITR fails due to missing WAL
  - a new backup is needed!

Log error example without `archive-async`

```
P00   INFO: archive-push command begin 2.58.0: [pg_wal/000000010000000010000000E0] ...
P00   WARN: dropped WAL file '000000010000000010000000E0' because archive queue exceeded 128MB
P00   INFO: archive-push command end: completed successfully (8ms)
P00   INFO: archive-push command begin 2.58.0: [pg_wal/000000010000000010000000E1] ...
P00  ERROR: [103]: unable to find a valid repository:
...
P00   INFO: archive-push command end: aborted with exception [103]
LOG:  archive command failed with exit code 103
```

Side effect

```
$ ls data/pg_wal/archive_status |grep .ready

000000010000000100000E1.ready
000000010000000100000E2.ready
000000010000000100000E3.ready
000000010000000100000E4.ready
000000010000000100000E5.ready
000000010000000100000E6.ready
000000010000000100000E7.ready
000000010000000100000E8.ready
```

- 8 x 16MB <= 128MB
- Only the oldest WAL file gets purged

Log error example with `archive-async`

```
P00   WARN: dropped WAL file '000000010000000200000001' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000002' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000003' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000004' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000005' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000006' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000007' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000008' because archive queue exceeded 128MB
P00   WARN: dropped WAL file '000000010000000200000009' because archive queue exceeded 128MB

P00   INFO: archive-push command begin 2.58.0: [pg_wal/00000001000000020000000A] ...
P00  ERROR: [103]: unable to find a valid repository:
...
P00   INFO: archive-push command end: aborted with exception [103]
LOG:  archive command failed with exit code 103
```

Side effect

```
$ ls data/pg_wal/archive_status |grep .ready

000000010000000200000000A.ready
000000010000000200000000B.ready
```

- The entire 128MB waiting queue gets purged

Restore: pgBackRest + PostgreSQL

pgBackRest handles the restore;
PostgreSQL handles the recovery!

Let’s talk about `restore` command and recovery targets…

Restore type?

- `--type` (and `--target` to define the target)
  - `default`: end of the WAL archive stream
  - `immediate`: backup consistency point
  - `lsn`: LSN (*Log Sequence Number*), `recovery_target_lsn`
  - `name`: restore point, `recovery_target_name`
  - `xid`: transaction ID, `recovery_target_xid`
  - `time`: timestamp, `recovery_target_time`
  - …

Backup set

- `--set`
  - default: `latest`
  - auto-selected for `time` and `lsn` targets

Timeline

- `--target-timeline`
  - `recovery_target_timeline`
  - default: `latest`
  - useful value: `current`

What happens at the target?

- `--target-action` (`recovery_target_action`)
  - `pause`: pause when the recovery target is reached
  - `promote`: promote the instance and switch to a new timeline
  - `shutdown`: shut down the server

Delta mode: restore into an existing PGDATA

- `--delta` (CLI) or `delta=y` (config)
- Uses checksums to decide what to copy
  - **Restore**: allows restoring into a non-empty data directory
  - **Backup**: uses checksums instead of timestamps

```
Restore or backup using checksums.
During a restore, by default the PostgreSQL data and tablespace directories
are expected to be present but empty. This option performs a delta restore
using checksums.

During a backup, this option will use checksums instead of timestamps to
determine if files will be copied.
```

Delta restore is not incremental backup

- **Delta restore**
  - affects how files are copied during restore
  - allows restoring into a non-empty data directory
  - does **not** change backup format or WAL requirements
- **Incremental backup**
  - affects how backups are created
  - stores only changes since a previous backup
  - requires a dependency chain
  - **not related** to `--delta`
- `--delta` **backup**: uses checksums, not timestamps

Selective restore

- `--db-include`
  - databases not explicitly included are restored as sparse, zeroed files
  - system databases are always restored unless explicitly excluded
    - (`template0`, `template1`, `postgres`)
- `--db-exclude`
  - excluded databases are restored as sparse, zeroed files
  - with `--db-include`, this only applies to system databases

Example (1)

Name	Size
db1	1015 MB
db2	3047 MB

```
db1=# SELECT pg_create_restore_point('dont_fear_demo_effect');
 pg_create_restore_point
-------------------------
 4/91FF5B58
(1 row)

db1=# DELETE FROM pgbench_accounts;
DELETE 6500000
```

Example (2) - restore command

```
$ pgbackrest restore --stanza=demo --type=name --target='dont_fear_demo_effect' --db-include=db1
P00   INFO: restore command begin 2.58.0: ...
P00   INFO: repo1: restore backup set 20260130-105007F, recovery will start at ...
P00   INFO: write updated /var/lib/pgsql/18/data/postgresql.auto.conf
P00   INFO: restore global/pg_control
P00   INFO: restore size = 4GB, file total = 1603
P00   INFO: restore command end: completed successfully

$ cat /var/lib/pgsql/18/data/postgresql.auto.conf
# Recovery settings generated by pgBackRest restore on ...
restore_command = 'pgbackrest --stanza=demo archive-get %f "%p"'
recovery_target_name = 'dont_fear_demo_effect'

$ du -hs /var/lib/pgsql/18/data
1.1G  /var/lib/pgsql/18/data
```

Example (3) - the result

```
db1=# SELECT COUNT(*) FROM pgbench_accounts;
  count
---------
 6500000
(1 row)

db1=# \c db2
connection to server on socket "/run/postgresql/.s.PGSQL.5432" failed:
  FATAL:  relation mapping file "base/16440/pg_filenode.map" contains invalid data

db1=# DROP DATABASE db2; -- corrupted after selective restore
DROP DATABASE
```

Selective restore summary

- Saves disk space and restore time (not recovery time!)
- pgBackRest restore `--db-include` with `--type=immediate` usually works well
- Selective PITR can work; otherwise, if it does not, a regular restore is required
- Selective restore is an optimization, not a guarantee

Asynchronously fetch WAL segments

- With `archive-get` and `archive-async=y`:
  - prefetches up to `archive-get-queue-max` WAL segments to **speed up recovery**
  - uses `process-max` workers
  - stores them in `spool-path`

Conclusion

- Tune your archiving (async, compression, concurrency)
- Monitor backups and archives
- Regularly test restores

Schrödinger's Law of Backups

> *The condition/state of any backup is unknown until a restore is attempted.*

Questions?

contact@dataegret.com