Mastering PostgreSQL Recovery: Beyond Backup Basics

data egret

Your remote PostgreSQL DBA team

Stefan FERCOT

stefan.fercot@dataegret.com

SECURING YOUR DATABASE AVAILABILITY, SO THAT YOUR TEAM CAN FOCUS ON NEW FEATURE DEVELOPMENT.

- Migrations
- DB audit
- Performance optimisation
- Backup & restore
- Architectural review
- Advising Data Science teams
- Developer training

on premise & cloud

EXPERTISE

Senior DBA with **10+ years** of PostgreSQL administration **experience**

DEVELOPMENT

Involved in **new feature and extension development**

TAILORED APPROACH

Felxible approach and **dedicated team** focused on success of your project

COMMUNITY

Contributing Sponsor. Deeply involved in the PostgreSQL community

Data Egret GmbH | 66583 Spiesen-Elversberg | Germany

contact@DataEgret.com

Stefan Fercot

- Senior PostgreSQL Expert **@Data Egret**
- pgBackRest fan & contributor
- aka. pgstef
- https://pgstef.github.io

Need a Disaster and Recovery Plan? ;-)
*Contact **Data Egret** to talk to me about backups and high-availability!*

Mastering PostgreSQL Recovery

- continuous archiving and PITR
  - pretty well covered in PostgreSQL docs
  - but successful recovery examples are not

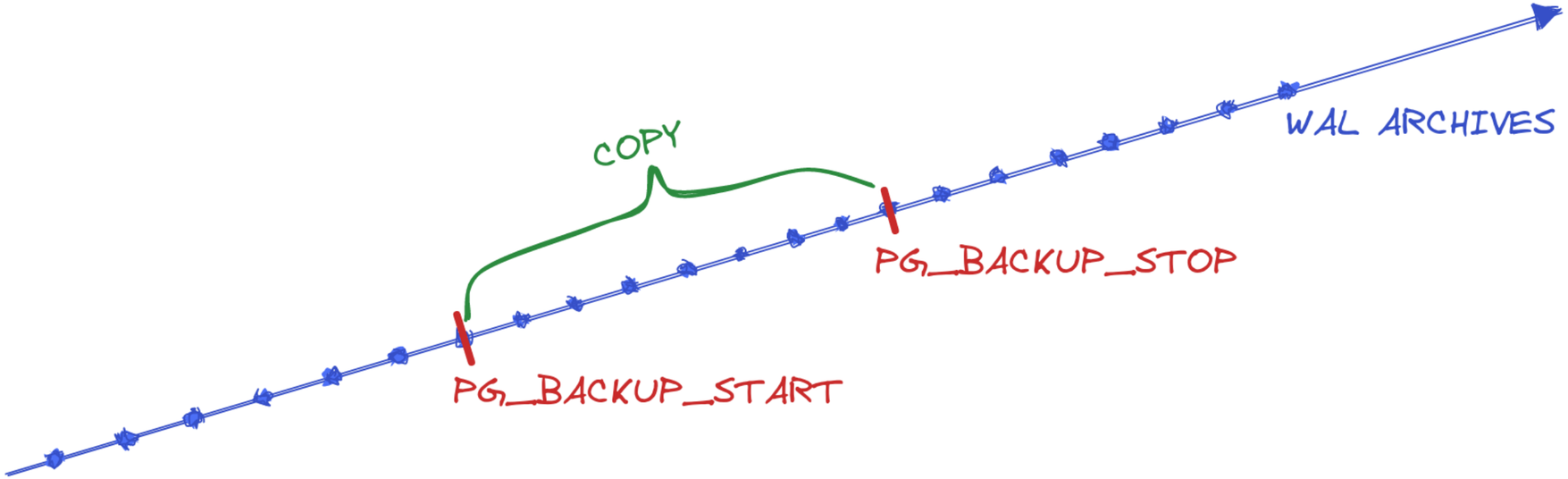
Agenda

- Backup basics quick recap
- Restore procedure
- Recovery settings
- Quick demo setup and examples

Restore vs Recovery

- *restore* process handled by community tools…
- *recovery* done by PostgreSQL itself!

Backup basics



- file-system-level backup (data files)
- continuous WAL archiving (data modifications)

Backup consistency

- to recover successfully
  - continuous sequence of archived WAL files needed…
  - from backup start to backup stop location

WAL archives

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

File-system-level backup

- `pg_basebackup`
- manual steps
  - `pg_backup_start()`
  - manual file-system-level copy
  - `pg_backup_stop()`

Restore procedure

- simple but… must be followed carefully!

Restore steps (1/4)

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

Restore steps (2/4)

- 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

Restore steps (3/4)

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

Restore steps (4/4)

- start the server
- watch the restore process
  - until consistent recovery state (or target) reached
- inspect your data

Recovery settings

- by default, recover to the end of the WAL stream
- how to specify an earlier stopping point?

Consistent state

- `recovery_target = 'immediate'`
  - recovery stops when consistent state is reached
  - (i.e. the point where taking the backup ended)

Restore point

- `recovery_target_name`
  - create a named restore point with `pg_create_restore_point()`

Timestamp

- `recovery_target_time`
  - timestamp with time zone format
  - recommended to use a numeric offset from UTC
    - example: `2024-05-07 09:00:00+02`
  - or write a full time zone name, e.g., *Europe/Brussels* not *CEST*

Transaction ID

- `recovery_target_xid`
  - transactions committed before (and optionally including) specified xid will be recovered

WAL location

- `recovery_target_lsn`
  - LSN of the write-ahead log location
  - parameter parsed as system data type pg_lsn

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)
```

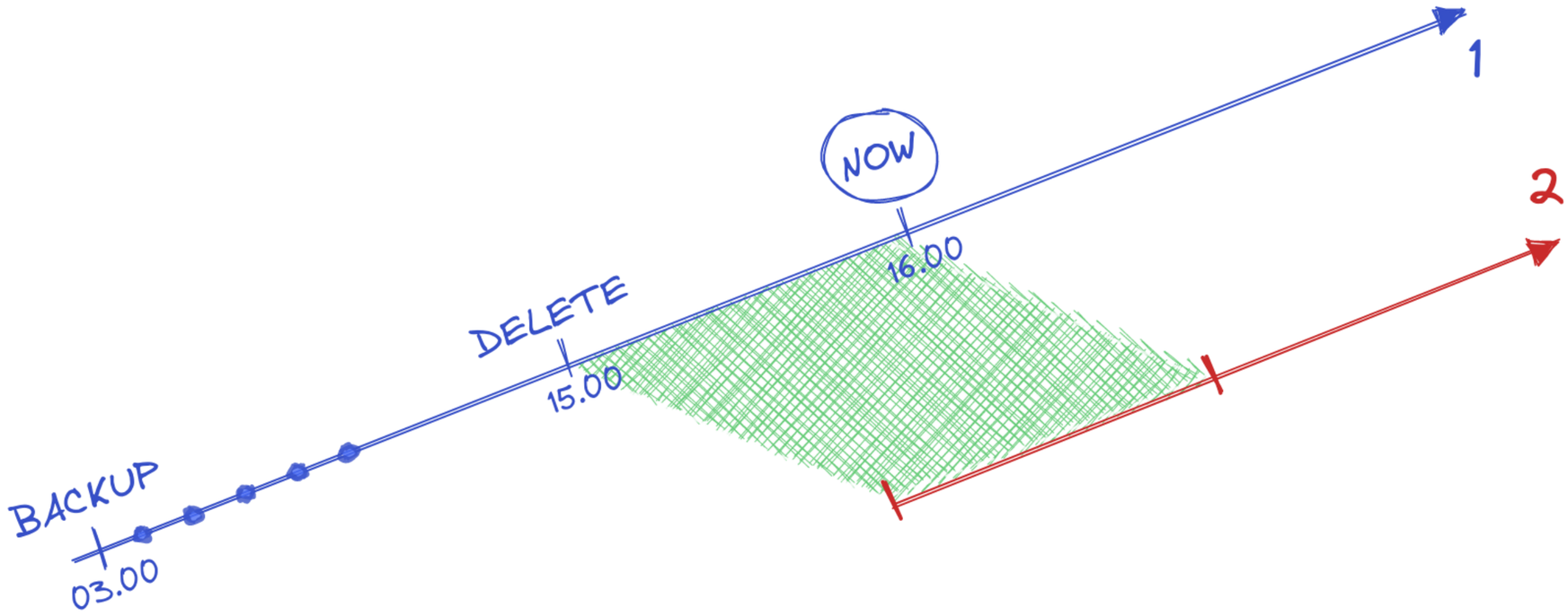
WAL filename

- 000000010000000200000003
  - 00000001 : timeline
  - 00000002 : wal
  - 00000003 : segment
- hexadecimal
  - 000000010000000000000001
  - 00000001000000000000FF
  - 000000010000000100000000
  - …

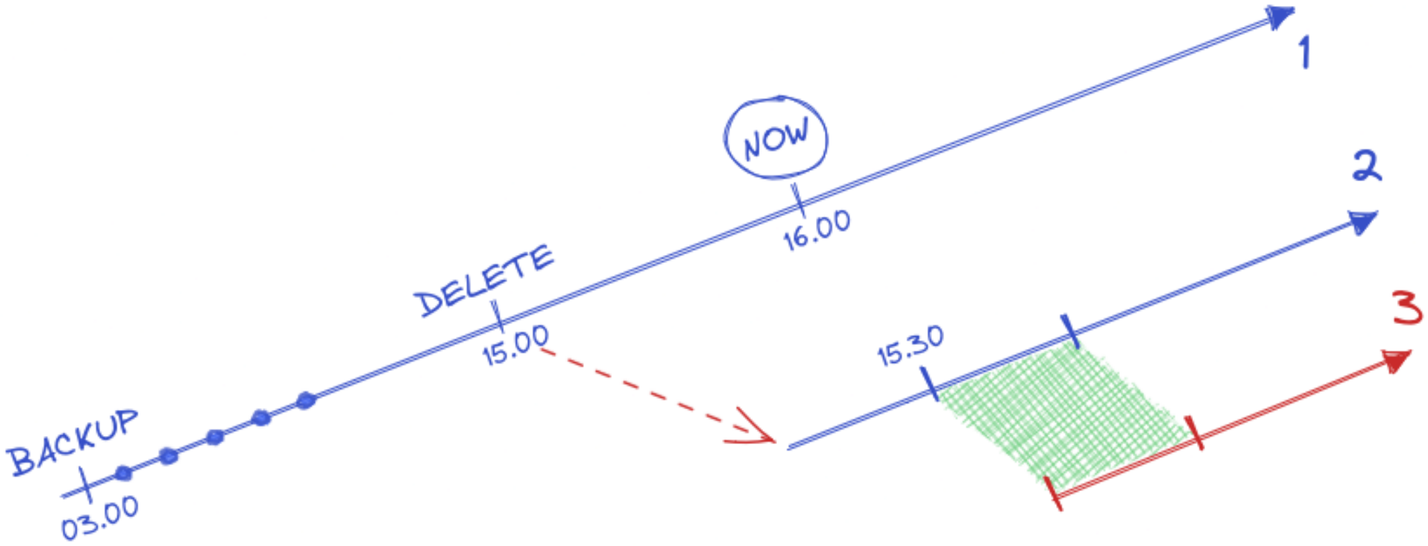
Timeline to follow

- 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)

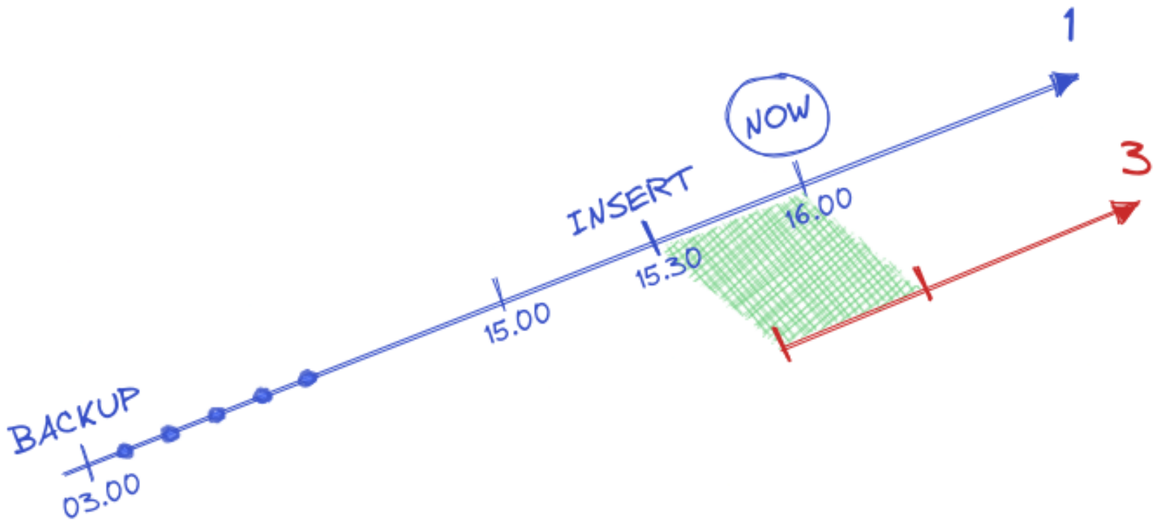
Timelines explanation



Timelines explanation (2)



Timelines explanation (3)



Stop after or before the target

- `recovery_target_inclusive`
  - recovery stops just after recovery target (`on`)…
  - …or just before (`off`)
  - works with LSN, time or xid
  - default is `on`

Action once recovery target is reached

- `recovery_target_action`
  - pause (`pg_wal_replay_resume()`)
  - promote
  - shutdown

Summary

- recovery targets:
  - `recovery_target = 'immediate'`
  - `recovery_target_name`, `recovery_target_time`
  - `recovery_target_xid`, `recovery_target_lsn`
- timeline to follow:
  - `recovery_target_timeline`
- stop after or before the target?
  - `recovery_target_inclusive`
- action once recovery target is reached?
  - `recovery_target_action`

Quick demo setup

```
$ createdb pgbench
$ /usr/pgsql-16/bin/pgbench -i -s 600 pgbench
$ /usr/pgsql-16/bin/pgbench -c 4 -j 2 -T 300 pgbench
```

```
archive_mode = on
archive_command = 'test ! -f /backup_space/archives/%f && cp %p /backup_space/archives/%f'
```

Take a backup

```
$ pg_basebackup -D "/backup_space/backups/$(date +'%F_%T')" \
 --format=plain --wal-method=none --checkpoint=fast --progress
NOTICE:  all required WAL segments have been archived
9233844/9233844 kB (100%), 1/1 tablespace
```

Oops time…

```
SELECT pg_create_restore_point('RP1');
BEGIN;
    SELECT pg_current_wal_lsn(), current_timestamp;
    DELETE FROM pgbench_tellers;
COMMIT;
BEGIN;
    CREATE TABLE important_table (field text);
    INSERT INTO important_table VALUES ('important data');
COMMIT;
SELECT pg_switch_wal();
```

Useful information from the output

```
pgbench=*# SELECT pg_current_wal_lsn(), current_timestamp;
 pg_current_wal_lsn |       current_timestamp
--------------------+-------------------------------
 2/B0786608         | 2024-05-07 08:50:10.316588+00
(1 row)
```

pg_waldump

```
$ /usr/pgsql-16/bin/pg_waldump /backup_space/archives/000000010000000200000B0
rmgr: XLOG        len (rec/tot):     98/    98, tx:          0,
    lsn: 2/B0786568, prev 2/B0786530, desc: RESTORE_POINT RP1
...
rmgr: Heap        len (rec/tot):     54/    54, tx:     259070, lsn: 2/B0786608,
    prev 2/B07865D0, desc: DELETE xmax: 259070, off: 1, infobits: [KEYS_UPDATED],
    flags: 0x01, blkref #0: rel 1663/16384/16400 blk 0
...
rmgr: Transaction len (rec/tot):     34/    34, tx:     259070, lsn: 2/B07D8A60,
    prev 2/B07D8A28, desc: COMMIT 2024-05-07 08:50:10.321494 UTC
```

How to identify our relation?

```
pgbench=# SELECT dattablespace AS tablespace, oid AS database,
          pg_relation_filenode('pgbench_tellers') AS table
          FROM pg_database
          WHERE datname=current_database();

 tablespace | database | table
------------+----------+-------
       1663 |    16384 | 16400
(1 row)
```

Findings…

- name: `RP1`
- lsn: `prev 2/B07865D0` (lsn before the first DELETE)
- xid: `tx: 259070`
- time: `2024-05-07 08:50:10.316588+00`
  - or `COMMIT 2024-05-07 08:50:10.321494 UTC`

Don’t forget to practice!

Schrödinger’s Law of Backups

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

Recovery example (1)

```
$ touch /var/lib/pgsql/16/data/recovery.signal
```

```
# postgresql(.auto).conf
archive_mode = off
restore_command = 'cp /backup_space/archives/%f %p'
recovery_target = 'immediate'
recovery_target_action = 'promote'
```

```
$ cat /var/lib/pgsql/16/data/backup_label
START WAL LOCATION: 2/23C700 (file 000000010000000200000000)
...
```

Look at the logs

```
LOG:  starting point-in-time recovery to earliest consistent point
LOG:  starting backup recovery with redo LSN 2/23C700,...
LOG:  restored log file "000000010000000200000000" from archive
LOG:  redo starts at 2/23C700
LOG:  restored log file "..." from archive
...
LOG:  consistent recovery state reached at 2/1EA6C7C8
LOG:  database system is ready to accept read-only connections
LOG:  recovery stopping after reaching consistency
...
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

Recovery example (2)

- what if we know exactly our recovery target?

```
# postgresql(.auto).conf
restore_command = 'cp /backup_space/archives/%f %p'
recovery_target_xid = '259070'
recovery_target_inclusive = off
recovery_target_action = 'promote'
```

Look at the logs

```
LOG:  starting point-in-time recovery to XID 259070
LOG:  starting backup recovery with redo LSN 2/23C700,...
LOG:  restored log file "..." from archive
...
LOG:  consistent recovery state reached at 2/1EA6C7C8
LOG:  database system is ready to accept read-only connections
...
LOG:  recovery stopping before commit of transaction 259070,
      time 2024-05-07 08:50:10.321494+00
...
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

Look at the backup space

- `archive_mode` was enabled this time!

```
$ cat /backup_space/archives/00000002.history
1   2/B07D8A60  before transaction 259070
```

Recovery example (3)

- use the named restore point

```
# postgresql(.auto).conf
restore_command = 'cp /backup_space/archives/%f %p'
recovery_target_name = 'RP1'
```

Look at the logs

```
LOG:  recovery stopping at restore point "RP1", time 2024-05-07 08:50:01.56315+00
LOG:  pausing at the end of recovery
HINT:  Execute pg_wal_replay_resume() to promote.
```

```
psql -c "SELECT pg_wal_replay_resume();"
```

```
LOG:  selected new timeline ID: 3
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

Recovery example (4)

What about our `important data`?

```
pgbench=# SELECT * FROM important_table;
ERROR:  relation "important_table" does not exist
LINE 1: SELECT * FROM important_table;
                      ^
```

What timeline to follow?

By default, PostgreSQL will follow the `latest` timeline!

```
$ cat /backup_space/archives/00000003.history
1   2/B07D8A60  before transaction 259070
2   2/B07865D0  at restore point "RP1"
```

```
# postgresql(.auto).conf
restore_command = 'cp /backup_space/archives/%f %p'
recovery_target_timeline = 'current'
```

Look at the logs

```
LOG:  starting archive recovery
LOG:  starting backup recovery with redo LSN 2/23C700,...
...
LOG:  restored log file "000000010000000200000B1" from archive
LOG:  redo done at 2/B1000148 ...
LOG:  restored log file "00000002.history" from archive
LOG:  restored log file "00000003.history" from archive
LOG:  selected new timeline ID: 4
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
```

Ta-da!

```
pgbench=# SELECT * FROM important_table;
     field
----------------
 important data
(1 row)
```

Conclusion

- tools make life easier…
- restore points are easy to use
- as usual, practice is the key to success
- the answer is in the PostgreSQL logs!

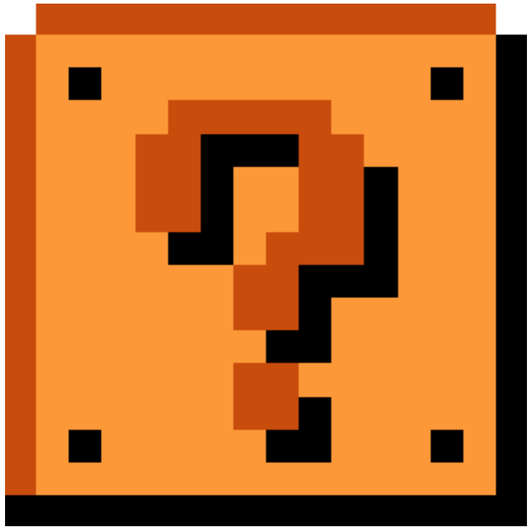
PGConf Belgium

PgBE PostgreSQL Users Group Belgium meetup group

Aachen meetup - May 15 (next Wednesday!)

PG Day France - Lille, June 11-12

Questions?



Thank you for your attention!