WAL for DBAs – (Almost) Everything you want to know

Devrim Gündüz

Principal Systems Engineer @ EnterpriseDB

devrim.gunduz@EnterpriseDB.com

Twitter : @DevrimGunduz

About me

- Who is this guy?

    – Using Red Hat (and then Fedora) since 1996.

    – Using PostgreSQL since 1998.

    – Responsible for PostgreSQL YUM repository.

    – Working at EnterpriseDB since 2011.

    – Living in London, UK.

    – The Guy With The PostgreSQL Tattoo!

Social media

Please tweet:

#PostgresVision

Please follow:

@PostgresVision

Social media

(Did you tweet? Thanks!)

Agenda (in random order)

- What is WAL?

- What does it include?

- How to read it?

- What about wal_level ?

- Replication and WAL

- Backup and WAL

- PITR and WAL

- Other topics

Before we actually start:

Please do not delete WAL files manually.

Please.

Before we actually start:

Please do not delete WAL files manually.

Please.
Please.

Before we actually start:

Please do not delete WAL files manually.

Please.
Please.

PLEASE.

What is WAL?

- Write Ahead Log:
  - Logging of transactions

  - a.k.a. xlog (transaction log),

  - 16 MB in most of the installations (can be configured, --with-wal-segsize)

      - v11: initdb will have a –wal-segsize parameter
    - 8 kB page size (can be configured, --with-wal-blocksize)
  - pg_xlog (<= 9.6)→ pg_wal (10+)

    - Because people deleted files under "log" directory.

What is WAL?

- Designed to prevent data loss in most of the situations
  - OS crash, hardware failure, PostgreSQL crash.

- Write transactions are written to WAL
  - Before transaction result is sent to the client
  - Data files are not changed on each transaction
  - Performance benefit

- Should be kept in a separate drive.
  - Initdb, or symlink

What is WAL?

- Built-in feature

- Life before WAL (not before B.C., though):
  - All changes go to durable storage (eventually), but:
    - Data page is loaded to shared_buffers
    - Changes are made there
    - Dirty buffers!
  - But not timely!
  - Crash → Data loss!

EDB
ENTERPRISEDB

What is WAL?

- Life after WAL:
  - Almost all "modifications" are "logged" to WAL files (WAL record)

    - Even if the transaction is aborted (ROLLBACK)

  - wal_buffers → WAL segments (files)

  - Ability to recover data after a crash

  - Checkpoint!

Where is it used?

- Transaction logging!

- Replication

- PITR

- REDO
  - Sequentially availability is a must.

  - REDO vs UNDO

  - No REDO for temp tables and unlogged tables.

Shared Buffers, Bgwriter and checkpointer

- shared_buffers in PostgreSQL
  - Dirty buffers

  - This is where transactions are performed

  - Side effect: Causes inconsistency(?) on durable storage, due to dirty buffers.

- Bgwriter: Background writer
  - LRU

- Checkpointer
  - Pushing all dirty buffers to durable storage

  - Triggered automatically or manually

- Backends may also write data to heap

WAL file naming

- 24 chars, hex.
  - 1st 8 chars: timelineID
    - 00000001 is the timelineID created by initdb
  - 2nd 8 chars: Block ID
  - 3rd 8 chars: WAL segment ID

- 000000010000000000000001 → 000000010000000000000002

- … 0000000100000000000000FF → 000000010000000100000000

- …and 00000001000000010000000FF → 000000010000000200000000

WAL file naming

- Use PostgreSQL's internal tools to manage them
  - pg_archivecleanup

  - pg_resetwal

  - pg_waldump

  - ....

pg_waldump

- We are all human.

- Use pg_waldump, if you want to see contents of WAL files

- rmgr --help to get list of all resource names, -f for follow, -n for limit. -z for stats.

- pg_waldump -n 20 -f 00000001000000700000033

- rmgr: Heap   len (rec/tot):   3/   59, tx: 389744, lsn: 7/33B66228, prev 7/33B661F0, desc: INSERT+INIT off 1, blkref #0: rel 1663/13326/190344 blk 0

- rmgr: Heap   len (rec/tot):   3/   59, tx: 389744, lsn: 7/33B66268, prev 7/33B66228, desc: INSERT off 2, blkref #0: rel 1663/13326/190344 blk 0

- rmgr: Transaction len (rec/tot):   8/   34, tx: 389744, lsn: 7/33B662A8, prev 7/33B66268, desc: COMMIT 2017-02-03 03:03:49.482223 +03

- rmgr: Heap   len (rec/tot):   14/   69, tx: 389745, lsn: 7/33B662D0, prev 7/33B662A8, desc: HOT_UPDATE off 1 xmax 389745 ; new off 3 xmax 0, blkref #0: rel 1663/13326/190344 blk 0

- rmgr: Transaction len (rec/tot):   8/   34, tx: 389745, lsn: 7/33B66318, prev 7/33B662D0, desc: COMMIT 2017-02-03 03:03:54.091645 +03

- rmgr: WAL   len (rec/tot):   80/ 106, tx: 0, lsn: 7/33B66340, prev 7/33B66318, desc: CHECKPOINT_ONLINE redo 7/33B66340; tli 1; prev tli 1; fpw true; xid 0/389746; oid 198532; multi 1; offset 0; oldest xid 1866 in DB 129795; oldest multi 1 in DB 90123; oldest/newest commit timestamp xid: 388437/389745; oldest running xid 0; online

- rmgr: WAL   len (rec/tot):   0/   24, tx: 0, lsn: 7/33B663B0, prev 7/33B66340, desc: SWITCH

WAL: LSN

- Log Sequence Number
  - Position of the record in WAL file.

  - Provides uniqueness for each WAL record.

- Per docs: "Pointer to a location in WAL file"

- LSN: Block ID + Segment ID:

- During recovery, LSN on the page and LSN in the WAL file are compared.
  - The larger one wins.

WAL: Finding current WAL file

- Probably not the last one in ls list!

- postgres=# SELECT * from pg_current_wal_location();

  pg_current_wal_location

  40E6/2C85AC10

- postgres=# SELECT pg_walfile_name('40E6/2C85AC10');
  pg_walfile_name

  00000003000040E60000002C

  So:

- postgres=# SELECT pg_walfile_name(pg_current_wal_location());

  pg_walfile_name

  00000003000040E60000002C

Checkpoint, and pg_control

- As soon as the checkpoint starts, REDO point is stored in shared buffers.

- A WAL record is created referencing checkpoint start, and it is first written to WAL buffers, and then eventually to pg_control.

  - pg_control is under $PGDATA/global

- Unlike bgwriter, checkpointer writes **all of the** data in the shared_buffers to durable storage.

- PostgreSQL knows the latest REDO point, by looking at pg_control file.

Checkpoint, and pg_control

- **pg_controldata**:

    - Latest checkpoint location: 40E7/E43B16B8

    - Prior checkpoint location: 40E7/D8689090

        They are LSN.

- When checkpoint is completed, **pg_control** is updated with the position of checkpoint.

- After checkpoint, old WAL files are either recycled, or removed.

- An "estimation" is done while recycling (based on previous checkpoint cycles)

- 9.5+: In minimum, **min_wal_size** WAL files are always recycled for future usage

pg_control and REDO

- postmaster reads pg_control on startup.

    /usr/pgsql-10/bin/pg_controldata -D /var/lib/pgsql/10/data | grep state

    - "Database cluster state":

        - starting up

        - shut down

        - shut down in recovery

        - shutting down

        - in crash recovery

        - in archive recovery

        - in production

- If pg_control says "in production", but db server is not running, then this instance is eligible for a recovery!

pg_control and REDO

- pg_control is the critical piece

    - Should not be corrupted

    - Per docs: "...theoretically a weak spot"

- REDO: All WAL files must be sequentially available for complete recovery.

- UNDO: Not available in Postgres.

Moving to the new WAL

- A WAL segment may be full

- PostgreSQL archiver will switch to the new WAL, if PostgreSQL reaches archive_timeout value.

- DBA issues **pg_switch_wal()** function.

WAL: Archiving

- Replication, backup, PITR
- archive_mode
- archive_command
- archive_timeout

WAL: Point-In-Time Recovery (PITR)

- A base backup (pg_basebackup!) and the WAL files are needed.

- WAL files must be sequentially complete – otherwise PITR won't be finished.

- "Roll-forward recovery"

WAL: Point-In-Time Recovery (PITR)

- PITR: Replaying WAL files on base backups, until **recovery target.**
  - **recovery_target_{time, xid, name, lsn}**
  - If not specified, all archived WAL files are replayed.

- **recovery.conf** and **backup_label:** Enters recovery mode.
  - restore_command, recovery_target_XXX,recovery_target_inclusive

- backup_label: Also includes checkpoint location (starting point of recovery)

- **Almost** like regular recovery process (WAL replay)

- Up to recovery_target_XXX is replayed.

WAL: Point-In-Time Recovery (PITR)

- After recovery process, timelineID is increased by 1 (also physical WAL file name is also increased by 1)

- A .history file is created.

- $ cat 00000003.history
    - 1   403F/58000098 no recovery target specified

    - 2   4048/43000098 before 2017-01-28 11:13:21.124512+03

    "WAL files were replayed until the given time above, and their replay location is 4048/43000098.

-

-

    -

Full page writes

- A WAL record cannot be replayed on a page which is corrupted during bgwriter and/or checkpointer, because of hardware failure, OS crash, kernel failure, etc.

- Full page writes IYF.

- Enabled by default.

    - Please turn it off, if you want to throw a lot of money to PostgreSQL support companies. Otherwise, don't do so ;)

Full page writes

- PostgreSQL writes header data + the entire page as WAL record, when a page changes after **every** checkpoint.

  - Increasing checkpoint_timeout helps.

  - Full-page image, backup block.

- PostgreSQL can even recover itself from write failures (not hw failures, though)

WAL parameters

- wal_level: Minimal, replica or logical
  - Must be > minimal for archiver to be able to run

- fsync : Always on, please.

- synchronous_commit: May lose some of the latest transactions
  - Server returns success to the client

  - Server waits **a bit** to flush the data to durable storage.

  - Less risky than fsync

- wal_sync_method : fdatasync is usually better. Use pg_test_fsync for testing.

WAL parameters

- wal_log_hints: When this value is set to on , the server writes the entire content of each disk page to WAL after a checkpoint and during the first modification of that page, even for non-critical modifications of so-called hint bits.

- wal_compression: off by default. Less WAL files, more CPU overhead.

- wal_buffers: -1: Automatic tuning of wal buffers: 1/32 of shared_buffers (not less than 64kB or no more than 16 MB (1 WAL file)

- wal_writer_delay : Rounds between WAL writer flushes WAL.

- wal_writer_flush_after: New in 9.6

Questions, comments?

WAL for DBAs – Everything you want to know

Devrim Gündüz

Principal Systems Engineer @ EnterpriseDB

devrim.gunduz@EnterpriseDB.com

Twitter : @DevrimGunduz