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.
    - Cheers for 21st year!
  - Responsible for PostgreSQL YUM (RHEL, CentOS, Fedora) and Zypp (SLES) repositories.
  - Fedora and EPEL packager.
  - Working at EnterpriseDB since 2011.
  - Living in London, UK.
  - The Guy With The PostgreSQL Tattoo! (Please discard imitations)

PGConf.EU 2019

- The largest PostgreSQL conference in Europe.
- Milan, Italy.
- 15 October: Training day.
- 16-18 October: Conference
- Registration is open: https://2019.pgconf.eu/registration
- CfP is also open: https://2019.pgconf.eu/callforpapers/

Social Media

Please tweet:

#PostgreSQL

#PostgresLondon

Please follow:

@Postgres_London

@PostgreSQL

@PGConfEU

Alternative Hashtag and account

#BlameMagnus

@BlameMagnus

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
- Full page writes!
- 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 in ancient times (transaction log),
  - 16 MB in most of the installations (can be configured, --with-wal-segsize)
    - v11+: initdb has a --wal-segsize parameter
      - Initdb --wal-segsize=64 ← in MB
  - 8 kB page size (can be configured, --with-wal-blocksize during configure)

What is WAL?

- Write Ahead Log:
  - Logging of transactions
  - a.k.a. xlog in ancient times (transaction log),
  - 16 MB in most of the installations (can be configured, --with-wal-segsize)
    - v11+: initdb has a --wal-segsize parameter
      - Initdb --wal-segsize=64 ← in MB
  - 8 kB page size (can be configured, --with-wal-blocksize during configure)
  - 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.

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

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!

What is WAL?

- Life after WAL:
  - Almost all of the “modifications” are “logged” to WAL files (WAL record)
    - Even if the transaction is aborted (ROLLBACK)

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)

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!

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: LSN

- Log Sequence Number
  - Position of the record in WAL file.
  - Provides uniqueness for each WAL record.
- 64-bit integer (historically 2x32-bit) (We'll need this info soon)
- Per docs: "Pointer to a location in WAL file"
- LSN: Block ID + Segment ID (See next slides)
- During recovery, LSN on the page and LSN in the WAL file are compared.
  - The larger one wins.

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: Segment ID
- 000000010000000000000001 → 000000010000000000000002
- … 000000010000000000000FF → 000000010000000100000000
- ...and 000000010000000100000FF → 000000010000000200000000

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: Segment ID
- 000000010000000000000001 → 000000010000000000000002
- … 000000010000000000000FF → 000000010000000100000000
- ...and 000000010000000100000FF → 000000010000000200000000

WAL file naming

- Default WAL file: 16 MB
  - Location within a WAL file can be expressed using 24 bits (because 2^24 = 16MB).
  - Take 64, split it into 32 + 32, subtract 24 from the second 32, you get 8, which is the number of bits from the low-order 32-bit integer that have to be stored in the WAL file name.
  - In hexadecimal, each character represents 4 bits, so to find the number of characters required to represent 8 bits, we take 8 / 4 = 2. And 2 is the number of 2 F's in the previous slide.

WAL: Finding current WAL file

- Probably not the last one in ls list!

- postgres=# SELECT * from pg_current_wal_lsn();

  pg_current_wal_location

  40E6/2C85AC10

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

  000000030000040E60000002C

  So:

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

  pg_walfile_name

  000000030000040E60000002C

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.
- More will come with full page writes.

Checkpoint, and pg_control

- **pg_controldata (before v11)**:

  Latest checkpoint location: 40E7/E43B16B8

  Prior checkpoint location: 40E7/D8689090

- **pg_controldata (v11+)**:

  Latest checkpoint location: 40E7/E43B16B8

  They are LSN.

Checkpoint, and pg_control

- **pg_controldata (before v11)**:

  Latest checkpoint location: 40E7/E43B16B8

  Prior checkpoint location: 40E7/D8689090

- **pg_controldata (v11+)**:

  Latest checkpoint location: 40E7/E43B16B8

  They are LSN.

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

Checkpoint, and pg_control

- **pg_controldata (before v11)**:

  Latest checkpoint location: 40E7/E43B16B8

  Prior checkpoint location: 40E7/D8689090

- **pg_controldata (v11+)**:

  Latest checkpoint location: 40E7/E43B16B8

  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.

Checkpoint, and pg_control

- **pg_controldata (before v11)**:

  Latest checkpoint location: 40E7/E43B16B8

  Prior checkpoint location: 40E7/D8689090

- **pg_controldata (v11+)**:

  Latest checkpoint location: 40E7/E43B16B8

  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)

Checkpoint, and pg_control

- **pg_controldata (before v11)**:

  Latest checkpoint location: 40E7/E43B16B8

  Prior checkpoint location: 40E7/D8689090

- **pg_controldata (v11+)**:

  Latest checkpoint location: 40E7/E43B16B8

  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-12/bin/pg_controldata -D /var/lib/pgsql/12/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”, but no issues reported yet!
  - There is a way to recover, but not implemented yet.
- REDO: All WAL files must be sequentially available for complete recovery.
- UNDO: Not available in Postgres yet.
  - See:
    - https://github.com/EnterpriseDB/zheap/
    - https://wiki.postgresql.org/wiki/Zheap

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 management

- 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 000000010000000700000033
- 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

pg_waldump

- rmgr: XLOG len (rec/tot): 30/ 30, tx: 0, lsn: 0/0CE268C8, prev 0/0CE26890, desc: NEXTOID 26914

  rmgr: Storage len (rec/tot): 42/ 42, tx: 0, lsn: 0/0CE268E8, prev 0/0CE268C8, desc: CREATE base/14012/18722

  rmgr: Heap len (rec/tot): 54/ 1338, tx: 1829, lsn: 0/0CE26918, prev 0/0CE268E8, desc: INSERT off 7, blkref #0: rel 1663/14012/1247 blk 15 **FPW**

  rmgr: Btree len (rec/tot): 53/ 6393, tx: 1829, lsn: 0/0CE26E58, prev 0/0CE26918, desc: INSERT_LEAF off 315, blkref #0: rel 1663/14012/2703 blk 2 **FPW**

  ---

  rmgr: **Standby** len (rec/tot): 42/ 42, tx: 1833, lsn: 0/0CE57300, prev 0/0CE572C8, desc: LOCK xid 1833 db 14012 rel 18731

  rmgr: Heap len (rec/tot): 54/ 54, tx: 1833, lsn: 0/0CE57330, prev 0/0CE57300, desc: DELETE off 14 KEYS_UPDATED , blkref #0: rel 1663/14012/1247 blk 15

  rmgr: Heap len (rec/tot): 54/ 54, tx: 1833, lsn: 0/0CE57368, prev 0/0CE57330, desc: DELETE off 26 KEYS_UPDATED , blkref #0: rel 1663/14012/2608 blk 62

  rmgr: Standby len (rec/tot): 42/ 42, tx: 0, lsn: 0/0CE573A0, prev 0/0CE57368, desc: LOCK xid 1833 db 14012 rel 18731

pg_waldump

- pg_waldump -r list
  - src/include/access/rmgrlist.h
- pg_waldump -r sequence…

- Parameter changes:
- rmgr: XLOG len (rec/tot): 50/ 50, tx: 0, lsn: 2/9410C4A8, prev 2/9410C438, desc: PARAMETER_CHANGE max_connections=100 max_worker_processes=8 max_prepared_xacts=0 max_locks_per_xact=64 wal_level=replica wal_log_hints=off track_commit_timestamp=off

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 (R.I.P as of v12) :** 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 2018-08-28 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.
  - A failure can cause parts of old data still remain on the data page!
- Full page writes IYF
  - Header data + entire page as a WAL record during the first change of each page after every checkpoint: Backup block / full page image
  - During replay, backup block overwrites data.
- 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

- Increases WAL I/O
- PostgreSQL writes header data + the entire page as WAL record, when a page changes after **every** checkpoint.
  - Increasing checkpoint_timeout and / or max_wal_size helps.
    - Low values has a side effect: More WAL activity, per above.-
  - Full-page image, backup block.
- PostgreSQL can even recover itself from write failures (not hw failures, though)

Full page writes

- Also needed by:
  - pg_basebackup, if you want to take backups from the standby node.
  - pg_rewind
- Increasing wal_buffers will help in busy environments.

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?

Photo time!

@CheerPostgreSQL

WAL for DBAs – Everything you want to know

Devrim Gündüz

Principal Systems Engineer @ EnterpriseDB

devrim.gunduz@EnterpriseDB.com

Twitter : @DevrimGunduz