Write-Ahead Logging (WAL):

The Internals of Reliability and Recovery

Hamid Akhtar,
Engineering Lead, PostgreSQL

December 12–15, 2023
Prague, Czechia

Want to know more about
Percona Software For Postgres?

PERCONA Distribution for PostgreSQL

PERCONA Distribution for MongoDB

PERCONA Distribution for MySQL

PERCONA Monitoring and Management

About Myself

- More than two decades of professional software development.
- Engineering Lead for PostgreSQL at Percona:
  - pg_stat_monitor
  - And, yes, we are working on ***TDE (Transparent Data Encryption).***
- Prior to joining Percona, I had worked for some other PostgreSQL companies:
  - HighGo, and
  - EnterpriseDB.

Presentation Outline

The expected takeaways from this session.

Outline

- Transactions and Failures
- A System Without WAL
- WAL in PostgreSQL
- The Internals
- Following Transaction Processing
- Checkpoint
- Backup and PITR

Transactions and Failures

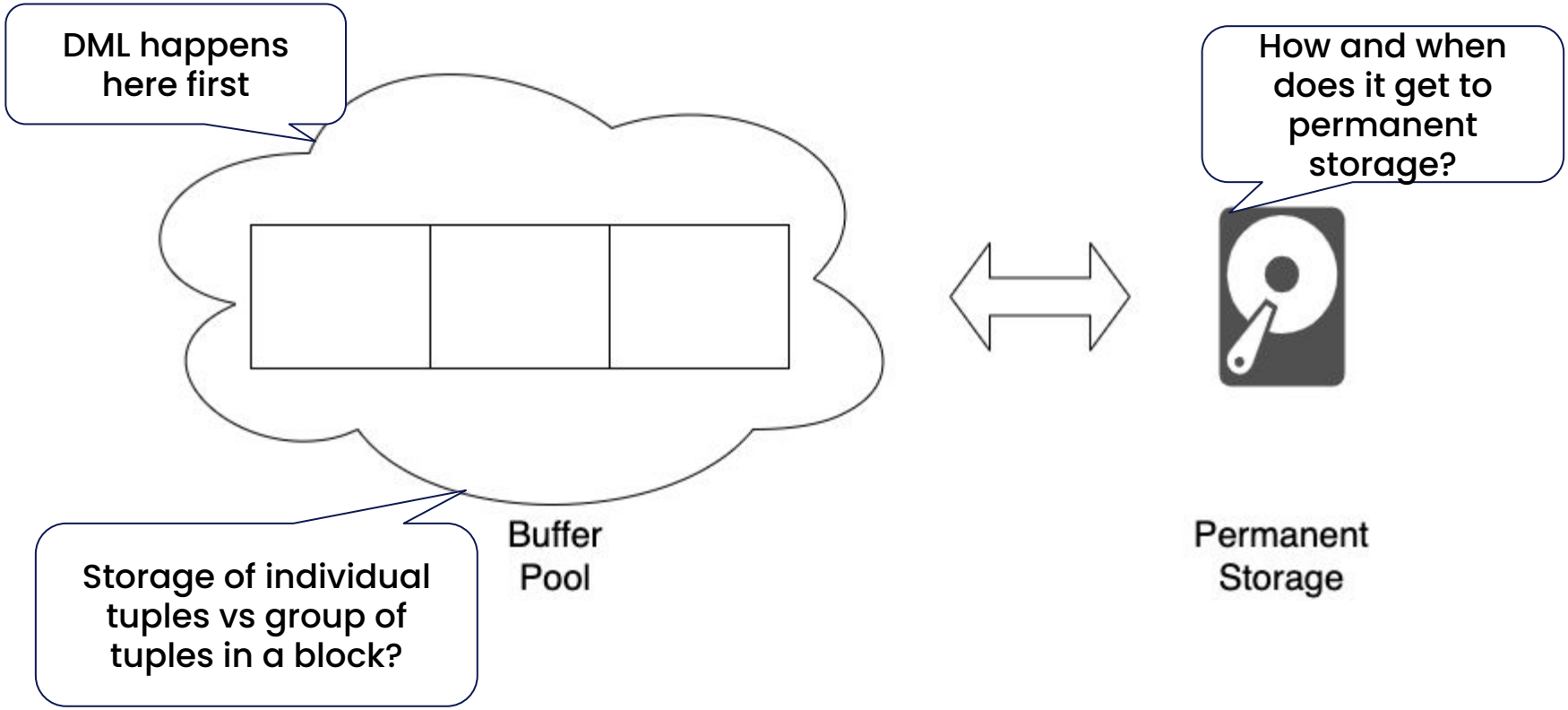
Making the database ACID compliance

The Types of Failures

- Transaction
  - Logical errors
  - Internal state errors
- System
  - Database server crashed
  - Host OS crashed
- Hardware
  - Recoverable failures; e.g. power loss
- Storage Failure
  - Oops. Nothing we can do here unless we have a backup of the data.

Let's see if we can somehow avoid data loss during these failures.

Challenge in Making Data Persistent



The Common Sense Factors

- RAM isn't really permanent storage.
- Data is updated in RAM first, and then written to disk.
- Reading and writing data in blocks is faster.
  - Changes performed on the data must be saved to persistent storage.
- The expectation is that when we tell a backend that their transaction is committed, the data is written to persistent storage.

Challenge with Data Blocks (Pages)

- Can we write this block onto persistent storage?
- When should we write this block onto persistent storage?

A System Without WAL

Durability can be ensure in many ways.

Possible Options

- Write the data page whenever a transaction makes a change.
- Shadow copy.

Saving Individual Commits

STEAL	FORCE

- PG < 7.1 - PostgreSQL could not guarantee
  - Consistency
    - Index tuples may point to non-existent table rows
    - Index tuples lost in split operations
    - Corrupt index or table pages because of partial writes.
  - All open data files had to be fsync'ed on every commit.
    - Now that's a serious performance issue.

Shadow Copy

NO STEAL	FORCE

- Primary page table point to valid pages.
- A transaction executing DML creates a copy of the page table (shadow).
  - Makes a copy of the pages where change is required.
  - Makes the changes.
  - When it commits, the database root is changed to point to the shadow page table.

WAL in PostgreSQL

It must bring the system back to the same stable state it was before the crash.

The Strategy

STEAL	NO FORCE

- ARIES
  - Algorithms for Recovery and Isolation Exploiting Semantics
- It mandates:
  - Write Ahead Logging
    - WAL data is written to disk before data page is written.
  - REDO
    - It is able to retrace the actions to bring the system back to the same stable it was before crashing.
  - UNDO
    - Any incomplete transactional data written to persistent storage can be undone.

WAL in PostgreSQL

- Write Ahead Log
  - Introduced in version 7.1 in 2001.
  - With REDO, and without UNDO.
  - Introduced checkpoint as well.
  - Added UNDO in 7.4
- Version 10 changed the directory to pg_wal from pg_xlog.
  - Fun Fact: rm -rf data/*log*
- Not just relations but indexes too!

The Internals

Pages and WAL under the hood.

Log Sequence Number (LSN)

- Refers to a location in the log file.
  - LSN is a 64 bit unsigned integer. In the PG code, it is:

```
typedef uint64 XLogRecPtr;
```

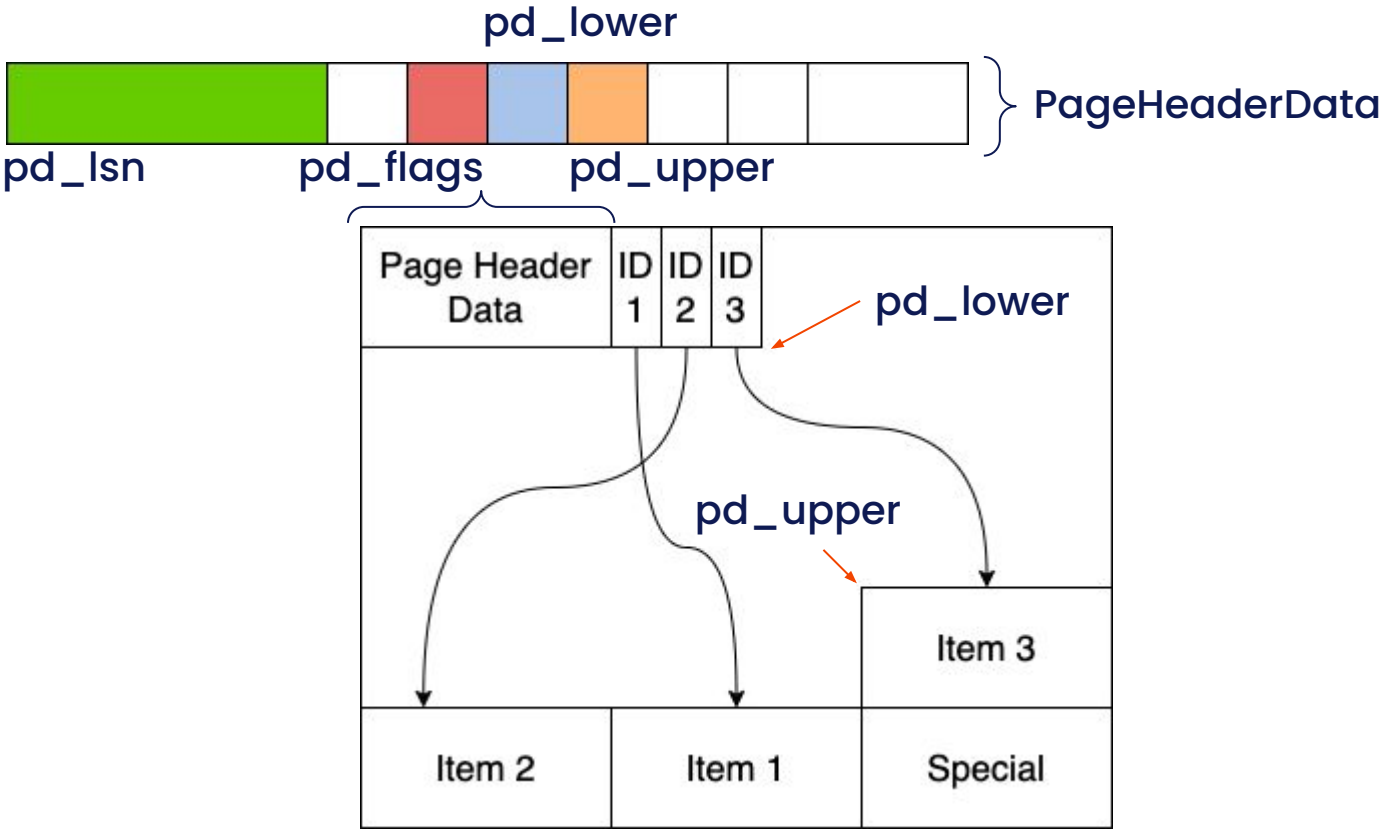
- LSN Format:
  - 00000000 00000000 00000000

- Some useful functions:
  - pg_current_wal_lsn
  - pg_walfile_name

Buffer Page

- Buffer Tag
  - Identifies which disk block the buffer contains. It's internally defined as a structure consisting of:
    - Tablespace OID
    - Database OID
    - Relation file number
    - Relation fork number
    - Block number in the relation.
- Buffer Descriptor
  - Location of a buffer page in the buffer pool slot.
  - Contains Buffer Tag, index, and state (flags, refcount, etc).

Data Page Layout



XLog API Functions

- XLogBeginInsert
  - Must be called when you intend to add an xlog record.
- XLogRegisterData
  - Arbitrary data that in the WAL record that will be available to the redo routine.
- XLogRegisterBuffer
  - Add information about the buffer to the WAL record. Contains information to re-find the page during the redo routine.
- XLogRegisterBufData
  - Included data associated with a registered buffer.
- XLogInsert
  - Insert the WAL record.

XLog Record Layout

- Fixed-size header (XLogRecord)
  - Includes total length of the record, transaction ID and resource manager ID.
- Block Header (XLogRecordBlockHeader)
  - Block ID, fork flags and data length
- <zero or more block headers>
- Data Header (XLogRecordDataHeader[Short|Long])
  - ID and data length of the main data part.
- Block Data
- <zero or more block data>
- Main Data

Full Page Images

- Written immediately to WAL when a page is modified for the first time after a checkpoint.
  - Avoids torn pages.
- The entire page is added to the WAL record.
  - Space is saved by omitting the empty space in the page.
  - And also through compression of data.
- If compression is enabled, a compression header is added to the xlog record.
  - It contains the size of the hole in the page.

WAL Resource Manager List

- The resource managers are used to identify the type of actions so that required routines may be invoked.
  - For more details, see the file “rmgrlist.h”.
- Resource manager entry for “heap”:

```
PG_RMGR(RM_HEAP_ID, "Heap", heap_redo, heap_desc, heap_identify,
        NULL, NULL, heap_mask, heap_decode)
```

- We’ll see how this is used in heap_redo.

WAL Resource Manager List

Category	Resource Manager ID
Heap	RM_HEAP_ID, RM_HEAP2_ID
Indexes	RM_BTREE_ID, RM_HASH_ID, RM_GIN_ID, RM_GIST_ID, RM_SPGIST_ID, RM_BRIN_ID
Replication	RM_STANDBY_ID, RM_REPLORIGIN_ID, RM_GENERIC_ID, RM_LOGICALMSG_ID
Sequences	RM_SEQ_ID
Storage	RM_SMGR_ID, RM_DBASE_ID, RM_TBLSPC_ID, RM_RELMAP_ID
Transactions	RM_XACT_ID, RM_MULTIXACT_ID, RM_XLOG_ID, RM_CLOG_ID, RM_COMMIT_TS_ID

Following Transaction Processing

How the elements of WAL get engaged in transaction processing.

Running Insert Transaction Command

- Let's consider a table "foo" with only one integer column.

```
INSERT INTO foo VALUES(1);
```

- Access method is "heap"
  - StartTransactionCommand
  - heap_insert function is called. Internally, it does the following:
    - GetCurrentTransactionId -> ExtendCLOG
    - Prepare the Tuple [heap_prepare_insert]
    - Get the buffer from the buffer pool [RelationGetBufferForTuple]
    - Enter critical section
    - …

Running Insert Transaction Command

- Add the Tuple to the Page [RelationPutHeapTuple]
- If page is all visible,
  - Unset the PD_ALL_VISIBLE flag in the page header
  - Update visibility map
- Mark the Page as Dirty [MarkBufferDirty]
- Add the WAL record by calling:
  - Begin Insert
  - Register Data [CTID and flags]
  - Register Buffer [The buffer containing the new tuple]
  - Register Buffer Data [Trimmed tuple header]
  - Register Buffer Data [Tuple data]
  - XLog Insert [RM_HEAP_ID, flags | XLOG_HEAP_INSERT]

Running Insert Transaction Command

- Update the pd_lsn to match the WAL lsn. [PageSetLSN]
- End critical section [Nothing to be done on the page]
- XactLogCommitRecord
- TransactionIdCommitTree

- The transaction is committed now.
- WAL Writer process is triggered by:
  - Commit or Abort
  - wal_writer_delay

Let's Recap

- We created and committed a transaction.
- A tuple was inserted into a heap page.
  - heap_insert added the tuple and generated the WAL record.
    - The page was marked as dirty.
    - Sets pd_lsn to point to the new LSN location; start of the next record.
  - Transaction was marked committed.
    - WAL record was generated for it.

- The data page is NOT written to disk yet.

Checkpoint and PITR

Durability and recovery

Checkpoint

- The primary functions are:
  - Cleaning dirty pages in the buffer pool.
  - Preparing server for recovery.
- When a “checkpoint” starts, it needs to flush all the dirty pages to disk.
  - But before it can do that, it saves the current WAL insert location.
- After completing the checkpoint, it writes into the WAL a checkpoint record with the redo location.

Following REDO

- Resource manager entry for “heap”

```
PG_RMGR(RM_HEAP_ID, "Heap", heap_redo, heap_desc, heap_identify,
        NULL, NULL, heap_mask, heap_decode)
```

- heap_redo function is called.
  - Gets the opcode from the XLogRecord and calls the required function. The opcode is XLOG_HEAP_INSERT in this case.
    - heap_xlog_insert in this case.
- Before changes on the page are made, LSN of the record is compared with pd_lsn of the page. Change is only IFF:
  - pd_lsn < LSN of the WAL Record

PITR

- Very similar to the recovery process except for:
  - It reads the WAL files from an archive directory.
  - The location of the redo point is read from a backup label file.

TDE is coming.

Feel free to reach out to me or any of the Percona folks here if you want to discuss TDE (transparent data encryption).

GitHub

Thank you!

hamid.akhtar@percona.com
https://www.linkedin.com/in/engineeredvirus/

PERCONA

PERCONA Distribution for PostgreSQL

PERCONA Distribution for MongoDB

PERCONA Distribution for MySQL

PERCONA Monitoring and Management