PostgreSQL Internals Part 4: A Beginner's Guide to Understanding WAL in PostgreSQL

Welcome to the fourth part of our PostgreSQL internals series.

In this blog, we'll dive into the concept of Write-Ahead Logging (WAL) in PostgreSQL, discussing its importance and examining the internal structure of WAL files.

Before going further if you're interested in exploring the earlier blogs in this series, you can read them here.

- PostgreSQL Internals Part 1: Understanding database cluster, database and tables
- PostgreSQL Internals Part 2: Understanding Page Structure
- PostgreSQL Internals Part 3: Understanding Processes in PostgreSQL

Write-Ahead Logging (WAL) is a critical feature in PostgreSQL that ensures data integrity and durability. It records changes to the database before they are applied, allowing PostgreSQL to recover data and restore the database to its most recent state in case of a crash or failure.

Before the introduction of Write-Ahead Logging (WAL) in PostgreSQL, PostgreSQL relied on a simpler mechanism for ensuring data integrity, which was less robust and did not support advanced features like point-in-time recovery and replication.

The primary method for ensuring data integrity was through the use of fsync operations to ensure that data was written to disk. However, this approach had limitations in terms of performance and did not provide the same level of reliability and flexibility as WAL.

PostgreSQL 7.1 (2001): The concept of Write-Ahead Logging was introduced in PostgreSQL WAL significantly improved the reliability and performance of PostgreSQL by ensuring that all changes to the database were first written to a log before being applied to the data files. This allowed for better crash recovery and more efficient data management.

Each major PostgreSQL release since the introduction of WAL has brought new features and improvements, showcasing the ongoing evolution and enhancement of the database system.

- **PostgreSQL 8.0 (2005)**: Introduced Point-in-Time Recovery (PITR).
- **PostgreSQL 9.0 (2010)**: Introduced Streaming Replication and Hot Standby.
- **PostgreSQL 9.6 (2016)**: Improved replication and recovery mechanisms.
- **PostgreSQL 10 (2017)**: Introduced Logical Replication.
- **PostgreSQL 13 (2020)**: Enhanced WAL archiving and replication.

How WAL works in PostgreSQL

Let's try to understand the working of WAL in PostgreSQL with a simple example. Imagine you have a PostgreSQL database with a table called accounts

```
CREATE TABLE accounts (

    id SERIAL PRIMARY KEY,

    name TEXT,

    balance NUMERIC

);Copy to Clipboard
```

We perform a series of transactions to update the balances of various accounts. Let's say the accounts table initially has the following data

```
INSERT INTO accounts (name, balance) VALUES ('Alice', 1000), ('Bob', 1500);Copy to Clipboard
```

We start a transaction to transfer $200 from Alice's account to Bob's account:

```
BEGIN;

UPDATE accounts SET balance = balance - 200 WHERE name = 'Alice';

UPDATE accounts SET balance = balance + 200 WHERE name = 'Bob';Copy to Clipboard
```

As these UPDATE statements are executed, PostgreSQL generates WAL records for each change. These records are initially stored in the in-memory WAL buffer.

```
WAL Record 1: Decrease Alice's balance by $200.

WAL Record 2: Increase Bob's balance by $200.Copy to Clipboard
```

And we COMMIT the transaction

```
COMMIT;Copy to Clipboard
```

PostgreSQL ensures that the WAL records are written to disk before the transaction is considered committed. This is done using the fsync system call to flush the WAL buffer to WAL segment files on the disk.

System Crash

Imagine a system crash that occurs immediately after the transaction is committed but before the changes are fully written to the data files. The WAL records are safely stored on disk, but the data files might not reflect the latest changes.

When PostgreSQL restarts after the crash, it begins the recovery process. PostgreSQL identifies the REDO point from the most recent checkpoint. The REDO point is the position in the WAL from which recovery should start.

PostgreSQL then replays the WAL records starting from the REDO point. In this case, it replays the WAL records for the transaction that updated Alice's and Bob's balances.

The changes recorded in the WAL are applied to the data files, ensuring that the database is brought to a consistent state.

After recovery, the accounts table reflects the committed transaction

```
SELECT * FROM accounts;

 id | name  | balance

----+-------+---------

  1 | Alice |    800

  2 | Bob   |   1700Copy to Clipboard
```

Question

What happens if a crash occurs in the in-memory WAL buffer? Could this lead to data loss?

Answer

The potential for data loss in the event of a crash while WAL data is still in the WAL buffer depends on the configuration of your PostgreSQL instance, particularly the synchronous_commit setting.

- **Synchronous Commit Enabled**: If synchronous_commit is set to on (which is the default), PostgreSQL ensures that WAL records are flushed to disk before a transaction is considered committed. This minimizes the risk of data loss because the WAL records will be safely stored on disk even if a crash occurs. Upon recovery, PostgreSQL will replay the WAL records to bring the database to a consistent state.
- **Synchronous Commit Disabled**: If synchronous_commit is set to off, transactions are considered committed as soon as they are written to the WAL buffers, but before the WAL records are flushed to disk. This can improve performance but increases the risk of data loss in the event of a crash, as any WAL records still in the buffer and not yet written to disk would be lost.

How PostgreSQL read WAL files for crash recovery

PostgreSQL uses XLOG records from WAL files and compares LSNs to ensure that all committed transactions are applied, maintaining database consistency. Here's the breakdown:

- PostgreSQL reads the XLOG (WAL) records from the WAL segment files during the recovery process. These records contain information about changes made to the database.
- When PostgreSQL encounters an XLOG record that pertains to a specific page, it loads that page into the shared buffer pool if it is not already present. The shared buffer pool is an in-memory cache that holds frequently accessed data pages.
- Each page in PostgreSQL has a Log Sequence Number (LSN) that indicates the last WAL record applied to that page. The LSN in the XLOG record is compared with the LSN of the page:
    - If the XLOG record's LSN is greater than the page's LSN, it means the changes in the XLOG record have not yet been applied to the page.
    - If the XLOG record's LSN is less than or equal to the page's LSN, it means the changes have already been applied, and no further action is needed.
- If the XLOG record's LSN is larger than the page's LSN, PostgreSQL replays the XLOG record. This involves applying the changes described in the XLOG record to the page. After applying the changes, the page's LSN is updated to match the LSN of the XLOG record.

This recovery process ensures that all committed transactions are applied to the database, bringing it to a consistent state.

WAL Segment File

PostgreSQL has a giant virtual file for changes, theoretically 16 exabytes in size. That's an incredibly large number! To give you a sense of scale, it's approximately 16 billion terabytes.

To manage this practically, PostgreSQL divides the virtual file into smaller 16 MB WAL segment files. A typical WAL segment file The WAL segment filename is a 24-digit hexadecimal number that looks like this: 000000010000000000000001.

This name is divided into three parts:

Timeline ID: 00000001

- The timeline ID is used to track different versions of the database's history. Think of it like saving different versions of a document. Each saved version has a unique identifier (timeline ID) so you can go back to any previous version if needed.

Log File ID: 00000000

- This is the WAL file number, which shows the sequence of the WAL file within that timeline

  Segment ID: 00000001

- The segment ID is the specific segment within the log file. Each segment is typically 16 MB in size.

  The first WAL segment file is named 000000010000000000000001. When the first file (000000010000000000000001) is filled up with changes, PostgreSQL moves to the next file, named 000000010000000000000002. This continues in ascending order 000000010000000000000003, 000000010000000000000004, and so on.

  After filling up 00000001000000000000FF (where FF is the hexadecimal representation of 255), the next file is 000000010000000100000000.

  Here, the middle part (00000000) increases by one to 00000001, and the last part resets to 00000000.

  This pattern continues. After 00000001000000010000000FF is filled, the next file is 000000010000000200000000.

  Again, the middle part increases by one (00000001 to 00000002), and the last part resets to 00000000.

  PostgreSQL starts with an empty WAL segment file. When we make a change to the database (e.g., insert a row). PostgreSQL writes an XLOG record for this change in the current WAL segment file. The XLOG record gets a unique address (e.g., LSN1). We then continue making changes, and PostgreSQL keeps writing XLOG records to the current WAL segment file. Each XLOG record gets a unique address (e.g., LSN2, LSN3, etc.).

  When the current WAL segment file reaches 16 megabytes, PostgreSQL starts a new WAL segment file. The new WAL segment file starts with the next XLOG record (e.g., LSN4).

  Get the current WAL file name

```
SELECT pg_walfile_name(pg_current_wal_lsn());

     pg_walfile_name

--------------------------

 000000010000000000000001

(1 row)Copy to Clipboard
```

I hope this guide has helped you understand WAL and its role in PostgreSQL. We'll dive deeper into WAL in future blogs, so stay tuned. Thank you for reading!