DATABASE MAMANGEMENT SYSTEM - CO3021

PostgreSQL vs HBase

Authors
Đỗ Nguyễn An Huy - 2110193
Phạm Võ Quang Minh - 2111762
Nguyễn Ngọc Phú - 2114417
Nguyễn Xuân Thọ - 2112378
Trần Nguyễn Phương Thành - 2110541

Mentor
MEng. Lê Thị Bảo Thu

Software Engineering
Academic year: 2024-2025

Contents

Table of figures

Table of tables

Work division

0.1 Theory work

Meaning of symbols:

- ✅ - Assigned and done (100%)
- 🟨 - Secondary contribution

Name	**Physical storage**	**Query processing**	**Transaction**	**Concurrency control**	**Recovery**
Đỗ Nguyễn An Huy (2110193)		🟨	🟨		✅
Phạm Võ Quang Minh (2111762)	✅				
Nguyễn Ngọc Phú (2114417)			✅		
Nguyễn Xuân Thọ (2112378)				✅	
Trần Nguyễn Phương Thành (2110541)		✅			

Table 1: Theory work division

0.2 Application work

Name	Role
Đỗ Nguyễn An Huy (2110193)	• Terminal frontend web interface. • Command execution model so that commands can be implemented. • `ls` command • `cat` command
Phạm Võ Quang Minh (2111762)	• Setup database • `cp` command • `mv` command
Nguyễn Ngọc Phú (2114417)	• `unalias` command • `alias` command
Nguyễn Xuân Thọ (2112378)	•
Trần Nguyễn Phương Thành (2110541)	

Table 2: Application work division

Chapter 1
Introduction

1.1 Foreword

HBase and PostgreSQL are two representatives of two flavors of DBMS - SQL and NoSQL. This should be enough to suggest that there exists *some* differences between the two. In fact, they are very far from each other on the spectrum - further than, say, PostgreSQL to MongoDB. Therefore, we decide to conduct a comparison between PostgreSQL and HBase to see why they are so different.

To sum up, most of the differences between PostgreSQL and HBase derive from the different use cases they serve, and the kind of processing workload they are optimized for.

From the perspective of processing, we can identify two categories:

Transactional processing: Computation pertaining to the daily operation.
- Real-time processing: Must complete in a few seconds at most.
- Relevant data ratio: Only a few records in a database are relevant to a query.
- Frequency: Very high.
- Dataset size: Small to medium.
- Data-access pattern: Random access.

Analytical processing: Computation that answers business questions.
- Batch processing: Can take from minutes to hours to complete.
- Relevant data ratio: A large portions of records are relevant to an analysis.
- Frequency: Very low - typically on a days/weeks/months' granularity's basis.
- Dataset size: Large to very large.
- Data-access pattern: Sequential.

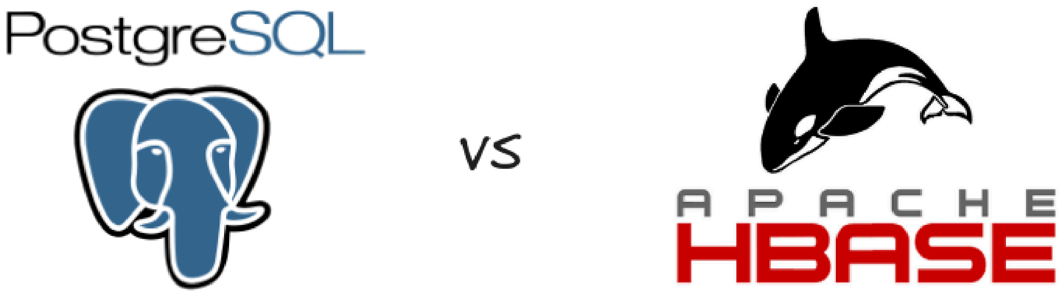


Figure 1: PostgreSQL vs HBase

PostgreSQL and HBase are designed for different use cases:

- **PostgreSQL**: Like good ‘ol traditional RDBMs, PostgreSQL are geared towards transactional use cases.
- **HBase**: An in-betweener between transactional & analytical processing. It was conceived of as a database engine for efficient distributed random access in the Hadoop ecosystem - which is designed for high-performance analytical batch processing on very big datasets.

This explains why they are different:

- **PostgreSQL** works well on small-to-medium datasets and mostly serves day-to-day business operations, such as banking, purchases, etc.
- **HBase** is designed for Big data use cases - efficient data analysis - but it’s also used for storing big transactional datasets, such as customer’s clickstream.

Because they are different, the conceptual models they use are different, thus are their schema design processes.

The reasons we decided to choose these two DBMSs are three-fold:

- Highlight the characteristics of each DBMS that are optimized for their use cases: data model, concurrency control & recovery, query processing, etc.
- Contrast the schema design processes of SQL and NoSQL databases.
- Considerations into when to use each.

1.2 Overview

1.2.1 PostgreSQL

PostgreSQL is a popular open-source object-relational DBMS which (kind of) implements the SQL language. In fact, no production DBMS so far has completely conformed to the SQL standard. However, in this regard, PostgreSQL is known for its high SQL conformance.

A bit of history, PostgreSQL dates back to 1986 and has been constantly developed for over 35 years. Therefore, PostgreSQL has undergone many big transformations. The latest major stable version as of 2024 is PostgreSQL 17.

According to Stackoverflow survey of 2023, PostgreSQL is the most popular DBMS.

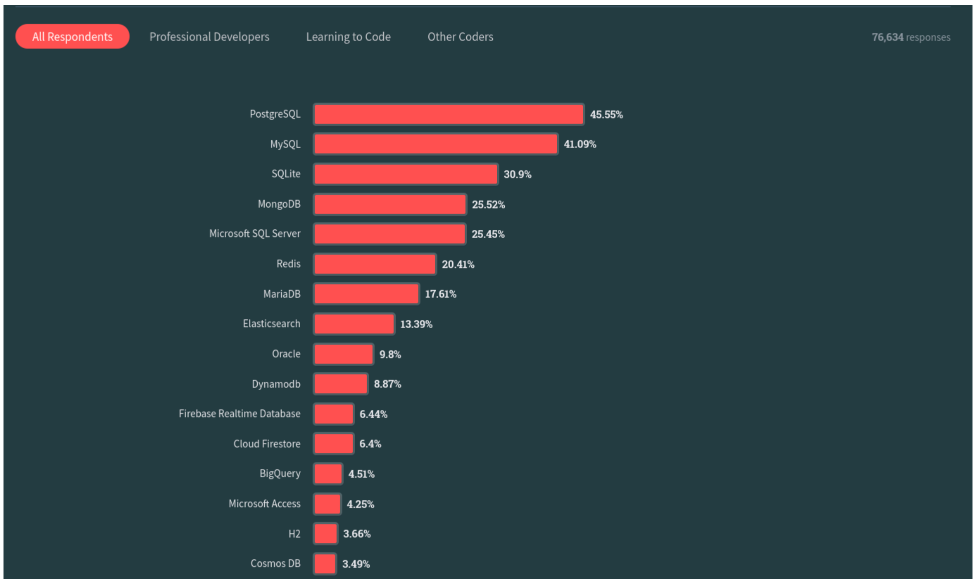


Figure 2: 2023 Stackoverflow's ranking of DBMSs

Being relational, PostgreSQL is suitable for most real-time transaction use cases.

The details of when to apply PostgreSQL rather than other RDBMS is left to the investigation of how PostgreSQL handles:
- Physical data storage.
- Indexing.
- Query processing.
- Transaction processing.
- Concurrency control.
- Recovery.
- Other advanced usages.

1.2.1.1 Philosophy

PostgreSQL emphasizes on:
- High SQL compliance: PostgreSQL tries to conform to the SQL standard as long as it doesn't badly hurt performance or hamper other well-known features of RDBMS.
- Extensibility: PostgreSQL has many plugins and mechanisms to extend its behavior.
  - Plugins & extensions. For example, foreign data wrapper (FDW), which allows a SQL server to query a remote relation.
  - Procedural Languages: PL/pgSQL, Perl, Python, and Tcl.
  - User-defined data types.

    - Custom functions, even from different programming languages.

1.2.1.2 NoSQL support

PostgreSQL is an **object-relational** DBMS so its capabilities extend beyond traditional features of RDBMS.

For example, PostgreSQL provides support for storing and efficiently querying the JSON data type:

- Native support for the `json` and `jsonb` data types.
- Access operators for JSON data such as `->`, `->>`, etc.
- A bunch of functions for building, iterating, transforming JSONs: `jsonb_build_object`, `jsonb_object_agg`, `json_each`, etc.

These make working with JSONs very expressive and flexible. In a sense we can work with JSON documents as in document-based NoSQL DBMSs. This means that we can model data in a schema-less fashion in PostgreSQL.

1.2.1.3 Summary

We can sum up the whole points as below:

- PostgreSQL is an object-relational DBMS.
- PostgreSQL has supports for modeling data like document-based DBMS.
- PostgreSQL is extensible.
- PostgreSQL is highly SQL-conformant.

1.2.2 HBase

HBase is a distributed wide-column NoSQL DBMS. It's part of the Hadoop ecosystem. In fact, it's the DBMS for the Hadoop Filesystem (HDFS), in the same sense as in PostgreSQL is a DBMS for the ext3, NTFS filesystems.

In the below picture, we can see that HBase is built upon HDFS:

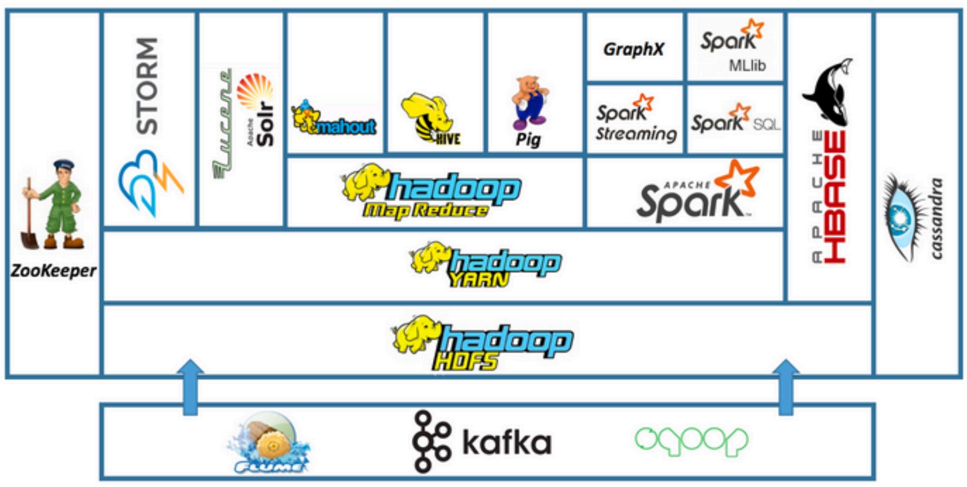


Figure 3: HBase in Hadoop landscape

HBase is still very young - first introduced in 2008. The latest version as of 2024 is 3, and it’s still in beta.

HBase is optimized for a mix of transactional and analytical use cases. Therefore, it should perform better on very big datasets than PostgreSQL.

In summary, HBase is conceived of as:

- Providing random access on top of the sequential access provided by HDFS filesystem.
- Providing real-time, random access to very large files.
- Integrating well with the Hadoop ecosystem.

Chapter 2
Data storage and management

2.1 What is data storage?

Logical and Physical Data Storage Strategy or something like that

2.2 HBase

2.2.1 Data Storage

1. Hadoop Distributed File System (HDFS)

Master-Slave

2.3 Postgres

1. Database Cluster

a) Logical View

A PostgreSQL server is a single process that runs in a SINGLE HOST and manages a *single* database cluster.

A database cluster is basically a collection of databases managed by a PostgreSQL server.

All the databases in cluster are internally managed by a 4-byte integers, called Object Identifiers (OIDs).

The relations between database objects and their respective OIDs are stored in appropriate system catalogs, depending on the type of objects. For example, OIDs of databases and heap tables are stored in pg_database and pg_class respectively.

b) Physical View

A database cluster is basically a single directory - or a **base directory**. For each database in the cluster, there is a subdirectory within PGDATA/base, named after the database's OID in pg_database.

This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there.

Each database is stored in a sub directory of **base directory**.

1.3 Layout of a database cluster

The layout of database cluster has been described in the official document

2.4 Comparison

Chapter 3
Query processing

Every data storage system needs to provide some way for users to specify their data needs, i.e to read/update/insert, on which fields/tables/records, etc. One way to achieve this is through queries - which can be thought as requests to the data storage system. Query processing is the process where these requests are analyzed to figure out how to satisfy those requests and also where these requests are run.

The way queries are specified varies:
- In PostgreSQL, we specify data needs via the structured query language (SQL).
- In HBase, only plain APIs are supported out-of-the-box.

3.1 HBase

By default, the only way to performs queries to HBase is via Java APIs. Although the Apache Hive provides a SQL layer on top of HBase, it's out of scope of this report.

The basic Java APIs for data operations:

- Get: Retrieves a single row or specific columns from a row.
- Scan: Retrieves multiple rows based on a range of keys.
- Put: Insert or updates a single row. Note that there is no API for Insert.
- Delete: Deletes a row, specific columns, or timestamped versions.

What happens when the user calls a Java API? Note that HBase is a distributed DBMS, it needs to locate where a piece of data is stored. In more details:

1. The HBase client library forms a request, which includes:
   - Operation: `Get`, `Put`, `Delete`, or `Scan`.
   - Target: Table, row, region, column.

   This request is serialized into a predefined format.
2. The client contacts the ZooKeeper to locate the meta-information about the requested table, including which region server holds the region for the target row.

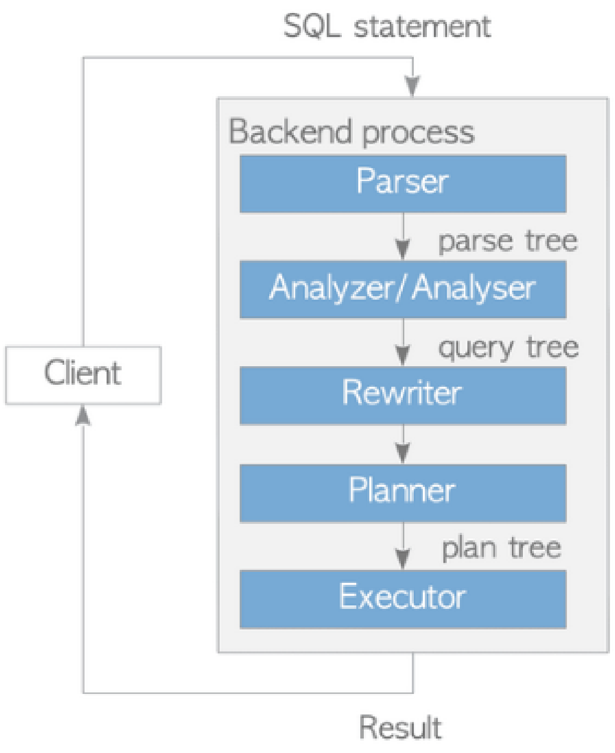
3. The client connects directly to the region server hosting the target region for the operation.
4. The region server then reads the request and performs the appropriates operations according to the request.

3.2 Postgres

Query processing in PostgreSQL is more involved, because:
- The SQL interface is a complex language on its own, which demands efforts for parsing.
- The SQL interface is very expressive and more flexible: users can specify whatever shape of data they desire. Therefore, it's more complicated to figure out an efficient way to execute the queries and to actually carry out query execution.

In fact, query processing the most complicated subsystem in PostgreSQL.

Every query issued by a connected client is handled by a type of process call the **backend process** in PostgreSQL. This process consists of 5 subsystems:



,

Figure 4: Query processing phases [1]

1. Parser

   This subsystem's role:
   - Check syntax of the SQL statement in plain text.

- Generate a **parse tree** from the SQL statement.

  Note that this phase does not check for semantics error, for example, selecting from a non-existent table or column.

2. Analyzer

   This subsystem performs **semantic check** from the **parse tree** and generates a **query tree**. To do this, it needs to fetch the metadata about relations in the system catalog.

3. Rewriter

   This phase implements the **rule system** in PostgreSQL. It takes the **query tree** from the analyzer and the **user-defined rules** (in the `pg_rules` system catalog) to rewrite it into another **query tree**.

   A rule is simply an instruction on how to rewrite the query tree. Underneath, it's just *another* **query tree** [2].

   To illustrate, views in PostgreSQL are implemented using the rule system [1], [2].

   For example, a view like:

   ```
   CREATE VIEW myview AS SELECT * FROM mytab;
   ```

   is very nearly the same thing as:

   ```
   CREATE TABLE myview (same column list as mytab);
   CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
       SELECT * FROM mytab;
   ```

   although you can't actually write that as tables cannot have `ON SELECT` rules.

   Therefore, a select from `myview` is rewritten into a select from `SELECT * FROM mytab`.

4. Planner

   This is where the query processor decides on how to execute the queries *efficiently*. To do this, it must consider (almost) all strategies to find the desired records, for example:
   1. How to locate the rows satisfying a condition effectively, i.e using index, sequential scan, etc.
   2. How to fetch the desired rows efficiently, i.e fetching the table's pages containing the rows or if index-only scan is applicable, fetching the index pages containing the rows.
   3. How to sort the rows efficiently if sorting is specified.
   4. How to perform joins efficiently if join is specified.

   From all of these considerations, the planner chooses the most approximately efficient plan and generates a corresponding plan tree.

5. Executor

   The executor accepts the **plan tree** and executes the query by accessing the tables and indexes in the order that was created by the **plan tree**.

3.2.1 Planner: Query optimization

As mentioned, query optimization happens in the planner.

PostgreSQL's query optimization is **cost-based**: there is no rule-based optimization. Besides, there is currently no way to hint a specific plan to PostgreSQL except for using extensions.

3.2.1.1 Terminology

Cost in PostgreSQL is:
- A dimensionless value. By default, a page read has a cost of `1`.
- Not an absolute performance indicator.
- Indicator to compare the relative performance of operations.

Cost is estimated by predefined functions (i.e. `cost_seqscan`, `cost_index`) using the statistics stored in the system catalog.

There are 3 kinds of costs:
- Startup cost:
  - The cost expended before the first tuple is fetched.
  - For example, the start-up cost of the index scan node is the cost of reading index pages to access the first tuple in the target table.
- Run cost: The cost of fetching all tuples.
- Total cost = Startup cost + Run cost.

Example: `0.00` and `145.00` are respectively the start cost and the run cost.

```
db=# EXPLAIN SELECT * FROM tbl;
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on tbl  (cost=0.00..145.00 rows=10000 width=8)
(1 row)
```

3.2.1.2 Cost estimation algorithm

In this section, for simplicity, we'll only illustrate cost estimation algorithm for single-table queries. The planner in PostgreSQL performs 3 steps:

1. Carry out **preprocessing**.
    1. Simplify the expressions in the SELECT clause, the LIMIT clauses, etc.

       For example, constant folding such as rewriting `2 + 2` to `4`.
    2. Normalize boolean expressions.

       For example, `NOT (NOT a)` is rewritten to `a`.
    3. Flatten `AND`/`OR` expressions.

       In the SQL standard, `AND` and `OR` are binary operators. In PostgreSQL internals, they are n-ary operators. Therefore, this step transforms the original representation to one that use n-ary versions of `AND` and `OR`.
2. Estimates the costs of all possible access paths and chooses the cheapest one.
3. Create the plan tree from the cheapest access path.

3.2.2 Executor: Sequential scan & Index scan & Index-only scan

Scans is a common operation in PostgreSQL, which scans the tuples of a relation searching for a matching tuple based on some conditions.

A sequential scan is the simplest scan. It simply scan the table page by page, tuples by tuples.

```
db=# EXPLAIN SELECT * FROM tbl;
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on tbl  (cost=0.00..145.00 rows=10000 width=8)
(1 row)
```

An index scan scan the index of the table to look for an appropriate match. Based on the availability of an appropriate index, an index scan may or may not be used:
- The indexed attributes must be specified in the scan filter condition.
- The scan filter operation should be compatible with the index. For example, equality filter or range filter can be applicable for B-Tree index.
- The statistics about selectivity: If PostgreSQL expects the scan to return a large portion of the table, it shall use sequential scan:
    - Sequential access is more efficient than random access in index scan.
    - After scanning the index, the table pages have to be fetched, but because the majority of tuples are expected, there's a high chance that all of the pages are fetched anyways. Therefore, the index scan in this case is just an unnecessary overhead.

Note that using an index scan, we have fetch both the index pages and the table pages. Why do we have to fetch the table pages also? Because the users requests some fields that are not present in the index file. By default, the index file contains only the indexed field, but it can be made to include additional fields (but not used for indexing). In cases that the requested fields are all present in an index, **index-only scan** may be used. This reduces the overhead of having to do another table fetch.

We'll give an example to sum up the above points:

- Table definition:

  ```
  db=# CREATE TABLE test (
    id INT,
    name TEXT
  );
  CREATE TABLE
  Time: 15.128 ms
  ```

- Populate data into the table:

  ```
  db=# CREATE SEQUENCE seq START 1;
  CREATE SEQUENCE
  Time: 9.864 ms

  db=# INSERT INTO test
    -# SELECT nextval('seq'), nextval('seq')::text || '_name'
    -# FROM generate_series(1, 10000000);
  INSERT 0 10000000
  Time: 29169.012 ms (00:29.169)

  db=# DROP SEQUENCE seq;
  DROP SEQUENCE
  Time: 9.085 ms
  ```

  We have to populate a large enough data - if all data just fits within a single page, **sequential scan** will always be used.

- Sequential scan (optimized into parallel sequential scan):

  ```
    db=# EXPLAIN SELECT id FROM test;
                                     QUERY PLAN
  --------------------------------------------------------------------------------
   Gather  (cost=1000.00..116286.39 rows=1 width=4)
     Workers Planned: 2
     ->  Parallel Seq Scan on test  (cost=0.00..115286.29 rows=1
  width=4)
           Filter: (id = 1)
   JIT:
     Functions: 4
     Options: Inlining false, Optimization false, Expressions true,
  Deforming true
  (7 rows)
  ```

```
  Time: 1.222 ms
```

- Create an index on `id`:

```
db=# CREATE INDEX idx_test ON test(id);
CREATE INDEX
Time: 4054.271 ms (00:04.054)
```

- Index scan:

```
db=#  EXPLAIN SELECT * FROM test WHERE id = 1;
 EXPLAIN SELECT * FROM test WHERE id = 1;
                              QUERY PLAN
----------------------------------------------------------------------
 Index Scan using idx_test on test  (cost=0.43..8.45 rows=1 width=17)
   Index Cond: (id = 1)
(2 rows)

Time: 0.566 ms
```

- Index-only scan:

```
db=# EXPLAIN SELECT id FROM test WHERE id = 1;
                                QUERY PLAN
--------------------------------------------------------------------------
 Index Only Scan using idx_test on test  (cost=0.43..4.45 rows=1
width=4)
   Index Cond: (id = 1)
(2 rows)

Time: 0.761 ms
```

- What if we want to use index-only scan when selecting both columns of the table while needing to index on only `id`?

```
db=# CREATE INDEX idx_all_test ON test(id) INCLUDE(name);
CREATE INDEX
Time: 4793.885 ms (00:04.794)

db=# EXPLAIN SELECT * FROM test WHERE id = 1;
                                  QUERY PLAN
--------------------------------------------------------------------------------
 Index Only Scan using idx_all_test on test  (cost=0.43..4.45 rows=1
width=17)
   Index Cond: (id = 1)
(2 rows)

Time: 0.776 ms
```

3.2.3 Executor: Join algorithms

Join algorithms are some of the more interesting algorithms in the executor. Therefore, a few join algorithms in PostgreSQL are presented here.

3.2.3.1 Nested loop join

This is the simplest join algorithm and can be used on all join conditions.

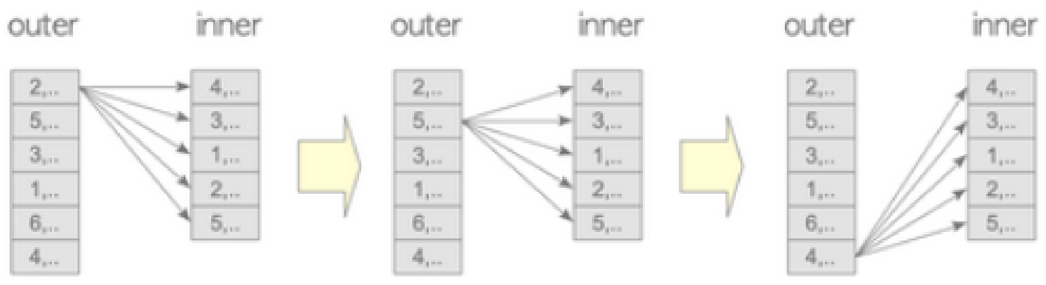


Figure 5: Nested loop join [1]

At its most basic form, for each row in the outer table, we loop over all rows in the inner table. This is costly as we have to scan all possible pairs.

PostgreSQL supports the materialized nested loop join to reduce cost of scanning the inner table:
- Before running a nested loop join, the executor writes the inner table tuples to the `work_mem` or a temporary file by scanning the inner table once.
- Inside the nested loop join, the executor can now load the inner table tuples with less I/O cost.

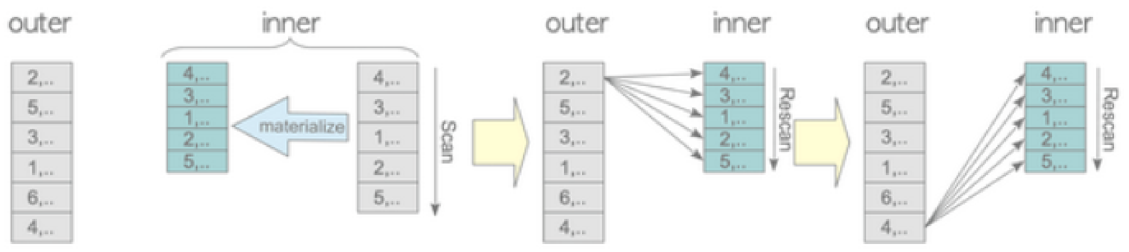


Figure 6: Indexed nested loop join [1]

PostgreSQL also supports the more complex indexed nested loop join, utilizing the index of the inner table if the join condition can be determined by that index. The inner loop uses the index of the inner table to lookup more efficiently.

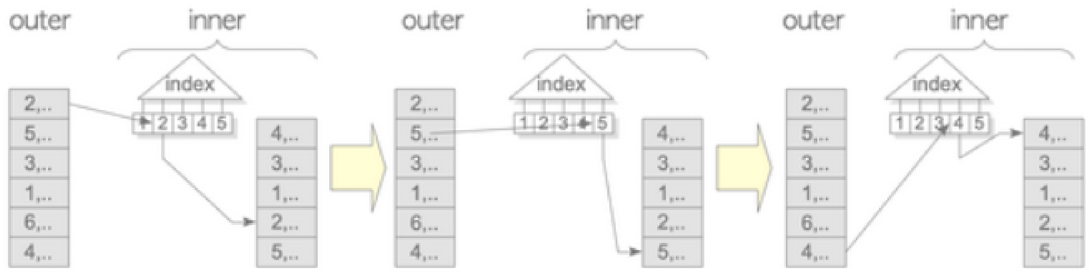


Figure 7: Indexed nested loop join [1]

3.2.3.2 Merge join

Merge join can only be used in equi-joins and natural joins.

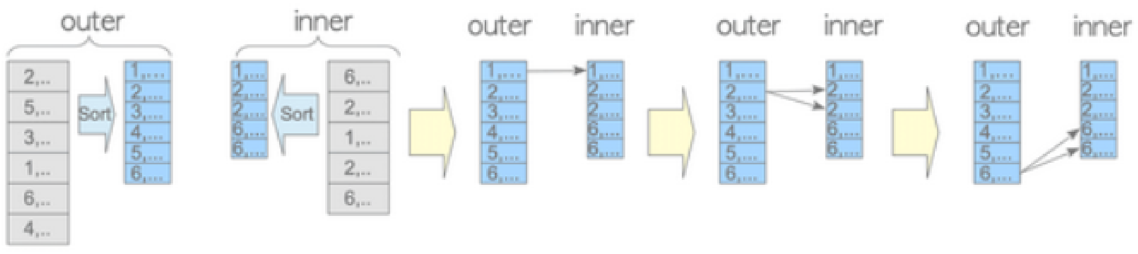


Figure 8: Merge join [1]

PostgreSQL first sorts the outer and inner tables. The resulting sorted tables will be stored in memory if they fit or else temporary files will be used. We can then keep 2 pointers to the current tuples of the outer and inner table and sequentially match them up.

There are some other variations of merge join supported by PostgreSQL.

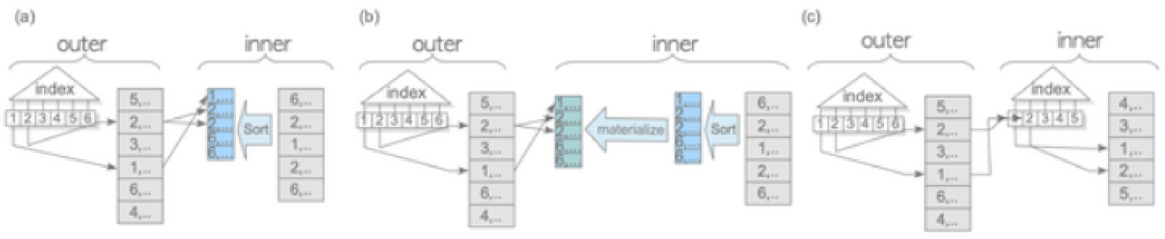


Figure 9: Merge join variations [1]

3.2.3.3 Hash join

Hash join can only be used in equi-joins and natural joins.

The hash join in PostgreSQL behaves differently depending on the sizes of the tables.

- If the target table is small enough (more precisely, the size of the inner table is ≤ 25% of the `work_mem`), it will be a simple two-phase in-memory hash join.
- Otherwise, the hybrid hash join is used with the skew method.

The **hash table area** is called a **batch** in PostgreSQL, which contains many **buckets**.

With in-memory hash joins, there are 2 phases [1]:
1. Build: All tuples of the inner table are inserted into a batch.
2. Probe: Each tuple of the outer table is compared with the inner tuples in the batch and joined if the join condition is satisfied.

The hybrid hash join with skew method is more complex, compared to in-memory hash join. Here are some differences:
- The outer table and the inner table are stored into multiple batches.
- The first batch of the inner table is kept in memory while the first batch of the outer table is not stored at all (we'll see why).
- Besides, an additional batch called the skew batch is kept in memory. This batch contains the inner table tuples whose attribute involved in the join condition appear frequently in the outer table.

The hybrid hash join happens in multiple rounds, with the first rounds very different from the remaining rounds [1]:

- First round:
  1. Build:
     1. Create a **batch** and a **skew batch** on `work_mem`.
     2. Create **temporary batch files** for storing the inner table tuples.
     3. Scan the inner table:
        1. If a tuple's joining attribute appears frequently in the outer table, it's inserted into the skew batch.
        2. Otherwise, calculate the hash key of the tuple and insert it into the corresponding batch.
  2. Probe:
     1. Create **temporary batch files** for storing the **outer table tuples**.
     2. Scan the outer table:
        - If the tuple appears frequently in the table, probe the skew batch and perform a join.
        - If the tuple hashes to the inner table's first batch, it's joined immediately. Therefore, the first outer batch file is never stored in `work_mem` or temporary file.
        - Otherwise, the tuple is written into an outer batch file, which corresponds to the inner batch file it hashes to.
- The second round:

1. Clear the skew batch and the first inner batch in `work_mem`.
2. Load the second inner batch into `work_mem`.
3. Scan the second outer batch and probe the inner batch file to perform joins.

- Remaining rounds:
  1. Load the corresponding inner batch into `work_mem`.
  2. Scan the corresponding outer batch and probe the inner batch file to perform joins.

3.3 Summary

Transaction processing in PostgreSQL is very heavyweight in terms of parsing, analyzing, planning and execution. This is due to the expressive power of SQL.

While in HBase, a simplistic interface is given via some plain APIs, this removes the need for parsing (other than deserializing the protocol serialized format). The APIs specify very simple operations: `Get`, `Put`, `Delete` - this make the planner & executor in HBase also simpler. The difficult part in HBase is how to communicate queries to a region server.

Chapter 4
Transaction processing

A transaction is a set of database reads and writes that is handled as a unit with a few crucial properties [3]. One such set of properties is ACID. Benefits of transactions [3]:
- Transactions make concurrency simple: Transactions relieve the burden of having to reason about potential interactions between operations from separate operations. The developers can think of transactions as executing sequentially.
- Transactions enable abstraction: *Application-defined* transactions are composable - the execution of one transaction can't affect the visible behavior of another. This allows us to mix-and-match transactions, compose transactions inside another, etc.

Transactions main use is maintaining ACID, enabling abstraction to the users so that they won't have to worry about transactions interfering with each other. To maintain ACID, transaction processing needs two subsystems: recovery subsystem and concurrency control subsystem. Because these two subsystems will be covered in more details in the next two chapters, this chapter only provides an overview of activities happening inside a transaction to allows for concurrency control.

4.1 HBase

In HBase, like typical NoSQL DBMS, has very limited transaction support. It does not have a built-in, explicit transactional framework for multi-row or multi-table operations. Each operation in HBase is a transaction itself [4]. Therefore, there's not much to say about transactions in HBase.

HBase provides **strict concurrency levels**, which means that:
- All reads immediately see the committed writes.
- All reads see writes in order of they are committed.

However all of these are guaranteed within a region only. Plus, we have the fact that each operation on a row is transaction (`Get`, `Put`, `Delete`) itself, except for `Scan` as it scans multiple rows and the rows are not guaranteed to be from the same snapshot in time. These two points make concurrency control in HBase significantly simpler:

It just needs to lock a row before updating. This lock blocks other concurrent updates on the same row, but doesn't block concurrent reads on the same row - reads don't see uncommitted changed on the rows as HBase uses some kind of **multiversion concurrency control** (MVCC). We'll explore it further and see its in Section 5.

For recovery, HBase utilizes the **write-ahead log** (WAL). For the same points above, the WAL in HBase is simplistic in both of its structure and associated operations. We'll explore it further in Section 6.

4.2 PostgreSQL

Unlike HBase, transactions is one of the core features of PostgreSQL and facilitate the ACID guarantees. In PostgreSQL, there's a component called the **transaction manager**, which is responsible for ensuring the ACID properties of transactions.

Each transaction can be user-defined by placing a set of operations between `BEGIN` and `END`.

```
BEGIN;

--- SQL statements

END;
```

`BEGIN` initiates a transaction block, that is, all statements after a `BEGIN` command will be executed in a single transaction until an explicit `COMMIT` or `ROLLBACK` is given [2].

Each transaction can also be started implicitly: By default (without BEGIN), PostgreSQL executes transactions in "autocommit" mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done) [2].

Each transaction in PostgreSQL is assigned a unique identifier called `txid` by the **transaction manager** when it starts [1]. We can query the `txid` of the current transaction like so:

```
db=# BEGIN;
BEGIN
Time: 8.349 ms
db=#* SELECT txid_current();
 txid_current
--------------
        30222
(1 row)

Time: 12.288 ms
```

```
db=#* END;
COMMIT
Time: 0.152 ms
```

The `txid` is a 32-bit unsigned integer, so it's limited. PostgreSQL reserves the following 3 special `txids`:

- `0`: Invalid `txid`.
- `1`: Bootstrap `txid`, which is only used in the initialization of the database cluster.
- `2`: Frozen `txid` (to avoid the transaction id wraparound problem - because txid is upper bound).

`txids` can be compared with each other. PostgreSQL views ordering as circular:

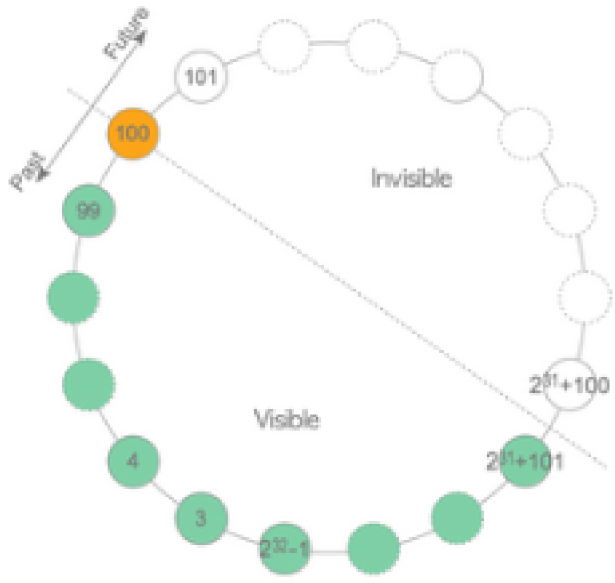


Figure 10: `txid` ordering [1]

- Half the cycle right before the `txid` is in the past and visible.
- Half the cycle right after the `txid` is in the past and invisible.

The "visibility" concept here has to do with how PostgreSQL performs concurrency control and will be explored in Section 5.

The transaction manager always holds information about currently running transactions.

There are two structures related to transactions & concurrency that PostgreSQL has to maintain: The `CLOG` and the Transaction snapshot. We'll defer it to Section 5. For now, we note that PostgreSQL uses MVCC for concurrency control.

There are 3 levels of isolation levels that can be specified for transactions in PostgreSQL: `READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`. `READ COMMITTED` is the default isolation level in PostgreSQL. `READ UNCOMITTED` is not present as the

MVCC concurrency model that PosgreSQL uses avoids dirty reads by default. See Section 5. We can specify the isolation level like so (for a single transaction):

```sql
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- Your SQL statements here
COMMIT;
```

We can also specify isolations at the session level:

```sql
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

For recovery, the PostgreSQL also uses the WAL like HBase, but it’s significantly more complex. We’ll defer it to Section 6.

Chapter 5
Concurrency control

5.1 Problems with concurrent execution

5.1.1 Lost update problem (write - write conflict)

5.1.2 Dirty read problem (write - read conflict)

5.1.3 Unrepeatable read problem (write - read conflict)

5.2 PostgreSQL

PostgreSQL uses Multi-version Concurrency Control (MVCC) to maintain data consistency. Each SQL statement sees a snapshot of data as it was some time ago, regardless of the current state of the underlying data. This provides transaction isolation by preventing statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows.

5.2.1 Transaction Isolation

Isolation level	**Dirty read**	**Non-repeatable read**	**Phantom read**	**Serialization anomaly**
`READ UNCOMMITTED`	Not possible	Possible	Possible	Possible
`READ COMMITTED`	Not possible	Possible	Possible	Possible
`REPEATABLE READ`	Not possible	Not possible	Not possible in PostgreSQL	Possible
`REPEATABLE READ`	Not possible	Not possible	Not possible	Not possible

Table 3: Isolation level in PostgreSQL

5.2.1.1 Read Committed Isolation Level

5.2.1.2 Repeatable Read Isolation Level

5.2.1.3 Serializable Isolation Level

5.2.2 Explicit Locking

5.2.2.1 Table-level Locks

5.2.2.2 Row-level Locks

5.2.2.3 Page-level Locks

5.2.2.4 Deadlocks

5.2.2.5 Advisory Locks

5.2.3 Data Consistency Checks at the Application Level

5.2.3.1 Enforcing Consistency with Serializable Transactions

5.2.3.2 Enforcing Consistency with Explicit Blocking Locks

5.2.4 Serialization Failure Handling

5.2.5 Caveats

5.2.6 Locking and Indexes

Chapter 6
Recovery

Any software systems can run into failures. DBMS is no exception and failures during a transaction can render the whole database corrupted.

There are many possible reasons why a DBMS may fail:
- Concurrency control: Multiple transactions running concurrently can potentially cause serialization anomalies or deadlocks. In those cases, the recovery subsystem may need to restart some transactions & rollback the database's state into a previous consistent state.
- Software interruption: Users may decide to interrupt a transaction mid-flight. A transaction must be rolled back in this case.
- Power failure: If the system is interrupted while a transaction is running, this would hang the transaction and thus violate the atomicity of transactions. This is one of those failure scenarios that we cannot avoid totally. In this case, the recovery system must either log enough information so that the completed operations can be rolled back or the not-completed operations can be carried on.
- File system/Disk failure: Another scenario where failures cannot be avoided. If the file system or the disk is faulty, any I/O can be unreliable. One way to recover from this is to restore the database state from a previous backup. This is not of ours concern in this section.

This is where recovery comes in: Its main role is to recover the database from a corrupted state. Specifically, recovery plays an important role in ensuring the ACID properties of transactions, specifically the A (atomicity) and D (durability). Recovery is unavoidable if reliability is to be achieved.

6.1 Recovery in PostgreSQL

Like many other RDBMS, PostgreSQL's main recovery mechanism is via the Write-ahead log (WAL).

6.1.1 General WAL concepts

The **WAL file** is an append-only sequential log file residing on disk, typically stored at `$PGDATA/pg_wal`.

When running, the PostgreSQL server also has a **WAL buffer** residing in shared memory to speed up logging. The **WAL buffer** is, at some appropriate time, flushed to the **WAL file**.

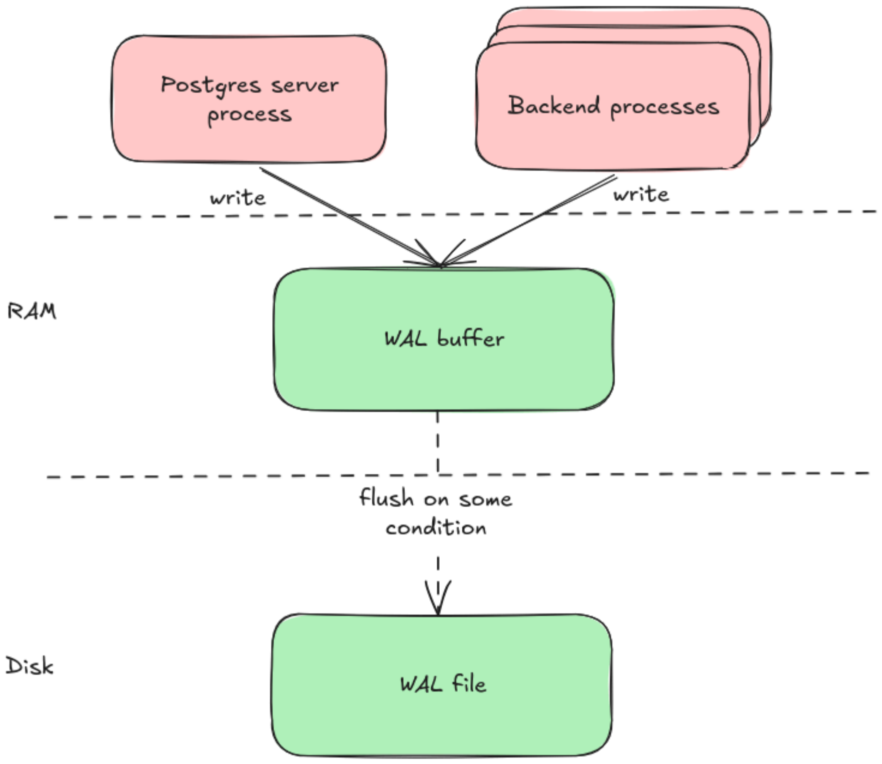


Figure 11: The WAL buffer & WAL file

The main idea of WAL is as follows:
- Each operation (in many cases, not all types of operations) in a transaction is logged to the WAL buffer.
- Before the data files are modified by an operation, the WAL record of that operation has to be written to the WAL buffer first. Only after the record has been written can the operation commence.
- A WAL record contain enough information so that its corresponding operation can be redone or rolled back idempotently (meaning that if we redo/rollback the same operation many times, we'll see the effect of redoing/rolling back exactly once).
- Before a transaction commits, the WAL buffer must be flushed to the WAL file first. [1]

Therefore, the WAL file completely captures at a moment in time which transactions are running & which operations have been or have not been done. If a crash happens, the database server can reload the WAL and replaying or rolling back to a consistent state.

Why does the WAL buffer has to be flushed to disk first before a transaction commits? To enforce Durability. Imagine after a transaction successfully commits, a power failure happens while the WAL is not flushed to disk. Essentially, the commit is lost and the database can only be recovered to a state before the commit happened - so the transaction will seem to have never committed. However, this violates Durability, which mandates that if a transaction has committed, its effect is always visible.

6.1.2 WAL structure in PostgreSQL

We'll be concerned of what information is stored in the WAL so that this can be used to recover the database to a consistent state.

In PostgreSQL, The WAL is called a **transaction log**. Logically, a **transaction log** is a virtual file whose bytes can be indexed by an 8-byte address. This means that this virtual file is 2^64 bytes (16 exabytes or 2^34 gigabytes) in size, which is vast enough for any real applications. However, physically, PostgreSQL cannot handle a file this big, so it splits the **transaction log** in chunks of 16 megabytes (by default). Each of these chunks is called a **WAL segment**. [1]

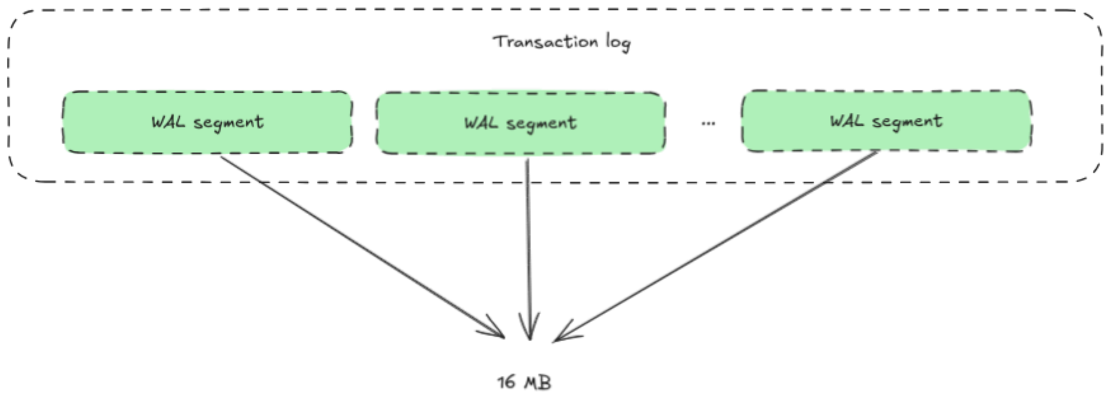


Figure 12: The transaction log structure

Each WAL segment is stored in a separate disk file [1]. This file is further divided to pages of 8 kilobytes:
- Each page has a page header.
- The XLOG records are written after the page header.

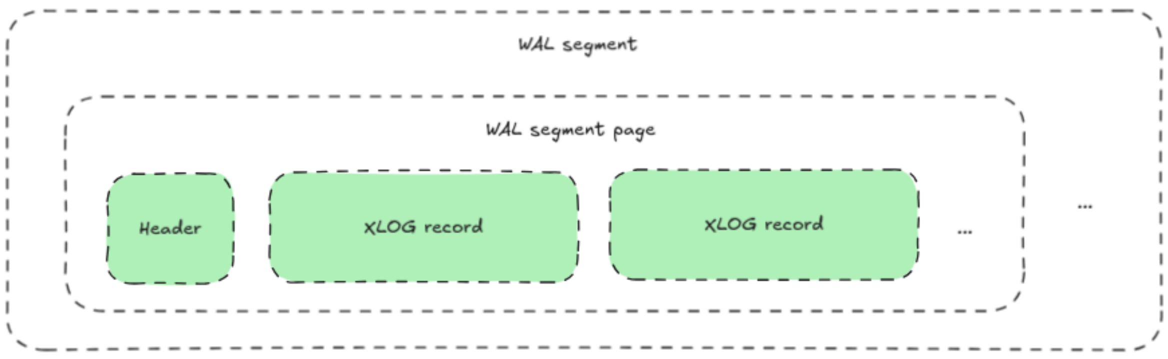


Figure 13: The WAL segment structure

Each XLOG record has a **log sequence number** (LSN) that is the index of the XLOG record in the transaction log. [1]

The XLOG record comprises a general header portion and each associated data portion. [1]

The header's most notable fields:
- `xl_xid`: The associated transaction ID.
- `xl_rmid` and `xl_info`: The information related to the resource manager for this record.
  - A resource manager is a collection of operations associated with the WAL feature, such as writing and replaying of XLOG records.
  - In short, these identifies if the XLOG record stands for:
    - A heap operation: `INSERT`, `UPDATE`, etc.
    - An index operation.
    - A transaction operation, like a `COMMIT`.
    - etc.

Therefore, the header completely identifies an operation in a specific transaction.

The data portion is highly dependent on the type of operation the XLOG presents. We just note that the data portion also contains a header and multiple data blocks. Each data block can be a **backup block** (storing an image of a full page of the database) or a **non-backup block**. [1]

6.1.3 Checkpoint

A checkpoint is a special XLOG record that marks the redo point if PostgreSQL is to recover from the WAL.

The latest checkpoint is the redo point.

In PostgreSQL there's a process called the checkpointer, which periodically performs checkpointing:

1. An XLOG record of this checkpoint is written to the WAL buffer.
2. All data in shared memory is flushed to the storage.
3. All dirty pages in the shared buffer pool are gradually written and flushed to the storage.

On recovery from a system failure, PostgreSQL **always rolls forward** from the REDO point and replays all XLOG records from there.

6.1.4 Recovery algorithm in PostgreSQL

Assume there's a transaction that inserts a tuple into `TABLE_A` and then commits. Assume that the latest XLOG record corresponding to `TABLE_A` has the log sequence number `LSN_0`.

Here's what happens if the transaction can be completed till end [1]:

1. Assume the checkpointer performs a checkpoint here.
2. PostgreSQL loads the `TABLE_A` page into the shared buffer pool in RAM.
3. PostgreSQL inserts a tuple into the page.
4. PostgreSQL creates and inserts an XLOG record of this statement into the WAL buffer at the location `LSN_1`.
5. PostgreSQL updates the `TABLE_A`'s `LSN` from `LSN_0` to `LSN_1`.
6. As this transaction commits, PostgreSQL:
    1. Creates and writes an XLOG record of this commit action into the WAL buffer.
    2. Writes and flushes all XLOG records on the WAL buffer to the WAL segment file, from `LSN_1`.

Assume that there's a power cut just after step 6. This means that:
- In the WAL file, the transaction appears to have committed.
- The shared buffer pool was lost, so the updated table pages are lost.

Here's what PostgreSQL does to recover [1]:
1. PostgreSQL reads all items in the `pg_control` file - an 8kb binary file that stores information about various aspects of the PostgreSQL server's internal state. If the `state` item is `in production`, PostgreSQL enters recovery-mode because this means that the database was not shut down normally. If it is `shut down`, PostgreSQL enters normal startup-mode.
2. PostgreSQL reads the latest checkpoint record in the `pg_control` file, which is the redo point.
3. PostgreSQL starts from the redo point (the latest checkpoint in the WAL file), which is at the start of the transaction.
4. PostgreSQL reads the XLOG record of the first `INSERT` statement from the WAL segment file.

5. PostgreSQL loads the `TABLE_A` page from the database cluster into the shared buffer pool.
6. PostgreSQL compares the XLOG record's LSN with the corresponding page's LSN. The rules for replaying XLOG records are as follows:
   - If the XLOG record's LSN is larger than the page's LSN, the data-portion of the XLOG record is inserted into the page, and the page's LSN is updated to the XLOG record's LSN.
   - If the XLOG record's LSN is smaller, there is nothing to do other than to read next WAL record.
7. PostgreSQL replays the remaining XLOG records in the same way, in this case, replays the COMMIT XLOG record.
8. After replaying all XLOG (WAL) records during the recovery process, PostgreSQL checks the state of each transaction. If it encounters a transaction that was incomplete at the time of the crash, PostgreSQL aborts the transaction. Because the COMMIT XLOG record is present, the transaction is successful upon recovery

6.1.5 Summary

1. PostgreSQL has a WAL file which serves as a history of all operations performed in every transaction.
2. In the WAL file, there are multiple checkpoints, the latest of which is the redo point to start recover from.
3. With most operations, PostgreSQL logs them in the WAL file so that they can be replayed upon recovery.
4. If after recovery, the transaction is found to be incomplete, it is rolled back.

6.2 Recovery in HBase

Because HBase is a distributed DBMS, we'll be concerned with 2 problems of recovery:
- How does the cluster recover to a consistent state if one of its data node fails?
- How does a region server recover locally to a consistent state after a failure?

6.2.1 How the cluster recovers from a region server's failure

We'll only consider how to recover from a region server's failure because these are where data can be stored. Although the master server may fail, this only causes availability problem & should not corrupt data, therefore, we skip the treatment for this type of failure.

There are some points worth reiterating about the HBase architecture:
- Data are assigned into regions - which is a partition of a table in HBase.

- Regions are further assigned to region servers - which serves all read and write requests for the region.
- By default, region server has no replica. This means that by default, each region is served by exactly one region server.

Although distributed in nature, the fact that each region is stored at one region server means that if one region server fails, the corresponding regions are not available until recovery.

The master server detects a region server failure via Zookeeper. Zookeeper will determine Node failure when it loses region server heartbeats. The master server will then be notified that the region server has failed.

The question here is that if a region is only assigned to one region server and that region server has crashed, how do we retrieve data in that region? In fact, although each region server is the sole server for some regions, HBase typically uses HDFS, which always performs data replication. Therefore, region data are unlikely to be lost. The master server can reassign the replicated region data to other active region servers.

However, there is one point yet to be addressed: What if the crashed server still has unflushed mutations in the Memstore? Each region server actually maintains a WAL file, and like region data, it's written to HDFS and is also replicated. The master will split the WAL file based on regions. Each split of the WAL file is then sent to a new region server that serves the split's region. Each new region server will then replay the WAL split.

The detail of how HBase organizes, maintains and replays WAL is fully treated when we consider the next question.

The remaining sections deal with the local recovery-related activities at each region server.

6.3 Revisit HBase concurrency model

These are the most notable points about the concurrency model of HBase:
- HBase has no mixed read/write transactions [5].
- HBase provides the strong concurrency level [6], which means that all reads always return the latest committed version & the clients can observe changes in the same order as committed writes.
- A transaction is based on a row-by-row basis, for example:
  - A mutation is atomic within a row, even spanning across column families [4].
  - All rows returned via any access API will consist of a complete row that existed at some point in the table's history [4].

    - A scan is not a consistent view of a table, that is, a mutation during a scan can cause the scan to return mutated rows [4].
- HBase does not guarantee any consistency between regions [4].

In short, HBase does not guarantee ACID across multiple rows and across multiple operations. Each operation mostly acts like its own transaction. This makes recovery much simpler compared to PostgreSQL.

6.3.1 WAL in HBase and how it differs from PostgreSQL

In HBase, the WAL is called a HLog [7]. The HLog in HBase is also an append-only sequential log.

However, there are a few notable differences from the WAL in PostgreSQL:
- HBase only maintains an HLog file, there's no WAL buffer.
- A single record is created for updates. There is no separate commit record.

6.3.2 Recovery algorithm in HBase

Assume that we're performing an insert to a row in an HBase table via a `PUT`.

1. The `PUT` is routed to the region server serving the region of the specified row.
2. The region server locks the row to prevent concurrent writes to the same row.
3. The region server retrieves the current WriteNumber.
4. The region server logs the change to the HLog **on disk**.
5. The region server applies the changes to the MemStore, tagging `KeyValues` with the retrieved WriteNumber.
6. The region server commits the transaction by attempting to roll the ReadPoint forward to the acquired WriteNumber.
7. The region server unlocks the row.

Note that the MemStore is in memory and changes to the MemStore is not flushed to disk until it is full. Note that the granularity of transaction in HBase is pretty low - short time between initiation and commit. Therfore, there's no need for an HBase buffer as it would be committed and flushed shortly after that, making the buffer rather less useful. In the same light, due to each operation being essentially its own transaction, there's no commit record as the single HLog record that we write at the start already represents the whole transaction - in another word that HLog record is its own commit record.

6.3.3 Summary

1. HLog also resides on disk but not in-memory.

2. For each operation, HBase writes a record to the HLog on disk before making mutations.

6.4 Comparison

The basic idea & operations of WAL are the same in PostgreSQL & HBase:
- There are append-only sequential logs on disk in both PostgreSQL & HBase.
- Before making a mutation to data in an in-memory buffer, it is logged into the WAL first.
- Recovery is done by replaying the records in the WAL file.

However, the WAL and its associated operations in PostgreSQL are far more complex and flexible compared to HBase. Besides recovery, the WAL in PostgreSQL can also be used in continuous archiving, point-in-time recovery, etc.

Nevertheless, in both PostgreSQL and HBase, the main point of the WAL existence is recovery.

Chapter 7
Conclusion

Bibliography

[1] Hironobu Suzuki, “The internals of PostgreSQL.” [Online]. Available: https://www.interdb.jp/pg/

[2] PostgreSQL org, “PostgreSQL 17.2 documentation.” [Online]. Available: https://www.postgresql.org/docs/17/index.html

[3] “Transaction manifesto.” [Online]. Available: https://apple.github.io/foundationdb/transaction-manifesto.html

[4] Apache HBase project, “ACID semantics.” [Online]. Available: https://hbase.apache.org/acid-semantics.html

[5] Lars Hofhansl, “ACID in HBase.” [Online]. Available: http://hadoop-hbase.blogspot.com/2012/03/acid-in-hbase.html

[6] Apache HBase, “Apache Hbase reference guide.” [Online]. Available: https://hbase.apache.org/book.html

[7] “HBase Gitbook.” [Online]. Available: https://nag-9-s.gitbook.io/hbase