How PostgreSQL Works: Internal Architectur Explained

ASHISH PRATAP SINGH AND ALEXANDRE ZAJAC
APR 08, 2025

101 6 18 Share

This post is a collaboration with Alexandre Zajac — Engineer at Amazon, tech c and author of the Hungry Minds newsletter.

In this post, we'll explore how PostgreSQL works under the hood and dive into t architecture that makes it a powerful choice for a wide range of use cases.

PostgreSQL has emerged as one of the most powerful and versatile open-source relational databases, trusted by software engineers to handle everything from sm applications to large-scale enterprise systems.

Its **robustness, flexibility, and rich feature set** make it a go-to choice for develop worldwide. But to truly harness its potential, understanding its internal architect and advanced features is essential.

In this blog post, we'll take a deep dive into PostgreSQL's core components and capabilities, with insights that will help you optimize performance, scalability, a reliability in your applications.

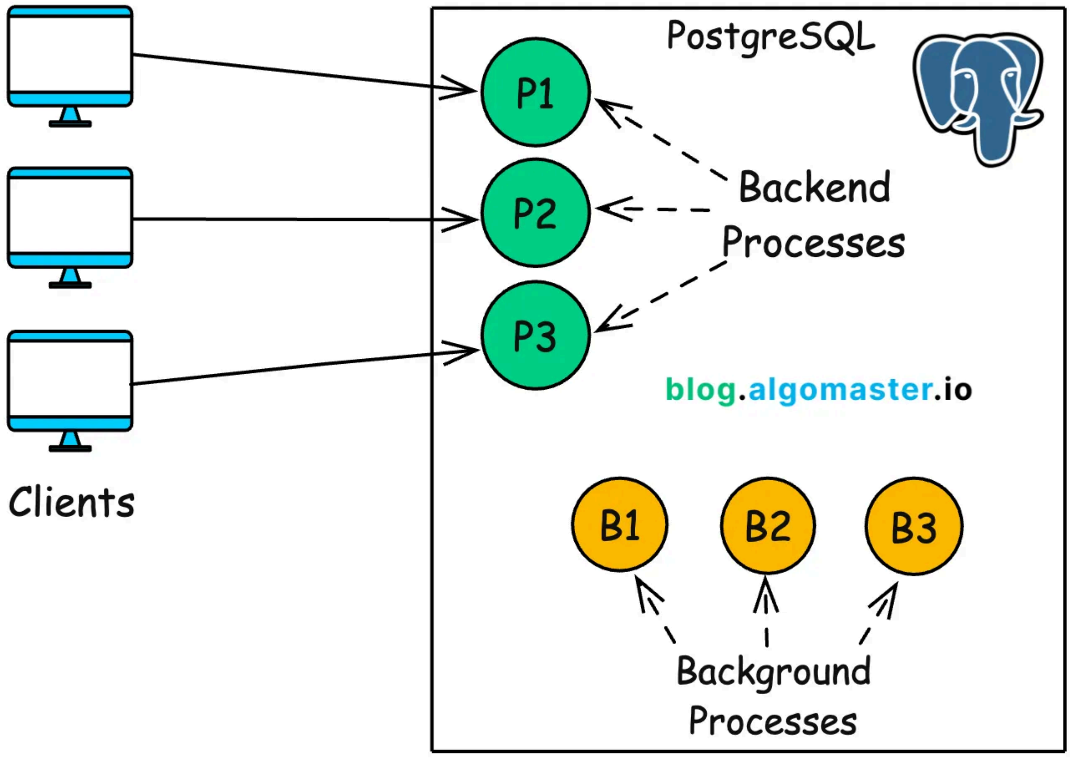
We'll explore:

1. **Process-Based Architecture**: How PostgreSQL manages connections for sta and isolation.
2. **Write Ahead Logging** (**WAL**): Ensuring data durability, crash recovery, and replication.
3. **Multi Version Concurrency Control** (**MVCC**): Allowing concurrent reads an writes without blocking.
4. **Query Execution Pipeline**: From parsing and planning to execution and resu delivery.
5. **Indexing System**: Choosing the right index for your data.
6. **Table Partitioning**: Managing large tables efficiently with range, list, or has based partitioning.
7. **Logical Decoding**: Streaming changes for replication and change data captu
8. **Extensions**: Extending PostgreSQL's capabilities with custom features.
9. **Statistics Collector**: Real-time insights for monitoring and optimizing datal performance.

Let's get started!

1. Process-Based Architecture

PostgreSQL follows a **process-per-connection architecture**, meaning each client connection is handled by a dedicated operating system process.



When the PostgreSQL server (postmaster) starts, it listens for incoming connecti on the configured port. For each new client connection, it **forks a new backend process** to handle that session. This backend process handles all communication query execution for that client.

Once the session ends (i.e., the client disconnects), the associated process termin

This architecture differs from **thread-based models** used in some other database MySQL or SQL Server), where a single process spawns threads for multiple connections.

Why PostgreSQL Chooses Processes?

- **Isolation**: Each connection runs in its own process, ensuring that if one sess crashes, it doesn't affect others. This design reduces the risk of memory corruption, race conditions, or resource conflicts between clients.
- **Stability**: The process model provides a higher level of stability, as issues in connection are contained within its process.
- **Simpler Internals:** Since each connection is isolated, PostgreSQL doesn't ne fine-grained locking on every internal data structure (as thread-based systen

This makes the system easier to maintain, debug, and extend.

Trade-Offs

While robust, this architecture does come with certain trade-offs:

- **Memory Overhead**: Each backend process maintains its own stack and local memory for query execution. This can consume significant RAM, especially many idle or parallel connections are open.
- **Connection Scalability**: Handling thousands of concurrent client connectio means creating thousands of OS processes. This can lead to high context-switching costs and pressure on kernel-level resources.

Best Practices

To maximize performance while retaining the benefits of PostgreSQL's architect

- Use a **connection pooling** like PgBouncer
  - PgBouncer sits between clients and PostgreSQL, reusing a small pool of persistent connections.
  - This drastically reduces process overhead while supporting thousands o clients.
- Tune **max_connections** parameter in `postgresql.conf`
  - Don't allow too many concurrent processes unless your hardware can ha them.
  - A common pattern: set `max_connections` to ~200–500, and let PgBounc manage the rest.

Background Processes

In addition to client backends, PostgreSQL runs **persistent background process** also forked by the postmaster at startup. Each has a distinct job and helps with s maintenance and performance:

- **Checkpointer:** Periodically flushes dirty pages from shared buffers to disk (t bound recovery time)
- **WAL Writer:** Writes WAL (Write-Ahead Log) changes from memory to disk
- **Autovacuum Workers:** Automatically cleans up dead tuples and refreshes statistics
- **Background Writer:** Continuously writes dirty buffers in the background to smooth out I/O spikes
- **Replication Workers:** Handles streaming WAL to replicas for physical/logic replication

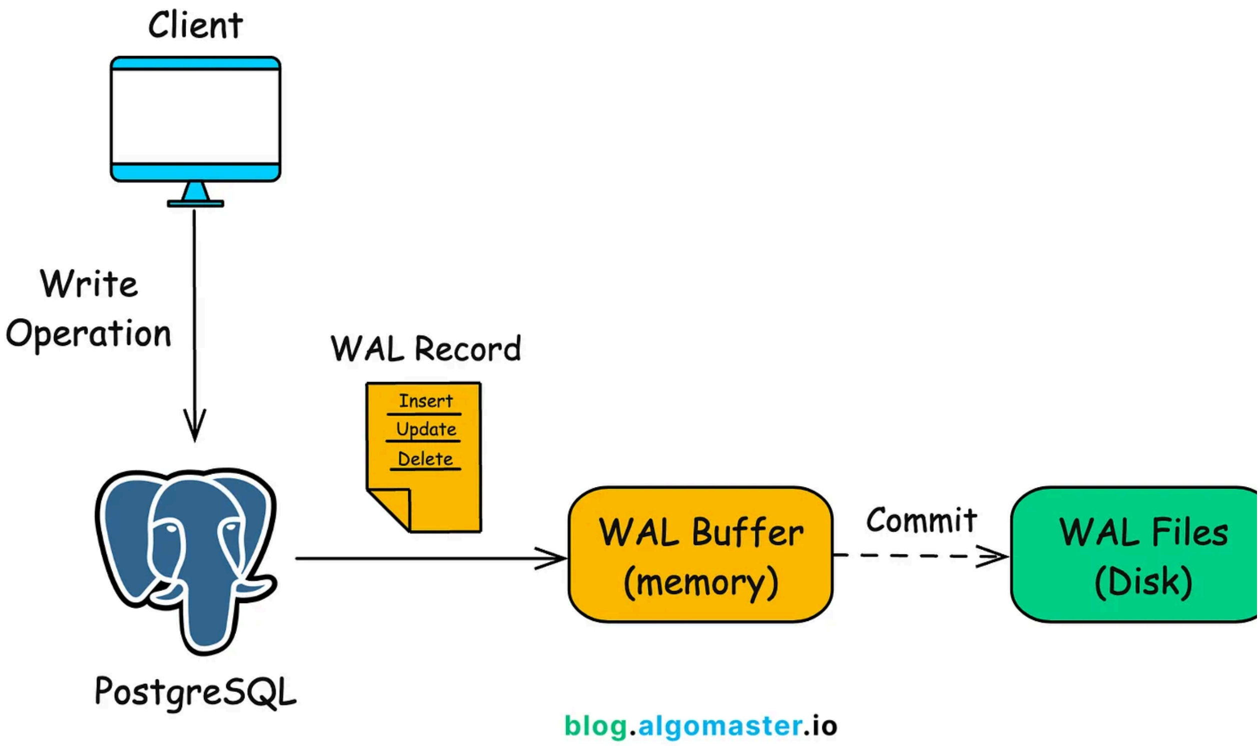
2. Write Ahead Logging (WAL)

Write-Ahead Logging (WAL) is one of PostgreSQL's most critical mechanisms f ensuring **data durability, consistency, crash recovery, and replication**.

What Is WAL?

WAL is a **sequential log of all database changes**, written *before* those changes ar applied to the actual data files (the heap or index pages). This ensures that, in the of a crash or power failure, PostgreSQL can **replay the log** and restore the databa a **safe and consistent state**.

How WAL Works?



- Client executes a write operation (INSERT, UPDATE, DELETE, DDL).
- PostgreSQL generates a WAL record describing the change.
- The WAL record is written to memory (WAL buffer).
- On commit, the WAL is flushed to disk (fsync)—this makes the transaction durable.
- The actual data files (heap pages) may be updated later, asynchronously by background processes (like the checkpointer or background writer).
- In case of a crash, PostgreSQL replays the WAL from the last checkpoint to recover the lost changes.

Key Benefits of WAL

- **Crash Recovery**: WAL replay brings the database to a consistent state after a crash. This ensures that no committed data is lost, even after a system failur
- **Replication**: Enables both synchronous and asynchronous replication for dis recovery and read scaling. A replica essentially replays WAL just like crash recovery, but continuously, to stay in sync with the primary.

- **Point-in-Time Recovery (PITR)**: By archiving WAL, you can restore a backu replay WAL to a specific point in time, essentially "time-traveling" the datal a desired state.

Managing WAL: Best Practices

If unmanaged, WAL can consume significant disk space. Here are ways to handl

- **Enable WAL archiving**

```
archive_mode = on
archive_command = 'cp %p /mnt/wal_archive/%f'
```

- **Use `pg_archivecleanup` or retention policies** to delete old WAL files.
- Set appropriate **checkpoint intervals** (e.g., `checkpoint_timeout`, `max_wal_` to balance recovery time vs. runtime I/O.
- **Use [replication slots]** to avoid prematurely removing WAL needed by standb logical consumers.

3. Multi Version Concurrency Control (MVCC)

PostgreSQL uses **MVCC** to handle simultaneous transactions without requiring locking. MVCC is a powerful mechanism that allows **reads and writes to occur concurrently** by maintaining multiple versions of a row.

This ensures that every transaction sees a **consistent snapshot** of the database as existed at the time the transaction began—regardless of ongoing changes by oth users.

Why MVCC?

Traditional databases often use locks to prevent conflicts between readers and w However, this leads to blocking and contention:

- Writers lock rows to prevent readers from seeing half-written data
- Readers may block writers while reading data

PostgreSQL avoids this by using MVCC to isolate transactions without locking–resulting in **high concurrency, better performance, and smoother scalability**.

How MVCC Works?

When a transaction starts, PostgreSQL assigns it a unique **transaction ID** (**XID**).

PostgreSQL tracks versions of rows using hidden system columns:

- `xmin`: The transaction ID that inserted the row
- `xmax`: The transaction ID that deleted (or updated) the row

Let's say we have a table called `accounts`:

```
id | balance
---+---------
1  | 1000
```

The current row has:

- `xmin = 100` → inserted by transaction ID 100
- `xmax = NULL` → it hasn't been deleted/updated yet

What Happens During Reads?

When a transaction reads a row, it sees the version that was current at the start c transaction:

- It checks `xmin` and `xmax` to determine if the row **was visible** at the time the transaction started.
- If another transaction later updates the row, it **creates a new version** with a `xmin`, leaving the original intact for other readers.

Transaction 1 (T1)

```
BEGIN;  -- Transaction ID = 200
SELECT * FROM accounts WHERE id = 1;
```

- T1 sees the row with `balance = 1000`
- Since `xmin = 100` and `xmax = NULL`, the row is visible to T1
- T1 keeps using this **snapshot** of the database until it commits, even if the da updated later

What Happens During Writes?

Writers create new versions of rows without blocking readers, ensuring high concurrency.

PostgreSQL **never overwrites rows**. Instead:

- **INSERT**: Adds a new row with the current transaction's XID in `xmin`
- **UPDATE**: Marks the old row's `xmax` and inserts a new row with a fresh `xmir`
- **DELETE**: Only sets `xmax`; the row remains until cleanup

This design ensures **snapshot isolation**—older transactions can still see the old versions of rows, even after they're updated. Readers **never block writers**, and wı **don't block readers**. Each sees the world as it existed at their transaction start.

Transaction 2 (T2) – A concurrent update

```
BEGIN;  -- Transaction ID = 201
UPDATE accounts SET balance = 1500 WHERE id = 1;
```

Here's what happens internally:

- The **old row** is updated:

- Its `xmax = 201` → marking it as deleted by T2

- A **new row** is inserted:
  - With `balance = 1500`
  - Its `xmin = 201` and `xmax = NULL`

So now there are **two versions** of the row:

```
Old version: balance = 1000, xmin = 100, xmax = 201
New version: balance = 1500, xmin = 201, xmax = NULL
```

What Each Transaction Sees

- **T1** is still active and keeps using its snapshot from the beginning:
  - It sees the old row: `balance = 1000`
  - It ignores the new row, because its `xmin = 201` (which is after T1 starte
- **T2**, meanwhile, sees:
  - The new row it just inserted: `balance = 1500`

Once both transactions are done, PostgreSQL will still keep both row versions u the old one is no longer needed.

Cleanup: The Role of VACUUM

PostgreSQL stores all row versions in the table (heap) until it's safe to remove the But old versions **don't disappear automatically**.

That's where **VACUUM** process comes in:

- Removes old row versions that are no longer visible to any transaction
- Updates the [Visibility Map] so index-only scans can skip dead pages
- Prevents **XID wraparound** by freezing old transaction IDs

PostgreSQL runs **autovacuum** in the background by default, but tuning its frequ is critical in high-write systems.

4. Query Execution Pipeline

When you run a query in PostgreSQL—whether it's a simple `SELECT * FROM us` or a complex join across multiple tables, it goes through a **well-defined five-stag pipeline**.

Each stage transforms the query from raw SQL into actual database operations t return results or modify data.

The Five Stages of Query Execution

1. **Parsing**
    - PostgreSQL takes the raw SQL string and checks it for **syntax errors**.
    - It then converts the query into a **parse tree** — a structured, internal representation of what the query is trying to do.
2. **Rewrite System**
    - Think of this step as preprocessing or query transformation—reshaping original request before planning.
    - PostgreSQL applies **rules and view expansions** to the parse tree.
    - For example, if the query targets a **view**, the system **rewrites the view reference** into its underlying query.
3. **Planner/Optimizer**
    - The planner generates **multiple candidate execution plans** for the query
    - It estimates the **cost** of each plan using table statistics and selects the **cheapest** one.
    - Cost is measured in terms of CPU, I/O, and memory usage—not time dir but an abstract "execution cost" unit.

4. **Execution**
    - PostgreSQL **executes the chosen plan** step by step.
    - Execution is **Volcano-style**, meaning each node requests rows from its cl processes them, and passes results upward.
5. **Result Delivery**
    - Once the executor produces result tuples, they are **returned to the client** SELECT), or **used to apply changes** (for INSERT/UPDATE).

Key Features in PostgreSQL's Execution Engine

- **Parallel Query Execution**: For large datasets, queries can be split across mul CPU cores.
- **JIT Compilation**: Complex queries can be compiled at runtime for faster execution.

Use the `EXPLAIN` command to analyze query plans and identify performance bottlenecks. For example:

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE customer_id = 123 AND order_date > now() - inter
'30 days';
```

5. Indexing System

Indexes are essential to database performance. PostgreSQL offers a variety of inc types to optimize query performance for different data types:

1. **B-tree** (**Default Index Type**)
    - **Best for:** Equality and range queries (=, <, <=, >, >=)
    - **Data types:** Numbers, strings, dates—anything with a natural sort order

```
CREATE INDEX idx_price ON products(price);
```

2. **Hash Index**
   - **Best for:** Simple equality comparisons (= only)
   - Slightly faster than B-tree for some equality lookups, but limited in capabilities.

```
CREATE INDEX idx_hash_email ON users USING HASH(email);
```

3. **GIN** (**Generalized Inverted Index**)
   - **Best for:** Documents or composite values (arrays, JSON, full-text search) you need to check if a document contains a value or perform membershi checks.
   - **Use case:** Indexes every element inside a value (like words in a text field keys in a JSON)

```
-- For full-text search
CREATE INDEX idx_gin_content ON articles USING
GIN(to_tsvector('english', content));

-- For JSONB
CREATE INDEX idx_jsonb_data ON items USING GIN(data jsonb_path_ops);

-- For array values
CREATE INDEX idx_tags_gin ON posts USING GIN(tags);
```

4. **GiST** (**Generalized Search Tree**)
   - **Best for:** Complex, non-scalar data types (geospatial, ranges, fuzzy text s
   - **Use case:** Stores **bounding boxes** or intervals and supports overlaps, proximity, containment. Underpins the PostGIS extension, range querie more.

```
-- For geospatial data (via PostGIS)
CREATE INDEX idx_location_gist ON places USING GiST(geom);

-- For range types
CREATE INDEX idx_price_range ON items USING GiST(price_range);
```

5. **SP-GiST** (**Space-Partitioned GiST**)
    - **Best for:** Data with natural space partitioning (e.g., tries, quadtrees)
    - **Use case:** Prefix searches, IP subnet matching, k-d trees

```
-- For text prefix matching
CREATE INDEX idx_prefix ON entries USING SPGIST(title);
```

6. **BRIN** (**Block Range Index**)
    - **Best for:** Huge, append-only tables where data is naturally sorted (e.g., ti series)

```
CREATE INDEX idx_log_time_brin ON logs USING BRIN(log_timestamp);
```

6. Table Partitioning

Table partitioning allows large tables to be divided into smaller, more manageal pieces based on range, list, or hash criteria. This improves query performance by enabling **partition pruning**, where only relevant partitions are scanned.

PostgreSQL supports **declarative partitioning**, which means you define partitior using standard SQL with the `PARTITION BY` clause.

Example SQL

To create a partitioned table by date range:

```
-- Create the partitioned parent table
CREATE TABLE measurements (
  city_id    INT NOT NULL,
  logdate    DATE NOT NULL,
  peaktemp   INT,
  unitsales  INT
) PARTITION BY RANGE (logdate);

-- Create child partitions for each year
CREATE TABLE measurements_2024 PARTITION OF measurements
  FOR VALUES FROM ('2024-01-01') TO ('2024-12-31');

CREATE TABLE measurements_2025 PARTITION OF measurements
  FOR VALUES FROM ('2025-01-01') TO ('2025-12-31');
```

When you insert into `measurements`, PostgreSQL automatically routes the row t correct partition based on the `logdate` value.

Partitioning Strategies

- **Range:** Splits data by ranges of values (e.g., dates, numbers).
- **List:** Divides data by discrete values (e.g., region codes {"APAC", "EMEA", ...}
- **Hash:** Distributes rows evenly using a hash function (when range/list isn't practical or balanced).

> Partitioning reduces query times by avoiding full table scans. Ideal for time-se data or datasets that can be logically grouped (e.g., by region or category).

7. Logical Decoding

Logical decoding is the process of transforming PostgreSQL's **low-level WAL re** into **high-level change events** like:

```
INSERT INTO customers (id, name) VALUES (1, 'Alice');
```

```
UPDATE orders SET status = 'shipped' WHERE id = 42;

DELETE FROM payments WHERE id = 7;
```

It allows changes from the WAL to be streamed in a logical format, making it us
for replication and **Change Data Capture** (CDC).

How Logical Decoding Works?

1. PostgreSQL WAL contains all changes, but in a binary, low-level format
2. Logical decoding interprets WAL into high-level row changes
3. The output is emitted using an **output plugin**, such as:
   - `pgoutput` (used for PostgreSQL logical replication)
   - `wal2json` (outputs changes as JSON)
   - `decoderbufs` (outputs Protocol Buffers)
   - `test_decoding` (for debugging/logging)
4. A **replication slot** is created to:
   - Track how much of the WAL has been consumed
   - Prevent PostgreSQL from deleting WAL segments that are still needed b consumer

Use Case

- **Change Data Capture (CDC):** Stream inserts/updates/deletes to message bro (e.g., Kafka, RabbitMQ)
- **Real-time analytics:** Sync changes into a data warehouse (like BigQuery or Snowflake)
- **Event-driven systems:** Trigger downstream services on data changes
- **Cross-database syncing:** Sync tables across PostgreSQL instances (or even t MongoDB/MySQL with external tools)

> **Note:** Logical decoding requires extensions like wal2json for specific output formats, as it's not built-in by default.

8. Extensions

One of PostgreSQL's greatest strengths is its **extensibility**.

From its inception, PostgreSQL was designed to be **modular and pluggable**, enal users to extend its functionality without modifying core source code.

This has turned PostgreSQL from a traditional relational database into a true **da platform**—capable of powering everything from analytics to full-text search, ma learning, and distributed systems.

What Are Extensions?

An **extension** in PostgreSQL is a package of additional functionality that can inc

- SQL objects (functions, types, tables, operators)
- Procedural language support
- Native C code for performance
- Background workers or hooks into the core engine

Once installed, extensions behave like built-in features, seamlessly integrated in database engine.

Installing an Extension

Extensions can be created by the community or bundled with PostgreSQL. To in one:

```
CREATE EXTENSION pgcrypto;
```

Extensions live in the `share/extension/` directory and are managed through `CF` `EXTENSION`, `ALTER EXTENSION`, and `DROP EXTENSION`.

Popular Extensions

- **pgcrypto**: Adds **cryptographic functions** for hashing, encryption, and passw handling
- **pgvector**: Enables vector **similarity search**, useful for machine learning and applications
- **PostGIS**: Turns PostgreSQL into a **geospatial database**, supporting geometr queries, spatial indexing
- **Citus**: Enables **distributed PostgreSQL**, allowing you to shard and scale out multiple nodes

> This flexibility makes PostgreSQL adaptable to a wide range of applications, f traditional relational databases to specialized data processing systems.

9. Statistics Collector

PostgreSQL's **statistics collector** gathers real-time data on database activity, hel you monitor and optimize performance.

There are two main types of statistics PostgreSQL collects:

1. **Cumulative Activity Statistics**: Used for *monitoring*, *autovacuum*, and *operati insights*
2. **Planner Statistics**: Used by the *query planner* to optimize execution plans

Cumulative Statistics System (pg_stat views)

These statistics are stored in *pg_stat_ views** and reflect what's happened since th server started or since stats were last reset.

Key views include:

- **pg_stat_activity**: Shows live queries and their status.
- **pg_stat_all_tables:** Shows table-level read/write/VACUUM stats.
- **pg_stat_all_indexes:** Tracks index usage (helps detect unused indexes).
- **pg_stat_statements**: Tracks execution metrics for SQL statements, helping identify slow or resource-intensive queries.

Example: Identify Top Slow Queries

Using **pg_stat_statements**:

```
SELECT query, calls, total_exec_time, mean_exec_time
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 5;
```

Planner Statistics (ANALYZE statistics)

This refers to the data collected by the `ANALYZE` command (automatically run by autovacuum or manually by DBAs) about the contents of tables.

These stats are stored in the `pg_statistic` system catalog and are critical for qu planning.

What PostgreSQL collects per column:

- Number of **distinct values**
- Percentage of **NULLs**
- List of **most common values** (**MCVs**) and their frequencies
- A **histogram** of value distribution

With **multi-column statistics** (using `CREATE STATISTICS`), PostgreSQL can also

- **Correlations** between columns
- **Functional dependencies** (e.g., if `state → zip_code`)

- **NDistinct** estimates for combinations of columns

PostgreSQL's query planner uses these stats to:

- Estimate **selectivity** (how many rows a WHERE clause will match)
- Choose between **index scan**, **seq scan**, **hash join**, or **merge join**
- Allocate memory for **hash tables** or **sorts**

Example:

If `ANALYZE` finds that `status = 'shipped'` occurs in 90% of rows, the planner r
avoid using an index because the query isn't selective.

Conclusion

PostgreSQL's robust architecture and feature set make it a powerful choice for developers. From its process-based model and MVCC for concurrency to advanc features like logical decoding and extensions, it offers the flexibility and reliabili needed for modern applications.

To take your understanding further, explore:

- [*PostgreSQL: High Performance* by Greg Smith]
- The [official PostgreSQL documentation]
- Community spaces like [Stack Overflow], [PostgreSQL mailing lists], and [r/PostgreSQL]

By mastering these concepts, you'll be well-equipped to design, optimize, and troubleshoot PostgreSQL databases, ensuring your applications run efficiently a reliably.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such co
every week.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a **paid subscriber**.

As a paid subscriber, you'll receive an **exclusive deep dive** every Thursday, access
[structured system design resource], and other [premium perks].

There are [group discounts], [gift options], and [referral bonuses] available.

A guest post by

Alexandre Zajac

SDE & AI @Amazon | Building Hungry Minds to 1M+ | Writing about Software Engineering, System Design, and AI

Discussion about this post

Write a comment...

Dep 2d

Liked by Alexandre Zajac, Ashish Pratap Singh

It's amazing and resourceful

LIKE (2) REPLY

1 reply

Varun 2d

Liked by Alexandre Zajac, Ashish Pratap Singh

Wow, very helpful for a developer to know the internals

LIKE (2) REPLY

1 reply

4 more comments...