PostgreSQL in a nutshell

ALL THE THINGS MATTER(?)

Disclaimer

- I am not a PostgreSQL specialist.
- This presentation should not be taken as an absolute source of truth.
- Everything here is a compilation of different knowledge sources.
- Last but not least: **we're only scratching the surface!**

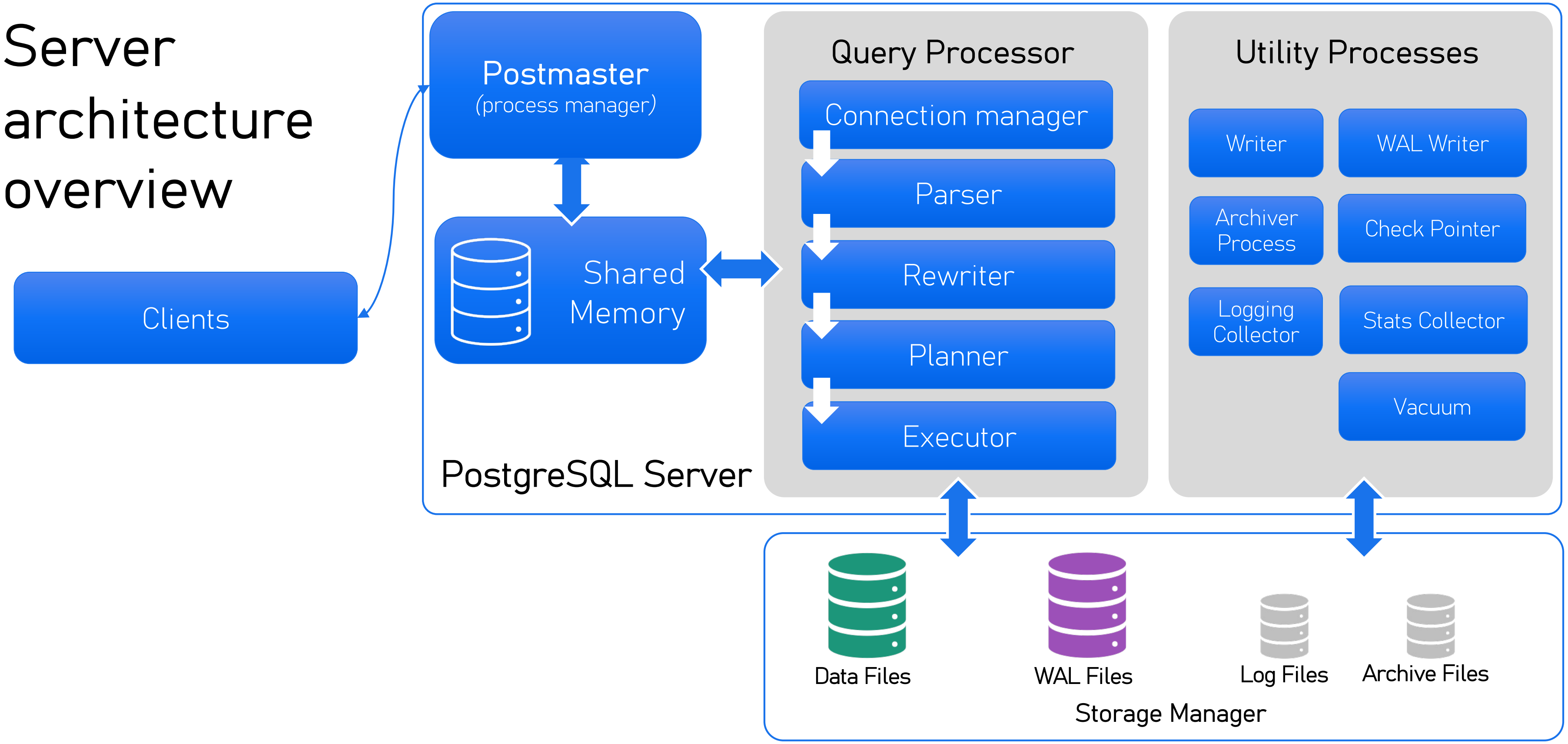
What matters?

- PostgreSQL Architecture
  - Why the server side matters
  - Server architecture overview
  - Why statistics matter
  - Hands-on
    - Playground
    - Statistics & ANALYZE
    - VACUUM vs VACUUM FULL
    - EXPLAIN
- Indexing
  - Concept
  - Index types
  - Hands-on
    - B-tree
    - GIN
- JSON Support
  - History & motivation
  - Types
    - JSON
    - JSONB
  - JSONPATH
  - TOAST
  - Hands-on
    - JSONB

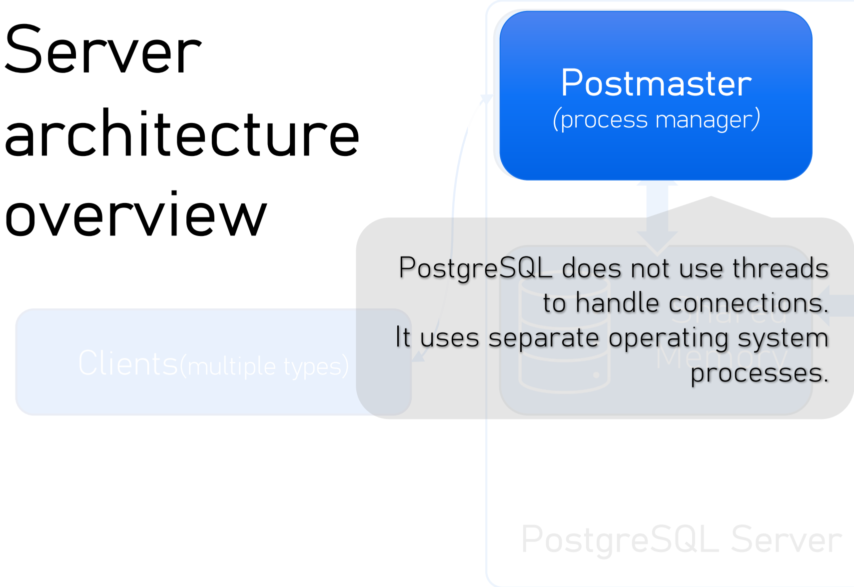
Why the server side matters

- Do you remember the last time you installed a PostgreSQL database and configured it for your application?
- Why does it matter?
- PostgreSQL's default configuration is not suitable for production.
- At the very least, buffer settings, max connections, and logging are things you should review.
- Up-to-date statistics allow the system to make better decisions — such as choosing the correct execution plan.
- Processes such as Autovacuum are crucial to keeping tables updated and providing accurate information to the query planner.

Server architecture overview



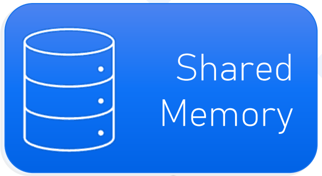
Server architecture overview



- It is the parent of all PostgreSQL processes. When PostgreSQL starts up, the **postmaster** (also known as the process manager) is the first process created.
- **Main responsibilities**
  - Accept new client connections.
  - Spawn new backend processes for each connected client.
  - Supervise processing and restart failed subprocesses.
  - Manage access to Shared Memory.

Hierarchically, it is the highest level — it coordinates everything.

Server architecture overview



Since these processes do not naturally share memory with each other, PostgreSQL needs a common memory area.

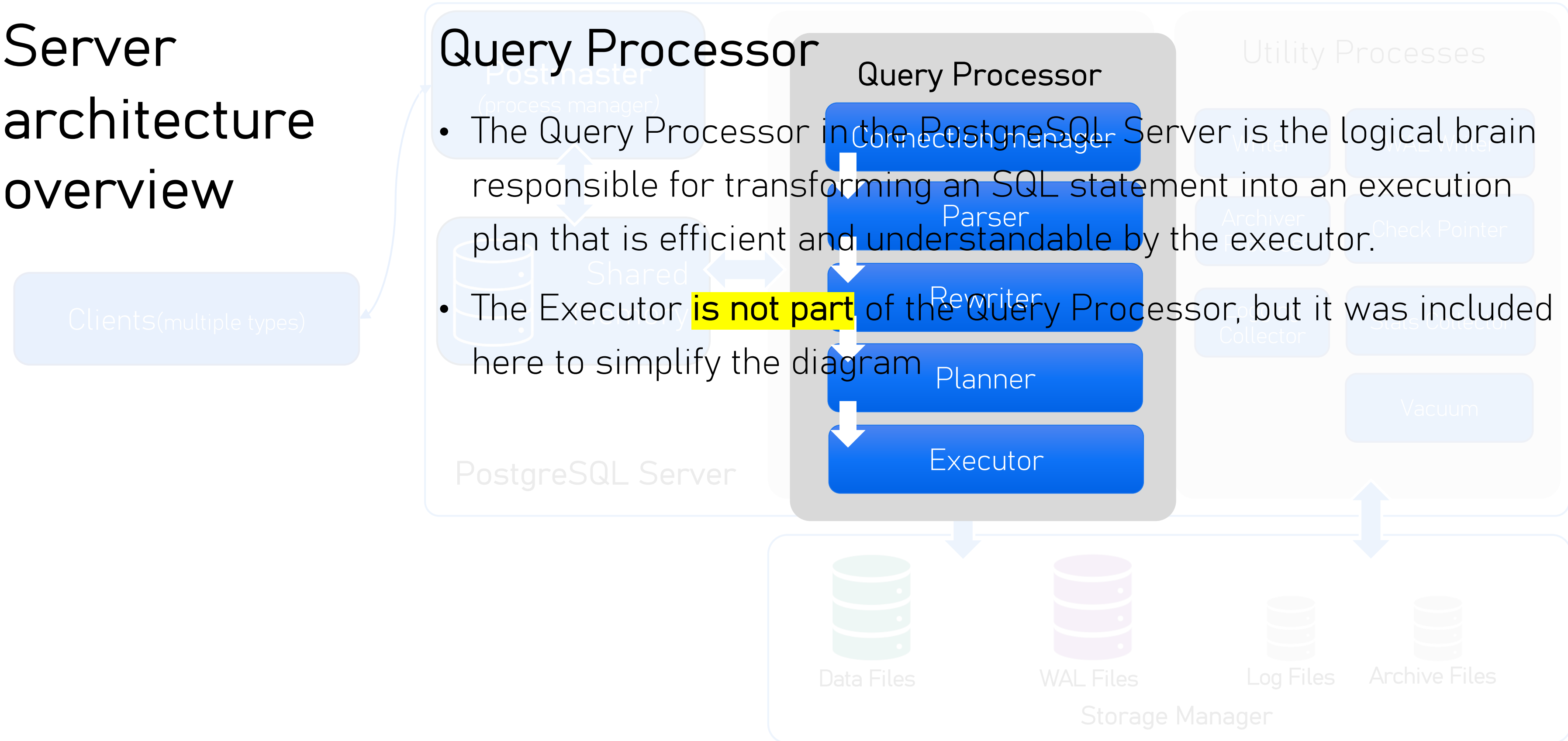
- It is the memory area shared by all PostgreSQL processes..
- Responsibility
- It stores core structures such as:
  - Buffer Cache (frequently accessed data pages)
  - WAL(Write-Ahead Log) Buffers
  - Lock Tables
  - Process info tables
  - Background worker info

It ensures that all processes have a consistent view of the database.

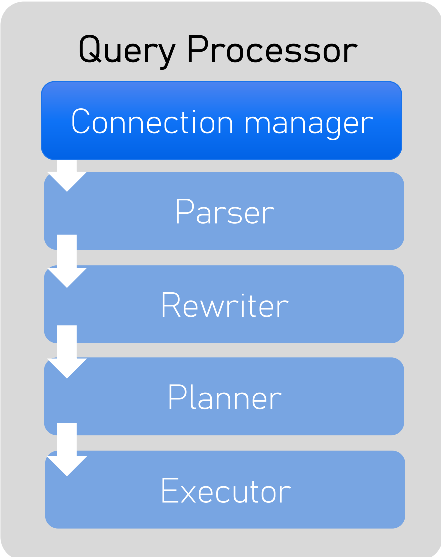
Server architecture overview

Query Processor

- The Query Processor in the PostgreSQL Server is the logical brain responsible for transforming an SQL statement into an execution plan that is efficient and understandable by the executor.
- The Executor **is not part** of the Query Processor, but it was included here to simplify the diagram



Server architecture overview

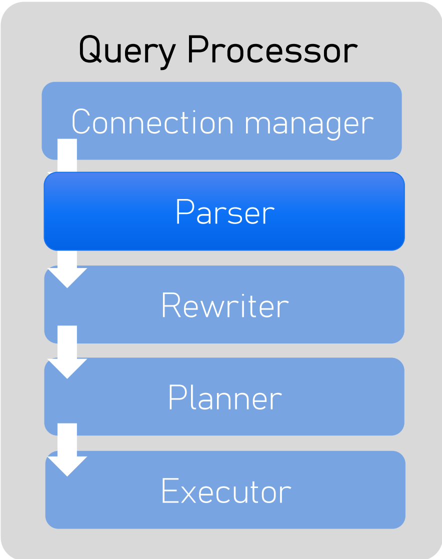


Connection Manager

- Responsible for:
  - Accepting the connection forwarded by the Process Manager.
  - Creating the client's specific backend process.
  - Associating the process's buffers, locks, and memory with the Shared Buffer.

It is the "entry point" where a query begins to exist inside the server.

Server architecture overview



Parsing (Syntactic and Semantic Analysis)

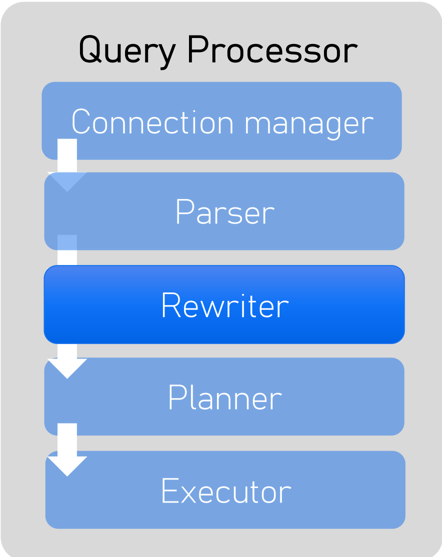
Responsibilities:

- Check whether the SQL is syntactically valid.
- Convert the textual SQL into a parse tree.
- Validate table names, columns, functions, types, etc.
- Resolve namespaces and validate basic permissions.

Components involved:

- Lexer (tokenizes the SQL)
- Parser (generates the parse tree based on PostgreSQL's grammar)
- Analyzer (performs semantic resolution)

Server architecture overview



Rewrite System (Query Rewrite)

Responsibilities:

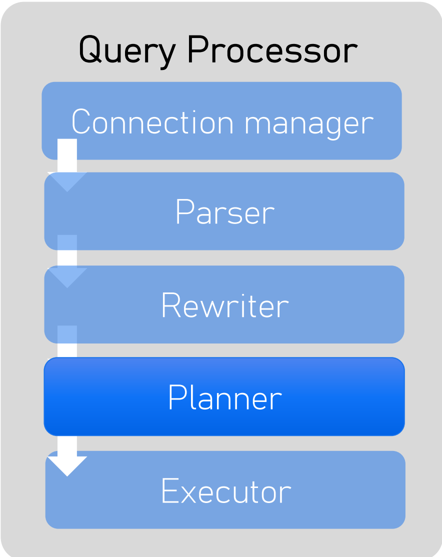
Apply rewrite rules such as:

- Expanding views
- Rewriting rules
- Internal logical transformations

Why does this exist?

- Because a *view*, for example, needs to be replaced by its underlying query before optimization or execution.

Server architecture overview



Optimizer / Planner (Optimization and Planning)

Responsibilities:

- Generate different execution strategies.
- Evaluate costs (based on catalog statistics).
- Choose the best possible execution plan.

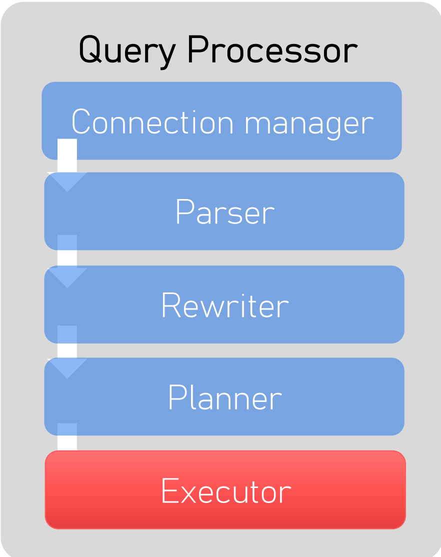
Processes performed:

- Choosing the join type (nested loop, hash join, merge join)
- Choosing indexes
- Deciding the join order
- Deciding table scan methods (seq scan, index scan, bitmap scan)
- Applying filter and projection pushdown
- Estimating cardinality.

Result:

- A plan tree: a tree detailing how each step will be executed.

Server architecture overview



Executor (Query Executor)

It is responsible for physically executing the execution plan generated by the Query Planner/Optimizer.

The execution plan is a tree of nodes (seq scan, index scan, hash join, sort, aggregate, limit, etc.).

Responsibilities:

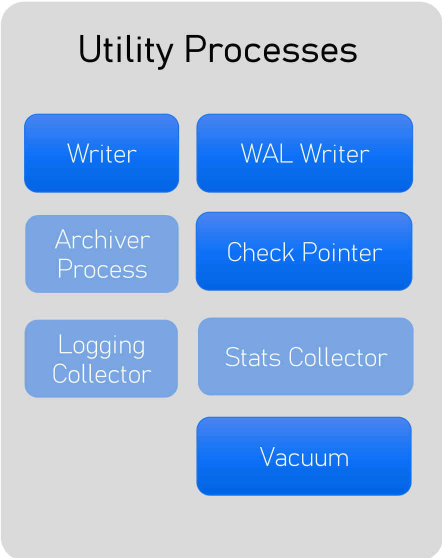
- Execute each node of the plan
- Manage the flow of tuples
- Apply filter conditions
- Manage buffers and interact with the Storage Manager
- Enforce MVCC rules
- Trigger firing
- Produce results for the client

Server architecture overview

Utility Processes

- The Utility Processes are a set of auxiliary p the overall functioning of the system.
- They are part of the general architecture of t within the PostgreSQL Server and are essen maintenance, reliability, cleanup, and interna for the database.
- They work continuously (or on demand) to ke stable, efficient, and safe.

Server architecture overview



Checkpointer Process

Ensures periodic persistence of data.

- Writes "dirty" pages from shared buffers to data files.
- Reduces recovery time after a crash.
- Operates based on checkpoint_timeout, among other parameters.

Essential role:
Minimize recovery effort and ensure on-disk consistency.

Background Writer

Gradually writes modified pages to disk, avoiding write bursts during checkpoints.

- Smooths out I/O over time.

Essential role:
Keep the buffer pool healthy and reduce latency for backends.

WAL Writer

Responsible for writing data to the Write-Ahead Log.

- Ensures the core durability principle of ACID.
- Backends do not write to WAL directly — they send buffers to the writer.

Essential role:
Reduce contention and ensure WAL is written efficiently and in order.

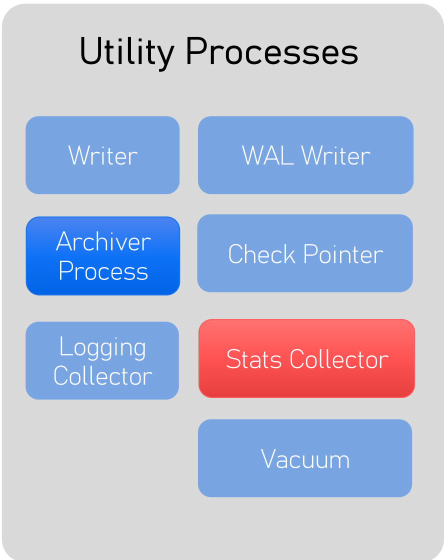
Auto Vacuum Launcher

Orchestrates autovacuum processes.

- Monitors tables, detects bloat, and triggers vacuum operations as needed.

Essential role:
Preserve space, update statistics, and keep MVCC functioning.

Server architecture overview



Archiver Process

Handles WAL archiving for replication or PITR backups.

- Moves completed WAL files to the location defined in archive_command.

Essential role:
Ensure extended durability and support for PITR.

Stats Collector

Collects and maintains usage statistics, access patterns, and read behavior.

- Feeds the Query Planner with statistical information.

Essential role:
Query optimization through up-to-date statistics.

Why statistics matters

PostgreSQL statistics are one of the essential pillars of database performance.

Without good statistics, the Query Planner literally goes blind, and the consequences are:

- Bad plans that can result in unnecessary seq scans.
- Incorrect index usage.
- Extremely expensive nested loops.
- Inefficient joins, massive sorts, memory explosions, and ultimately, timeouts

In short: a nightmare!

Why statistics matters

PostgreSQL uses a **cost-based query planner**.

Which means:

It chooses an execution plan based on cost estimates.

These estimates rely almost entirely on statistics.

So:

Good statistics → accurate estimates → good plans → fast queries

Bad statistics → wrong estimates → bad plans → slow queries

Why statistics matters

Statistics are produced in **two ways**:

Automatically via *Autovacuum*

Autovacuum triggers ANALYZE on tables that have undergone enough changes.

Manually, using the ANALYZE command

For example, when we run the command:

- ANALYZE my_table;
- ANALIZE my_table (column1, column2)

Hands-on

Our playground

- Enough theory — let's set up our environment.
- **What you'll need:**
  - Docker
  - A simple IDE — I'll use VS Code. Sometimes I'll run code directly in it, and sometimes I'll use the terminal to send commands to PostgreSQL.
  - VS Code extensions:
    - https://marketplace.visualstudio.com/items?itemName=ms-ossdata.vscode-pgsql
    - https://marketplace.visualstudio.com/items?itemName=inferrinizzard.prettier-sql-vscode
- To make things easier, clone the presentation repository and follow the instructions:
  https://github.com/isaacvitor/postgresql-nutshell

Hands-on

Statistics, ANALYZE, VACUUM and VACUUM FULL

- New statistics are generated when Autovacuum runs the Vacuum process together with ANALYZE.
- Autovacuum is an automated process that uses Vacuum. It runs automatically when certain thresholds are reached.
- Autovacuum settings can be changed in the ***postgresql.conf*** file.
- What is the Vacuum?
  - In summary, Vacuum is the process responsible for cleaning up dead tuples.
- There are two versions of the Vacuum process:
  - VACUUM – Marks spaces as being available for reuse. This process does not lock the table.
  - VACUUM FULL - Removes the deleted or updated records and reorders the table data. Note: VACUUM FULL requires an exclusive lock on the table.

Hands-on

Statistics, ANALYZE, VACUUM and VACUUM FULL

Let’s see them in action!

- Statistics and ANALYZE
- VACUUM
- VACUUM FULL

Hands-on

EXPLAIN command

What is the EXPLAIN command?

- The EXPLAIN command displays the **execution plan** — the strategy PostgreSQL will use to execute a query. It does **not** execute the query itself (unless you use EXPLAIN ANALYZE)
- In summary, EXPLAIN shows us:
  - which algorithms will be used (Seq Scan, Index Scan, Bitmap Scan, Hash Join, Merge Join, etc.);
  - the order in which operations will occur;
  - cost estimates, number of rows, and row width;
  - use of parallelism;
  - filtering, join conditions, rechecks;
  - which indexes were chosen;
  - how the hierarchical execution plan is structured.

Hands-on

EXPLAIN command

Basic syntax

- EXPLAIN [ANALYZE] [VERBOSE] [COSTS] [SETTINGS] [BUFFERS] [WAL] [TIMING] [SUMMARY] [FORMAT format] query

Main parameters:

- EXPLAIN query — Basic plan
- EXPLAIN ANALYZE query — Executes the query and shows actual times
- EXPLAIN VERBOSE query — Detailed information
- EXPLAIN (ANALYZE, BUFFERS) — Includes buffer usage
- EXPLAIN (ANALYZE, VERBOSE, BUFFERS) — Complete information
- EXPLAIN (ANALYZE true, TIMING false) — Disables timing for faster queries

Output formats:

- EXPLAIN (FORMAT TEXT) — Default text format
- EXPLAIN (FORMAT JSON) — JSON
- EXPLAIN (FORMAT XML) — XML
- EXPLAIN (FORMAT YAML) — YAML

Useful combinations:

EXPLAIN (ANALYZE true, BUFFERS true, TIMING true, COSTS true) query;

Let's see it in action!

Hands-on

EXPLAIN command

Running the command:

EXPLAIN SELECT * FROM explain_test WHERE id = 500000;

We'll get something like this:

```
 Gather  (cost=1000.00..11614.43 rows=1 width=14)
   Workers Planned: 2
   ->  Parallel Seq Scan on explain_test  (cost=0.00..10614.33 rows=1 width=14)
         Filter: (id = 500000)
(4 rows)
```

What the result means:

Gather – *Coordinator node that collects results from parallel workers*

cost*=1000.00..11614.43 – (startup cost .. total cost)*

rows*=1 – Estimated to return 1 row*

width*=14 – 14 bytes per row on average*

Workers Planned*: 2 – PostgreSQL will use 2 parallel worker processes*

Parallel Seq Scan – *Parallel sequential scan on the table*

cost*=0.00..10614.33 – Cost of the parallel scan*

Filter*: (id = 500000) – Condition applied during the scan*

(4 rows) – *Quantity of rows in the output*

Seq Scan? What?

Indexing

Concept

There's no substitute for a missing index. To achieve **good performance**, you need to have **proper indexing**.

To understand this statement, we must grasp the underlying concepts:

An index is an auxiliary **data structure** that **speeds up** the lookup of rows in tables, avoiding full table scans(sequential scan).
It works as a "shortcut" to find specific values without reading all the table's blocks.

Indexing

Concept

Why do indexes matter in PostgreSQL?

PostgreSQL stores tables as heap files by default, with no physical ordering.

Without indexes, any:

- filter (WHERE)
- lookup for a specific value
- key-based JOIN
- ORDER BY or GROUP BY

...will likely result in a Sequential Scan

Indexing

Concept

Indexes are not free!

An index is maintained in its own data structure, independent from the base table.

How do you choose the right index?

Operators(=, LIKE, >,<, !=, @>, <@, etc)

Data type (TEXT, JSONB, Array)

Scenario (Combination of multiple data types)

Indexing

Index types

PostgreSQL provides 6 index types:

B-tree, Hash, GiST, SP-GiST, GIN, and BRIN

The most commonly used types are:

- **B-tree** – default index type
- **GIN** – great for arrays, JSONB, full-text-like structures
- **GiST** – flexible, extensible, geometric, ranges
- **BRIN** – ideal for very large, naturally ordered tables

Hands-on

B-tree (Balanced multi-way search tree)

How to create it:

```
CREATE INDEX idx_name
ON table_name (column_name);
```

Best for:

- Equality lookups
- Range queries
- Sorting (ORDER BY), grouping
- Most typical queries
- Foreign keys

How it works:

- Balanced tree structure
- Keeps keys in sorted order
- Supports O(log n) lookups

Common operators supported:

- =
- <, <=, >, >=
- BETWEEN
- IS NULL, IS NOT NULL
- Pattern matching with left-anchored LIKE:
  - LIKE 'abc%'
  - ILIKE 'abc%' (with proper operator class)
- Supports ordering operators for ORDER BY

Hands-on

GIN (Generalized Inverted Index)

How to create it:

```
CREATE INDEX idx_name

ON table_name

USING gin (column_name);
```

Best for:

- Array fields
- JSONB data
- Full-text search (tsvector)
- Rows with multiple components (composite values)

Operators for arrays:

- && — overlap
- @> — array contains
- <@ — array is contained by
- = — equality

Operators for JSONB:

- ? — key exists
- ?| — any of these keys exist
- ?& — all keys exist
- @> — contains
- <@ — contained by

Operators for full-text (tsvector):

- @@ — match
- @@@ — phrase search (extensions)

JSON Support

History & motivation

2012

(PostgreSQL 9.2): Introduction of **JSON type**

- Textual JSON storage with validation.
- Motivation: Accept JSON/API payloads without losing structural integrity.

2014

(PostgreSQL 9.4): Introduction of **JSONB type**, the binary JSON

- Binary representation → faster indexing, queries, and manipulation.
- Made PostgreSQL competitive with document-oriented databases.

2019

(PostgreSQL 12): Introduction of **SQL/JSON Path**

- Full support for SQL/JSON Path (ANSI SQL:2016).
- Introduces jsonpath type and jsonb_path_* functions.

JSON Support

JSON Type

What it is:

textual (string) storage with JSON validity checks on insert.

Properties:

Preserves whitespace and exact key order as inserted

No efficient native indexing on content

Use cases:

When you must preserve the exact JSON text (formatting)

When you will never query based on JSON content

Limitations:

Expensive queries and extraction (re-parsed on every access)

No GIN support for inner-structure queries

JSON Support

JSONB type

What it is:

binary, normalized representation of JSON (key order not preserved, no whitespace). Designed for search/querying. Introduced in 9.4.

Pros:

Indexable with GIN/GIST (queries using @>, ?, ?|, ?&, etc.)

Much faster search and extraction operations compared to json

Supports efficient functions/operators (->, ->>, #>, #>>, jsonb_set, jsonb_insert, etc.)

Cons:

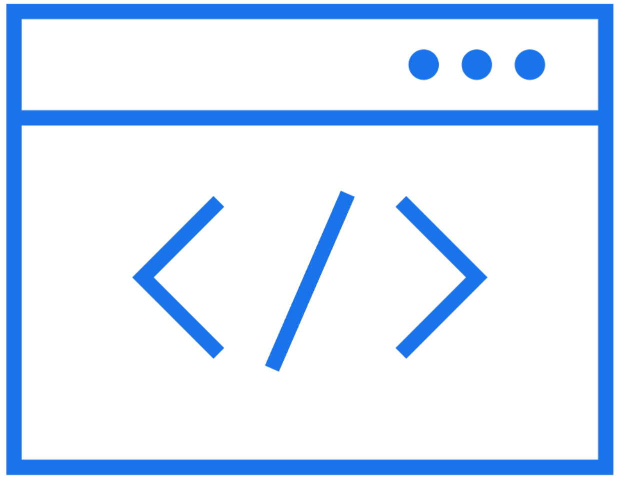
Updates that change the whole document are costly (full rewrite in most cases)

It is TOASTable — large documents may incur I/O overhead and degraded key-search performance when TOASTed

JSON Support

JSONPATH

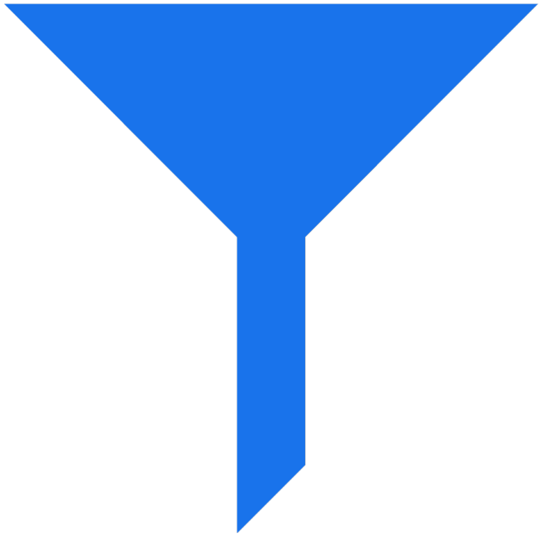
- What it is:
  - a type for storing SQL/JSON Path expressions; introduced together with SQL/JSON Path support (Postgres 12).
    Provides a standards-based JSONPath querying language. Useful with @?, @@, and jsonb_path_query* functions.



JSON Support

TOAST

- PostgreSQL stores data in **8 KB pages**.
  When a value (such as JSONB, text, bytea, arrays) becomes **too large to fit comfortably inside the page**, PostgreSQL activates **TOAST** — *The Oversized-Attribute Storage Technique*.
- Large JSONB objects are one of the most common triggers for TOAST



JSON Support

TOAST

What TOAST does:

- When a column exceeds roughly 2 KB, PostgreSQL may:
  - Compress the value
  - Store it externally in a TOAST table
  - Keep only a lightweight pointer in the main row

Why PostgreSQL uses TOAST:

- Prevent oversized rows from splitting across multiple pages
- Keep hot pages small (better caching)
- Avoid slow sequential reads due to bloated tuples
- Reduce excessive WAL generation during updates

JSON Support

TOAST

- Important consequences
  - Even if a query needs **only one field** inside a large JSONB document, PostgreSQL may need to **fetch and decompress the entire TOASTed value**.
  - Updates like jsonb_set often **rewrite the whole document**, which is expensive.
  - Queries that filter using jsonb -> 'key' may still force a full fetch + decompress of 200 KB
  - A single jsonb_set() on a 500 KB document generates 500 KB of WAL

JSON Support

TOAST

- Popular mistake:
  - Put everything inside a JSONB column
- CREATE TABLE bad_small (pk **serial primary key**, jb jsonb);
- CREATE TABLE good_small (pk **serial primary key**, id int, jb jsonb);

Hands-on

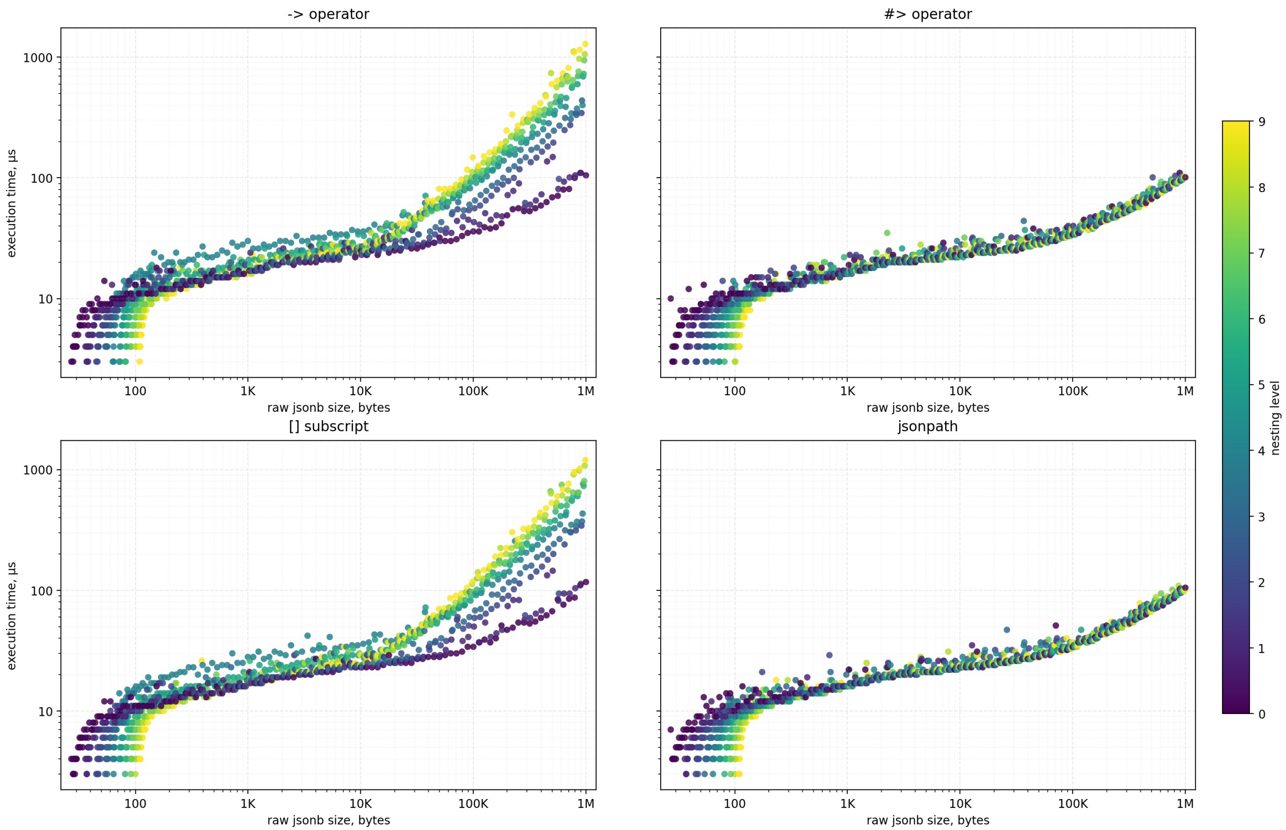
JSONB – Operators performance

- Nested containers performance
- How different JSONB navigation operators behave in terms of performance
  - Sample table with nested objects of various sizes (1 KB – 1 MB) and various nesting levels (0–9), containing one short key and one long key.
- Test expressions (operators being benchmarked)
  - -> (arrow):
    jb -> 'obj' -> 'obj' -> … -> 'obj' -> 'key'
  - #> (path):
    jb #> '{obj,obj,…,obj,key}'
  - [] (subscript):
    jb['obj']['obj']…['obj']['key']
  - JSONPath query:
    jsonb_path_query_first(jb, '$.obj.obj.… .obj.key')

Note: The graphs presented here are an attempt to reproduce the experiments shown at PGConf 2021.

Hands-on

JSONB – Operators performance



Conclusions:

- 1. JSONB access cost increases with document size
    - Extraction operations become significantly slower once JSONs exceed ~50–100 KB, and the effect becomes very pronounced near 1 MB.
- 2. Nesting depth has a direct and measurable impact
    - Deeper JSON structures (levels 6–9) consistently incur higher latency than shallow ones, even with the same overall document size.
- 3. The **-> operator is the most sensitive to size and depth**
    - It shows sharp performance degradation, reaching over 1000 µs for large and deeply nested documents.
- 4. The **#> operator delivers the best overall performance**
    - Its growth curve is smoother and remains stable even for large JSONs.
    - It is the most reliable operator for accessing nested structures efficiently.

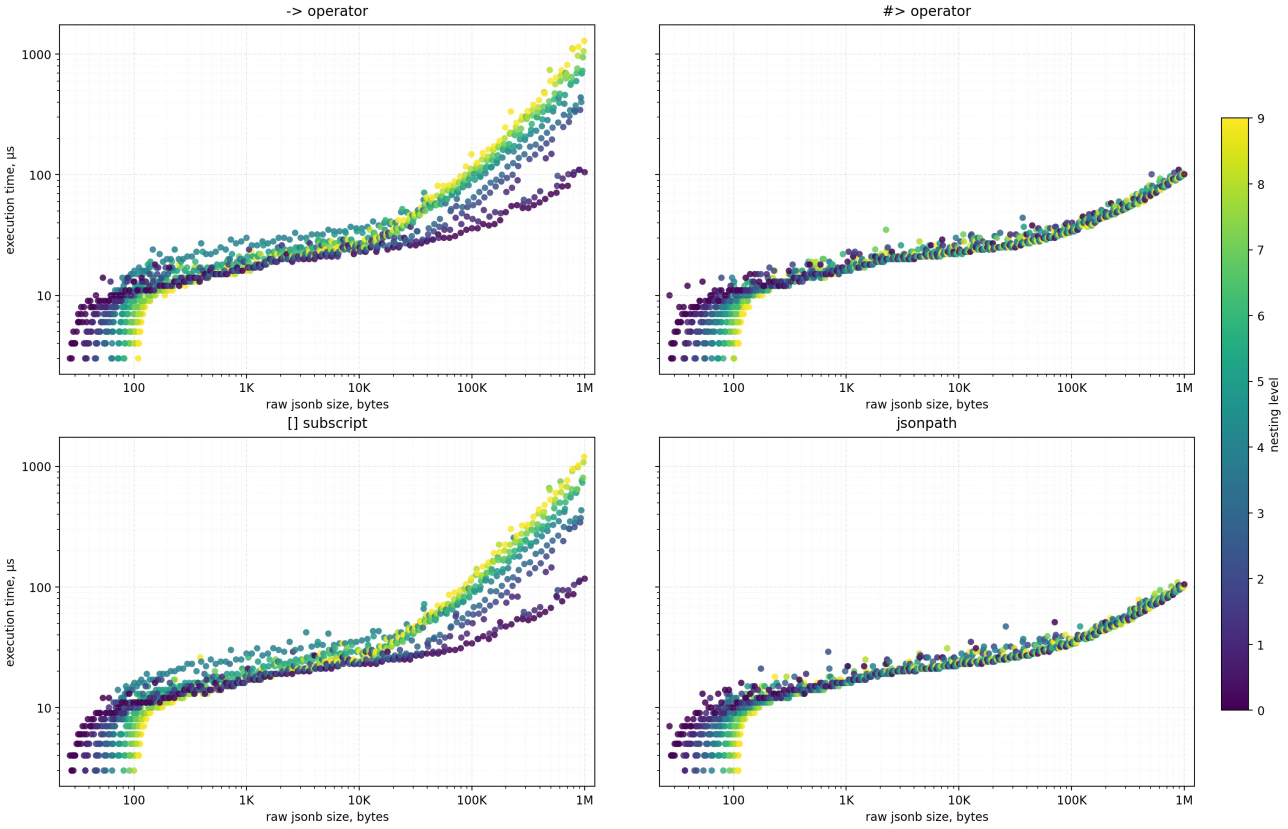
Hands-on

JSONB – Operators performance

Conclusions:

- 5. The **[] subscript** behaves essentially the same as ->
  - Since [] is syntactic sugar for ->, it inherits the same scaling issues and becomes slow on large nested objects.
- 6. **JSONPath** provides flexibility, but at a higher cost
  - . jsonb_path_query_first() is consistently slower than direct operators.
  - It is great for expressive queries but inefficient for simple lookups in large structures.
- 7. Operator choice **matters much more for large documents**
  - For small JSONs the difference between operators is minimal,
  - But in medium and large JSONs the differences grow by orders of magnitude.
- 8. For critical paths, prefer direct and explicit access methods

  Whenever possible:
  - **use #> for nested lookup**;
  - avoid **JSONPath when the path is fixed**;
  - **minimize excessive nesting in JSON design**.



Keep that in mind

ALL OPERATORS HAVE A COMMON OVERHEAD: DETOAST TIME + JSONB ITERATION TIME

That’s all folks