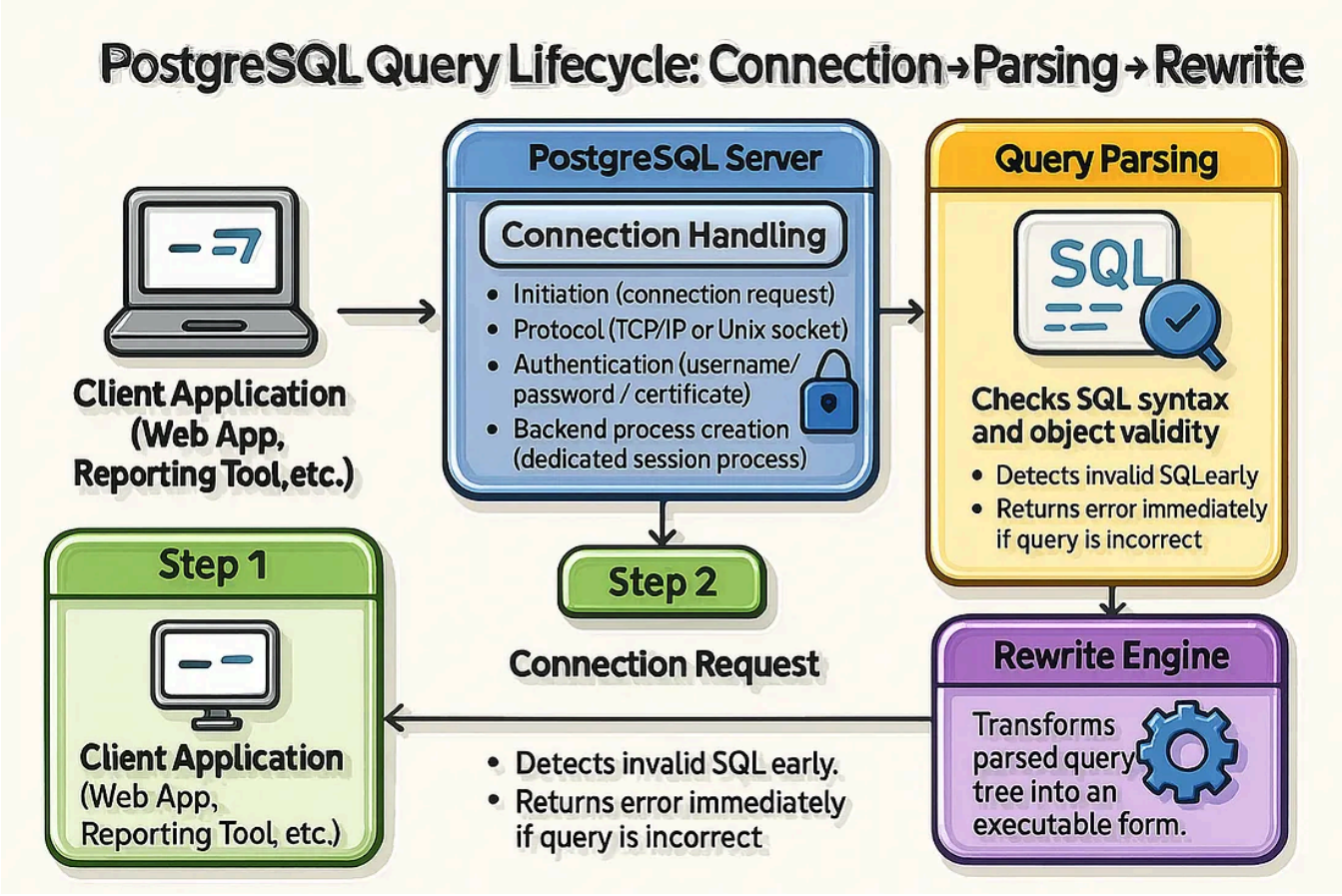
03 -PostgreSQL Performance Tuning: Step-by-Step Journey of a Query

16 min read · Aug 29, 2025

J Jeyaram Ayyalusamy Following



When you write a query in PostgreSQL, it feels almost magical — you type a SQL statement, press enter, and within moments, the result appears. But what seems instant on the surface is actually a carefully designed sequence of operations happening inside PostgreSQL.

In reality, every query goes through several **well-defined stages** before it produces a result. Each stage has its own purpose — to validate, optimize, and execute the query efficiently. Understanding these stages is important for performance tuning because it helps you see where PostgreSQL spends its time and how you can reduce overhead in large systems.

Let's take a simple query and trace how it begins its journey.

Example Query

Suppose we're working with an **e-commerce database** and want to fetch details of a particular customer:

First, create the `customers` table and insert the sample data.

```
CREATE TABLE customers (
    customer_id   INT PRIMARY KEY,
    name          TEXT NOT NULL,
    email         TEXT NOT NULL,
    status        TEXT CHECK (status IN ('active','inactive'))
);
```

```
INSERT INTO customers (customer_id, name, email, status) VALUES
(100, 'Alice Johnson',  'alice.johnson@example.com',  'active'),
(101, 'Bob Smith',      'bob.smith@example.com',      'inactive'),
(102, 'Charlie Lee',    'charlie.lee@example.com',    'active'),
(103, 'Diana Carter',   'diana.carter@example.com',   'inactive'),
(104, 'Ethan Brown',    'ethan.brown@example.com',    'active');
```

```
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

It looks like a straightforward request: *"Give me the name and email of the customer whose ID is 101."*

```
postgres=# SELECT name, email
FROM customers
WHERE customer_id = 101;
   name    |         email
-----------+-----------------------
 Bob Smith | bob.smith@example.com
(1 row)

postgres=#
```

But before PostgreSQL can even begin to check the syntax or optimize this query, one crucial step must happen first: the application must **connect** to the PostgreSQL server.

Step 1: Connection Establishment

Before any query can run, there has to be an **active connection** between your application and the PostgreSQL server.

🔹 How the connection works

1. **Initiation** — Your application (for example, a web app or reporting tool) sends a connection request to the PostgreSQL server.
2. **Protocol** — This request typically happens over **TCP/IP** if the server is remote, or via a **Unix socket** if it's on the same machine.
3. **Authentication** — PostgreSQL checks the credentials (username, password, or certificate) against its authentication rules.
4. **Backend process creation** — If authentication succeeds, PostgreSQL creates a **dedicated backend process** for that session.

- This backend process is unique to the connection.
- It will handle all queries from this client until the session ends.

🔹 Why this matters for performance

- Each connection consumes memory and CPU resources.

- If your system opens too many connections (e.g., thousands at once), PostgreSQL must create thousands of backend processes, which can hurt performance.
- That's why **connection pooling tools** like **PgBouncer** or **Pgpool-II** are often used — they reduce the overhead of creating and destroying backend processes for each client.

🔹 Real-world example

- A shopping website may have hundreds of customers logging in at once.
- Without pooling, each customer session could create a new backend process, straining the server.
- With pooling, connections are reused efficiently, so PostgreSQL doesn't get overloaded before queries even start.

🔹 Analogy

Think of this step as **calling a customer support hotline**.

- First, you dial the number (send a connection request).
- The system checks your identity (authentication).
- Then, you are connected to an **agent** (backend process) who will handle all your questions (queries) during that call.

Unless you're connected to an agent, you can't ask any questions — and similarly, unless your application is connected to PostgreSQL, no query can even begin to run.

Step 2 — Parsing Stage

After a connection is established, the next step in PostgreSQL's query lifecycle is the **Parsing Stage**. This is where PostgreSQL examines the query you typed to ensure it is **correctly written** and refers to valid objects in the database.

Parsing is essential because it prevents PostgreSQL from wasting resources on invalid queries. If the query fails at this stage, PostgreSQL immediately returns an error instead of trying to execute something that makes no sense.

What happens during Parsing?

1. Syntax validation

The parser first checks if your SQL statement follows the **rules of the SQL language.**

✅ Correct query:

```
SELECT name, email FROM customers WHERE customer_id = 101;
```

```
postgres=# SELECT name, email FROM customers WHERE customer_id = 101;
   name    |         email
-----------+-----------------------
 Bob Smith | bob.smith@example.com
(1 row)

postgres=#
```

❌ Incorrect query (keyword misspelled):

```
SELET name, email FROM customers WHERE customer_id = 101;
```

If the keyword `SELECT` is mistyped as `SELET`, PostgreSQL immediately throws an error:

```
postgres=# SELET name, email FROM customers WHERE customer_id = 101;
ERROR:  syntax error at or near "SELET"
LINE 1: SELET name, email FROM customers WHERE customer_id = 101;
        ^
postgres=#
```

👉 **Think of this as grammar checking in English.** Just as a sentence with missing words doesn't make sense, a query with incorrect syntax won't pass the parser.

2. Object validation

After confirming the grammar is correct, PostgreSQL checks whether the **database objects** you referred to actually exist.

For our query:

```
SELECT name, email FROM customers WHERE customer_id = 101;
```

- Does the table `customers` exist?
- Does it have the columns `name`, `email`, and `customer_id`?

If any of these are wrong, PostgreSQL immediately stops and shows an error.

For example, if you typed `customerid` instead of `customer_id`:

```
SELECT name, email FROM customers WHERE customerid = 101;
```

You'd get:

```
postgres=#
postgres=# SELECT name, email FROM customers WHERE customerid = 101;
ERROR:  column "customerid" does not exist
LINE 1: SELECT name, email FROM customers WHERE customerid = 101;
                                               ^
HINT:  Perhaps you meant to reference the column "customers.customer_id".
postgres=#
```

👉 **Analogy:** This is like filling out a form with field names. If the form expects `First Name` and you write `Nickname`, the system won't recognize it. PostgreSQL is just as strict.

3. Query tree creation

If both the syntax and the objects are valid, PostgreSQL then translates the SQL query into an internal structure called a **query tree**.

The query tree is not about *how* the query will be executed (that comes later in optimization). Instead, it's a **structured map** of what the query is asking for.

For our example query:

```
SELECT name, email FROM customers WHERE customer_id = 101;
```

The query tree looks like this:

- **Operation:** SELECT
- **Columns:** name, email
- **Source Table:** customers
- **Condition:** customer_id = 101

👉 **Analogy:** Think of the query tree as a teacher rewriting a messy math problem neatly on the board. The teacher hasn't solved it yet, but now it's clear and ready for the next steps.

Real-world example

Imagine a customer support chatbot that receives your request:

You type:

> “Show me John’s last 5 orders.”

Before giving you results, the chatbot:

1. Checks if your sentence is valid (grammar).
2. Confirms that “John” exists in its database.
3. Translates your request into a structured format so the system can actually fetch the right data.

PostgreSQL’s parsing stage does exactly the same for your SQL queries.

Why Parsing Matters for Performance

- Parsing ensures only **valid queries** reach the optimizer and executor, saving system resources.
- It provides a clean **query tree**, which becomes the foundation for later stages like rewriting, planning, and execution.
- Errors are caught early, avoiding wasted effort on invalid queries.

✅ **Summary:**

- **Parsing Stage** in PostgreSQL 17 checks for syntax and object validity.
- If correct, it creates a **query tree** that describes what the query is asking for.
- Example: `SELECT name, email FROM customers WHERE customer_id = 101;` → Parsed into a query tree with operation, columns, source table, and condition.
- Analogy: Like a teacher checking if a math problem is written correctly before solving it.

Step 3 — Rewrite System

After PostgreSQL parses a query and creates a **query tree**, the next step is the **Rewrite System**. This stage is often invisible to developers, but it is critical. Its main job is to transform the query tree so that it can actually be executed on **real base tables**.

In simpler words: if parsing makes sure your SQL is valid, the rewrite system makes sure it's **executable**.

What is the Rewrite System?

The rewrite system is like a **translator** inside PostgreSQL.

- It looks at the query tree created by the parser.
- It applies **rules** or **transformations** to adjust the query.
- It ensures that queries written against **abstract objects** (like views) are mapped to actual **base tables** that store the data.

The key point: The rewrite system does not change *what you asked for*, but it rewrites *how the query is expressed* so PostgreSQL can execute it.

The Role of Views

The most common place where rewriting happens is when you query a **view**.

- A **view** is like a saved query that looks like a table.
- But views **don't store data**. They are shortcuts to base tables.
- PostgreSQL has to rewrite queries on views into queries on the real tables.

Example with a View

Let's say you create a view for active customers in your e-commerce database:

```
CREATE VIEW active_customers AS
SELECT * FROM customers WHERE status = 'active';
```

```
postgres=# CREATE VIEW active_customers AS
SELECT * FROM customers WHERE status = 'active';
CREATE VIEW
postgres=#
```

Now you run this query:

```
SELECT name, email
FROM active_customers
WHERE customer_id = 102;
```

```
postgres=# select * from active_customers;
 customer_id |     name      |           email           | status
-------------+---------------+---------------------------+--------
         100 | Alice Johnson | alice.johnson@example.com | active
         102 | Charlie Lee   | charlie.lee@example.com   | active
         104 | Ethan Brown   | ethan.brown@example.com   | active
(3 rows)

postgres=#

postgres=# SELECT name, email
FROM active_customers
WHERE customer_id = 102;
    name     |          email
-------------+-------------------------
 Charlie Lee | charlie.lee@example.com
(1 row)

postgres=#
```

Here's what happens:

1. PostgreSQL sees that `active_customers` is a **view**, not a physical table.
2. The rewrite system steps in.
3. It rewrites the query into one that runs against the **base table** `customers`:

```
SELECT name, email
FROM customers
WHERE status = 'active' AND customer_id = 102;
```

```
postgres=# SELECT name, email
FROM customers
WHERE status = 'active' AND customer_id = 102;
    name     |          email
-------------+------------------------
 Charlie Lee | charlie.lee@example.com
(1 row)

postgres=#
postgres=#
```

Notice how PostgreSQL added the condition from the view (`status = 'active'`) and combined it with the condition you gave (`customer_id = 101`).

Why is this Important?

- Without the rewrite system, queries on **views** would fail, because PostgreSQL wouldn't know how to access the data.
- Rewriting ensures that your query always targets **real tables**, even if you wrote it against a shortcut like a view.
- This makes views powerful — they let developers write simpler queries, while PostgreSQL handles the complex transformations in the background.

Real-world Analogy

Imagine you go to a **travel agent** and say:

> "I'd like a Paris holiday package."

The travel agent doesn't stop there. Instead, they break it into specific bookings:

- Flight tickets to Paris ✈️
- Hotel reservations 🏨
- City tours 🎟️

You only asked for a **package**, but behind the scenes, the agent rewrote your request into multiple detailed actions.

👉 PostgreSQL's **rewrite system** does exactly the same thing. A view (holiday package) is rewritten into detailed instructions (base table queries) that PostgreSQL can actually execute.

Another Example: Rule-based Rewriting

Although views are the most common, rewriting can also apply **rules** that modify queries. For example, PostgreSQL allows you to define rewrite rules using the `CREATE RULE` command.

Suppose you define:

create the audit_log table

```
CREATE RULE no_delete AS
ON DELETE TO customers DO INSTEAD
INSERT INTO audit_log VALUES ('DELETE blocked', now());
```

```
postgres=# CREATE TABLE audit_log (
    log_id     SERIAL PRIMARY KEY,
```

```
    message    TEXT,
    logged_at  TIMESTAMPTZ DEFAULT now()
);
CREATE TABLE
postgres=#
```

Create the rule:

```
CREATE RULE no_delete AS
ON DELETE TO customers DO INSTEAD
INSERT INTO audit_log (message, logged_at)
VALUES ('DELETE blocked', now());
```

```
postgres=# CREATE RULE no_delete AS
ON DELETE TO customers DO INSTEAD
INSERT INTO audit_log (message, logged_at)
VALUES ('DELETE blocked', now());
CREATE RULE
postgres=#
```

Now, if someone tries to run:

```
DELETE FROM customers WHERE customer_id = 101;
```

The rewrite system will intercept it and rewrite it into:

```
postgres=# INSERT INTO audit_log (message, logged_at)
VALUES ('DELETE blocked', now());
INSERT 0 1
postgres=#
postgres=#
```

```
postgres=# INSERT INTO audit_log (message, logged_at)
VALUES ('DELETE blocked', now());
INSERT 0 1
postgres=#
postgres=# SELECT * FROM audit_log;
 log_id |    message     |           logged_at
--------+----------------+-------------------------------
      1 | DELETE blocked | 2025-08-29 17:46:15.489419+00
      2 | DELETE blocked | 2025-08-29 17:47:56.584537+00
(2 rows)

postgres=# DELETE FROM customers WHERE customer_id = 101;
DELETE 0
postgres=#
postgres=# SELECT * FROM audit_log;
 log_id |    message     |           logged_at
--------+----------------+-------------------------------
      1 | DELETE blocked | 2025-08-29 17:46:15.489419+00
      2 | DELETE blocked | 2025-08-29 17:47:56.584537+00
      3 | DELETE blocked | 2025-08-29 17:49:06.179487+00
(3 rows)

postgres=#
```

👉 The user thinks they tried to delete a row, but PostgreSQL rewrote it to insert into the audit log instead.

Summary

- The **Rewrite System** in PostgreSQL 17 takes the query tree from the parser and applies transformations.
- Most commonly, it rewrites queries on **views** into equivalent queries on **base tables.**
- It can also apply **rules**, such as modifying or redirecting queries.
- Example: Querying `active_customers` (a view) is rewritten into a query against `customers`.
- Analogy: Like a travel agent converting a “holiday package” request into flights, hotels, and tours.

✅ By the end of the rewrite stage, PostgreSQL has a **fully expanded query tree** that references real objects, ready to move on to the next step: optimization.

PostgreSQL 17 Performance Tuning: Step 4 — Optimizer (Planner)

After a query has been parsed and rewritten, PostgreSQL hands it over to the **Optimizer (or Planner)**. This stage is the **decision-maker** — it figures out the best way to actually run your query.

Think of it as the "strategy builder" of PostgreSQL. You tell PostgreSQL *what you want* in your SQL statement, but it's the optimizer that decides *how to get it efficiently*.

Why Do We Need the Optimizer?

SQL is **declarative**, meaning you describe the *result* you want, not the *steps* to get it. For example:

```
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

You don't tell PostgreSQL whether to scan the whole table or use an index — you only state the result you want.

The optimizer decides:

- Which tables to scan.
- Whether to use an index.
- How to join tables (if there are multiple).
- Whether sorting or filtering should happen early or late.

Without an optimizer, every query would default to the slowest method, wasting time and resources.

Common Execution Paths

There are usually multiple ways to run the same query. The optimizer explores these paths and compares them.

1. Sequential Scan

- PostgreSQL starts at the first row of the table and checks each row until it finds a match.
- Efficient for **small tables** because the overhead of using an index might be unnecessary.
- Problematic for **large tables** because it has to read every row.

Example:

- If `customers` has 100 rows, scanning all of them is cheap.

2. Index Scan

- If there is an index on `customer_id`, PostgreSQL can jump directly to the row where `customer_id = 101`.
- Much faster for **large tables** because it skips irrelevant rows.
- However, for small tables, the index lookup overhead might make it slower than a sequential scan.

Example:

- If `customers` has 10 million rows, the optimizer will almost always prefer an index scan.

3. Other Strategies (joins, sorts, etc.)

For more complex queries, PostgreSQL may also decide between:

- **Nested loop joins vs hash joins vs merge joins.**
- **Sorting with indexes vs in-memory sort operations.**
- **Parallel query execution** (added and improved in recent versions like PostgreSQL 17).

But for now, let's stick to the sequential vs index scan example, since it's the easiest to understand.

How PostgreSQL Decides

PostgreSQL doesn't guess. It uses **statistics** about your data, stored in the system catalog, to estimate the cost of each possible plan.

It considers factors such as:

- **Table size** → How many rows are in the table?
- **Row count estimate** → How many rows will match the filter (`WHERE customer_id = 101`)?
- **Index selectivity** → How selective or efficient is the index at finding rows?
- **Disk I/O vs memory usage** → How much data needs to be read from disk vs from cache?
- **CPU cost** → How many comparisons or calculations will be needed?

Once PostgreSQL calculates the estimated cost for each plan, it picks the **cheapest one.**

Example in Action

Let's look at how PostgreSQL might handle our query in two situations:

Case 1: Small Table (100 rows)

```
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

- The optimizer decides: **Sequential Scan.**
- Why? Because reading 100 rows directly is faster than the overhead of using an index.

Case 2: Large Table (10 million rows)

```
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

```
postgres=# SELECT name, email
FROM customers
WHERE customer_id = 101;
   name    |         email
-----------+-----------------------
 Bob Smith | bob.smith@example.com
(1 row)

postgres=#
```

- The optimizer decides: **Index Scan.**
- Why? Because jumping directly to the correct row via the index is much faster than scanning millions of rows one by one.

The Output: Query Plan

After analyzing all paths, the optimizer produces a **query plan**.

- This plan is a step-by-step execution strategy.
- The next stage (the executor) will follow this plan to actually run the query.

You can see this plan using PostgreSQL's `EXPLAIN` command:

```
EXPLAIN SELECT name, email
FROM customers
WHERE customer_id = 101;
```

```
postgres=# EXPLAIN SELECT name, email
FROM customers
WHERE customer_id = 101;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using customers_pkey on customers  (cost=0.15..8.17 rows=1 width=64
   Index Cond: (customer_id = 101)
(2 rows)

postgres=#
```

Depending on the table size and indexes, you might see either:

- `Seq Scan on customers`
- `Index Scan using customers_customer_id_idx on customers`

Analogy: Google Maps for Queries

Think of the optimizer like **Google Maps**:

- You tell it your destination (the SQL query).
- Google Maps checks all possible routes — highway, back roads, toll roads.
- It estimates traffic, distance, and time.
- Then it suggests the **fastest, cheapest route**.

👉 PostgreSQL's optimizer does exactly this. It looks at all possible ways to fetch your data and chooses the most efficient one.

Summary

- The **Optimizer (Planner)** in PostgreSQL 17 is responsible for deciding how a query should be executed.
- It evaluates multiple possible paths (like sequential scan vs index scan).
- It uses **statistics** to estimate costs and chooses the cheapest plan.
- The chosen plan is a detailed **execution strategy**, ready for the executor.
- Analogy: Like Google Maps suggesting the best route to your destination based on traffic and distance.

✅ By the end of this stage, PostgreSQL has built a **smart, optimized plan** for your query, ensuring that execution is as efficient as possible.

PostgreSQL 17 Performance Tuning: Step 5 — Executor

After PostgreSQL has checked your query for correctness (parsing), applied rules or views (rewrite system), and chosen the best plan (optimizer), the final step is the **Executor**.

The executor is the stage where **plans become reality**. It's the worker that takes the blueprint (query plan) and actually runs it step by step to produce results for your application.

What Happens in the Executor?

The executor's role is to **carry out the plan chosen by the optimizer.** It doesn't question the plan or try to optimize it further — it simply executes it.

Let's break this into sub-steps.

1. Accessing the Data

The executor begins by reading the data from the relevant tables.

- If the optimizer chose a **Sequential Scan,** the executor reads all rows one by one until the filter condition is met.
- If the optimizer chose an **Index Scan,** the executor uses the index to jump directly to the matching row(s).

👉 Example:

```
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

- **Sequential Scan** → Executor checks every row in `customers` until it finds the row where `customer_id = 101`.
- **Index Scan** → Executor jumps directly to the correct row using the index on `customer_id`.

2. Applying Filters

After rows are retrieved, the executor applies filters from the `WHERE` clause.

- In our query, the filter is `customer_id = 101`.
- Even if a scan brings in multiple rows, only those matching this condition are kept.

👉 Example: If the `customers` table has IDs 1 to 10,000, the executor ensures that only the row with `101` is passed to the next step.

3. Performing Additional Operations

If the query involves more than just fetching rows, the executor handles those too.

- **Joins** → If the query joins multiple tables, the executor combines rows according to the join strategy (nested loop, hash join, merge join).
- **Sorting** → For `ORDER BY`, it sorts rows into the required order.
- **Grouping** → For `GROUP BY`, it groups rows by the specified column(s).
- **Aggregations** → For functions like `COUNT`, `SUM`, or `AVG`, it computes the results.

👉 Example:

```
SELECT status, COUNT(*)
FROM customers
GROUP BY status;
```

```
postgres=# SELECT status, COUNT(*)
FROM customers
GROUP BY status;
  status  | count
----------+-------
 active   |     3
 inactive |     2
(2 rows)

postgres=#
```

Steps for executor:

1. Read rows from `customers`.
2. Group them by `status` (active, inactive).
3. Count the rows in each group.
4. Return the results.

4. Returning the Results

Once all operations are complete, the executor assembles the final result set and sends it back to the client application.

👉 Example output for our simple query:

name email Alice alice@email.com

At this point, the query lifecycle is complete — the application receives the exact data it asked for.

Analogy: Chef and Recipe

The executor is like a **chef in a kitchen**:

- The **optimizer** writes the recipe (query plan), carefully deciding the order of steps and ingredients.
- The **executor (chef)** follows the recipe exactly, chopping, mixing, and cooking step by step.
- The **final dish** (query result) is served to the customer (your application).

👉 Without the executor, the recipe is just an idea. The executor is the one who actually turns it into a finished meal.

Example with `EXPLAIN ANALYZE`

You can see the executor's work in action using `EXPLAIN ANALYZE`.

```
EXPLAIN ANALYZE
SELECT name, email
FROM customers
WHERE name = 'Alice Johnson';
```

If the table is small, you may see a **sequential scan**:

```
postgres=# EXPLAIN ANALYZE
SELECT name, email
FROM customers
WHERE name = 'Alice Johnson';
                                                QUERY PLAN
-----------------------------------------------------------------------------------------
 Seq Scan on customers  (cost=0.00..17.88 rows=3 width=64) (actual time=0.017..
   Filter: (name = 'Alice Johnson'::text)
   Rows Removed by Filter: 4
 Planning Time: 0.075 ms
 Execution Time: 0.033 ms
(5 rows)

postgres=#
```

If the table is large and indexed, you may see an **index scan**:

```
EXPLAIN ANALYZE
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

```
postgres=# EXPLAIN ANALYZE
SELECT name, email
FROM customers
WHERE customer_id = 101;
```

```
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using customers_pkey on customers  (cost=0.15..8.17 rows=1 width=64
   Index Cond: (customer_id = 101)
 Planning Time: 0.055 ms
 Execution Time: 0.032 ms
(4 rows)

postgres=#
```

Here, you can clearly see the executor carrying out the plan chosen by the optimizer.

Summary

- The **Executor** in PostgreSQL 17 is the final stage that **executes the query plan**.
- It:
- Retrieves data (sequential scan or index scan).
- Applies filters (`WHERE`).
- Handles joins, sorting, grouping, and aggregations.
- Returns the results to the client application.
- Example: For `WHERE customer_id = 101` , the executor fetches only the matching row and returns `name` and `email` .
- Analogy: Like a **chef following a recipe** written by the optimizer — the executor "cooks" the query into actual results.

✅ Once the executor finishes, your SQL query has traveled through the full PostgreSQL lifecycle, from parsing to execution, and the data is ready in your application.

Full Flow Recap

So, when you run:

```
SELECT name, email FROM customers WHERE customer_id = 101;
```

PostgreSQL does the following:

1. **Connection** → The application connects to the PostgreSQL server.
2. **Parser** → Query is validated and turned into a query tree.
3. **Rewrite System** → Views and rules are expanded into real queries.
4. **Optimizer** → The best execution plan is chosen (sequential scan vs index scan).
5. **Executor** → The plan is executed, data is fetched, and results are returned.

📢 Stay Updated with Daily PostgreSQL & Cloud Tips!

If you've been finding my blog posts helpful and want to stay ahead with daily insights on PostgreSQL, Cloud Infrastructure, Performance Tuning, and DBA Best Practices — I invite you to subscribe to my Medium account.

🔔 Subscribe here 👉 https://medium.com/@jramcloud1/subscribe

Your support means a lot — and you'll never miss a practical guide again!

🔗 Let's Connect!

If you enjoyed this post or would like to connect professionally, feel free to reach out to me on LinkedIn:
👉 Jeyaram Ayyalusamy

I regularly share content on **PostgreSQL, database administration, cloud technologies, and data engineering.** Always happy to connect, collaborate, and discuss ideas!

Postgresql Oracle MySQL AWS Mongodb

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

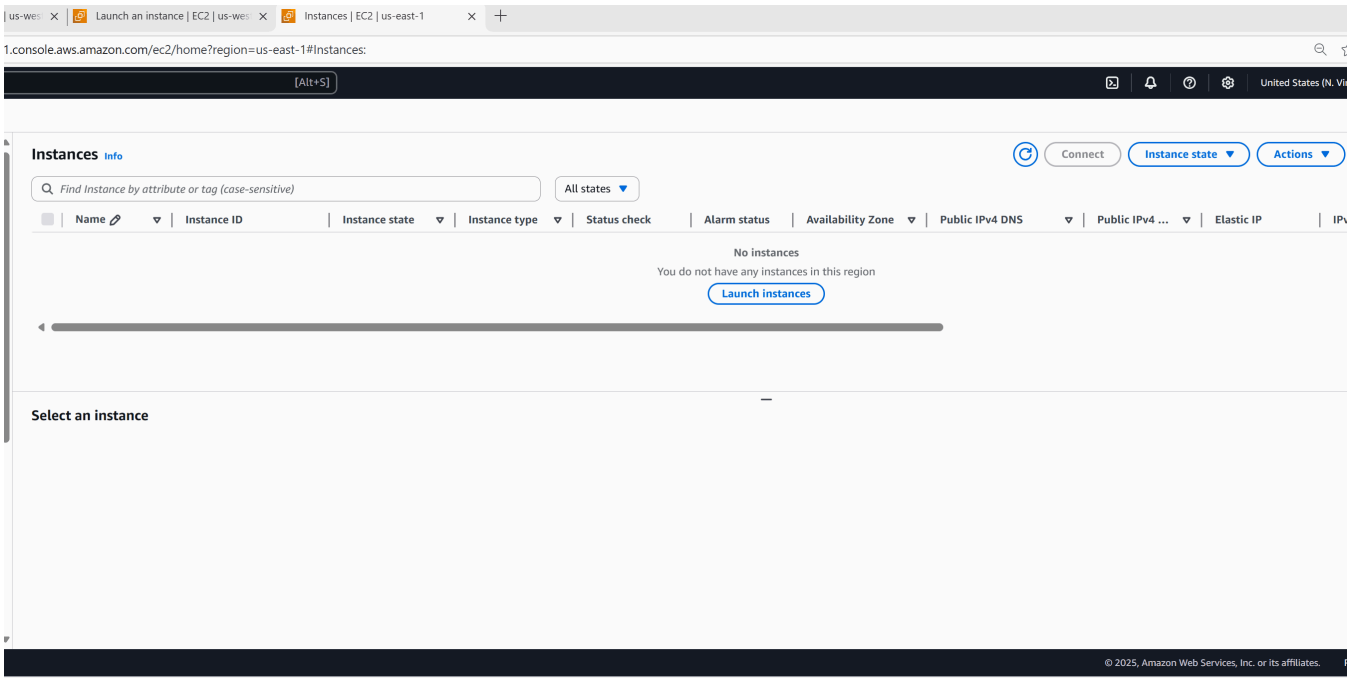
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

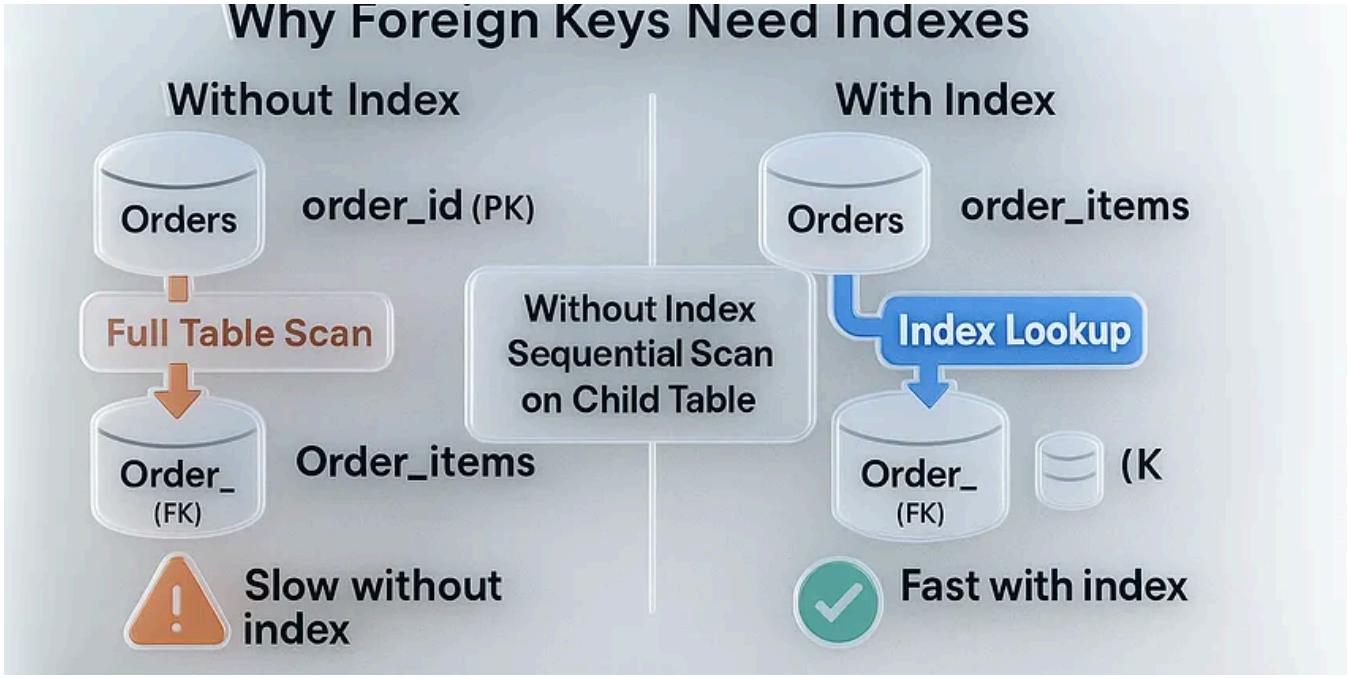


Jeyaram Ayyalusamy

Upgrading PostgreSQL from Version 16 to Version 17 Using pg_upgrade on a Linux Server AWS EC2...

🛠️ A Complete Step-by-Step Guide to Installing PostgreSQL 16 on a Linux Server (With Explanations)

Aug 4 40

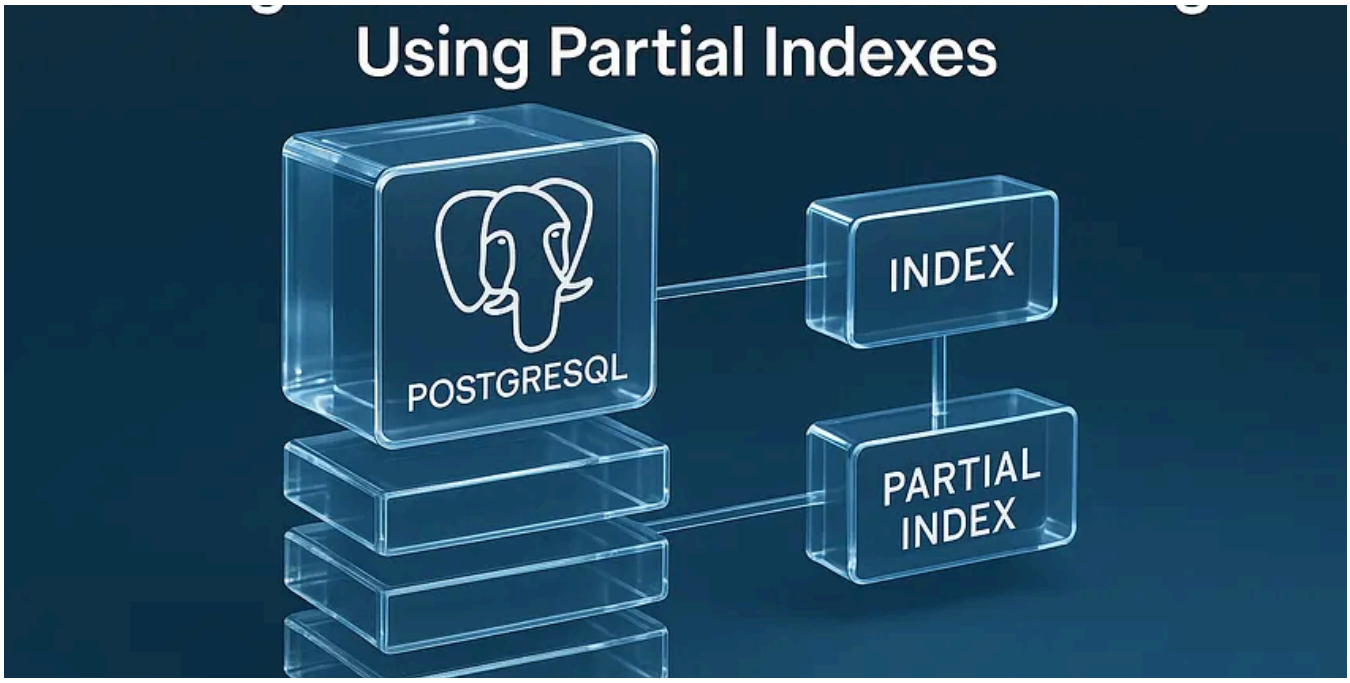


Jeyaram Ayyalusamy

16 -PostgreSQL 17 Performance Tuning: Why Foreign Keys Need Indexes

When designing relational databases, parent-child relationships are common. A classic case is the Orders table (parent) and the Items table...

Sep 3 3 2

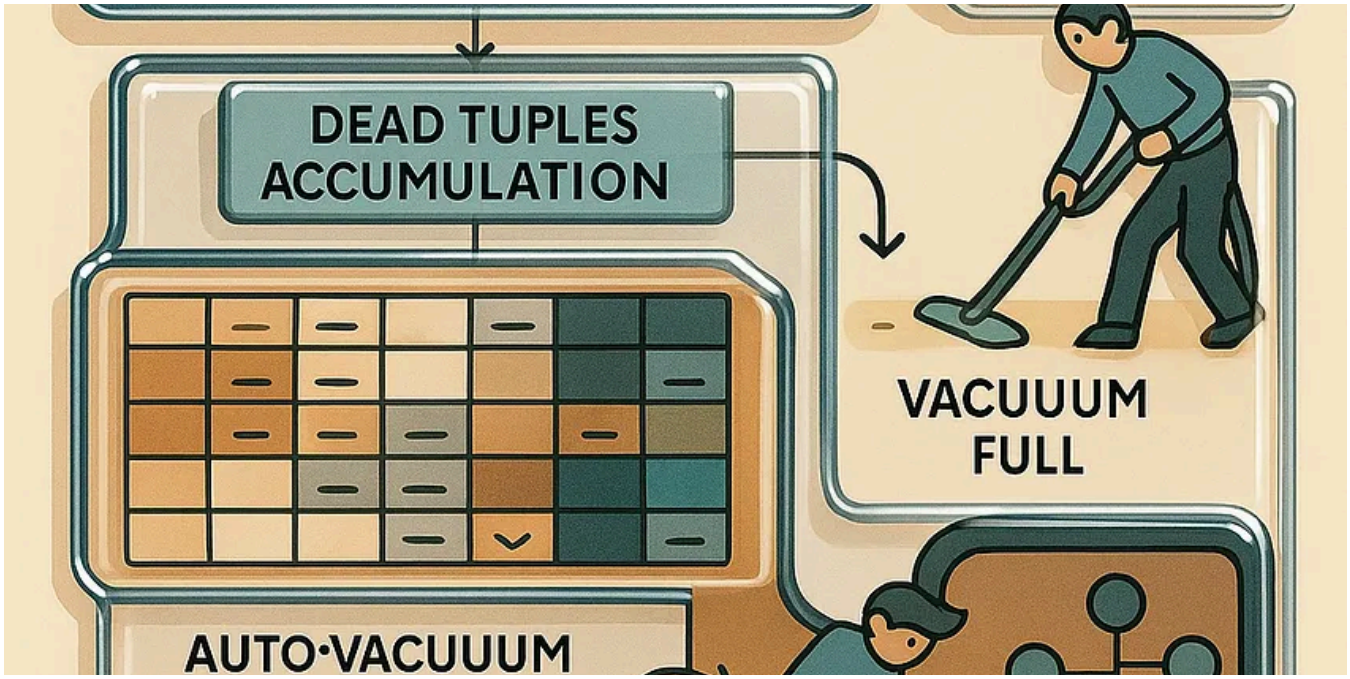


Jeyaram Ayyalusamy

17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes

Even though we usually create indexes on all rows of a table, PostgreSQL also allows you to create an index on just a subset of records...

Sep 4 3



Jeyaram Ayyalusamy

08-PostgreSQL 17: Complete Tuning Guide for VACUUM & AUTOVACUUM

PostgreSQL's MVCC design creates dead tuples during UPDATE/DELETE. VACUUM reclaims them; AUTOVACUUM schedules that work. Get these knobs...

Sep 1 26

See all from Jeyaram Ayyalusamy

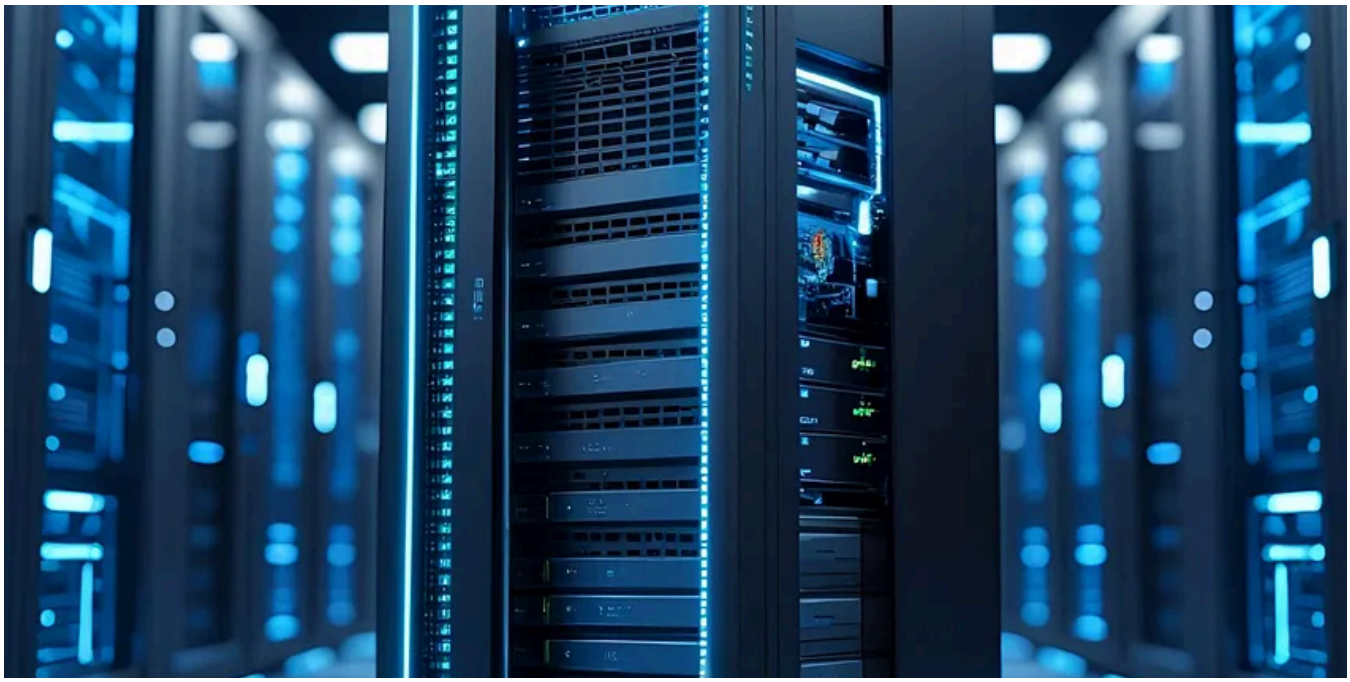
Recommended from Medium

Tomasz Gintowt

Security in PostgreSQL

PostgreSQL is a powerful open-source database. Security is very important when working with sensitive data like passwords, customer...

6d ago 5

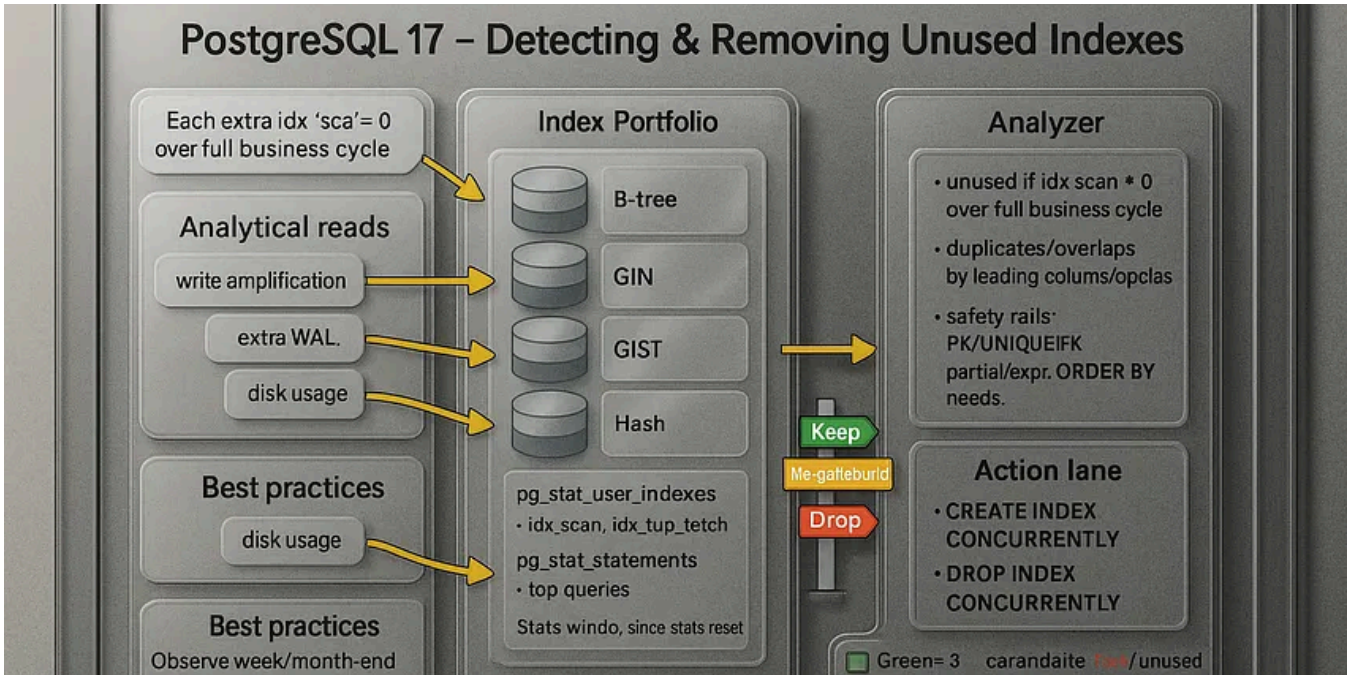


Rizqi Mulki

PostgreSQL Index Bloat: The Silent Killer of Performance

How a 10TB database became 3x faster by fixing the invisible problem that plagues 90% of production PostgreSQL deployments

Sep 15 11 1

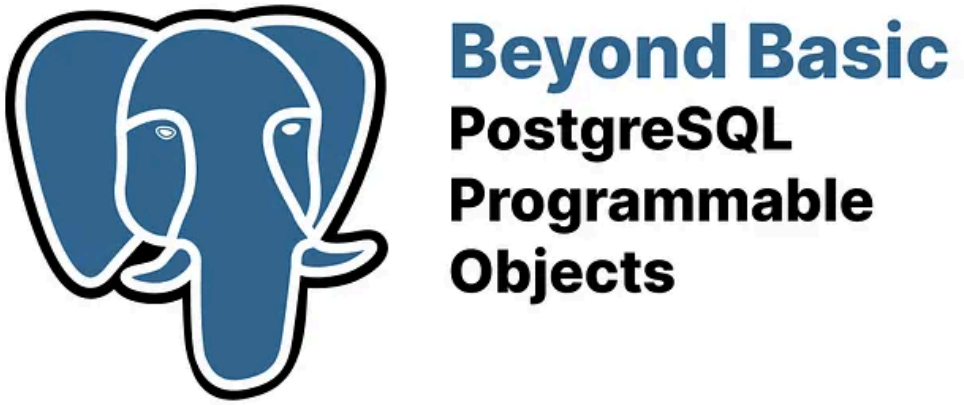


Jeyaram Ayyalusamy

25 - PostgreSQL 17 Performance Tuning: Detecting and Removing Unused Indexes

Indexes are one of the most important tools for query performance in PostgreSQL. But while a missing index slows down queries, an unused or...

Sep 10 40

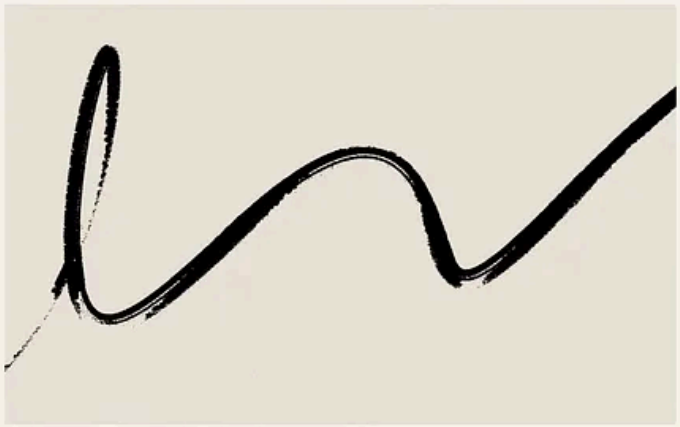


In Stackademic by bektiaw

Beyond Basics: PostgreSQL Programmable Objects (Automate, Optimize, Control)

Tired of repeating the same SQL logic? Learn how PostgreSQL can do the work for us.

Sep 1 68 1



Rohan

JSONB in PostgreSQL: The Fast Lane to Flexible Data Modeling 🚀

“Why spin up yet another database just to store semi-structured data?”—Every engineer who discovered JSONB

Jul 18 12 1

Vijay Gadhave

Master SQL’s QUALIFY Clause for Cleaner, Faster Window Queries

Note: If you’re not a medium member, CLICK HERE

May 20 12 1

See more recommendations