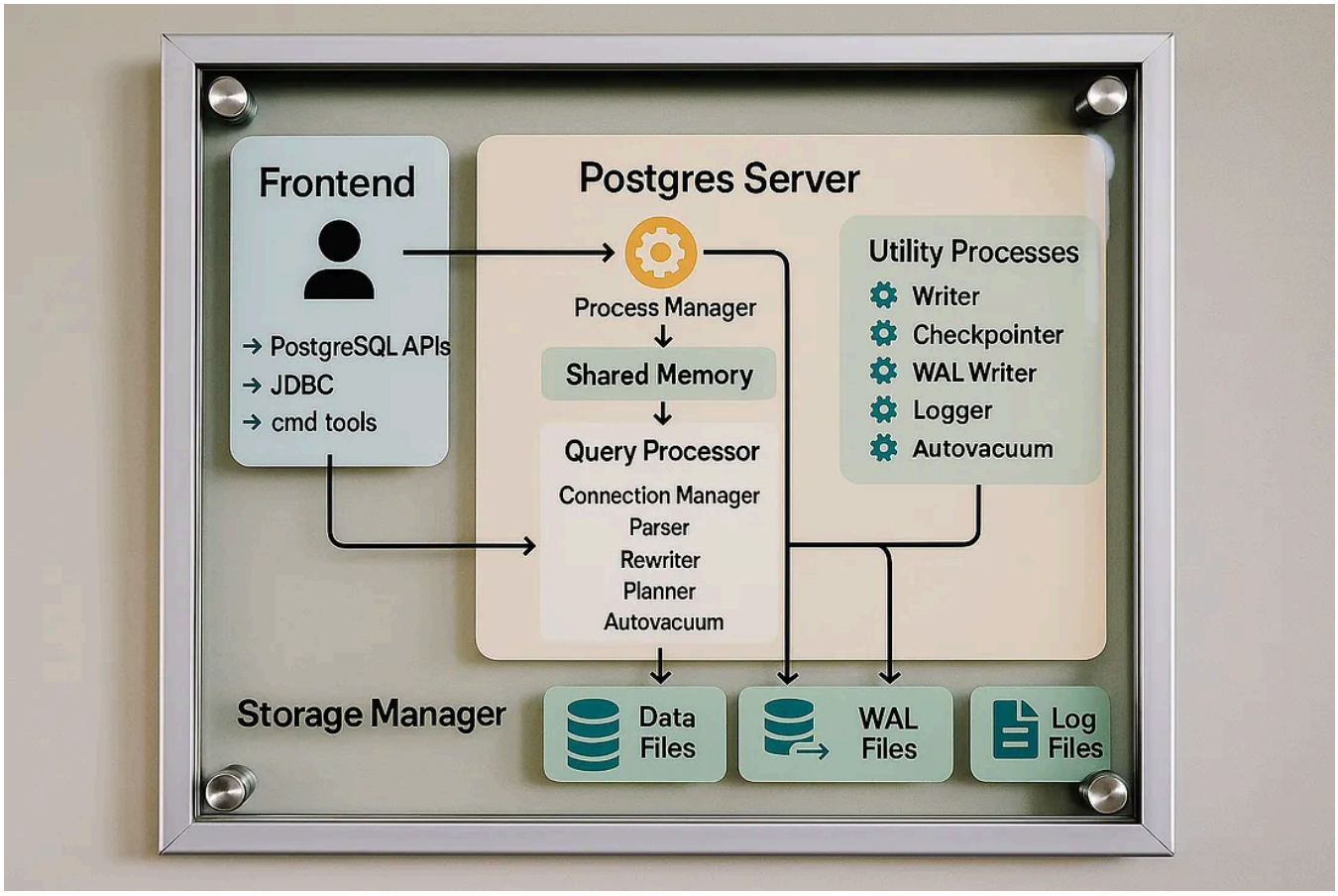
Understanding the Client-Server Model in PostgreSQL 17

14 min read · Aug 29, 2025

Jeyaram Ayyalusamy Following



Introduction

PostgreSQL 17, like most modern relational databases, is built on the **client-server model.** This model is what allows PostgreSQL to handle **multiple users, multiple**

applications, and multiple requests all at the same time without losing control or data consistency.

If you're new to PostgreSQL administration, the easiest way to think about this model is to compare it to a **restaurant**:

🍽️ 1. The Client = The Customer

- In a restaurant, the customer places an order for food.
- In PostgreSQL, the client (which could be `psql`, an application, or a GUI tool) places an **SQL query** like:

```
SELECT * FROM orders WHERE status = 'pending';
```

- The client never prepares the meal (or the data) itself — it only **requests** what it wants.

👩‍🍳 2. The Server = The Kitchen Staff

- Once the customer places the order, the kitchen staff gets to work.
- In PostgreSQL, the **server processes (daemon and backends)** take the query, **parse it, plan it, execute it, and then return results.**
- Just like cooks may chop, grill, or fry depending on the meal, PostgreSQL decides whether to use an index, scan a table, or join data depending on the query.

👨‍💼 3. The Daemon = The Restaurant Manager

- Every restaurant needs a manager who:

- Opens the restaurant each day.
- Assigns waiters and chefs to customers.
- Ensures the kitchen runs smoothly.
- Keeps track of inventory and makes sure the place is clean.
- Similarly, in PostgreSQL:
- The **daemon process** (often called `postmaster`) is the **master controller**.
- It **accepts new connections**, **spawns backend processes** for each client, and **runs background tasks** like:
- **Background writer** → writes changes to disk gradually.
- **Checkpointer** → ensures data consistency at regular intervals.
- **Autovacuum launcher** → cleans up old/dead data to prevent "table bloat."
- **Logger** → records errors and system events.

👉 Without the daemon, no one could "open the doors," and clients wouldn't even be able to place their orders.

Putting It All Together

Here's how it works in practice:

1. **The client (customer)** → Sends a request (`SELECT`, `INSERT`, etc.).
2. **The daemon (manager)** → Accepts the request and assigns it to a **backend process**.
3. **The backend process (kitchen staff)** → Executes the SQL query:

- Parses it (understands what's being asked).
- Plans it (chooses the most efficient way).
- Executes it (fetches or modifies data).

4. **Results are returned to the client** → Just like food is delivered to the customer.

Why This Matters for Performance Tuning

When tuning PostgreSQL performance in version 17, it's important to understand this model because:

- **Too many clients (customers)** can overwhelm the daemon → leading to connection errors (`too many clients`).
- **Backend processes (kitchen staff)** can get stuck on slow queries → delaying results for everyone else.
- **Background processes (cleaners)** like autovacuum or checkpoints need to run smoothly, or else the database slows down over time.

Just like a busy restaurant must balance staff, orders, and resources to keep customers happy, PostgreSQL must balance **clients, backends, and background processes** to keep queries fast and reliable.

PostgreSQL Performance Tuning: Client-Side — Where Requests Begin

Client-Side: Where Requests Begin

In PostgreSQL, everything starts with the **client**. A client is any tool, program, or interface that a user or application uses to send SQL queries to the database.

Think of the client as the **mouthpiece**: it doesn't do the actual cooking (query execution), but it places the order that kicks off the entire process.

PostgreSQL supports different types of clients, and understanding how they work is the first step in performance tuning, because poorly designed client requests can overwhelm the server no matter how powerful it is.

1. Command-Line Tools

- PostgreSQL includes **`psql`**, its default command-line client.
- DBAs and developers rely on `psql` because it gives **direct, low-level access** to the database.

What you can do with `psql`:

- Run SQL queries interactively.
- Check and modify configuration settings.
- Automate routine tasks through scripts (e.g., nightly reports, vacuuming tables).

👉 Example:

```
psql -U postgres -d mydb
```

This connects you to the database `mydb` as the user `postgres`. From here, you can execute queries directly.

Why it matters for performance tuning:

- `psql` is often used to **test queries**, analyze their execution plans (`EXPLAIN`), and monitor system stats.
- It is the most powerful way to **see how the server is behaving** without layers of abstraction.

2. Graphical User Interfaces (GUIs)

Not everyone is comfortable typing commands. For those users, PostgreSQL supports **GUI tools** like:

- **pgAdmin**
- **DBeaver**
- **DataGrip**

What GUIs provide:

- A **point-and-click environment** to explore the database.
- Query editors with autocomplete and visual execution plans.
- Dashboards for monitoring performance and activity.

👉 Example:
Instead of typing `SELECT * FROM customers;` into a terminal, you can run the same query inside pgAdmin with a click, and the results appear in a table view.

Why it matters for performance tuning:

- GUIs help you **visualize bottlenecks** in queries (e.g., missing indexes, long-running joins).
- They're great for **analysts or junior DBAs** who need insights but aren't command-line experts.

3. Application Programs

In the real world, most PostgreSQL requests don't come directly from humans — they come from **applications**.

- Applications can be written in almost any modern language:
- **Java** → uses JDBC to connect.
- **Python** → uses `psycopg2` or `asyncpg`.
- **Node.js** → uses `pg` library.
- **C#**, **Go**, and others also have PostgreSQL drivers.

👉 Example:
A Java application might run this query in the background:

```
SELECT * FROM orders WHERE status = 'pending';
```

The end-user only sees a web page with pending orders, but behind the scenes, the application is acting as the **client** that communicates with PostgreSQL.

Why it matters for performance tuning:

- Poorly optimized queries from applications are **the most common cause of slow performance.**
- For example, if an application runs a `SELECT *` on a table with millions of rows when it only needs a few columns, it can overload the server.

How Clients Communicate with PostgreSQL

Once the client prepares a request, how does it actually reach PostgreSQL?

There are two main communication methods:

1. **TCP/IP Protocol**

- Used when the client and PostgreSQL server are on **different machines.**
- Example: A web app hosted on AWS connecting to a PostgreSQL instance on another server.
- Standard network communication ensures PostgreSQL can be accessed remotely.

2. **Unix Domain Sockets (Linux Sockets)**

- Used when the client and PostgreSQL server are on the **same machine.**
- Faster than TCP/IP because there's no network overhead.

- Common in local development or single-host deployments.

👉 No matter how the connection is made, PostgreSQL enforces the same workflow:

- **Client sends the request** →
- **Server daemon receives it** →
- **Backend process executes it** →
- **Result is sent back to the client.**

Key Takeaway

It doesn't matter whether the client is:

- A DBA typing queries in `psql`.
- An analyst using pgAdmin.
- Or a Python app fetching data from a web form.

The PostgreSQL workflow is always the same:

Client → Server Daemon → Backend → Result.

For performance tuning, this is crucial:

- Optimizing starts at the **client level** (e.g., efficient queries).
- Every poorly written request increases the load on the server.
- Every well-structured query makes PostgreSQL faster and more scalable.

✅ This section sets the stage for performance tuning by showing that **good performance begins with the client's request**. Even the most powerful server cannot perform well if it's constantly receiving inefficient queries.

PostgreSQL Performance Tuning: Server-Side — The Daemon Process

Introduction

Every PostgreSQL database has a **master process** running in the background, called the **daemon process** (or historically the *postmaster*).

This process is the **heartbeat** of PostgreSQL. Without it, the database cannot start, no client can connect, and none of the background tasks that keep PostgreSQL healthy will run.

To make it easier to understand, think of the daemon as the **restaurant manager** in our earlier analogy:

- It **opens the restaurant** (initializes memory and structures).
- It **assigns staff roles** (launches background workers).
- It **greets customers and seats them** (listens for connections and spawns backend processes).

If the manager doesn't show up, the restaurant stays closed. In PostgreSQL, if the daemon fails, the entire database system goes down.

1. Initializing Memory and Shared Structures

When PostgreSQL 17 starts, the daemon's first responsibility is to **set up memory and shared structures**.

Imagine a **large shared whiteboard** in a busy kitchen:

- All the chefs (backend processes) use it to coordinate.
- They write down orders, mark completed ones, and track what's still cooking.

In PostgreSQL, this shared whiteboard is called **shared memory**.

It is used for:

- **Caching data:** Frequently accessed rows or pages are stored in memory so PostgreSQL doesn't have to fetch them from disk repeatedly (which is much slower).
- **Locks and coordination:** Ensures two users don't update the same record at the same time.
- **Transaction control:** Tracks whether a transaction is committed or rolled back so that all processes agree on the database state.

👉 **Performance tuning impact:**

- The parameter `shared_buffers` determines how much memory PostgreSQL uses for caching.
- Setting it too low forces PostgreSQL to read from disk too often.
- Setting it too high may starve the operating system of memory.
- A balanced configuration improves query speed dramatically.

2. Launching Utility Background Processes

PostgreSQL isn't just one process — it's actually a **family of cooperating processes**. The daemon is like a manager assigning roles to different employees. Each utility process has a specialized job:

Background Writer

- Writes modified data from memory (dirty pages) to disk in small, steady amounts.
- Prevents sudden bursts of disk activity.

Checkpointer

- At certain intervals, flushes all dirty pages to disk.
- Guarantees that if PostgreSQL crashes, the database can recover to a safe point.

Autovacuum Launcher

- Removes “dead tuples” (old row versions left behind by updates or deletes).
- Prevents **table bloat**, which slows down queries and increases storage.

Logger Process

- Records all important activity, warnings, and errors into log files.
- These logs are critical for diagnosing slow queries and performance bottlenecks.

Stats Collector

- Gathers information about how queries, indexes, and connections are being used.
- Provides data to the query planner so PostgreSQL can choose faster execution strategies.

👉 **Performance tuning impact:**

- If **autovacuum** runs too slowly, tables grow bloated, and queries crawl.
- If **checkpoints** happen too often, disk I/O spikes and slows down the system.
- If logging isn’t tuned, you may miss important performance data.

By carefully adjusting parameters like `checkpoint_timeout`, `autovacuum_vacuum_scale_factor`, and logging levels, DBAs can keep these processes working in harmony.

3. Listening for Client Connections

The daemon also acts as a **gatekeeper**.

- It waits for client requests on:

- **TCP/IP port (default: 5432)** for remote connections.
- **Unix sockets** for local, same-machine connections (faster since there's no networking overhead).

When a new connection request arrives:

1. The daemon **accepts the request**.
2. It **spawns a new backend process** for that client.
3. The backend process takes over and handles all queries for that client session.

👉 **Performance tuning impact:**

- PostgreSQL creates one **operating system process** per client.
- If too many clients connect at once, the server can run out of resources.
- The parameter `max_connections` sets the maximum number of clients.
- In busy systems, DBAs often use **connection pooling** (e.g., PgBouncer or Pgpool-II) to reuse connections and reduce overhead.

Why the Daemon is Critical

The daemon must always be running. If it stops:

- No new connections are allowed.
- Existing backend processes die.
- Background processes stop working.
- PostgreSQL is effectively **down**.

This is why PostgreSQL is usually managed as a service (`systemctl start postgresql-17`) with **automatic restart policies**, ensuring that if the daemon crashes, it comes back up quickly.

Key Takeaways

- The **daemon (postmaster)** is the master process of PostgreSQL 17.
- Its responsibilities are threefold:

1. **Initialize memory and shared structures** for faster query execution.
2. **Launch background processes** that keep PostgreSQL healthy.
3. **Listen for connections** and spawn backend processes for each client.

- From a **performance tuning perspective**:
- Memory settings like `shared_buffers` directly impact query speed.
- Background process tuning (checkpoints, autovacuum, logging) keeps the system stable and fast.
- Connection management (`max_connections` and pooling) ensures scalability under load.

PostgreSQL Performance Tuning: Connection Lifecycle Step by Step

Introduction

Every time a user or application connects to PostgreSQL 17, a **series of behind-the-scenes steps** takes place before a single query can even run.

Understanding this connection lifecycle is crucial for performance tuning because:

- Each connection consumes memory and CPU.
- Every backend process adds overhead.
- Poorly managed connections can slow down or even crash a database server.

Let's explore the **connection process in PostgreSQL 17 step by step**.

Step 1: Connection Request

Imagine a developer launching a connection with:

```
psql -h localhost -U postgres
```

- This request is sent to the **PostgreSQL daemon (postmaster)**.
- The daemon is always listening for incoming connections — like a **receptionist at a hotel** waiting to check in new guests.

👉 **Performance insight:**
If too many connection requests arrive at once, the daemon can become overwhelmed. This is why many production systems use **connection poolers** (like **PgBouncer**) to reuse existing connections instead of creating new ones each time.

Step 2: Daemon Creates a Backend Process

When the daemon receives the request, it doesn't handle queries itself. Instead, it:

- **Forks a new backend process** specifically for that client.
- Each client always gets its **own dedicated backend process.**

This is like the receptionist assigning each guest their **own hotel room.** Guests don't share rooms, which ensures privacy and stability.

👉 **Performance insight:**
While this model ensures isolation, it's also resource-heavy. Each backend is a full **operating system process**, not just a lightweight thread. Too many connections = too many processes, which can strain memory and CPU.

Step 3: Authentication

Before the client is allowed in, PostgreSQL must **verify identity**.

- Supported authentication methods include:
- `md5` → basic password hashing.
- `scram-sha-256` → stronger and more secure password hashing.
- **LDAP/Kerberos** → for enterprise single sign-on.
- **SSL certificates** → for encrypted, certificate-based logins.
- If authentication fails → the connection is **immediately denied.**

This is like the hotel receptionist checking your **ID and reservation** before giving you the room key.

👉 **Performance insight:**

- Strong authentication methods (like `scram-sha-256`) balance **security and speed.**
- Poorly configured authentication (like repeated DNS lookups in LDAP) can **slow down connection times.**

Step 4: Backend Takes Over

Once authentication succeeds:

- The backend process becomes the **personal waiter/chef** for that client.
- From now on, this backend:
- **Parses** SQL queries (checks syntax).
- **Plans** the execution (decides the best path, like using indexes).
- **Executes** the query (retrieves or modifies data).
- **Sends results** back to the client.

The backend will continue serving the client until the session ends.

👉 **Performance insight:**

- Optimizing queries (via indexes, caching, and query rewrites) makes backends faster.
- A slow query in one backend doesn't crash others, but it can **hog resources** like CPU or locks, affecting overall performance.

Step 5: Concurrency and Limits

This cycle repeats for **every client connection**.

- PostgreSQL uses the `max_connections` parameter to control how many clients can connect at the same time.
- Default: **100**.
- Beyond this limit, new clients see an error:

```
FATAL: sorry, too many clients already
```

- Unlike some databases that use threads, PostgreSQL uses **full OS processes** for each backend.
- **Advantage:** Stability and isolation — if one process crashes, others survive.
- **Disadvantage:** Heavy resource usage — too many processes can overload the OS.

👉 **Performance insight:**

- For high-traffic systems:
- Use a **connection pooler** (PgBouncer, Pgpool-II) to recycle connections.
- Lower `max_connections` to a safe level instead of raising it too high.

- Monitor memory and CPU usage closely, because each backend consumes resources.

Key Takeaways

- The PostgreSQL connection lifecycle is **step-by-step and resource-intensive.**
- Each client gets a dedicated backend process, ensuring stability and isolation.
- Performance tuning at the connection level focuses on:
- **Managing connection counts** (`max_connections`).
- **Using pooling** to avoid overhead.
- **Optimizing queries** so backends work efficiently.

PostgreSQL Performance Tuning: User Operations, the Daemon Process, and Real-World Insights

User Operations After Connection

Once a client successfully connects to PostgreSQL 17, the user can start working with the database. What they can do depends on their **role and permissions** (for example, read-only users cannot run `UPDATE` or `DELETE` queries).

The most common operations include:

`SELECT` → Fetch rows from a table or view.

- Example: `SELECT * FROM customers WHERE city = 'New York';`
- This reads data and returns results.

`INSERT` → Add new rows into a table.

- Example: `INSERT INTO orders (id, customer, amount) VALUES (101, 'Alice', 250);`

`UPDATE` → Modify existing rows.

- Example: `UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';`

`DELETE` → Remove rows.

- Example: `DELETE FROM sessions WHERE last_login < now() - interval '30 days';`

Behind the scenes, PostgreSQL doesn't just run the query directly. Instead, the **backend process** assigned to that user goes through several steps:

1. **Parsing**

- PostgreSQL checks if the SQL syntax is correct.
- Example: If you type `SELEC` instead of `SELECT`, parsing fails immediately.

2. **Planning/Optimization**

- PostgreSQL decides the **best way to execute the query**.
- Should it use an index? Should it scan the entire table? Should it join tables with a nested loop or a hash join?
- This is where the **query planner** comes into play.

3. **Execution**

- Once the plan is ready, PostgreSQL executes it.
- This could mean fetching rows, writing new rows, or updating values on disk.

4. **Result Delivery**

- Finally, the backend sends results back to the client — whether that's rows of data, a confirmation message ("1 row updated"), or an error.

👉 **Why this matters for performance:**

- If queries are poorly written or lack proper indexes, the **planning and execution stages** become slow.
- DBAs use tools like `EXPLAIN` and `EXPLAIN ANALYZE` to see query plans and optimize them.

Why the Daemon Process is Mandatory

The **daemon process** (also known as the **postmaster**) is the **master controller** of PostgreSQL. It must always be running for the database to function.

The daemon is responsible for:

- **Accepting new connections** from clients.
- **Spawning backend processes** for each connection.
- **Launching background workers** like autovacuum and checkpointer.
- **Managing shared memory and system-wide resources** used by all backends.

If the daemon stops:

- No new clients can connect.
- All existing backend processes are terminated.
- PostgreSQL is effectively **down**.

That's why PostgreSQL is always managed as a **system service**. For example:

```
systemctl start postgresql-17
```

In production environments, DBAs configure **automatic restart policies** so that if the daemon crashes for any reason, it restarts automatically without human intervention.

👉 **Key takeaway:** The daemon is like the **power switch** of PostgreSQL. If it's off, the entire system is unavailable.

Real-World Example

Let's put this into perspective with a practical scenario:

- Imagine **50 users** connecting to a PostgreSQL 17 database at the same time.
- The **daemon** listens for requests on the default port **5432**.
- Each connection request is accepted, and the daemon spawns a **new backend process** for each user.
- If `max_connections` is set to **100**, all 50 users are served simultaneously, with 50 backend processes running in parallel.

Now, what happens next?

- Each backend handles its client's queries independently. For example:
- User 1 runs a `SELECT` query on the `customers` table.
- User 2 runs an `UPDATE` query on the `orders` table.
- User 3 runs a complex join across multiple tables.
- Meanwhile, **background processes** like the **checkpointer** (writing dirty pages to disk) and **autovacuum** (cleaning dead tuples) keep the database stable.

From the **end-user's perspective**, it feels simple:

- *"I typed a query and got results."*

But from PostgreSQL's perspective, a **multi-process architecture** is working hard in the background to ensure stability, performance, and isolation.

Conclusion

The **client-server model** in PostgreSQL 17 is what makes it **reliable, secure, and scalable.**

- **Clients** → Any tool or application (e.g., psql, pgAdmin, Python apps) that issues SQL commands.
- **Daemon (postmaster)** → The master controller that initializes memory, launches workers, and spawns backends.
- **Backends** → Dedicated per-client processes that parse, plan, execute, and return results.
- **Background Workers** → Processes like checkpointer, autovacuum, stats collector, and logger that keep PostgreSQL healthy.

This architecture provides PostgreSQL with:

- **Isolation** → Each client has its own backend process.
- **Resilience** → The daemon ensures service continuity.
- **Scalability** → PostgreSQL supports many clients in parallel, up to configured limits.

👉 For DBAs and developers, understanding this architecture is not just theory — it's essential for **performance tuning, troubleshooting, and efficient system design.**

📢 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 AWS Cloud Computing Database

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

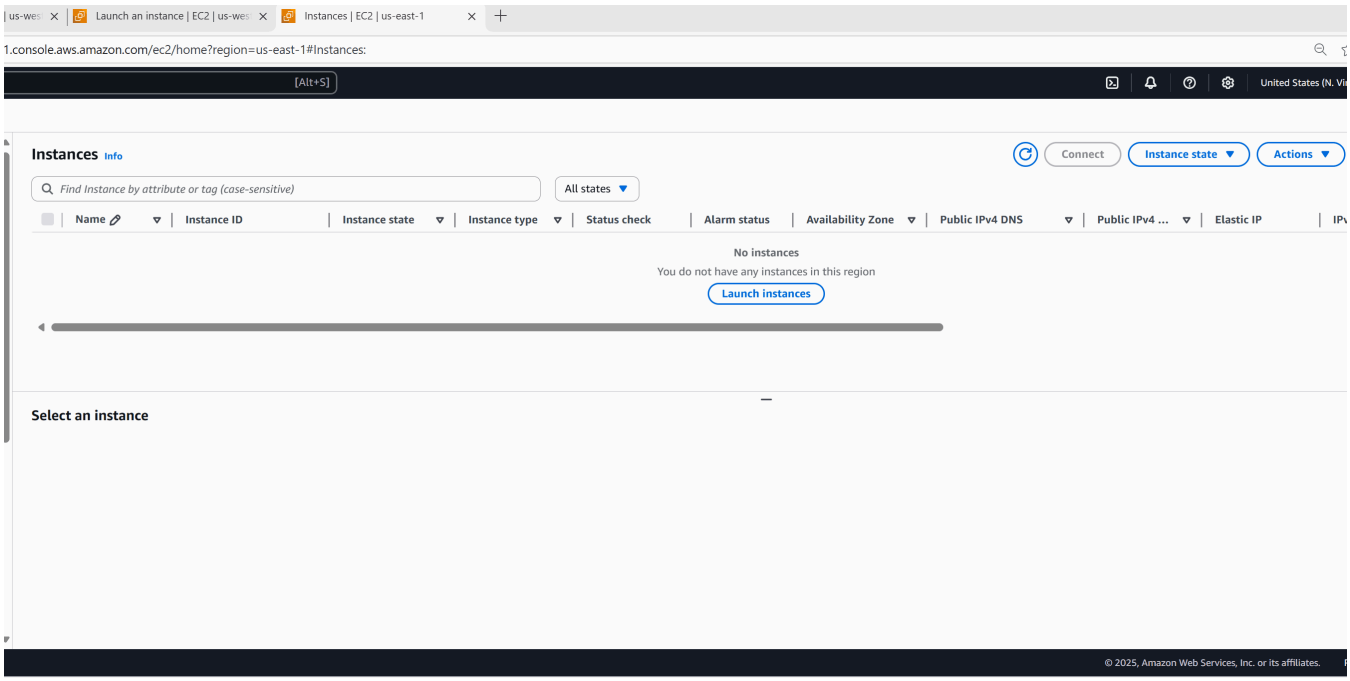
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

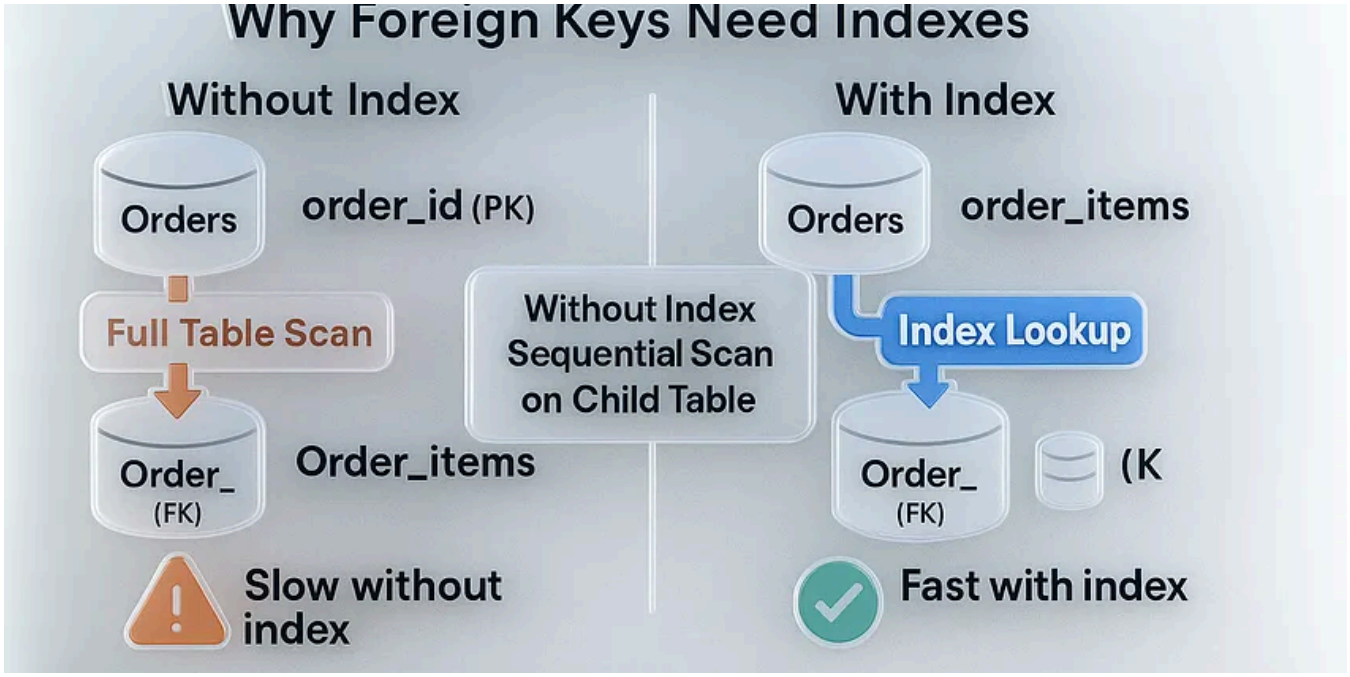


Jeyaram Ayyalusamy

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

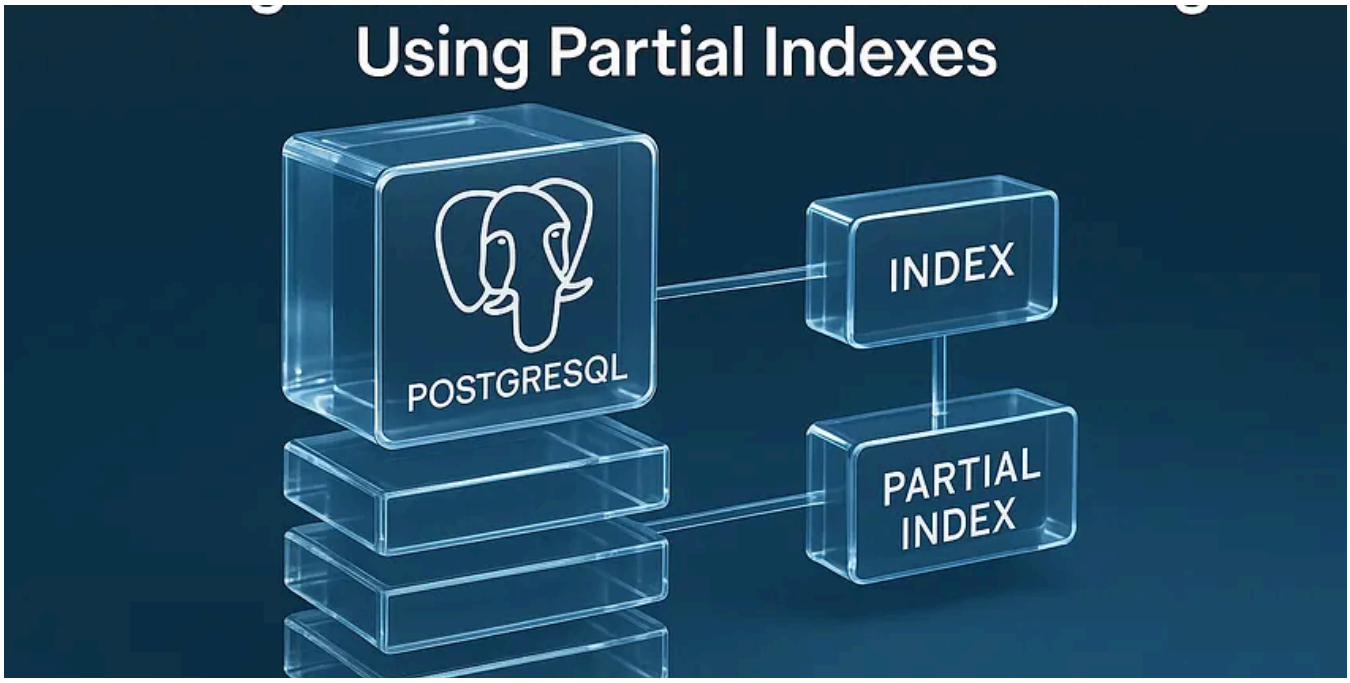


Jeyaram Ayyalusamy

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

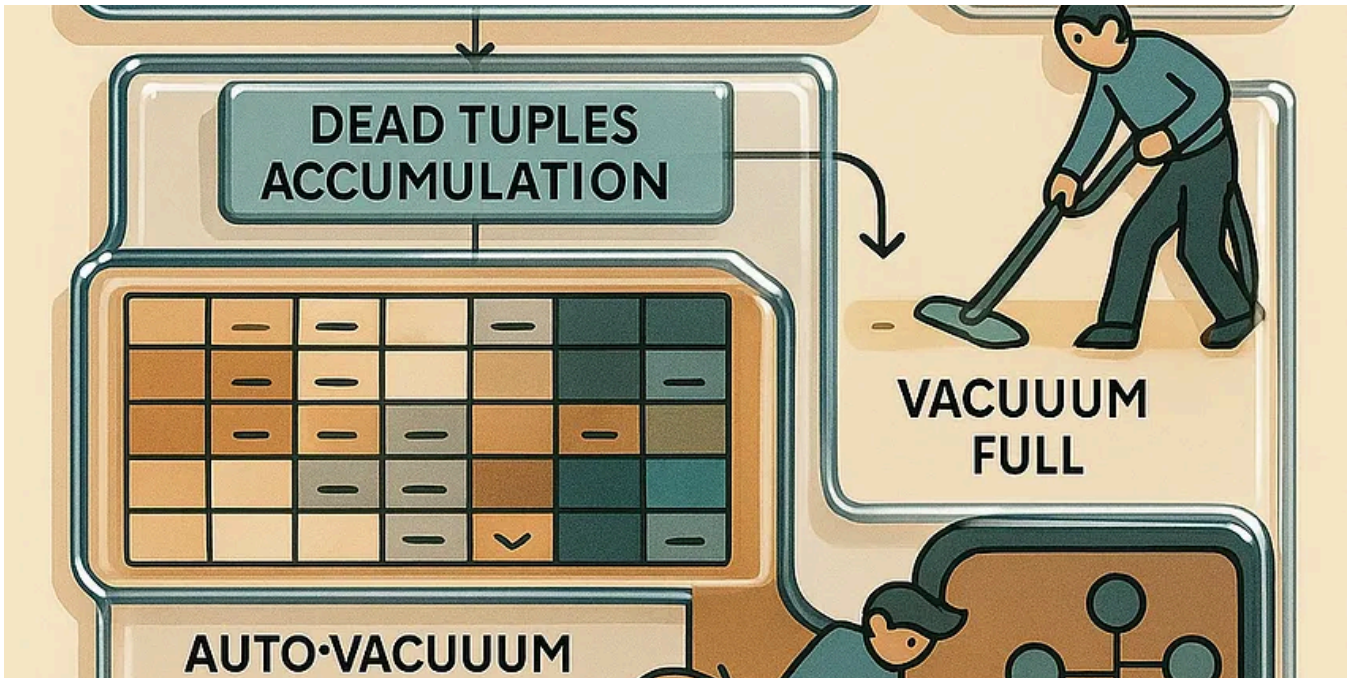


Jeyaram Ayyalusamy

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

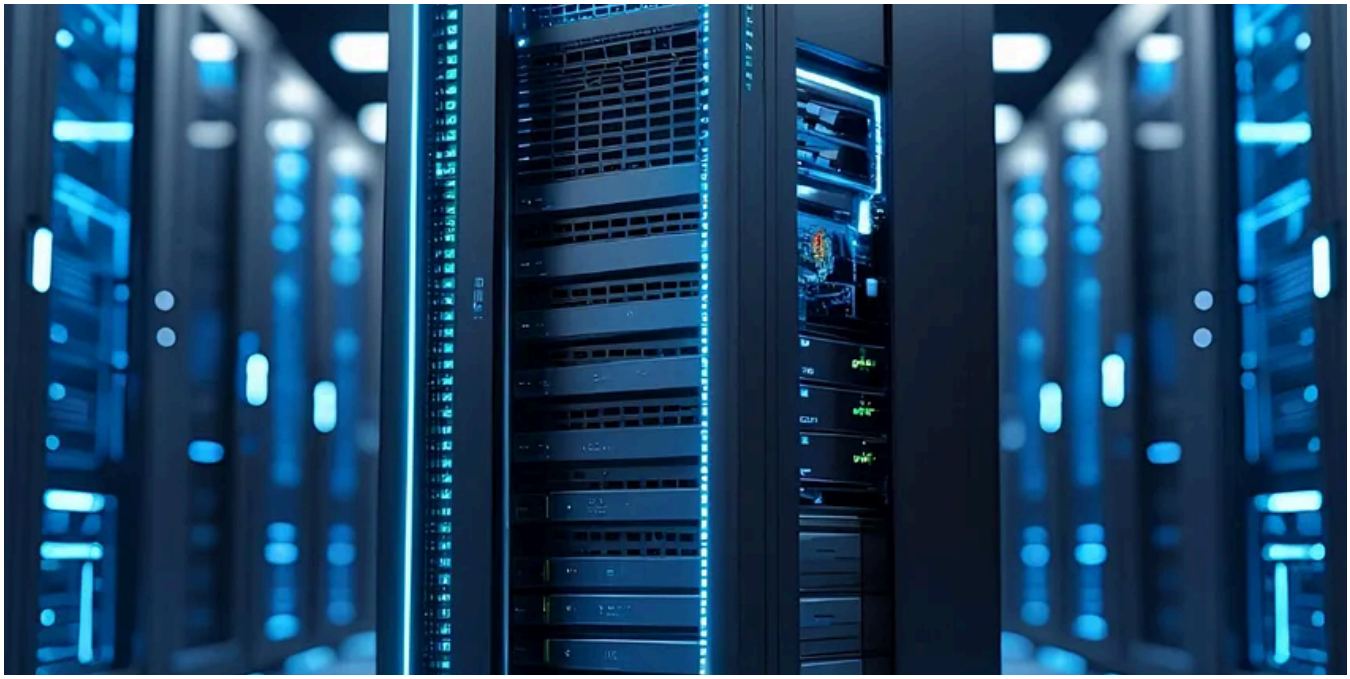
Recommended from Medium

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

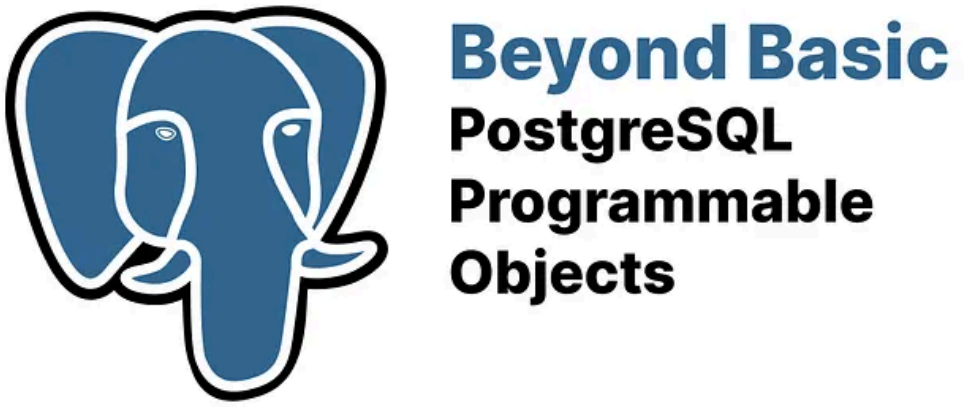


Rizqi Mulki

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



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

In CodeX by MayhemCode

The Secret Weapons That Separate Bash Beginners from Command Line Legends

Ever wondered how senior developers seem to have magical powers when it comes to bash? They write scripts that validate data, find patterns…

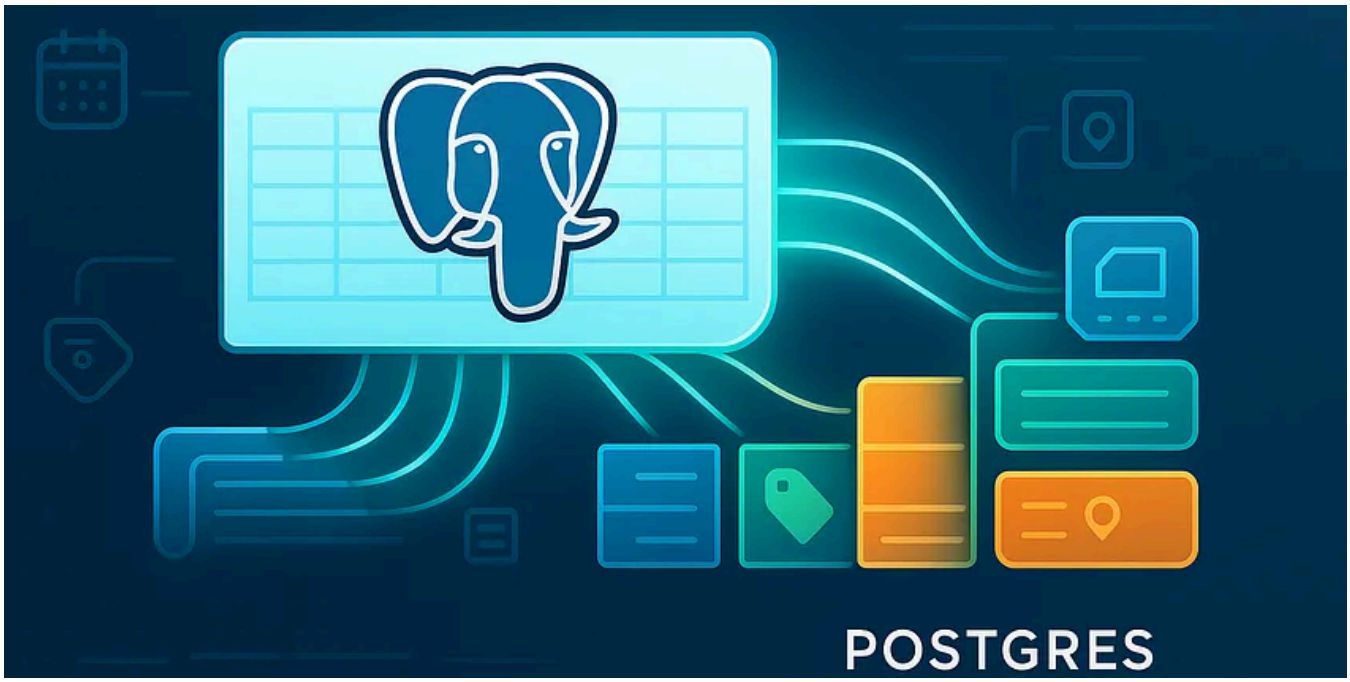
4d ago 7

Ajaymaurya

PostgreSQL 18 Features That Change Everything

When a new PostgreSQL release drops, the developer world always pauses. PostgreSQL 18 is no exception—it's bold, innovative, and packed…

Sep 14 19 2



Thinking Loop

10 Postgres Partitioning Patterns for Internet-Scale Apps

Keep Postgres fast past a billion rows with real patterns and ready-to-run SQL.

Aug 13 88 2