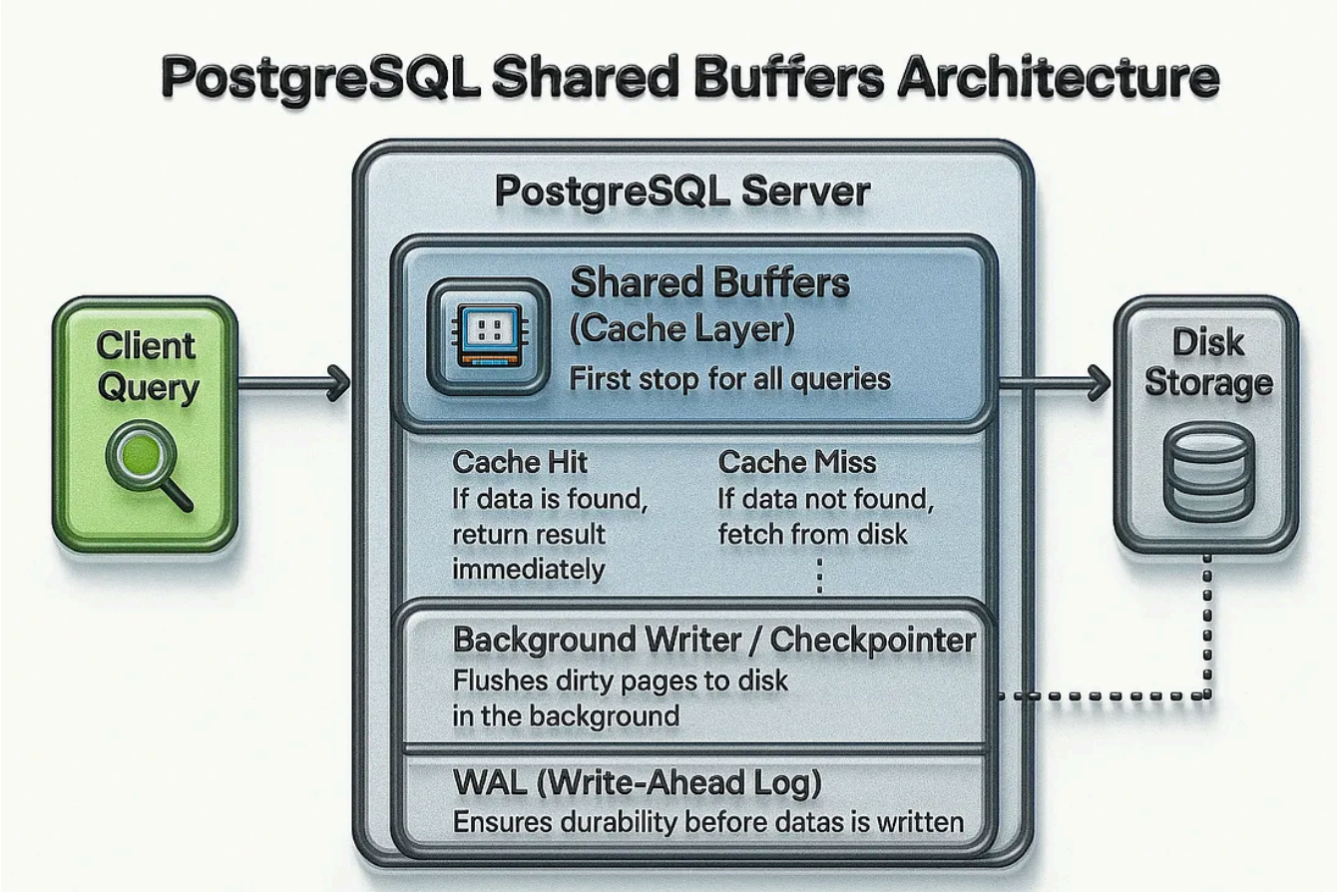
02 - PostgreSQL Performance Tuning: Understanding PostgreSQL Shared Buffers for Performance Tuning

7 min read · Aug 29, 2025

Jeyaram Ayyalusamy Following



When there are thousands of users trying to read or write data across many different tables, performance quickly becomes a challenge. Imagine a large application where hundreds or thousands of transactions are hitting the database

every second. If PostgreSQL were to directly read and write from physical files on disk for every query, the system would very quickly become overloaded.

This is because file operations are inherently slow and non-scalable. Each request would require PostgreSQL to search through the file system, open the right file, locate the specific data record inside it, apply locks to prevent conflicts, edit the data, and then unlock it. Every one of these steps adds latency. While this might be manageable for small workloads, it creates serious overhead at scale and cannot keep pace with modern applications that demand low-latency, high-throughput performance.

To solve this, PostgreSQL introduces the concept of **shared buffers**. Instead of having backend processes constantly interacting with disk files, they work with a dedicated block of memory (RAM) known as the **shared buffer cache**. By routing all reads and writes through this memory region, PostgreSQL avoids a large portion of slow disk I/O operations. The result is a dramatic improvement in performance, faster query responses, and a database engine that can scale to meet the demands of thousands of concurrent users.

How Shared Buffers Work

The amount of memory PostgreSQL sets aside for these shared buffers is controlled by the `shared_buffers` parameter in the configuration file (`postgresql.conf`). When the PostgreSQL server starts, it allocates a fixed-sized block of shared memory that acts as this buffer cache. Every query that comes into the system interacts with this buffer pool first rather than directly touching the disk.

Here's what this means in practice:

- When a query requests data, PostgreSQL first looks inside the buffer cache. If the requested data is already there, the system can return it immediately without hitting the disk.
- If the data isn't in the buffer cache, PostgreSQL will pull it from the disk and then store it in the buffer so that subsequent queries can access it quickly.
- Updates and writes also go through the buffer. PostgreSQL writes the changes to the buffer first and then eventually flushes them to disk in the background,

reducing the amount of time spent waiting on slow storage operations.

This caching mechanism gives PostgreSQL a huge performance advantage because queries are resolved in memory most of the time, rather than waiting for slower physical reads and writes.

The Role of Other Caches

It's important to understand that PostgreSQL doesn't work in isolation. While the shared buffer cache is extremely powerful, it is not the only caching layer that contributes to performance. The **operating system cache** plays a significant role as well.

The OS cache keeps frequently accessed files and blocks in memory at the operating system level, which means that even if PostgreSQL has to ask the OS for data, there's still a good chance that it will be served from memory rather than disk. This provides another opportunity to avoid costly physical I/O.

In modern systems, caching doesn't stop there. Many servers also use **hardware-level caches**, such as disk controller caches or drive-level caches. These layers all work together to minimize the number of times PostgreSQL actually needs to perform a true physical read or write from the storage device.

Why This Matters

The bottom line is simple: **physical I/O is the biggest bottleneck in database systems.** Memory is orders of magnitude faster than disk, and shared buffers, together with OS-level and hardware caches, are designed to take advantage of that fact. By serving most queries from memory, PostgreSQL reduces latency, increases throughput, and delivers the performance needed for demanding workloads.

Without these caching layers, every query would involve repeated file system operations, and PostgreSQL would not be able to handle the scale of modern applications. With them, most queries are served in microseconds rather than milliseconds, creating a more responsive and scalable system.

✨ In short, **shared buffers are at the heart of PostgreSQL performance tuning.** They provide a crucial layer of memory-based optimization, ensuring that most queries never have to touch the disk at all. Combined with the operating system and hardware caches, they allow PostgreSQL to handle thousands of concurrent operations efficiently while minimizing one of the biggest performance drains: physical I/O.

PostgreSQL Performance Tuning: Large Cache Risks and How Queries Travel Through Caches

The Risks of Large Caches

Caches are essential for speeding up PostgreSQL because they keep frequently used data in memory, which is much faster than reading it from disk. However, bigger does not always mean better. Allocating too much memory to the database cache can create its own set of problems.

Here's why:

When PostgreSQL stores data changes in its shared buffer cache, those changes eventually need to be **written back to disk.** If the cache is very large, a lot of data can pile up before it's flushed. When the time comes to write that data out, PostgreSQL may suddenly push a huge amount of I/O to the disk all at once.

This results in **I/O spikes** — sudden bursts of disk activity that can temporarily slow down the system. Applications may notice longer response times during these flushes, even if the database had been running smoothly just moments before.

So, tuning the `shared_buffers` parameter becomes a balancing act:

- **Too small:** PostgreSQL can't take full advantage of memory, and queries will hit the disk more often, making them slower.
- **Too large:** PostgreSQL risks unpredictable slowdowns whenever a large batch of cached data has to be flushed to disk.

👉 The best performance comes from finding the right middle ground, where PostgreSQL makes effective use of memory without creating dangerous spikes in I/O.

Example: How a Simple SELECT Travels Through Caches

To see how this works in practice, let's take a simple example. Suppose you run the following query:

```
SELECT * FROM orders WHERE order_id = 5001;
```

Behind the scenes, PostgreSQL doesn't immediately go to the disk to find this row. Instead, it follows a smart, layered approach to minimize slow I/O.

Step 1: Check the Database Buffer Cache

The first stop is PostgreSQL's **shared buffer cache**. This is memory space inside PostgreSQL that stores recently accessed or frequently used rows and blocks of data.

- ✅ If the data is already here, PostgreSQL can return it instantly.
- ⏱️ No disk access is needed, which makes this the fastest possible outcome.

Think of this like checking your desk for a paper you're working on. If it's sitting right there, you don't need to go anywhere else.

Step 2: Check the Operating System Cache

If the requested row isn't in PostgreSQL's own buffer, the next place to check is the **operating system cache**. The OS often keeps copies of recently used files and blocks in memory.

- ✅ If the OS cache has the file, it quickly hands it over to PostgreSQL.
- ⏱️ This is still very fast and avoids going to the disk.

This is like asking your coworker if they already picked up the document you need. If they have it, you save yourself a trip to the filing cabinet.

Step 3: Perform Physical I/O (Disk Read)

If the data is missing from both caches, PostgreSQL has no choice but to go to the **disk**.

- ⚠️ This is the slowest option because disk operations take much longer than memory.
- ⏱️ PostgreSQL must open the file, locate the block, read it, and then return it to the query.

This is like walking to the filing room, opening a cabinet, and searching through folders. It works, but it's much slower than grabbing it from your desk or a coworker.

Why This Matters in Real Workloads

In real-world databases, most queries are served from **caches, not disk**. This is why PostgreSQL can handle thousands of queries per second.

- If many queries focus on the same tables or rows (for example, checking the latest orders in an e-commerce app), PostgreSQL can serve them directly from memory most of the time.
- Even under heavy load, physical disk I/O stays limited as long as queries are concentrated on similar datasets.

However, if queries are spread out across very different tables or indexes, PostgreSQL has to keep pulling in new data and flushing out old data from the cache. This leads to more frequent disk flushes, higher I/O, and reduced performance.

✅ **In summary:**

- Caches are powerful but must be sized carefully — too small wastes memory, too large creates dangerous I/O spikes.
- A `SELECT` query doesn't always hit the disk. It first checks PostgreSQL's buffer cache, then the operating system cache, and only if both are empty does it perform a slow physical read.
- Most real workloads benefit from caching, but scattered queries across different datasets can still cause frequent disk flushes.

Final Thoughts

PostgreSQL shared buffers are central to performance tuning. By reducing direct disk I/O and leveraging multiple levels of caching, PostgreSQL ensures faster query responses and greater scalability. However, effective tuning requires striking the right balance: enough buffer space to handle workload patterns efficiently, but not so much that cache flushes overwhelm the system.

📢 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 | Cloud Computing | Oracle | AWS | Database

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

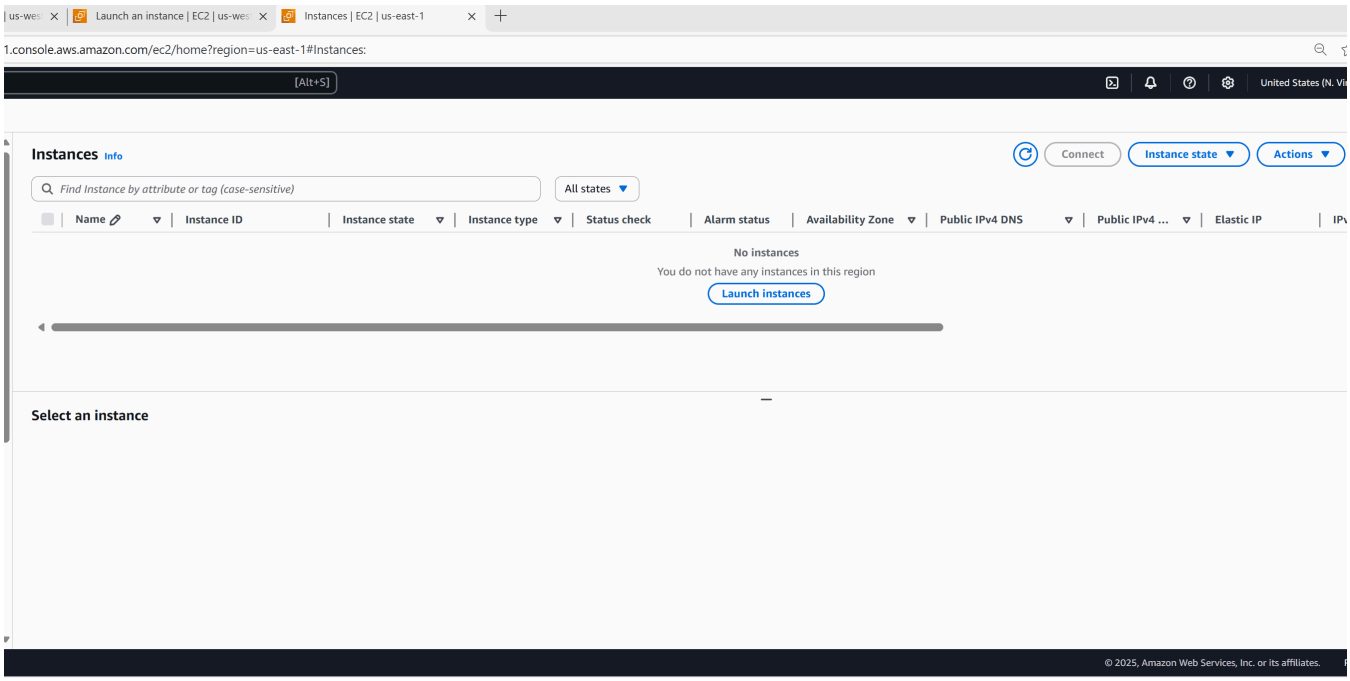
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

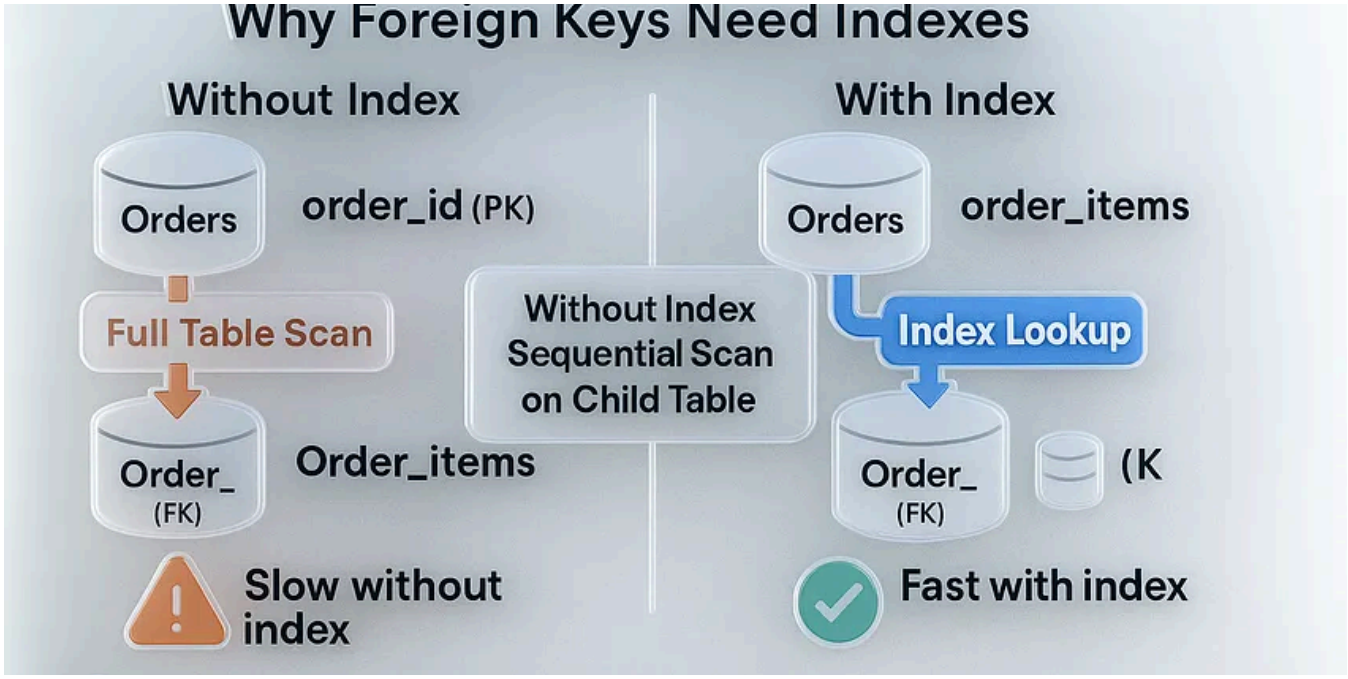


Jeyaram Ayyalusamy

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

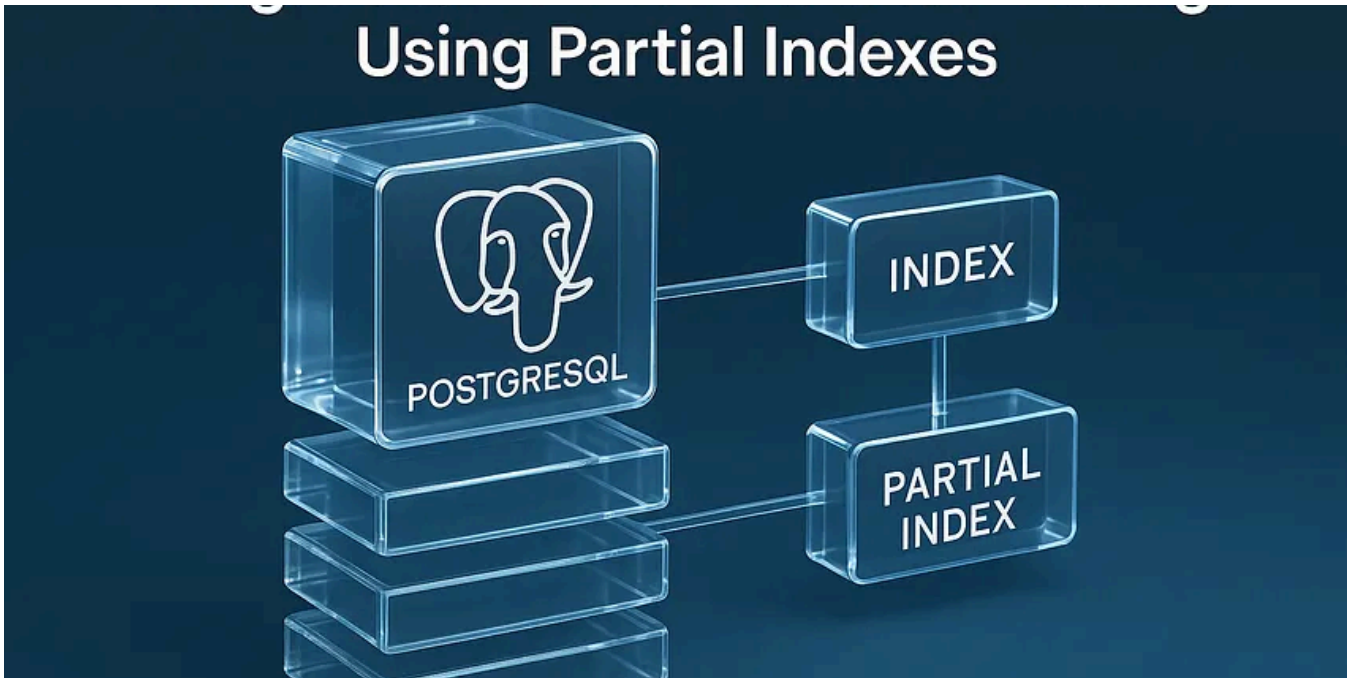


Jeyaram Ayyalusamy

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

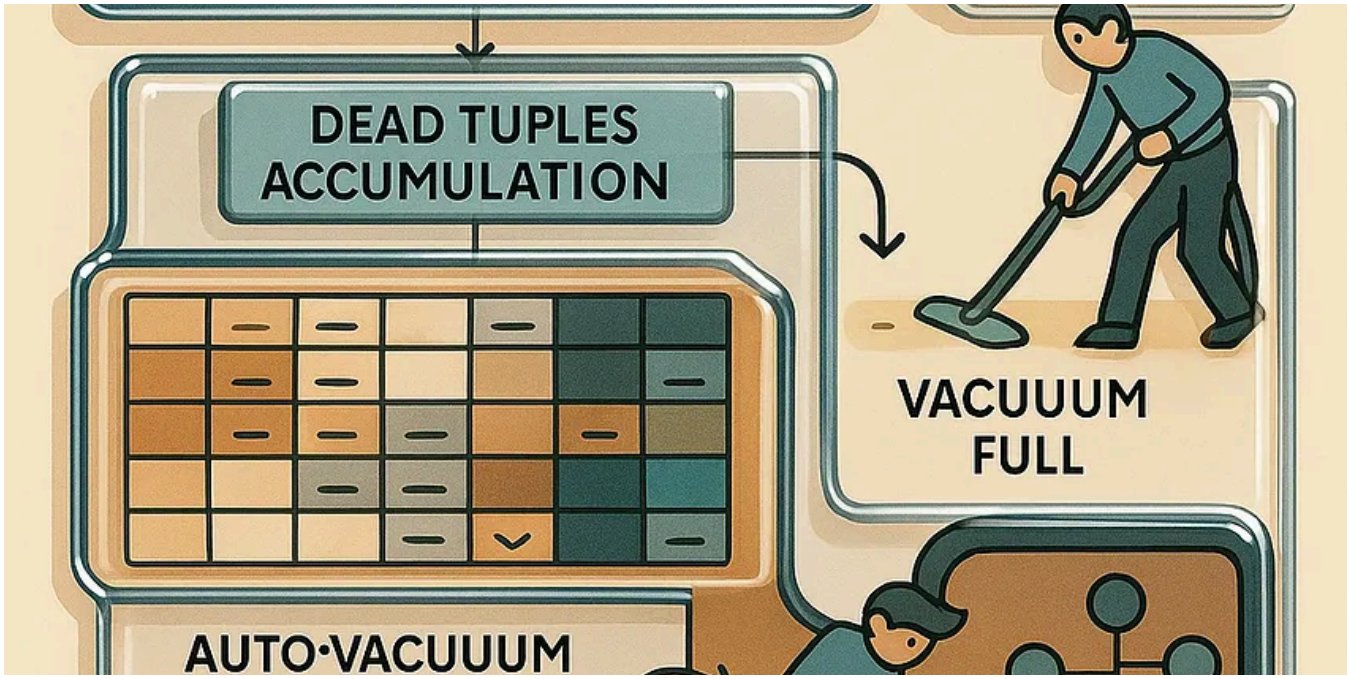


Jeyaram Ayyalusamy

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

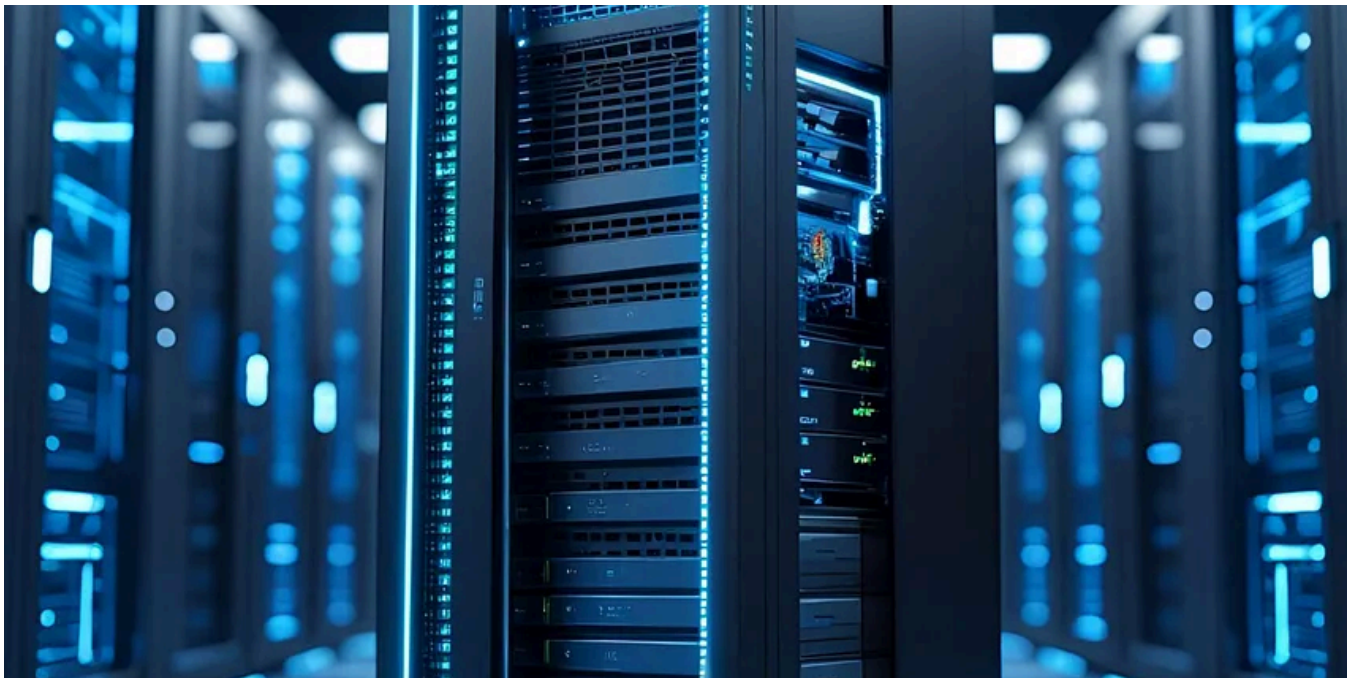
Recommended from Medium

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

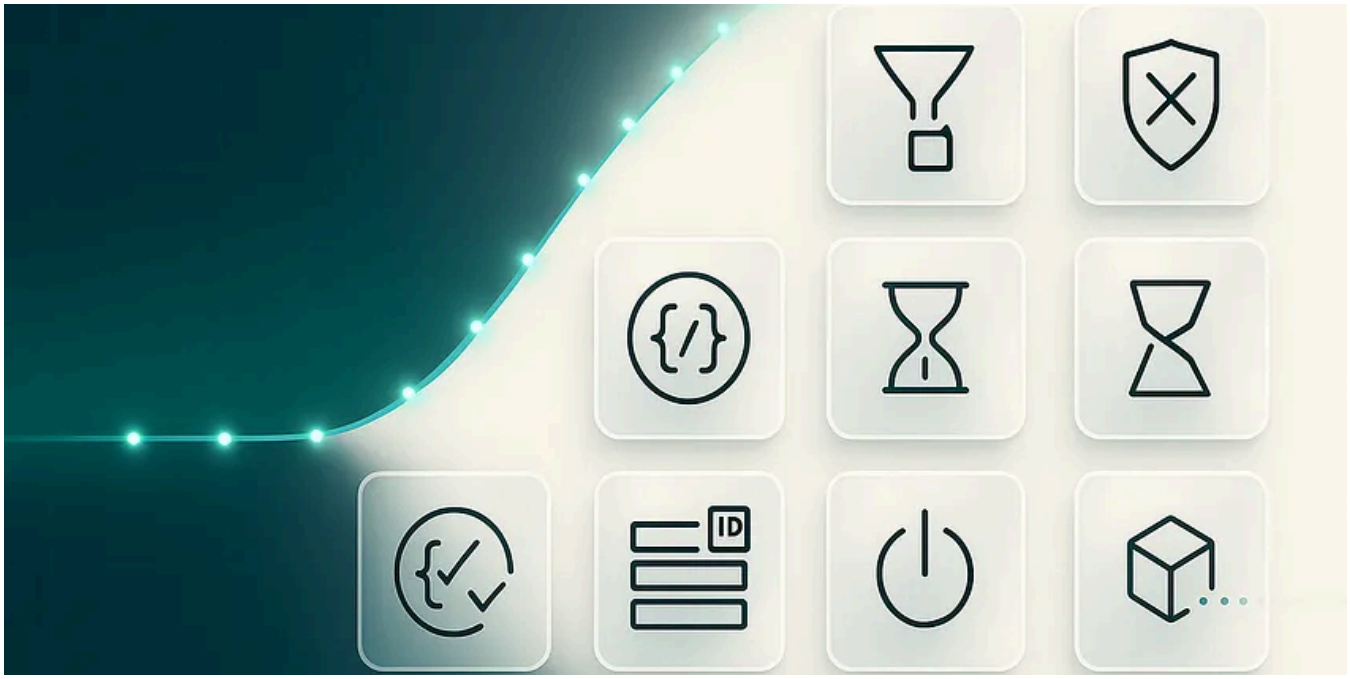


Rizqi Mulki

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



Hash Block

10 Node.js Error-Handling Tricks That Saved Me

Practical patterns to catch bugs early, keep services alive, and turn chaos into readable logs.

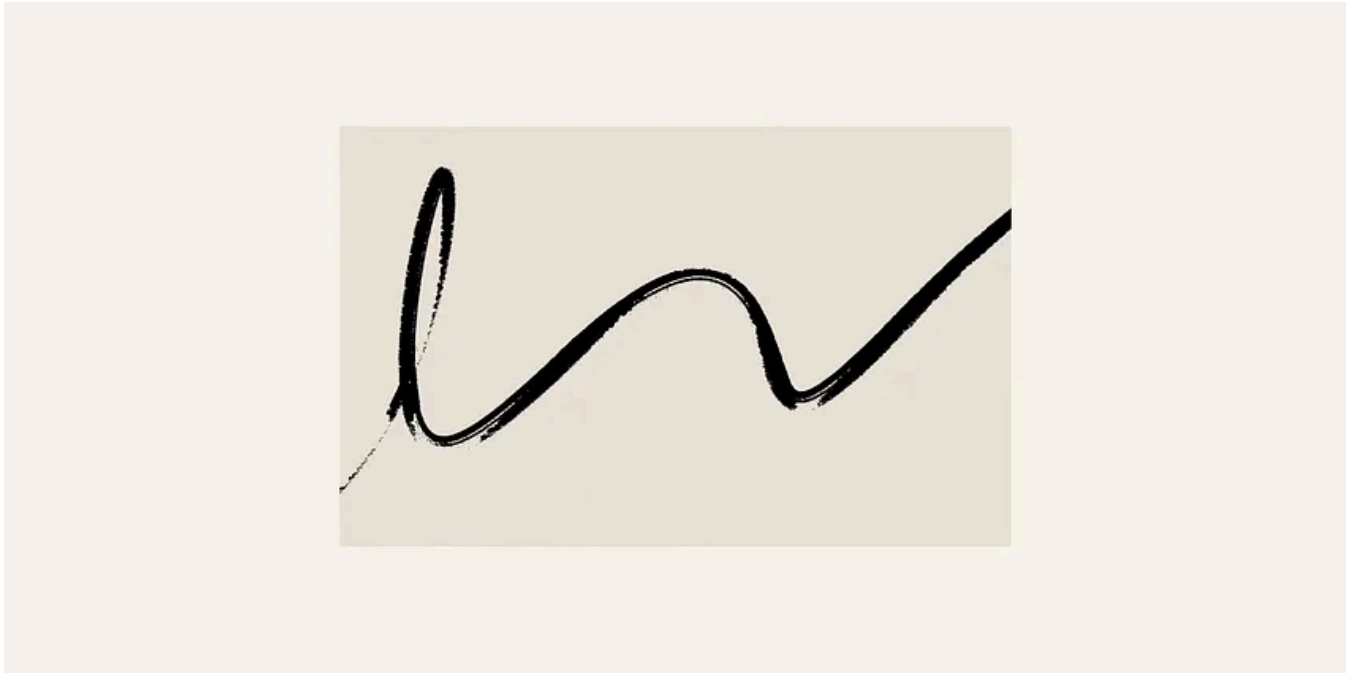
4d ago 17

Thread Whisperer

Postgres 18 Arrives: Async I/O You Should Turn On First

Turn disk waits into throughput with a few safe switches

Sep 15 77



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

Thinking Loop

5 DuckDB–Arrow–Polars Workflows in Minutes

Turn day-long pipelines into small, local, reproducible runs without clusters or drama.

5d ago 14

See more recommendations