PostgreSQL MVCC: The Secret Behind Its Powerful Concurrency

12 min read · Jun 11, 2025

J Jeyaram Ayyalusamy Followin

Listen Share ••• Mo

J Following

Jeyaram Ayyalusamy

76 followers

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

Book Author

PostgreSQL is renowned for its performance, consistency, and concurrency. But at the core of this magic lies one of its most important — yet often misunderstood — features: **MVCC (Multi-Version Concurrency Control).**

In this post, we'll dive deep into how MVCC works in PostgreSQL, why it matters, its benefits, limitations, and how you can leverage it to build scalable, high-performance systems.

🚀 What is MVCC in PostgreSQL?

In modern database systems, ensuring high performance and consistency — especially in multi-user environments — is a complex challenge. How do we allow multiple users to **read and write to the same data** without causing delays, blocks, or conflicts?

Enter **MVCC** — *Multi-Version Concu* makes PostgreSQL one of the most robust day.

🔍 The Problem: Locking and

In traditional databases, concurrer or example:

- A user writing to a record locks it.
- A user reading a record may bl

While this guarantees consistency, it comes at a cost:

- Slower performance under heavy loads
- Users waiting for locks to be released
- Risk of deadlocks and contention

This model doesn't scale well for high-traffic systems or real-time analytics where data is constantly read and modified.

🚀 The MVCC Solution

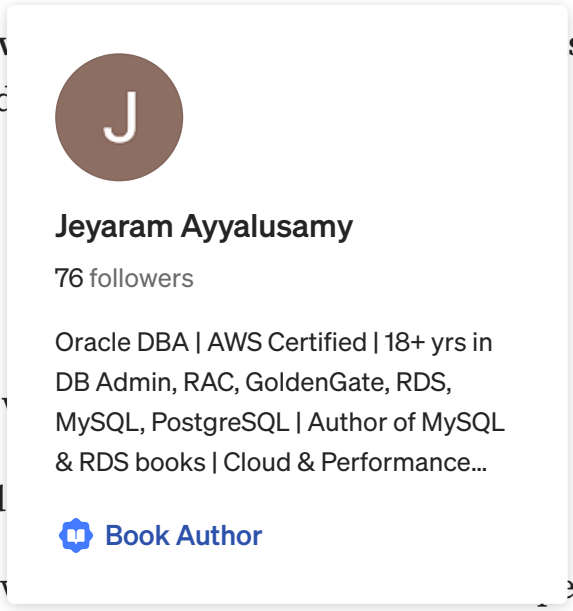
MVCC (Multi-Version Concurrency Control) solves this by taking a completely different approach:

> Instead of locking rows, PostgreSQL **creates a new version of a row** for every data change.

This allows:

- Multiple transactions to **read and write simultaneously**
- Each transaction to see a **consistent snapshot** of the database
- Readers to **never block writers**, and vice versa

In other words, **your read doesn't v** **sh** — and your write doesn't stop others from read



🧠 How It Works (Simplified)

Let's say two users are interacting v

1. **User A** starts a transaction and
2. **User B** updates that same row v en.
3. PostgreSQL **doesn't overwrite the original row** — instead, it creates a **new version** of it.
4. User A continues to see the original version (the one from when their transaction started).
5. User B sees their updated version.

Each transaction operates as if it has a **private, isolated view** of the data — this is what we call a **snapshot.**

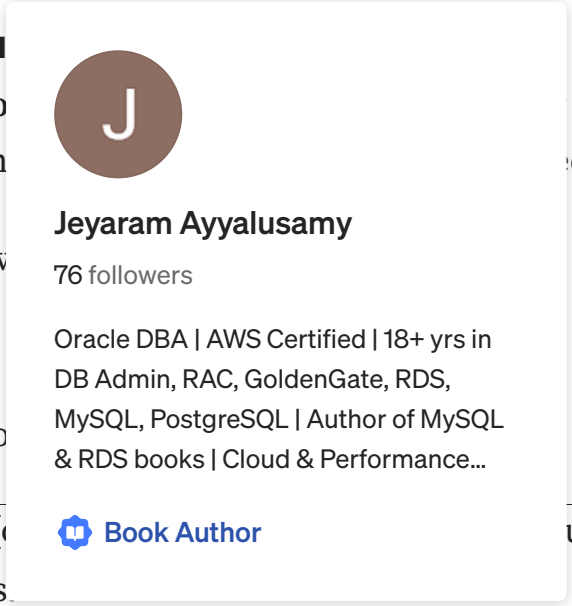
🔄 Real-World Benefits of MVCC

- ✅ **High concurrency:** Multiple users can work without interfering with each other.
- ✅ **Fewer locks:** Reduces contention and improves throughput.
- ✅ **Better performance:** Ideal for web apps, OLAP systems, and high-read environments.
- ✅ **Transactional consistency:** Each transaction sees a consistent view of data throughout its execution.

🧹 Behind the Scenes: Cleanu

Because MVCC creates **multiple ro** need to be cleaned up. PostgreSQL handles th d `VACUUM`.

- It **removes dead tuples** (older v
- Frees up space
- Maintains table performance o



> ⚠️ If `VACUUM` is not run regularly (ur tables can bloat — which slows down queries.

🏁 Summary

MVCC is the secret sauce that allows PostgreSQL to handle massive workloads without sacrificing performance or correctness. By managing **multiple versions of data** behind the scenes, it allows:

- Writers to update data without blocking readers
- Readers to query data without being blocked by writers
- Every user to see a stable, consistent snapshot of the database

> 💡 In a multi-user world, MVCC is not just a feature — it's a necessity.

🧭 MVCC Lifecycle in PostgreSQL: How Transactions Really Work

PostgreSQL's **MVCC (Multi-Version Concurrency Control)** isn't just a clever feature — it's the core of how the database handles concurrent transactions with safety, speed, and consistency.

But what does a transaction look like under the hood when MVCC is in action?

In this article, we'll walk through the **life of a transaction** using MVCC, step by step.

1️⃣ Transaction Starts

Every time a transaction begins in identifier called a **Transaction ID (XID)**.


```
BEGIN;
```

At this moment, the database engi database state. This snapshot includes:

- Which rows are visible
- Which transactions are active or completed
- Which changes should be ignored

> 💡 This snapshot remains consistent throughout the life of the transaction — regardless of what others are doing at the same time.

2️⃣ Reading Data

When your transaction reads data, PostgreSQL doesn't just read rows blindly. It checks two hidden system columns on each row: `xmin` and `xmax`.

- `xmin` : the XID that **created** the row
- `xmax` : the XID that **deleted or updated** the row (if any)

Visibility Rules:

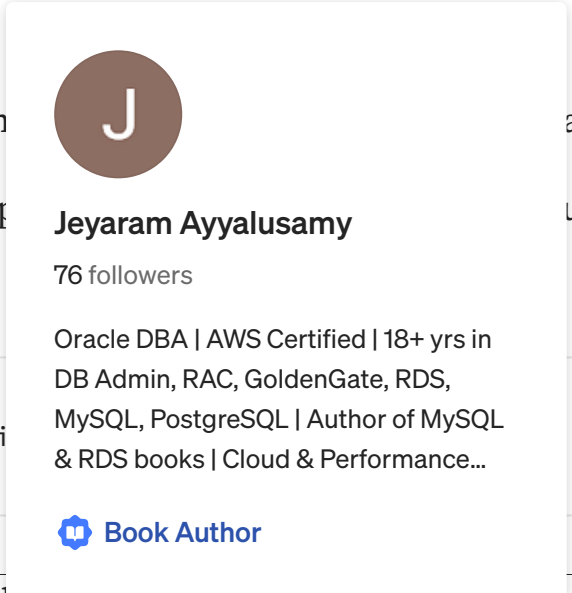
A row is **visible** to your transaction if:

```
xmin <= your_transaction_id
AND
(xmax IS NULL OR xmax > your_transaction_id)
```

This ensures:

- You only see data that was com an
- You don't see rows that were u ur transaction started

```
SELECT * FROM my_table WHERE i
```



> 🔍 Each transaction sees the database as if it's frozen in time — this is MVCC in action.

3️⃣ Modifying Data

When you modify data, PostgreSQL doesn't update the row in-place. Instead, it **creates a new version** of the row.

Here's what happens internally during an update:

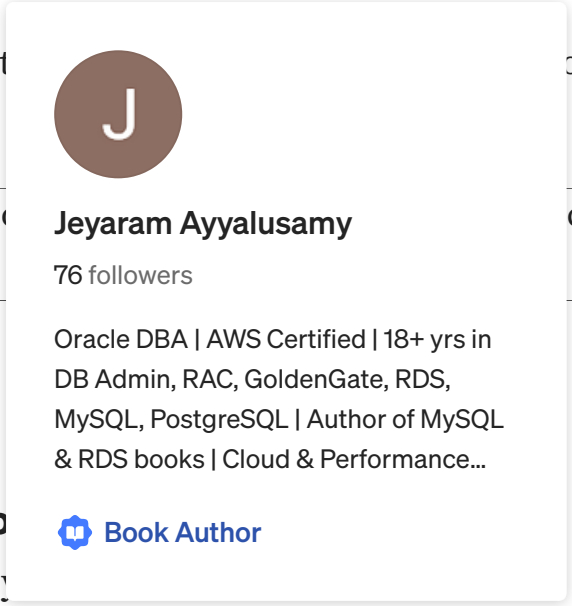
```
UPDATE my_table SET value = 'new_value' WHERE id = 1;
```

- The **existing row's** `xmax` is set to your current transaction ID → marking it as obsolete.
- A **new row** is created with:
- the updated value
- a new `xmin` (your transaction ID)
- a `NULL xmax` (means the row is still active)

This approach means:

- Old versions remain in the database but are **invisible** to new transactions
- Readers can continue reading t on is working on the new one

> 🔄 PostgreSQL is essentially versi cation ever knowing it.



4️⃣ Committing the Transactio

Once all operations are complete, y

```
COMMIT;
```

At this point:

- All your changes become **visible to future transactions**
- PostgreSQL records the transaction as complete
- Other users can now see the rows your transaction added or modified (if they start new transactions after this point)

> 🔒 Importantly, until you commit, **no other transaction sees your changes** — ensuring transactional isolation.

5️⃣ Visibility to Other Transactions

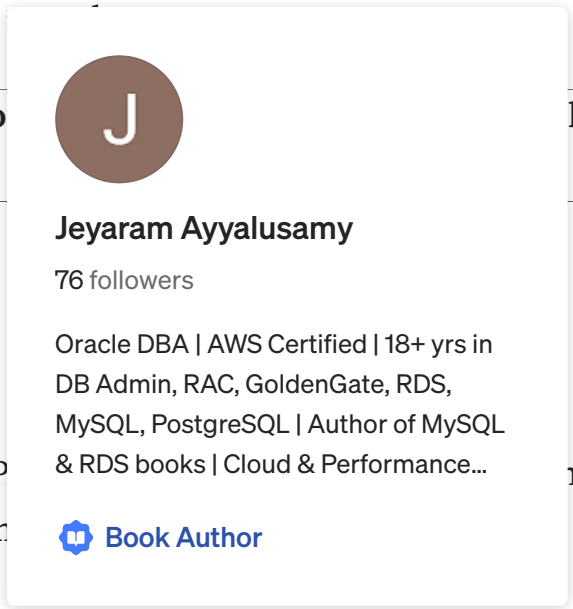
Each transaction operates based on the **snapshot** it took when it began. That means:

- If your transaction started **before** someone else committed their change, you won't see that change.
- If your transaction starts **after** a commit, you'll see the latest committed version.

Summary of Who Sees What:

Transaction A Transaction B Visibility Starts first Updates row A sees old data Commits Starts after commit Sees

> 🧠 MVCC ensures that **readers do** **block readers** — a win-win for performance and



🏁 Final Thoughts

The **MVCC lifecycle** is what gives P **n-concurrency** environments with ease. By walkin

- You understand how PostgreSQL keeps **transactions isolated**
- You see why updates don't block reads
- You appreciate the internal machinery — `xmin`, `xmax`, and snapshots — that make it all possible

> 🔬 Underneath your simple SQL commands lies a powerful versioning engine — and that's the magic of PostgreSQL's MVCC.

💡 Why MVCC is Brilliant in PostgreSQL: Unlocking the Power of True Concurrency

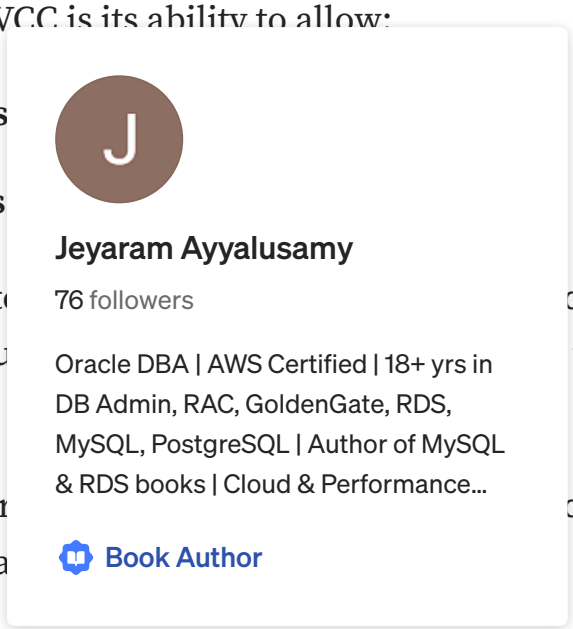
PostgreSQL's **Multi-Version Concurrency Control (MVCC)** is one of its most powerful and elegant features — and it's often what sets PostgreSQL apart from other relational databases.

MVCC enables **high performance, consistency, and scalability** in transactional systems without sacrificing concurrency. Here's a closer look at *why it's brilliant* and how it benefits real-world applications:

✅ True Concurrency

One of the standout features of MVCC is its ability to allow:

- **Reads without blocking writes**
- **Writes without blocking reads**



In many databases, reads and writ down performance under heavy load. Bu that.

🧠 How?

PostgreSQL maintains multiple ver ds data, they see a snapshot from when their tra are writing at the same time.

This makes PostgreSQL a top choice for **highly concurrent systems**, like financial apps, e-commerce platforms, and real-time analytics engines.

✅ Consistent Snapshots

Every transaction in PostgreSQL using MVCC sees a **stable and isolated view** of the database for the entire duration of that transaction. That means:

- No **dirty reads** (reading uncommitted data)
- No **non-repeatable reads** (seeing different data on repeated reads)
- No **phantom reads** (new rows appearing in a re-executed query)

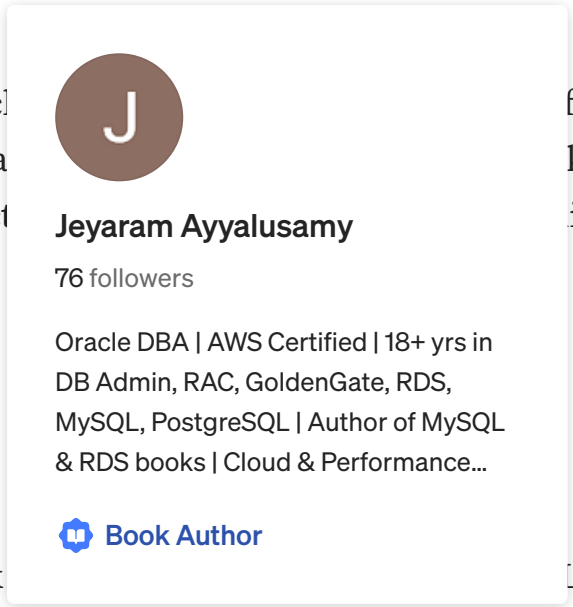
This snapshot consistency gives developers peace of mind — they can trust the data they query will stay consistent until their transaction ends.

✅ High Availability

MVCC plays nicely with:

- **Hot standby nodes**
- **Streaming replication**
- **Real-time read replicas**

Because read operations never loc fficiently without interfering with the prima kes MVCC ideal for **high availability architect** ity and disaster recovery.



✅ Isolation Made Easy

MVCC simplifies the complex task L provides all the standard SQL isolation levels (like *Read Committed* and *Serializable*) **without relying on heavyweight locking.**

This lets developers build complex transactional logic without worrying about lock contention or performance bottlenecks.

✅ Zero Read Locks

In PostgreSQL with MVCC, **read operations never acquire locks on rows.** That's a huge win for systems where read performance is critical.

📊 Whether you're running reporting dashboards, machine learning feature pipelines, or user-facing queries, the lack of read locks keeps latency low and throughput high.

🚀 Final Thoughts

MVCC isn't just a technical detail — it's a cornerstone of PostgreSQL's performance, reliability, and developer-friendly nature. By avoiding locking wherever possible and ensuring consistent, isolated views of the data, MVCC empowers PostgreSQL to scale effortlessly across modern workloads.

If you're building apps that need **high concurrency**, **data integrity**, and **scalability**, then MVCC is the reason PostgreSQL should be at the top of your stack.

⚠️ The Trade-Offs: Understan greSQL

While PostgreSQL's **Multi-Version** s impressive benefits like true concurrency, sna ility, it doesn't come without trade-offs.

Just like any powerful system, MV base administrators must understand a ions and how to mitigate them effectively.

1️⃣ Storage Overhead (Table Bloat)

Every time a row is updated or deleted in PostgreSQL, MVCC doesn't overwrite it. Instead, it creates a **new version** of the row and leaves the old version behind — still stored on disk.

🔍 Why?

This is what enables transactions to read a consistent snapshot — they can access the version that was valid when they began.

📦 The downside?

Over time, this leads to **table bloat**: unnecessary storage used by outdated row versions. Bloat can degrade performance by increasing disk usage, index size, and scan times.

2️⃣ Vacuuming is Essential

PostgreSQL doesn't automatically remove the old row versions left behind by MVCC. That's where the **VACUUM** process comes in.

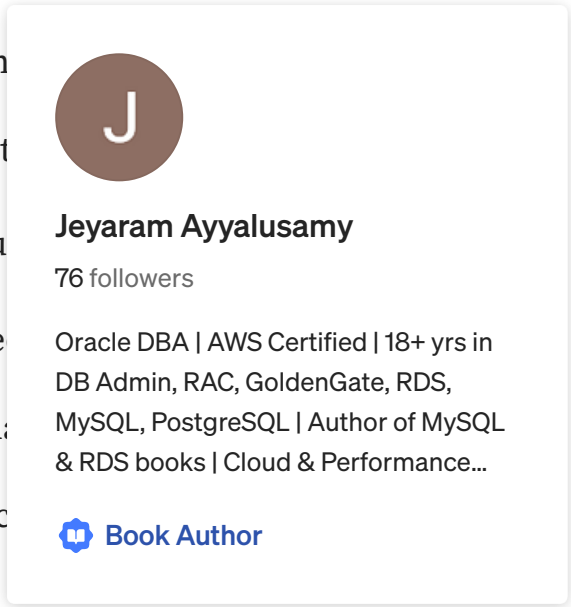
```
VACUUM my_table;
```

🧹 **What it does:**

- Identifies obsolete row version
- Marks space as reusable for fut

🚨 **Without regular VACUUM**, you

- Continue growing in size unne
- Suffer from degraded perform
- Risk transaction ID issues (disc



💡 **Solution:**

Enable and monitor **autovacuum**, PostgreSQL's background process that automatically vacuums tables based on activity and thresholds.

3️⃣ Transaction ID Wraparound

PostgreSQL assigns a unique **Transaction ID (XID)** to every transaction — but these IDs are 32-bit integers. That means:

- Only around **2 billion** transaction IDs are available before wraparound occurs.
- Once it wraps, PostgreSQL could **recycle IDs** and think an old row version is still active — a dangerous situation that can lead to **data corruption**.

🛡️ **PostgreSQL's safeguard:**

It uses a process called **freezing**, which marks old row versions as permanently visible and no longer tied to an XID.

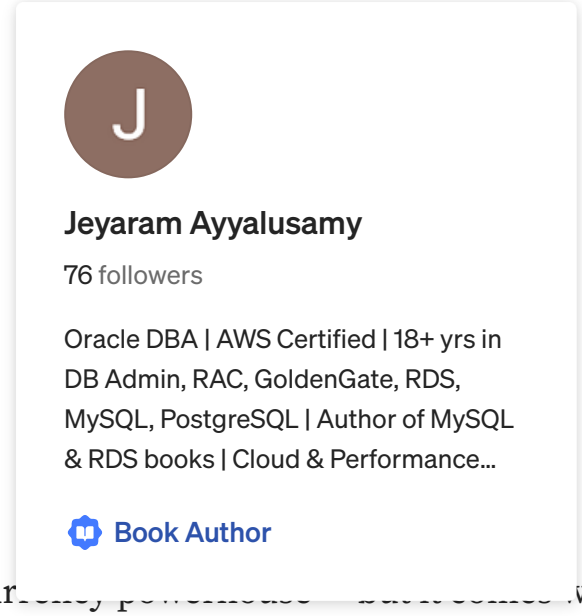
You can trigger it manually:

```
VACUUM FREEZE my_table;
```

Or rely on autovacuum to do it automatically — **if configured properly**.

🔧 Important Parameters:

- `autovacuum_freeze_max_age`
- `vacuum_freeze_min_age`
- `vacuum_freeze_table_age`



🧠 Final Thoughts

MVCC makes PostgreSQL a concur[illegible] with a **maintenance cost.** Ignoring vacuuming or not understanding transaction ID wraparound can lead to serious issues over time.

📌 To get the best from PostgreSQL:

- Regularly monitor table bloat
- Ensure autovacuum is running effectively
- Educate your team on the importance of storage management

MVCC is brilliant — but like all brilliance, it demands a bit of respect.

🔧 Monitoring MVCC in PostgreSQL: Stay Ahead of Bloat and Wraparound

PostgreSQL's MVCC (**Multi-Version Concurrency Control**) delivers excellent performance and concurrency, but as discussed earlier, it requires careful maintenance — especially around **dead tuples**, **autovacuum activity**, and **transaction ID age**.

To keep your database healthy and performant, you should regularly **monitor MVCC-related metrics**. Fortunately, PostgreSQL offers several powerful system views to help you do just that. Below are some essential queries and what they tell you.

📊 1. Check for Dead Tuples (F

```
SELECT relname, n_dead_tup
FROM pg_stat_user_tables;
```

🧠 Explanation:

This query shows how many **dead tuples** (obsolete row versions) exist in each user table. These are rows that were updated or deleted and are no longer visible to any active transaction.

⚠️ Why it matters:

A high count of `n_dead_tup` indicates potential **table bloat** and suggests the need for a `VACUUM`.

🔍 **Tip:** Monitor trends over time — sudden spikes may signal heavy update/delete activity or autovacuum falling behind.

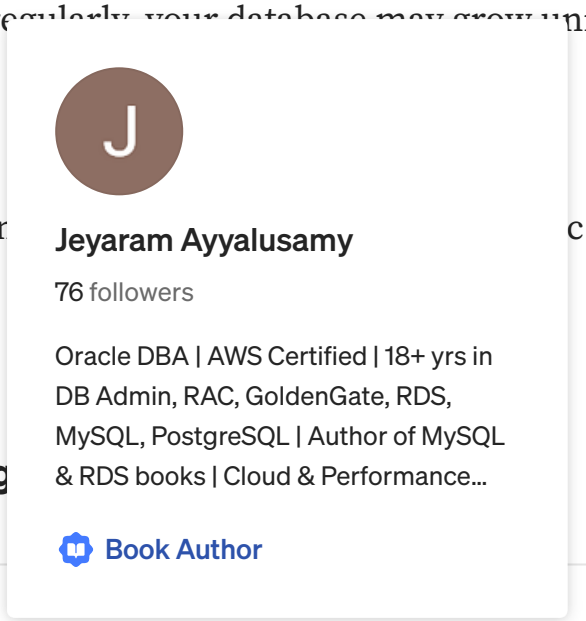
🚀 2. View Autovacuum Activity

```
SELECT relname, last_autovacuum
FROM pg_stat_user_tables;
```

🧠 **Explanation:**
This query shows the **last time autovacuum ran** on each table. If you notice tables with many dead tuples and a `NULL` in `last_autovacuum`, that's a red flag — autovacuum might be misconfigured or disabled.

⚠️ **Why it matters:**
Autovacuum is your primary defense against table bloat and transaction ID wraparound. If it's not triggering regularly, your database may grow unnecessarily or even risk downtime.

🔧 **Actionable Insight:**
Adjust autovacuum thresholds or r c tables.



⏳ 3. Check Transaction ID Ag

```
SELECT datname, age(datfrozenxid)
FROM pg_database;
```

🧠 **Explanation:**
This query helps monitor how **old** the oldest unfrozen transaction ID is in each database. The function `age(datfrozenxid)` tells you how close you are to the **2-billion transaction ID limit.**

⚠️ **Why it matters:**
If this number gets too high (e.g., above 1.5 billion), your database is at risk of **transaction ID wraparound**, which can corrupt data if not handled.

✅ **Best Practice:**
Ensure regular `VACUUM FREEZE` operations are occurring, either via autovacuum or

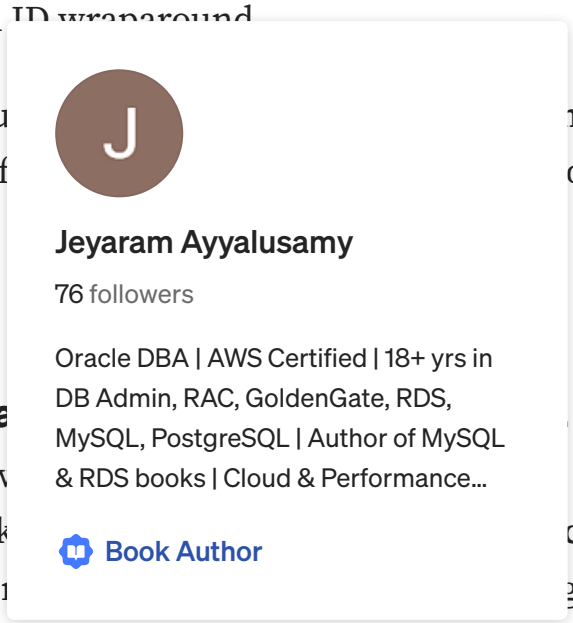
scheduled maintenance.

📌 Final Thoughts

Monitoring MVCC is not optional in a production PostgreSQL environment — it's essential. These simple yet powerful queries can help you:

- Detect table bloat early
- Confirm that autovacuum is working
- Avoid catastrophic transaction ID wraparound

🔁 Integrate these checks into you **n jobs**, or **alerting systems.** A few seconds of owntime later.



🔬 Why Developers Should Ca

As developers, we often focus on w or building APIs — while database internals lik **cy Control)** quietly do their job behind the scer greSQL, **understanding MVCC is more than just optional knowledge — it's a superpower.**

Here's why developers should care about MVCC and how it can help you build faster, more reliable, and scalable applications.

🚀 Optimize Transactions

MVCC manages data visibility by creating **multiple versions of rows**, enabling transactions to run concurrently without interfering with each other. But the way **you structure transactions** can have a big impact.

🧠 **Understanding MVCC lets you:**

- Keep transactions short and efficient

- Avoid long-running transactions that prevent dead tuple cleanup
- Choose the right isolation levels to balance performance and consistency

🔍 For example, a long idle transaction may block autovacuum and lead to table bloat — a subtle bug that can cause major performance issues.

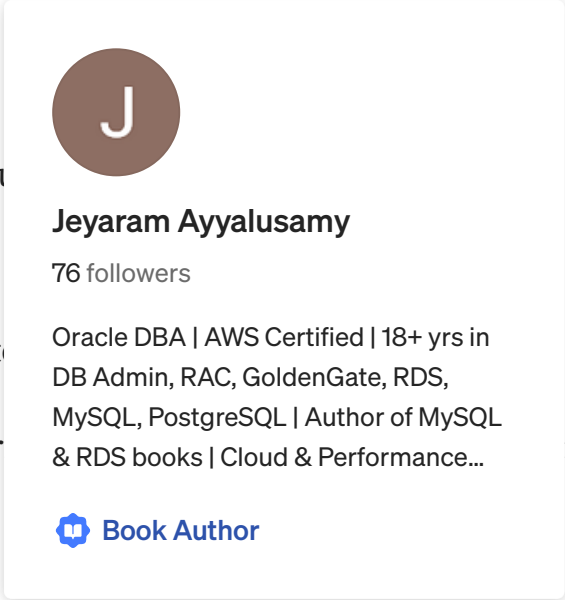
🛠️ Tune Vacuum Settings with Confidence

If you understand that MVCC creates dead tuples during updates/deletes, you'll also realize the importance of **autovacuum** and **manual VACUUM** routines.

💡 **As a developer, you can:**

- Work with DBAs to fine-tune au
- Ensure your high-churn tables
- Monitor vacuum lag using syst

This knowledge leads to better per n the long run.



🧪 Write Safer, More Efficient Code

MVCC allows PostgreSQL to avoid locking for reads, but that doesn't mean you're immune to issues.

🔐 **With MVCC awareness, developers can:**

- Avoid unnecessary row-level locks
- Write idempotent update logic
- Prevent anomalies in high-concurrency environments

Knowing how row visibility works under the hood helps you write code that's both safe and scalable.

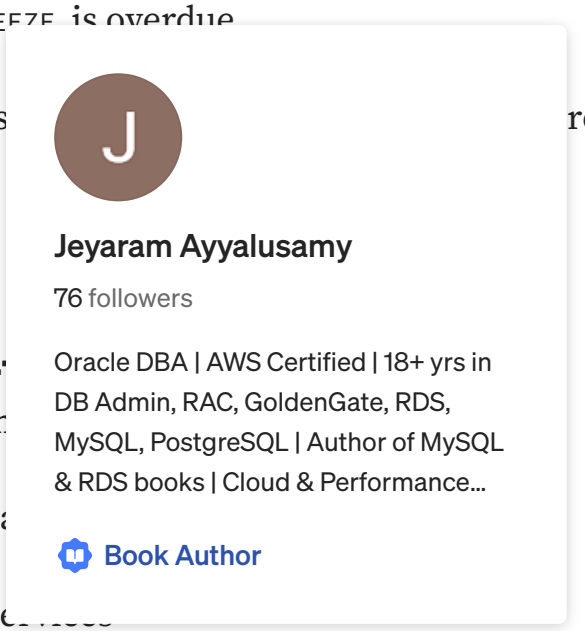
🔎 Diagnose Performance Issues Quickly

MVCC issues often **hide in plain sight** — like table bloat, delayed autovacuum, or slow queries caused by old versions of rows being scanned.

👷 If you know how MVCC works, you'll be able to:

- Use `pg_stat_user_tables` to spot dead tuples
- Monitor `pg_stat_activity` for long-running transactions
- Recognize when `VACUUM` or `FREEZE` is overdue

Instead of guessing, you'll troubles recision.



🧱 Design Better PostgreSQL

Ultimately, understanding MVCC h

✅ You'll make smarter decisions

- Transaction scoping in microse
- Data modeling for high-update workloads
- Replication and failover strategies in MVCC-friendly ways

You don't need to be a DBA to care about MVCC — just a developer who wants their app to scale smoothly and run efficiently.

💭 Final Thoughts

MVCC isn't just a backend detail — it's **PostgreSQL's secret sauce** for performance, consistency, and concurrency.

As a developer, investing just a little time to understand how MVCC works will give you massive returns in code quality, system design, and debugging agility.

So next time your query slows down or your table starts bloating — you'll know **exactly where to look.**

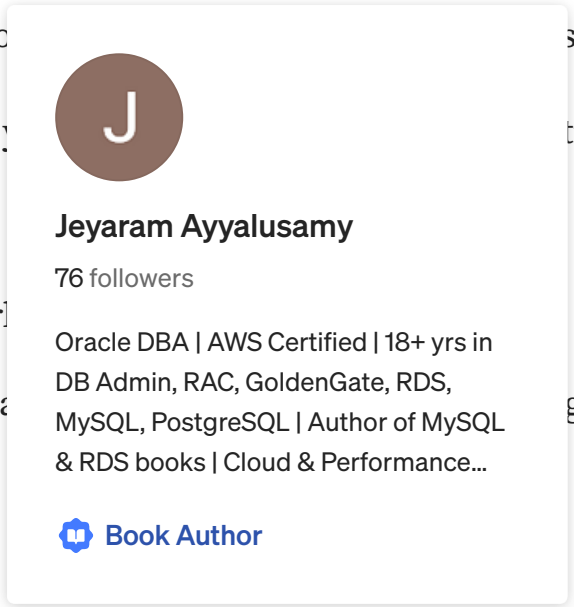
🏁 Conclusion

MVCC is PostgreSQL's superpower.

It enables PostgreSQL to handle thousands of concurrent transactions safely and efficiently — without the painful lo ses.

But remember: while MVCC gives till need to:

- Monitor bloat.
- Configure autovacuum properl
- Understand how your queries a g systems.



👉 *If you found this guide helpful, follow me(**medium**) for more practical PostgreSQL tutorials, database architecture guides, and hands-on DBA content.*

🔗 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!

Open Source | Postgresql | Oracle | Sql | MySQL

Following

Written by Jeyaram Ayyalusamy

76 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

Responses (1)

Gvadakte

What are your thoughts?

Jeyaram Ayyalusamy

76 followers

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

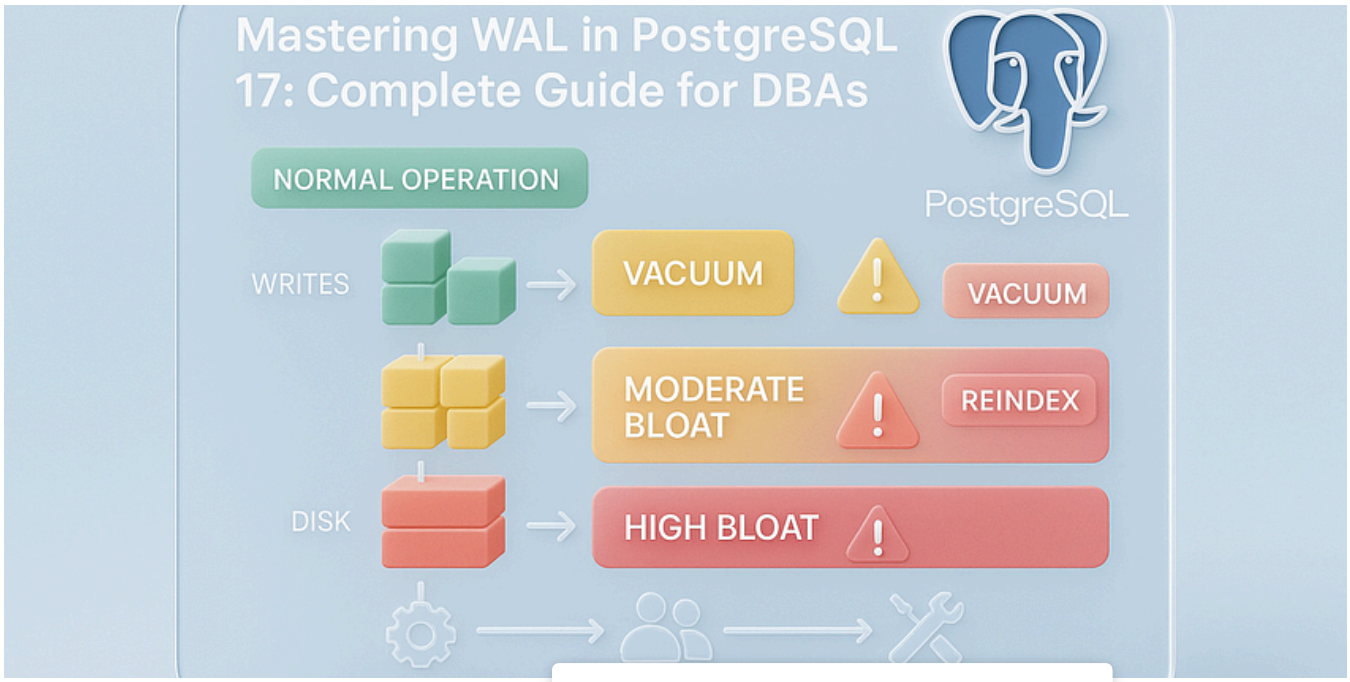
Book Author

Loïc Lefèvre
Jun 12

Might bring some interests to you: https://www.cs.cmu.edu/~pavlo/blog/2023/04/the-part-of-postgresql-we-hate-the-most.html

Reply

More from Jeyaram Ayyalusamy



Jeyaram Ayyalusamy

Understanding and Managing … nplete Guide for DBAs

PostgreSQL is a powerful, reliable, and … abase system. It's praised for its extensibility, ACID…

Jun 25 52

J

Jeyaram Ayyalusamy

76 followers

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

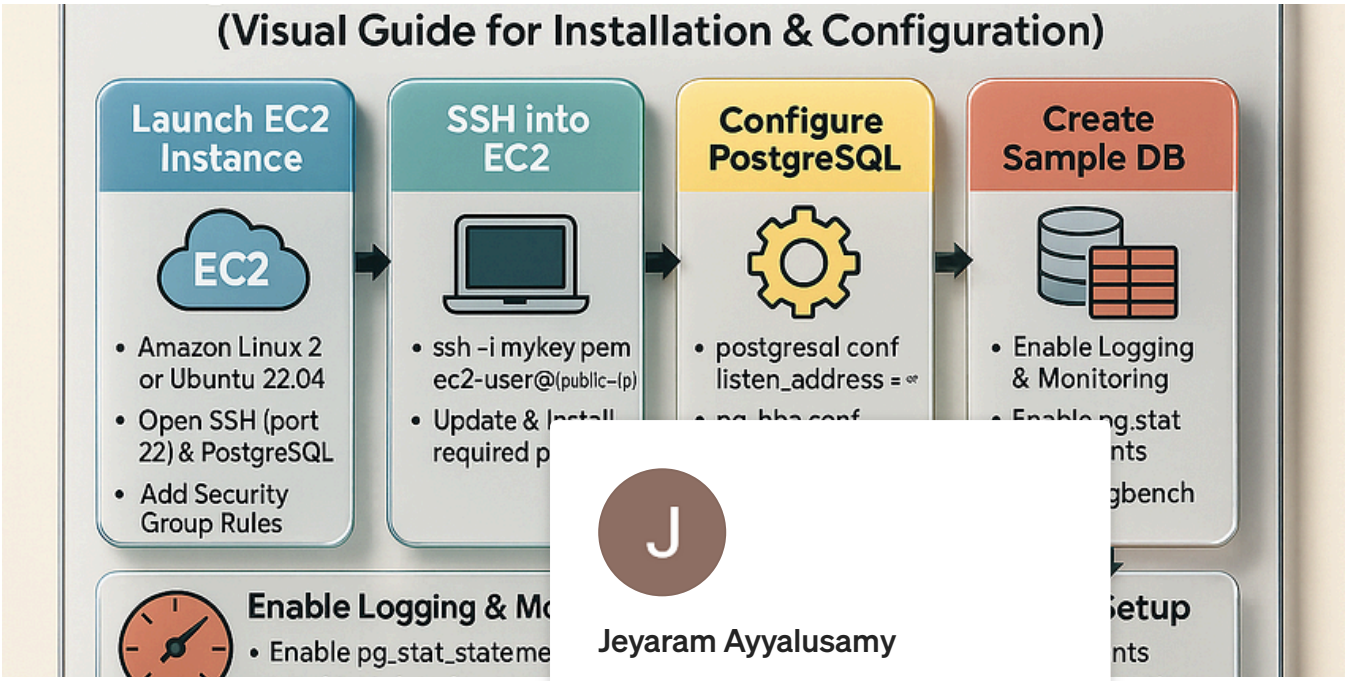
Book Author

Jeyaram Ayyalusamy

PostgreSQL Tablespaces Explained: Complete Guide for PostgreSQL 17 DBAs

PostgreSQL offers a powerful feature called tablespaces, allowing database administrators to take control of how and where data is...

Jun 12 8



Jeyaram Ayyalusamy

PostgreSQL 17 on AWS EC2—
Walkthrough

Setting up PostgreSQL 17 on AWS EC2 you break down each component, it becomes an efficie

1d ago 50

J

Jeyaram Ayyalusamy

76 followers

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

Book Author

Jeyaram Ayyalusamy

How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

PostgreSQL 17 is the latest release of one of the world's most advanced open-source relational databases. If you're using a Red Hat-based...

Jun 3

See all from Jeyaram Ayyalusamy

J

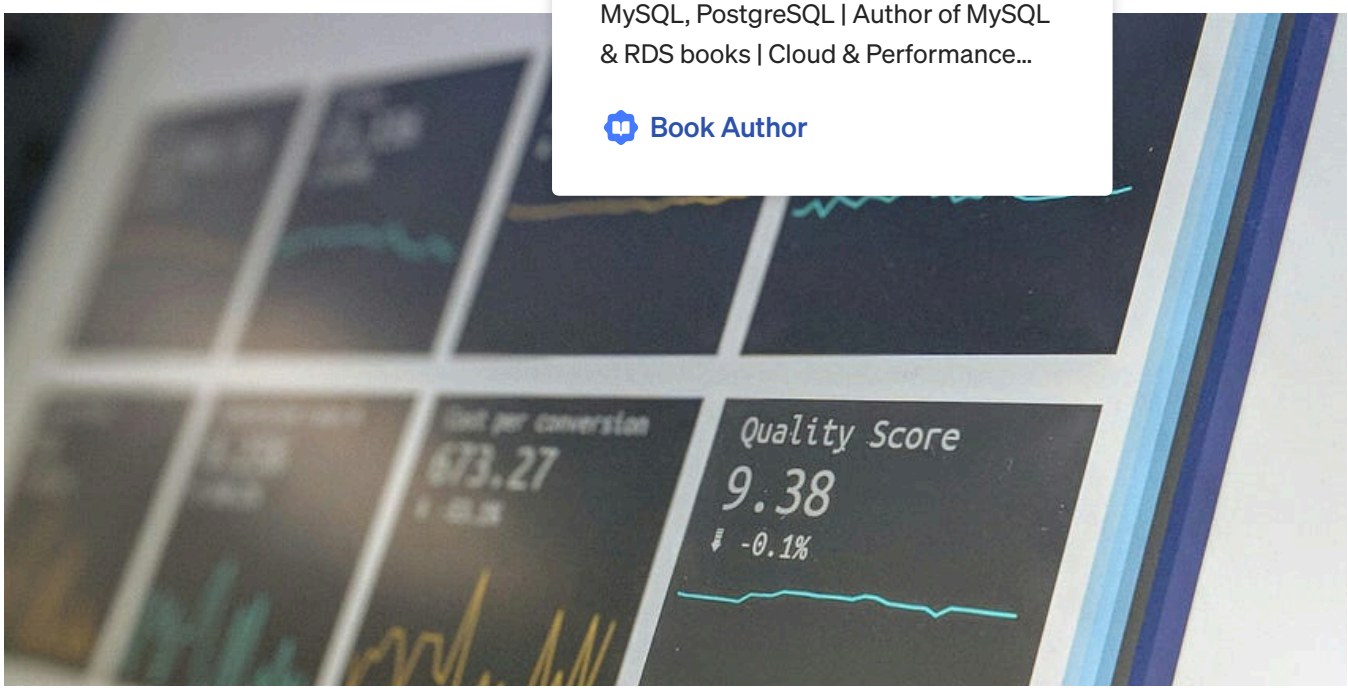
Jeyaram Ayyalusamy

76 followers

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

Book Author

Recommended from Medium

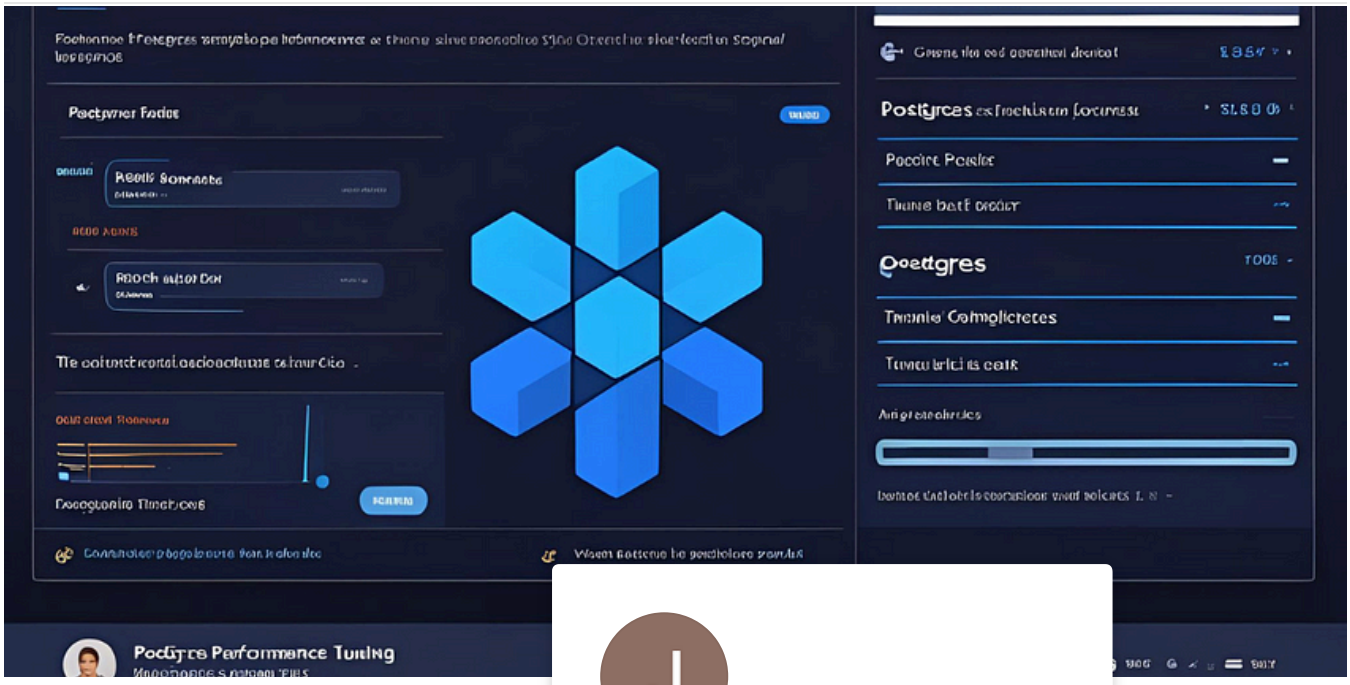


Azlan Jamal

Stop Using SERIAL in PostgreSQL: Here's What You Should Do Instead

In PostgreSQL, there are several ways to create a PRIMARY KEY for an id column, and they usually involve different ways of generating the...

Jul 12 33



Rizqi Mulki

Postgres Performance Tuning

The advanced techniques that separat

6d ago 55

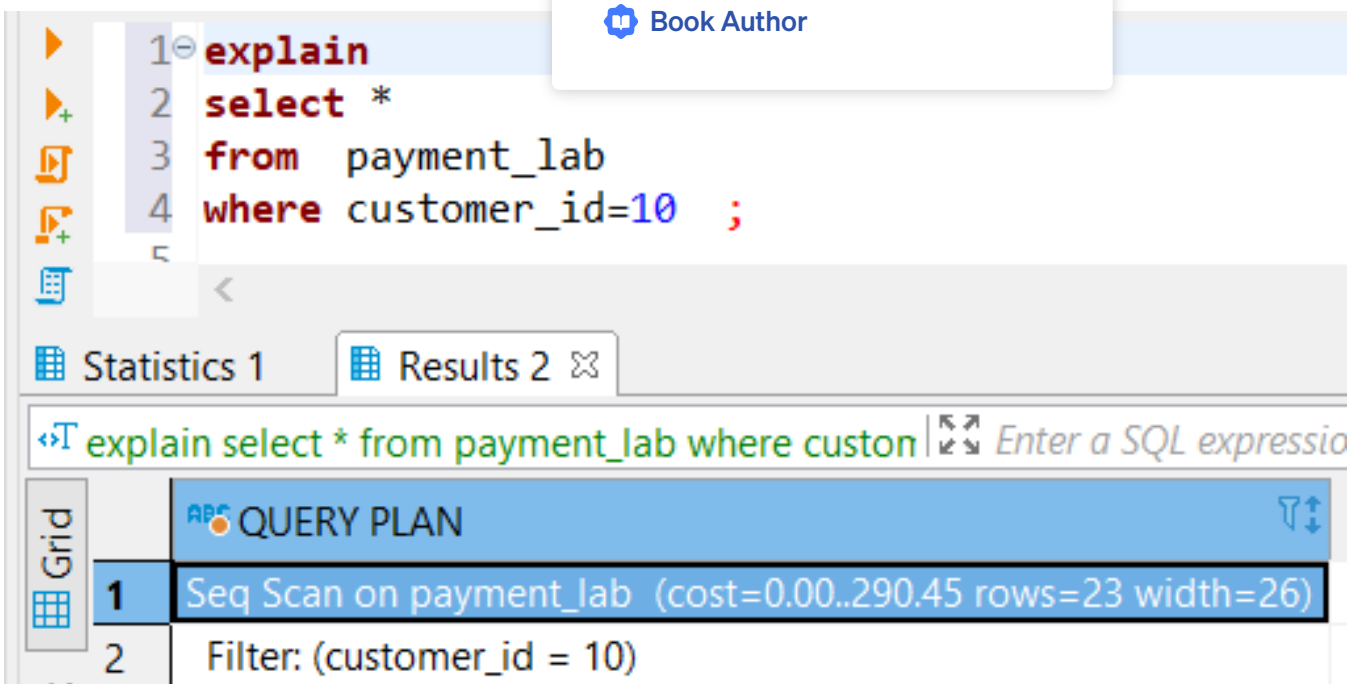
J

Jeyaram Ayyalusamy

76 followers

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

Book Author



Muhammet Kurtoglu

Postgresql Query Performance Analysis

SQL tuning is critically important for ensuring database performance, reliability, and scalability. In most cases, performance issues in a...

6d ago 10

techWithNeeru

This SQL Query Took 20 Secor What I Changed

Last week, I was troubleshooting a perf n crazy. Our analytics dashboard was timing out, us

Jul 10 66

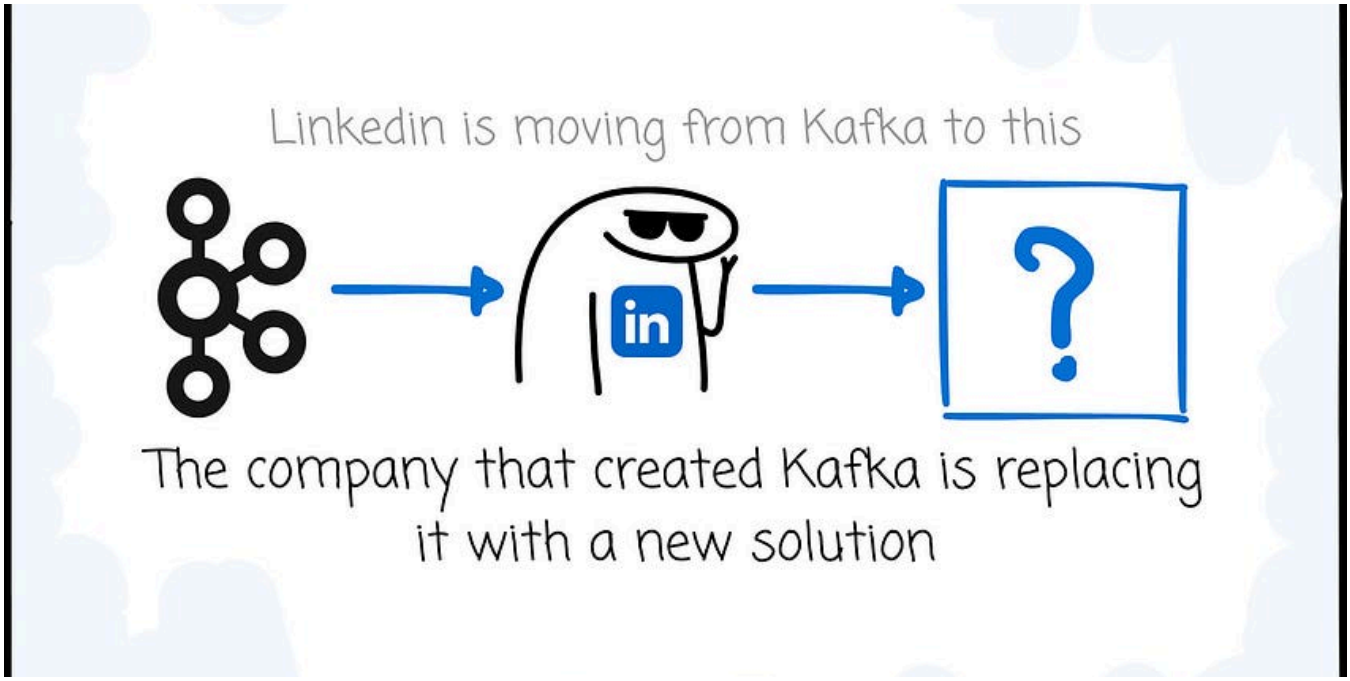
J

Jeyaram Ayyalusamy

76 followers

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

Book Author

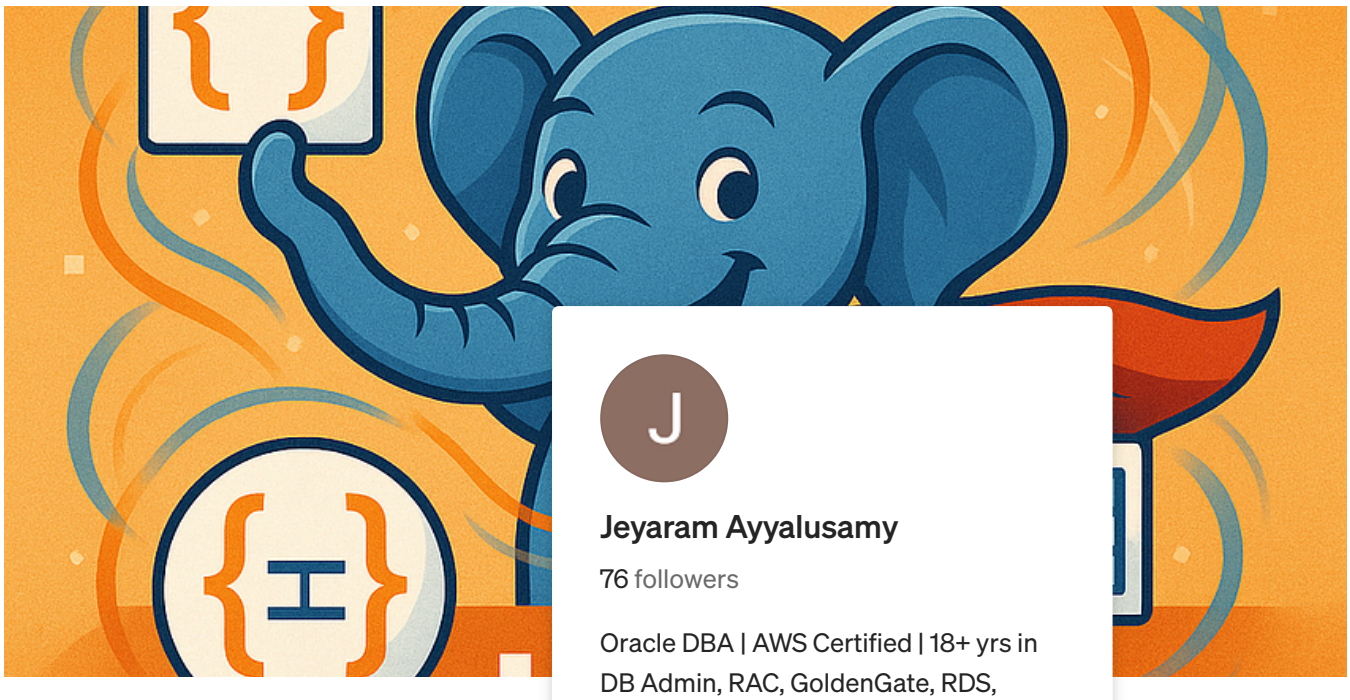


In Data Engineer Things by Vu Trinh

The company that created Kafka is replacing it with a new solution

How did LinkedIn build Northguard, the new scalable log storage

Jul 17 330 6



J

Jeyaram Ayyalusamy

76 followers

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

Book Author

Rick Hightower

JSONB: PostgreSQL's Secret W ing

Have you ever stared at your database e more column to this table, I'm going to lose it"? As you...

May 4 8

See more recommendations