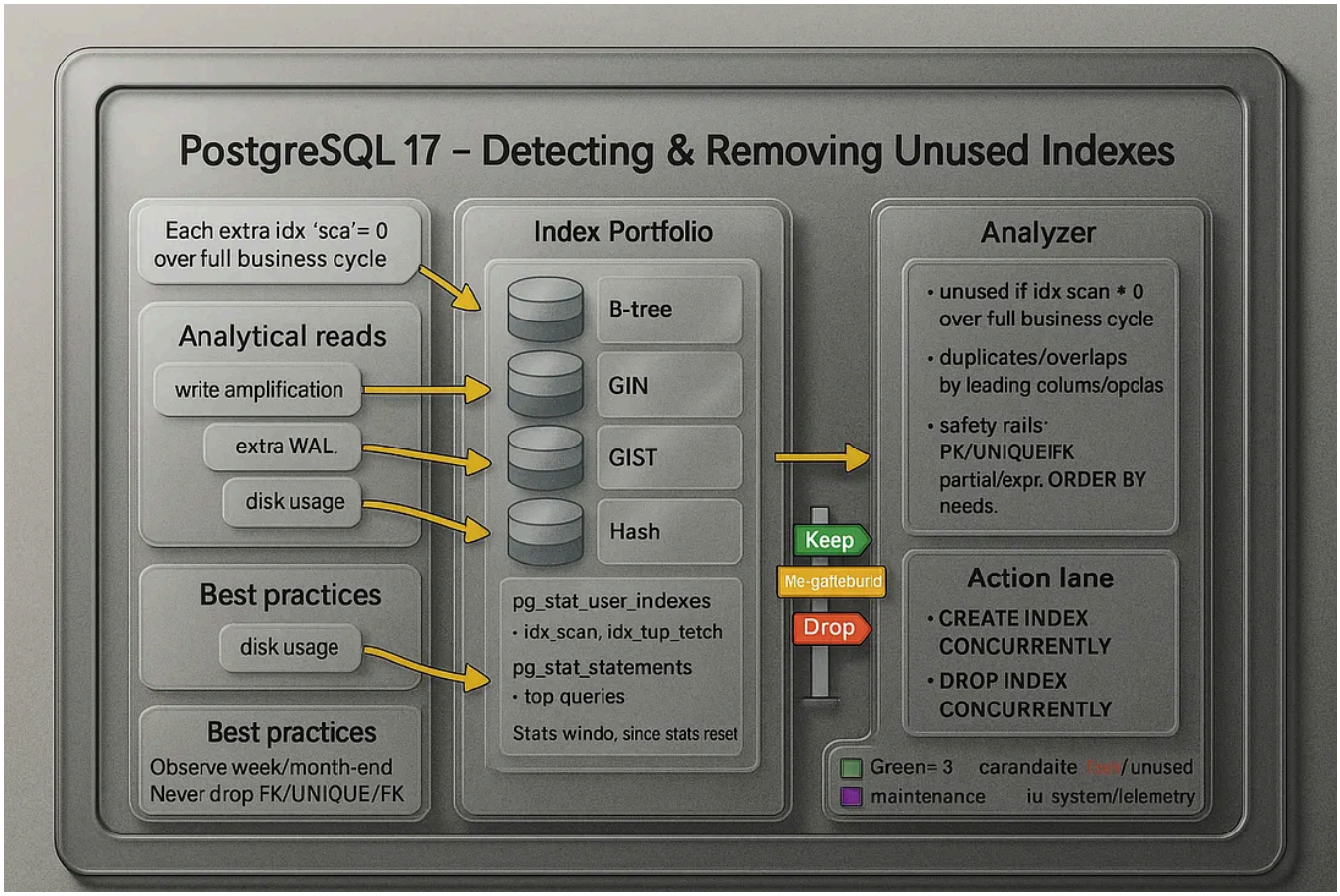
25 - PostgreSQL 17 Performance Tuning: Detecting and Removing Unused Indexes

5 min read · Sep 10, 2025

Jeyaram Ayyalusamy Following



Indexes are one of the most important tools for query performance in PostgreSQL. But while a missing index slows down queries, an **unused or pointless index** can be just as harmful.

If your database is very small, a few extra indexes don't matter much. But in a large production environment, **unused indexes can waste hundreds of gigabytes and put unnecessary strain on the system**. Every extra index comes with hidden costs: more disk usage, slower writes, and more work for background processes like VACUUM.

That's why tuning isn't just about **adding indexes**. It's equally about **finding indexes that shouldn't exist**.

Why Unused Indexes Are a Problem

1. **Disk Space Waste**

- An index duplicates column values plus pointers to rows.
- On a 10M row table, each index can take hundreds of MB or even several GB.
- Multiple unused indexes = huge wasted storage.

2. **Slower Write Performance**

- Every time you `INSERT`, `UPDATE`, or `DELETE`, PostgreSQL must also update all related indexes.
- More indexes = more overhead = slower write-heavy workloads.

3. **Maintenance Overhead**

- VACUUM, REINDEX, and CLUSTER must also handle these indexes.
- Index bloat and dead entries pile up even if the index is never queried.

👉 In short: while indexes make reads faster, unused indexes **slow everything else down**.

Step 1: Create a Products Table

Let's simulate a real-world dataset with 10 million rows:

```
CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    price        NUMERIC(10,2),
    stock_qty    INT
);
```

```
postgres=# CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    price        NUMERIC(10,2),
    stock_qty    INT
);
CREATE TABLE
postgres=#
```

```
-- Insert 10 million rows
INSERT INTO products (product_name, category, price, stock_qty)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    (random()*500)::NUMERIC(10,2),
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
```

```
INSERT INTO products (product_name, category, price, stock_qty)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
```

```
    (random()*500)::NUMERIC(10,2),
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
INSERT 0 10000000
ANALYZE
postgres=#
```

Step 2: Add Some Indexes

For demonstration, we'll add three indexes:

```
-- Useful index for frequent category lookups
CREATE INDEX idx_products_category ON products(category);
```

```
postgres=# CREATE INDEX idx_products_category ON products(category);
CREATE INDEX
postgres=#
```

```
-- Rarely used index on product_name
CREATE INDEX idx_products_name ON products(product_name);

-- Probably useless index on stock quantity
CREATE INDEX idx_products_stock_qty ON products(stock_qty);
```

```
postgres=# -- Rarely used index on product_name
CREATE INDEX idx_products_name ON products(product_name);
```

```
CREATE INDEX
postgres=#
```

```
postgres=# -- Probably useless index on stock quantity
CREATE INDEX idx_products_stock_qty ON products(stock_qty);
CREATE INDEX
postgres=#
```

At this point, we don't yet know which indexes are helpful and which are wasteful.

Step 3: Check Index Usage with `pg_stat_user_indexes`

PostgreSQL tracks how often indexes are used. We can query this data:

```
SELECT relname AS table_name,
       indexrelname AS index_name,
       idx_scan,
       idx_tup_read,
       idx_tup_fetch,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE relname = 'products'
ORDER BY idx_scan ASC;
```

Sample Output for using without index

```
postgres=# SELECT relname AS table_name,
       indexrelname AS index_name,
       idx_scan,
       idx_tup_read,
       idx_tup_fetch,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE relname = 'products'
ORDER BY idx_scan ASC;
 table_name |       index_name       | idx_scan | idx_tup_read | idx_tup_fetch
------------+------------------------+----------+--------------+---------------
 products   | products_pkey          |        0 |            0 |             0
 products   | idx_products_category  |        0 |            0 |             0
 products   | idx_products_name      |        0 |            0 |             0
 products   | idx_products_stock_qty |        0 |            0 |             0
(4 rows)

postgres=#
postgres=#
```

```
SELECT * FROM products WHERE product_id = 5000000;
```

```
postgres=# SELECT * FROM products WHERE product_id = 5000000;
 product_id |  product_name    |  category  | price  | stock_qty
------------+------------------+------------+--------+-----------
    5000000 | Product_5000000 | Category_0 | 152.44 |        78
(1 row)

postgres=#
```

```
SELECT * FROM products WHERE category = 'Category_20';
```

```
postgres=# SELECT * FROM products WHERE category = 'Category_20';
 product_id |  product_name   |  category   | price  | stock_qty
------------+-----------------+-------------+--------+-----------
         20 | Product_20      | Category_20 | 469.36 |        30
         70 | Product_70      | Category_20 | 481.26 |        31
        120 | Product_120     | Category_20 | 318.93 |        28
        170 | Product_170     | Category_20 |  12.32 |        93
        220 | Product_220     | Category_20 | 113.61 |        43
        270 | Product_270     | Category_20 | 470.95 |        68
        320 | Product_320     | Category_20 | 368.27 |        85
        370 | Product_370     | Category_20 | 464.31 |        34
        420 | Product_420     | Category_20 | 255.68 |        16
        470 | Product_470     | Category_20 |  88.36 |        58
        520 | Product_520     | Category_20 | 233.06 |        11
        570 | Product_570     | Category_20 |  47.56 |         4
```

```
SELECT relname AS table_name,
       indexrelname AS index_name,
       idx_scan,
       idx_tup_read,
       idx_tup_fetch,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE relname = 'products'
ORDER BY idx_scan ASC;
```

Step 4: Sample Output

Example results might look like this:

```
postgres=# SELECT relname AS table_name,
       indexrelname AS index_name,
       idx_scan,
       idx_tup_read,
       idx_tup_fetch,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE relname = 'products'
ORDER BY idx_scan ASC;
 table_name |       index_name       | idx_scan | idx_tup_read | idx_tup_fetch
------------+------------------------+----------+--------------+---------------
 products   | idx_products_stock_qty |        0 |            0 |
 products   | idx_products_name      |       12 |       500000 |        50000
 products   | idx_products_category  |    20000 |     25000000 |     25000000

postgres=#
```

👉 Interpretation:

- **`idx_products_category`** → scanned 20,000 times, fetched 25M rows. Clearly useful.
- **`idx_products_name`** → scanned only 12 times. Might be justified if occasional lookups by product name are business-critical.
- **`idx_products_stock_qty`** → **never used** (`idx_scan = 0`), yet it consumes 250 MB. This is a waste.

Step 5: Deciding What to Do

- If an index shows **high usage** → keep it.

- If it shows **low or occasional usage** → consult with developers/business before removing. It might be used in rare but critical reports.
- If it shows **zero usage over a long period** → it's a candidate for dropping.

Step 6: Dropping an Unused Index

If we decide that `idx_products_stock_qty` is wasteful, we can safely remove it:

```
DROP INDEX idx_products_stock_qty;
```

```
postgres=# DROP INDEX idx_products_stock_qty;
DROP INDEX
postgres=#
```

Now the database has:

- Less storage overhead.
- Faster writes (`INSERT` / `UPDATE` / `DELETE`).
- Lower maintenance costs during VACUUM and reindexing.

Step 7: Continuous Monitoring

Finding unused indexes is not a one-time task. Query patterns change over time. A rarely used index today may become critical tomorrow after an application change.

That's why it's best practice to:

- Regularly monitor `pg_stat_user_indexes`.
- Integrate index usage metrics into your monitoring dashboards (e.g., Grafana, pgBadger, or custom queries).
- Review index usage before major schema changes.

Key Takeaways

- **Unused indexes** harm performance by slowing writes and wasting space.
- PostgreSQL 17 provides **system views (`pg_stat_user_indexes`)** to spot them.
- Always validate with developers before dropping indexes — some may serve niche but important queries.
- Regular monitoring ensures your database remains lean and efficient.

✅ Index tuning is not only about **adding indexes** where they are missing, but also about **removing the ones you don't need.**

📢 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 Open Source MySQL

J

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

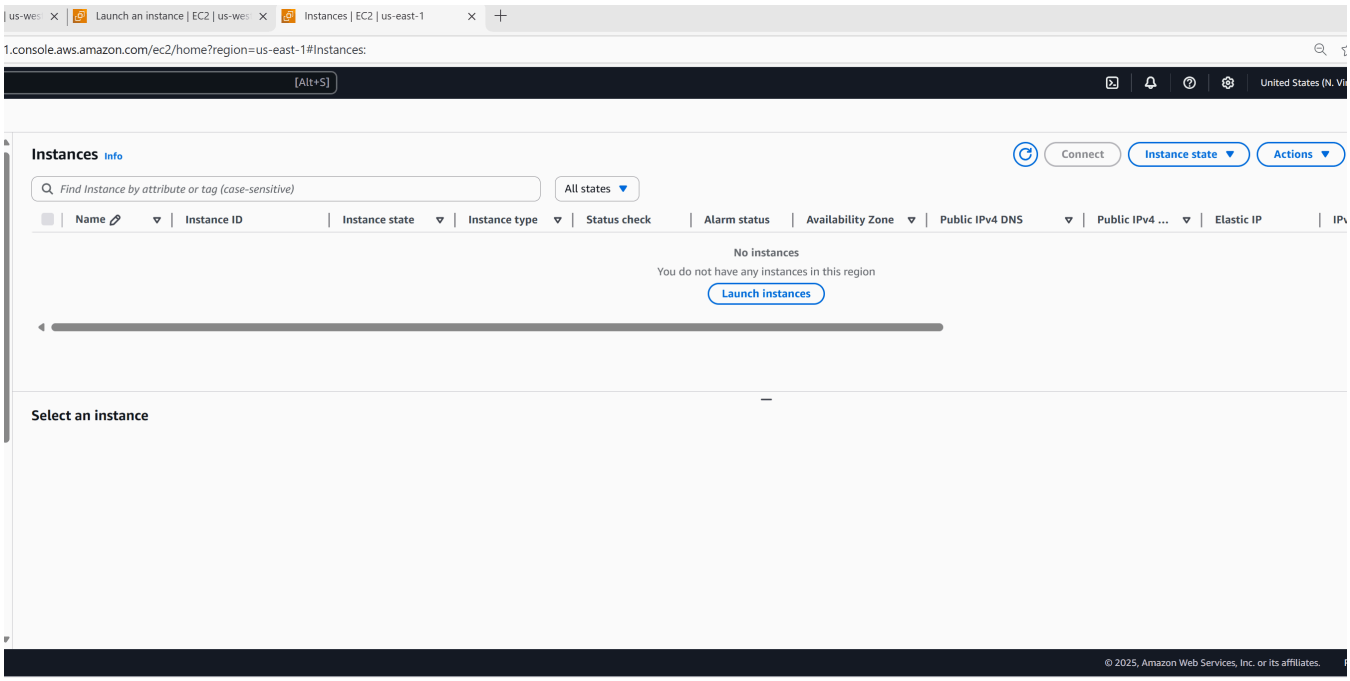
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

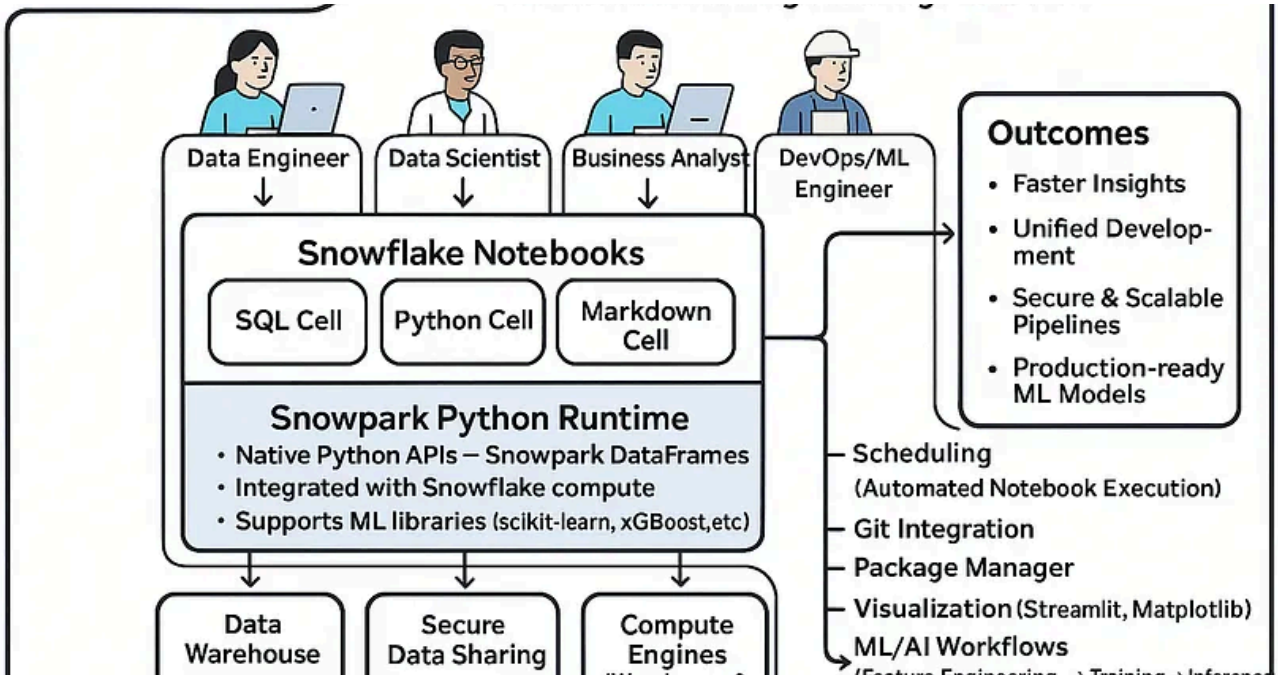


Jeyaram Ayyalusamy

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

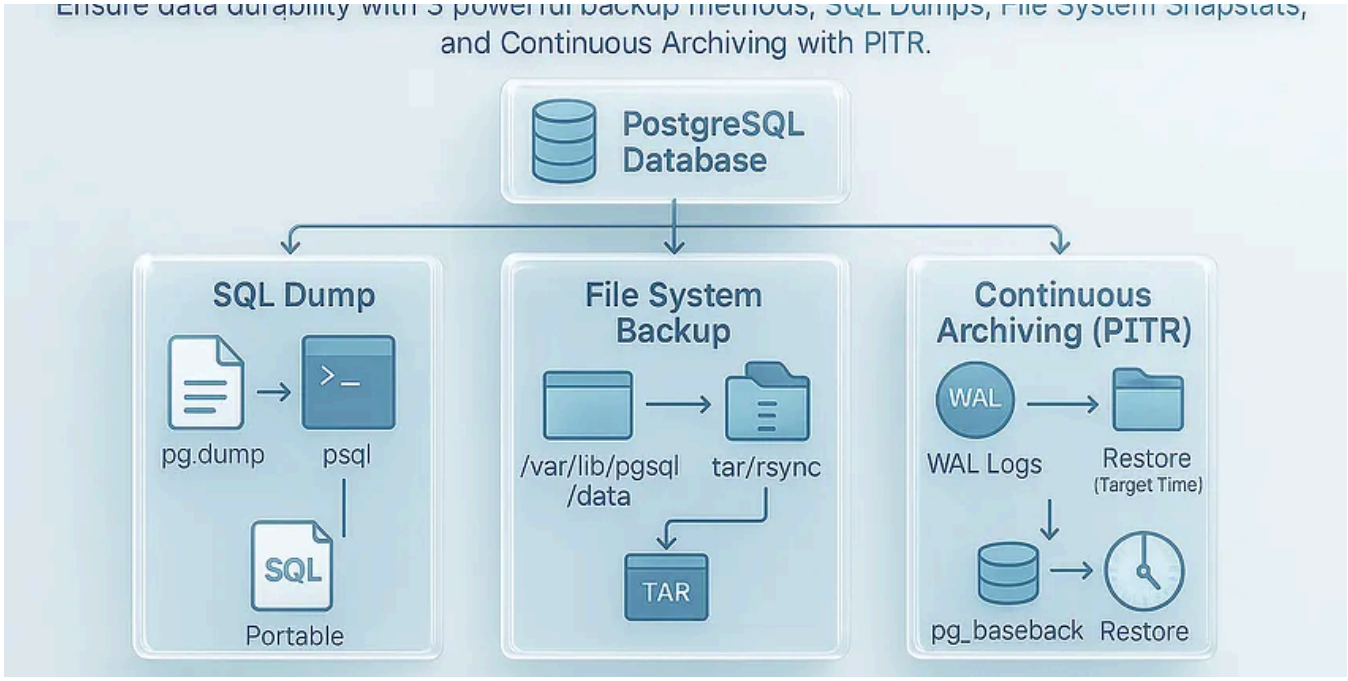


Jeyaram Ayyalusamy

16—Experience Snowflake with Notebooks and Snowpark Python: A Unified Data Engineering Platform

In the fast-moving world of data, organizations are no longer just collecting information—they're leveraging it to drive business...

Jul 13

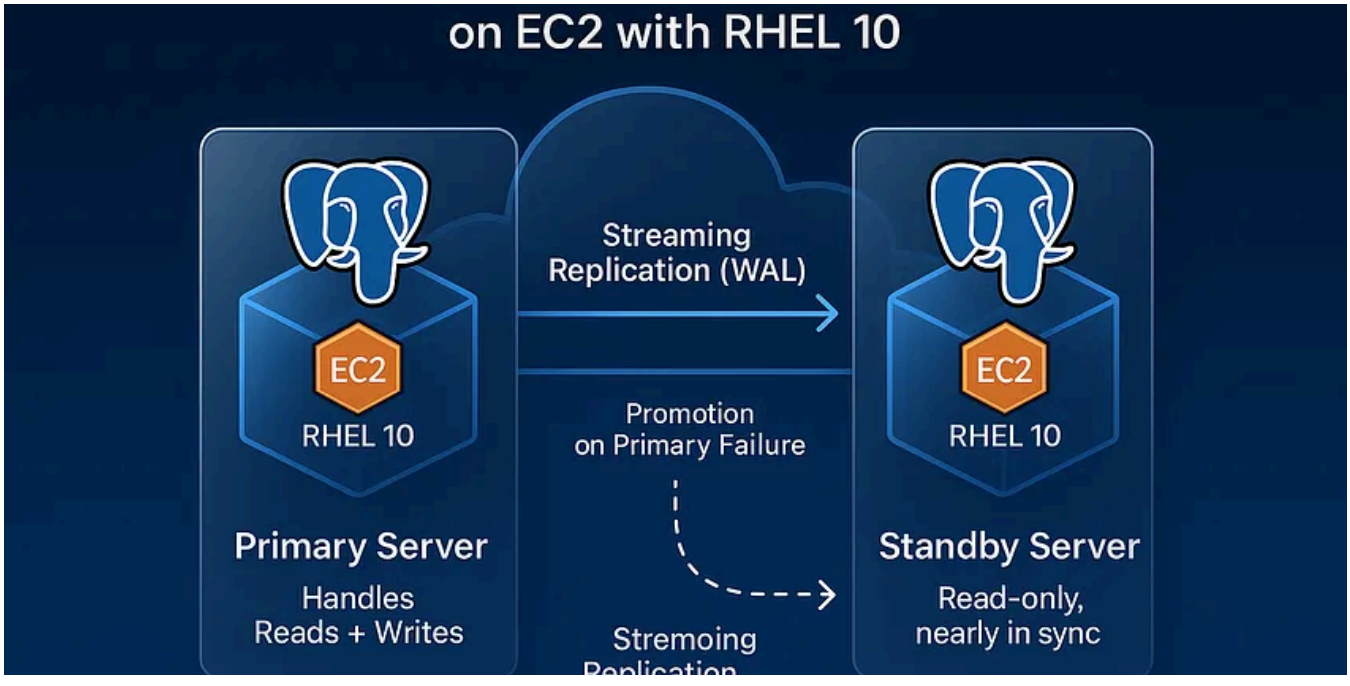


Jeyaram Ayyalusamy

🛡️ Essential Techniques Every DBA Should Know: PostgreSQL Backup Strategies

In the world of databases, data is everything—and losing it can cost your business time, money, and trust. Whether you're managing a...

Aug 20 26



Jeyaram Ayyalusamy

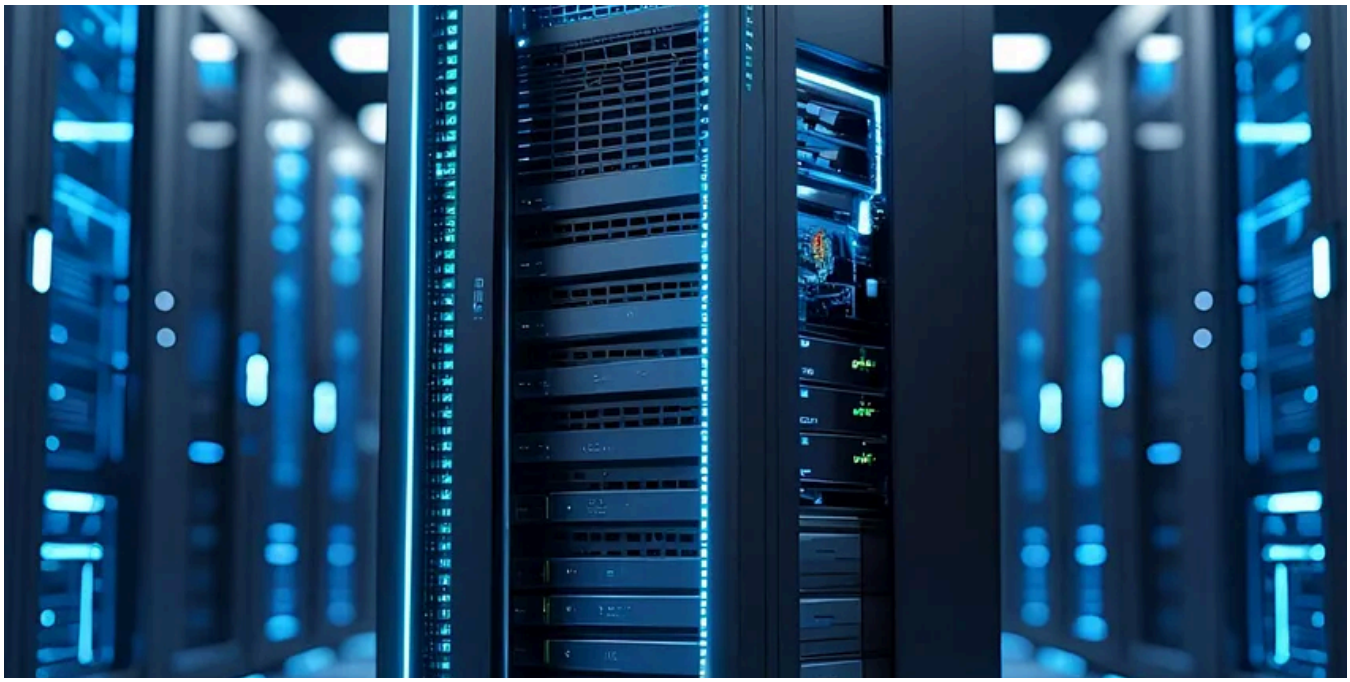
🚀 Streaming Replication in PostgreSQL 17 on EC2 with RHEL 10—A Deep Dive

🐘 Introduction: Why Streaming Replication Matters

Aug 20 50

See all from Jeyaram Ayyalusamy

Recommended from Medium



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

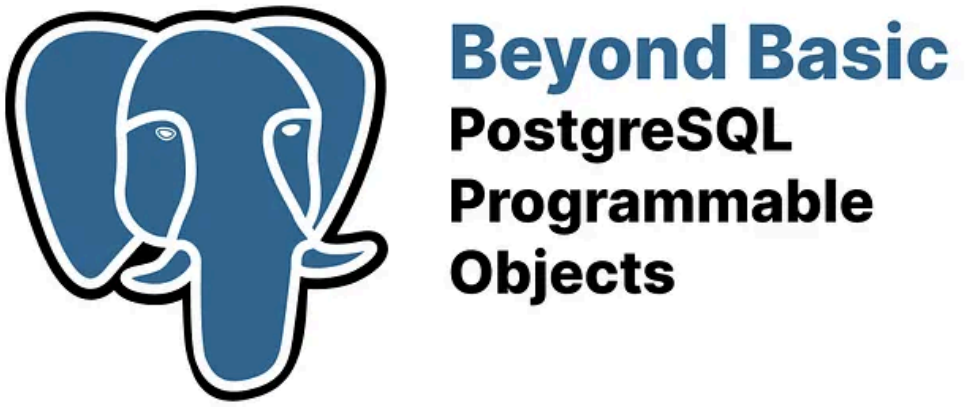
Sep 15 11 1

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



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.

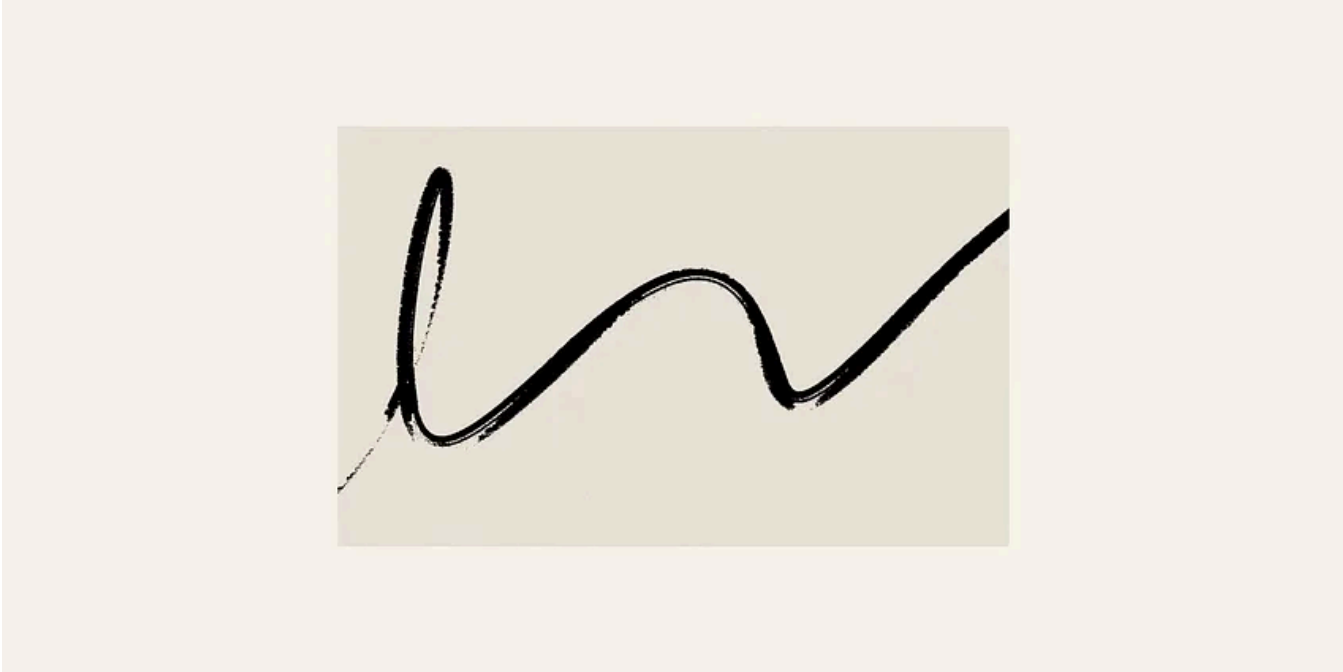
Sep 1 68 1

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

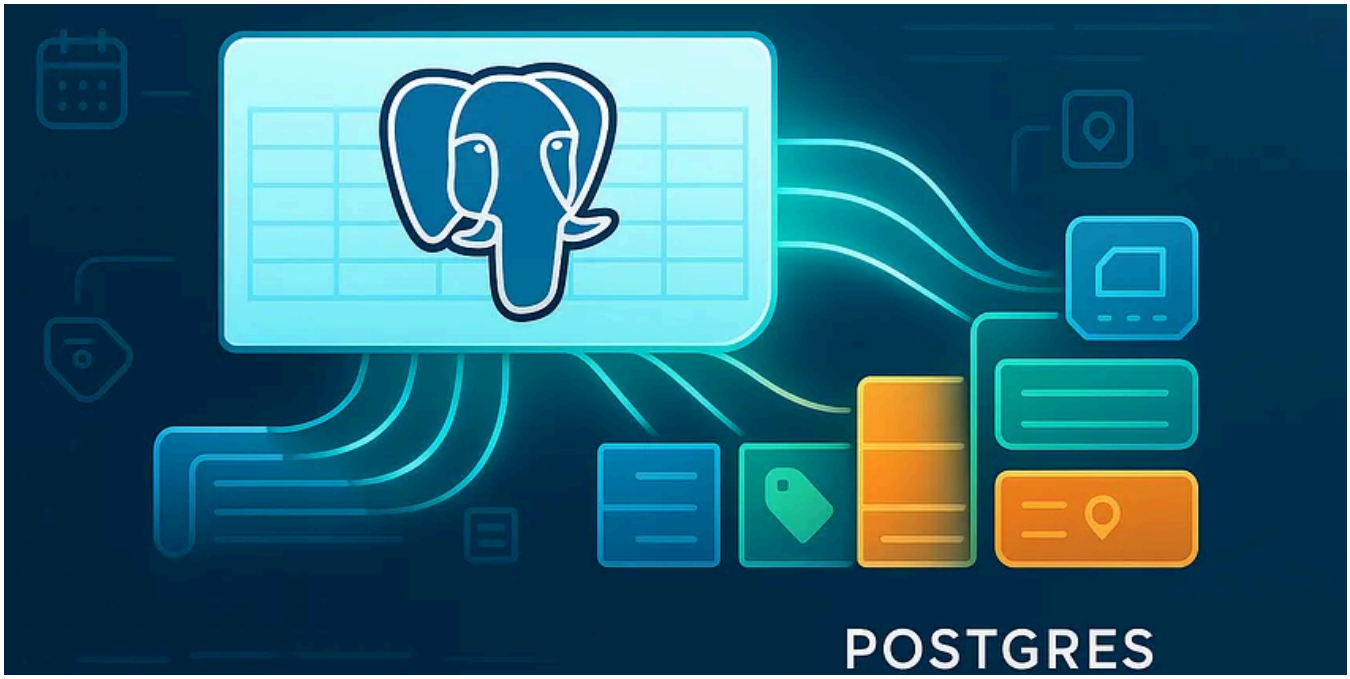


Rohan

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

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

See more recommendations