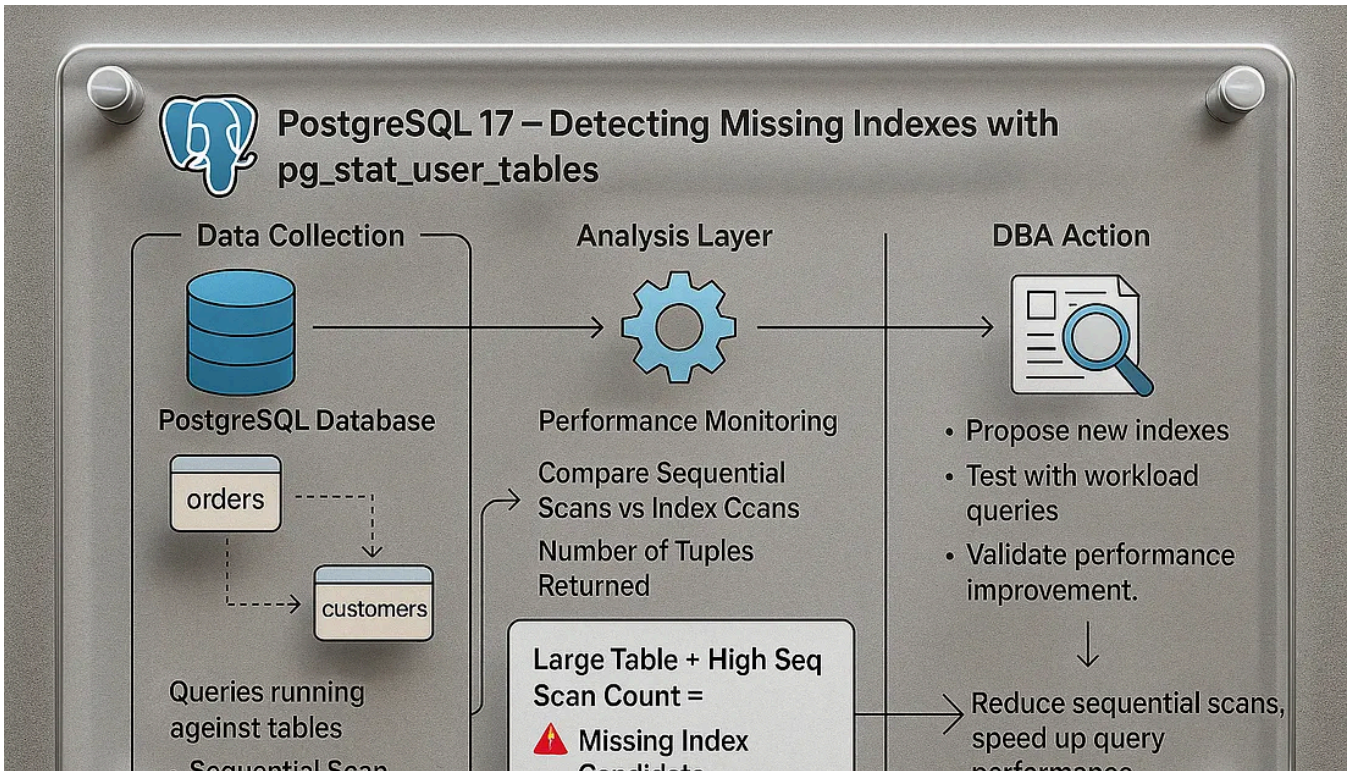
# 26 - PostgreSQL 17 Performance Tuning: Detecting Missing Indexes with `pg_stat_user_tables`

5 min read · 3 days ago

Jeyaram Ayyalusamy Following

Listen Share More






Collecting statistics is only the first step — the real challenge is making sense of the data. Simply reading counts of scans, tuples, or dead rows won't improve

performance unless you know how to interpret them.

One practical way to use `pg_stat_user_tables` is to **detect which tables might need an index.** The approach is simple:

- Look for **large tables.**
- Check if they are frequently accessed with **sequential scans.**
- If a big table consistently shows high sequential scan counts, it is a strong candidate for indexing.

### Step 1: Create a Products Table with 10 Million Rows

For demonstration, let's create a realistic table and load it with data.

```
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 % 100),     -- 100 categories
    (random()*500)::NUMERIC(10,2),
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
```

```
postgres=# -- Insert 10 million rows
INSERT INTO products (product_name, category, price, stock_qty)
SELECT
    'Product_' || g,
    'Category_' || (g % 100),     -- 100 categories
    (random()*500)::NUMERIC(10,2),
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
INSERT 0 10000000
ANALYZE
postgres=#
```

This gives us a large table that will generate meaningful scan statistics.

## Step 2: Run Some Queries to Generate Load

```
-- Category filter (no index on category yet)
```

```
SELECT * FROM products WHERE category = 'Category_42';
```

```
postgres=# SELECT * FROM products WHERE category = 'Category_42';
 product_id |  product_name   |  category   | price  | stock_qty
------------+-----------------+-------------+--------+-----------
         42 | Product_42      | Category_42 | 240.05 |        70
        142 | Product_142     | Category_42 |  12.28 |         1
        242 | Product_242     | Category_42 | 228.58 |        37
        342 | Product_342     | Category_42 | 341.42 |        58
        442 | Product_442     | Category_42 | 370.30 |        15
        542 | Product_542     | Category_42 | 100.69 |        73
        642 | Product_642     | Category_42 |  47.06 |        30
        742 | Product_742     | Category_42 | 434.37 |        94
        842 | Product_842     | Category_42 | 228.29 |         9
        942 | Product_942     | Category_42 | 197.54 |        28
       1042 | Product_1042    | Category_42 | 163.16 |        98
       1142 | Product_1142    | Category_42 | 244.24 |        27
       1242 | Product_1242    | Category_42 |  50.84 |        43
       1342 | Product_1342    | Category_42 |   0.04 |        79
```

```
-- Price filter
SELECT * FROM products WHERE price > 250;

-- Update by category
UPDATE products SET stock_qty = stock_qty + 1
WHERE category = 'Category_10';
```

```
postgres=# UPDATE products SET stock_qty = stock_qty + 1
WHERE category = 'Category_10';
UPDATE 100000
postgres=#
```

Without indexes on `category` or `price`, these queries will trigger **sequential scans.**

### Step 3: Identify Tables with Heavy Sequential Scans

You can run the following query to detect which tables might need an index:

```
SELECT relname AS table_name,
       seq_scan,
       seq_tup_read,
       idx_scan,
       idx_tup_fetch,
       n_live_tup
FROM pg_stat_user_tables
ORDER BY seq_tup_read DESC
LIMIT 10;
```

👉 This query highlights:

- **`seq_scan`** → number of sequential scans.
- **`seq_tup_read`** → total rows read by sequential scans.
- **`idx_scan`** → number of index scans.
- **`n_live_tup`** → approximate live row count in the table.

Large tables at the **top of this list** with **high sequential scan counts** are often missing indexes.

### Step 4: Interpreting the Results

Example output:

```
postgres=# UPDATE products SET stock_qty = stock_qty + 1
WHERE category = 'Category_10';
UPDATE 100000
postgres=# SELECT relname AS table_name,
       seq_scan,
       seq_tup_read,
       idx_scan,
       idx_tup_fetch,
       n_live_tup
FROM pg_stat_user_tables
ORDER BY seq_tup_read DESC
LIMIT 10;
 table_name | seq_scan | seq_tup_read | idx_scan | idx_tup_fetch | n_live_tup
------------+----------+--------------+----------+---------------+------------
 products   |        6 |     28639097 |        0 |             0 |   10000000
(1 row)

postgres=#
```

- The `products` table shows **millions of tuples read sequentially**.
- No index scans were used, meaning PostgreSQL has no choice but to scan the whole table.
- On a large dataset, this will **seriously degrade performance** for frequent queries.

### Important Note

Not all sequential scans are bad. They are natural in cases like:

- Full-table analytical queries (OLAP).

- Backups or reporting jobs.

But if you see **frequent large sequential scans in transactional workloads (OLTP)**, performance will inevitably degrade.

👉 Experience shows that **missing indexes are the #1 cause of poor performance in PostgreSQL**. By monitoring `pg_stat_user_tables` and focusing on tables with high sequential scans, you can systematically detect and fix indexing gaps.

✅ With this method, PostgreSQL performance tuning becomes **fact-driven**: you're not guessing, but identifying exactly which tables and queries need indexing support.

### Example: Missing Indexes with `pg_stat_user_tables`

### Step 1: Add Indexes

Now, let's fix it with proper indexes:

```
CREATE INDEX idx_products_category ON products(category);
CREATE INDEX idx_products_price ON products(price);
ANALYZE products;
```

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

```
postgres=# CREATE INDEX idx_products_price ON products(price);
CREATE INDEX
postgres=#

postgres=# ANALYZE products;
ANALYZE
postgres=#
```

## Step 5: Run Queries Again

```
-- Now this query should use an index
SELECT * FROM products WHERE category = 'Category_42';
```

```
-- This one too
SELECT * FROM products WHERE price > 300;
```

## Step 6: Check Statistics Again

```
SELECT relname AS table_name,
       seq_scan,
       seq_tup_read,
       idx_scan,
       idx_tup_fetch
FROM pg_stat_user_tables
WHERE relname = 'products';
```

### 📊 Sample Output (after indexing):

```
postgres=# SELECT relname AS table_name,
       seq_scan,
       seq_tup_read,
       idx_scan,
       idx_tup_fetch
FROM pg_stat_user_tables
WHERE relname = 'products';
 table_name | seq_scan | seq_tup_read | idx_scan | idx_tup_fetch
------------+----------+--------------+----------+---------------
 products   |       11 |     58584731 |        1 |       3562964
(1 row)

postgres=#
```

- Sequential scans remain for earlier queries, but **new queries now use indexes.**
- **`idx_scan = 120`** → PostgreSQL chose indexes instead of scanning the whole table.
- **`idx_tup_fetch`** shows how many rows were efficiently fetched via the index.

## Key Takeaway

- **Before indexing** → 10M rows were read repeatedly via sequential scans.
- **After indexing** → PostgreSQL switched to index scans, reading only relevant rows, saving massive I/O.
- **Practical lesson** → Monitor `pg_stat_user_tables`, and when you see frequent sequential scans on large tables, it's a strong signal you need indexes.

✅ This example shows how PostgreSQL 17 makes it easy to **spot missing indexes** using table statistics and how a single change can dramatically improve query performance.

### 📢 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!

AWS | Open Source | Oracle | Postgresql | MySQL

Following

## Written by Jeyaram Ayyalusamy

158 followers · 2 following

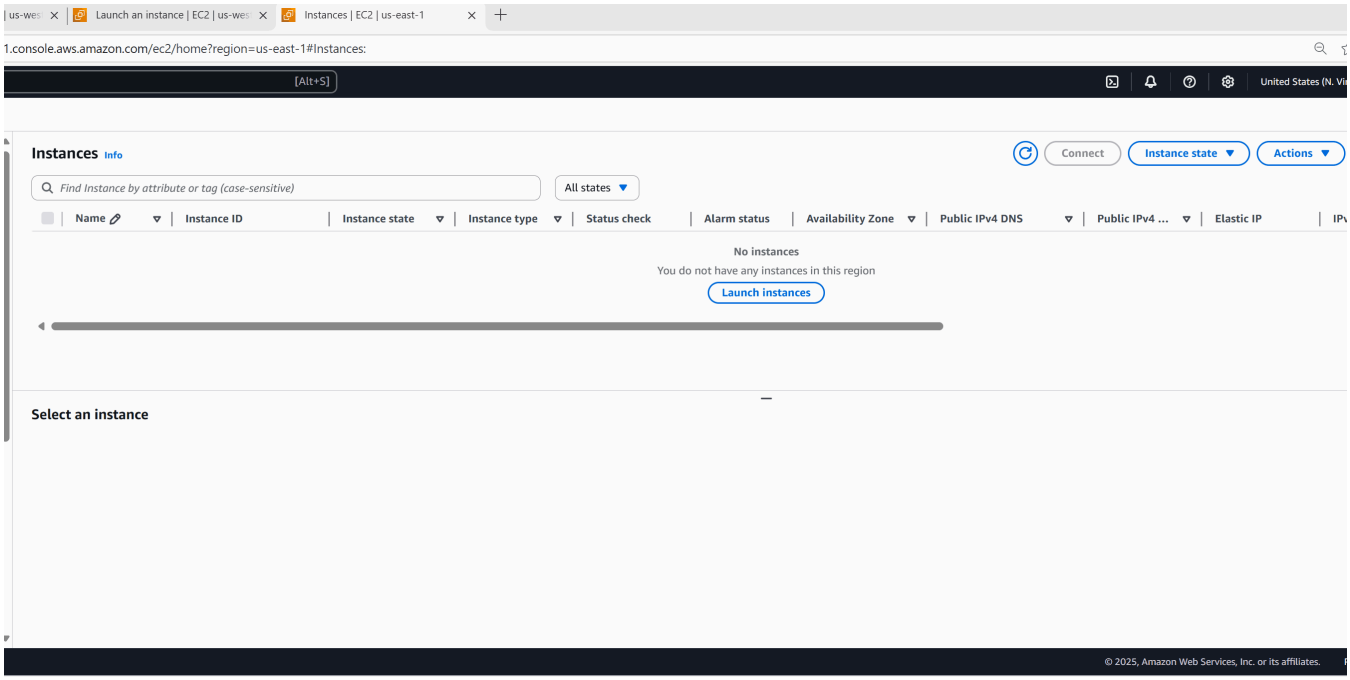
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

## No responses yet

Gvadakte

What are your thoughts?

## More from Jeyaram Ayyalusamy

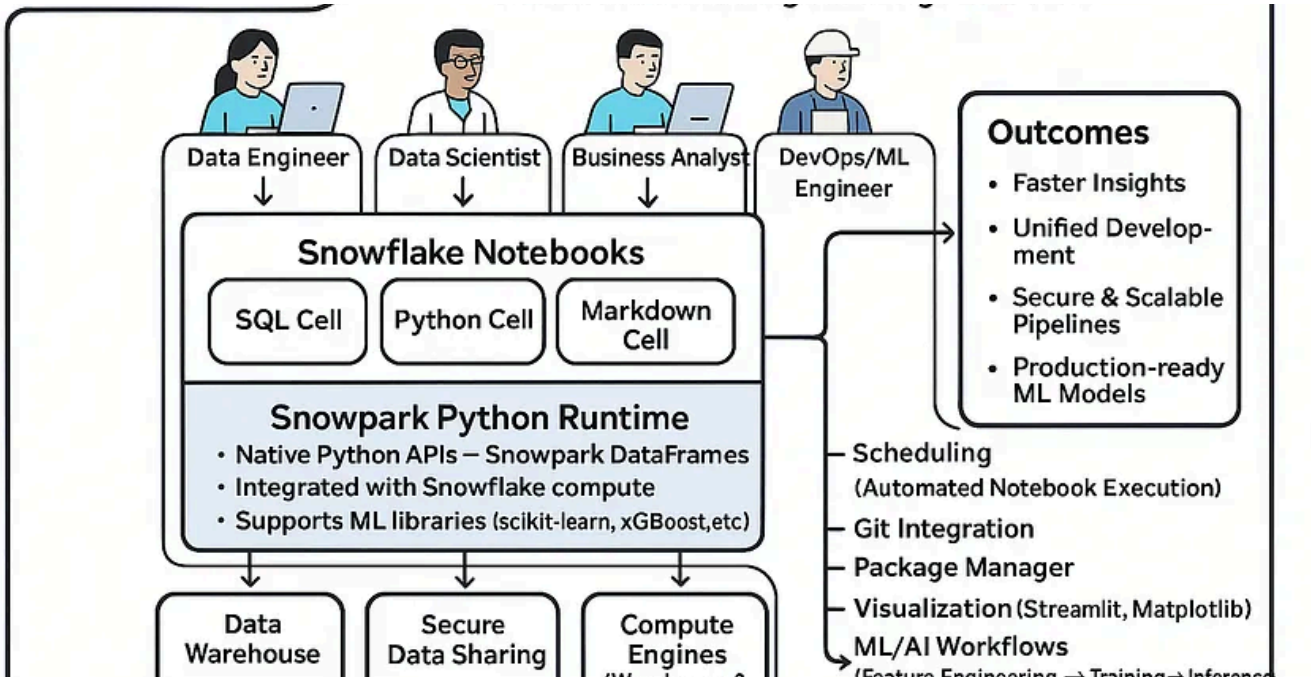



Jeyaram Ayyalusamy

### 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

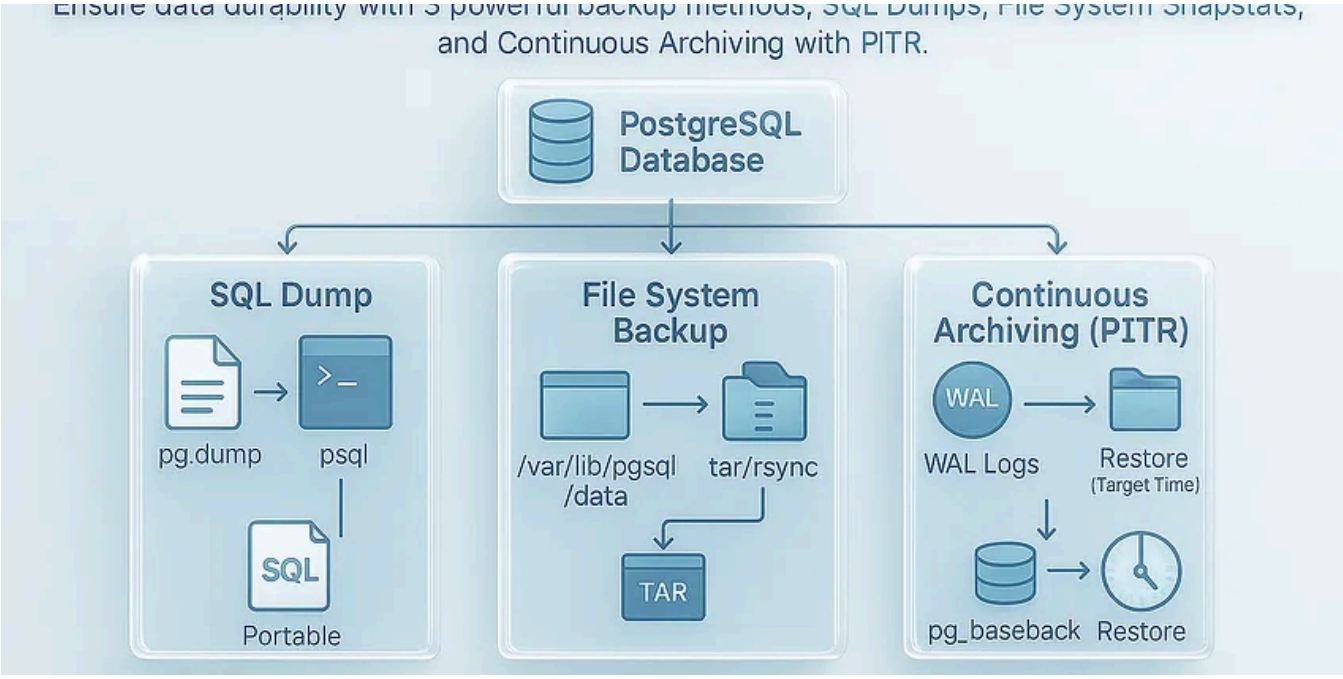



Jeyaram Ayyalusamy

## 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

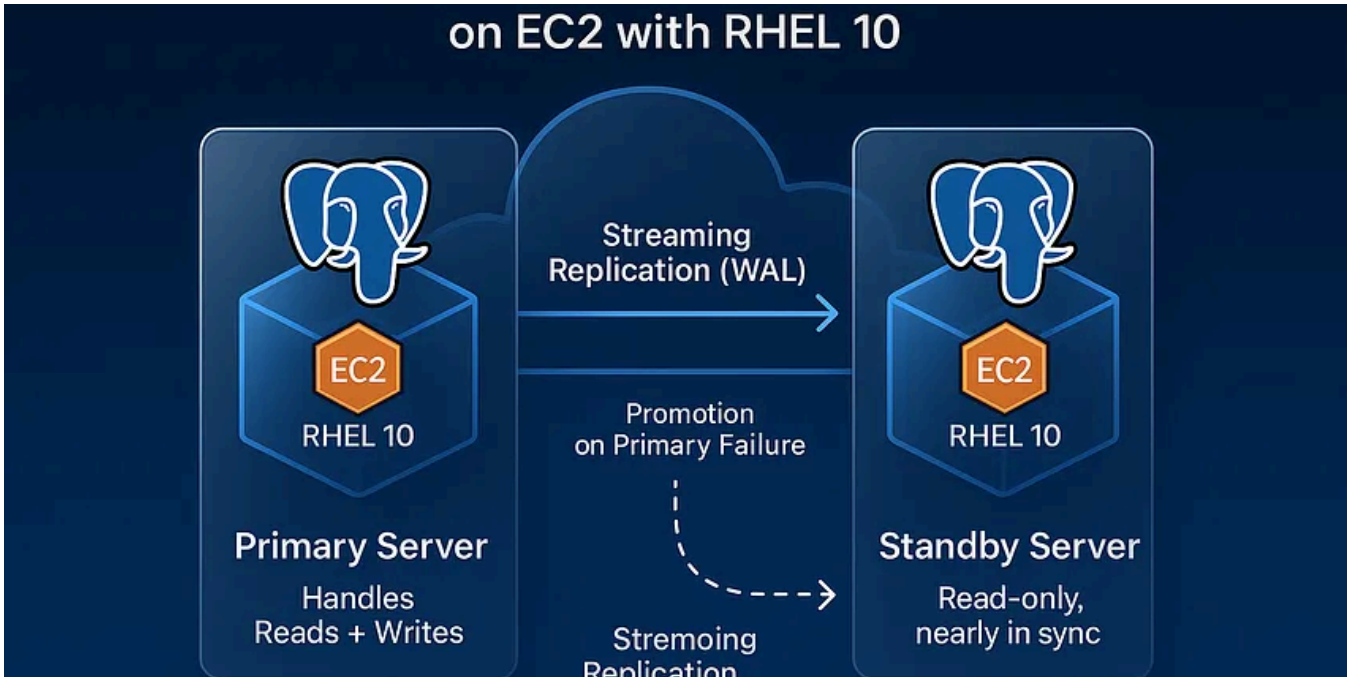



Jeyaram Ayyalusamy

## 🛡️ 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

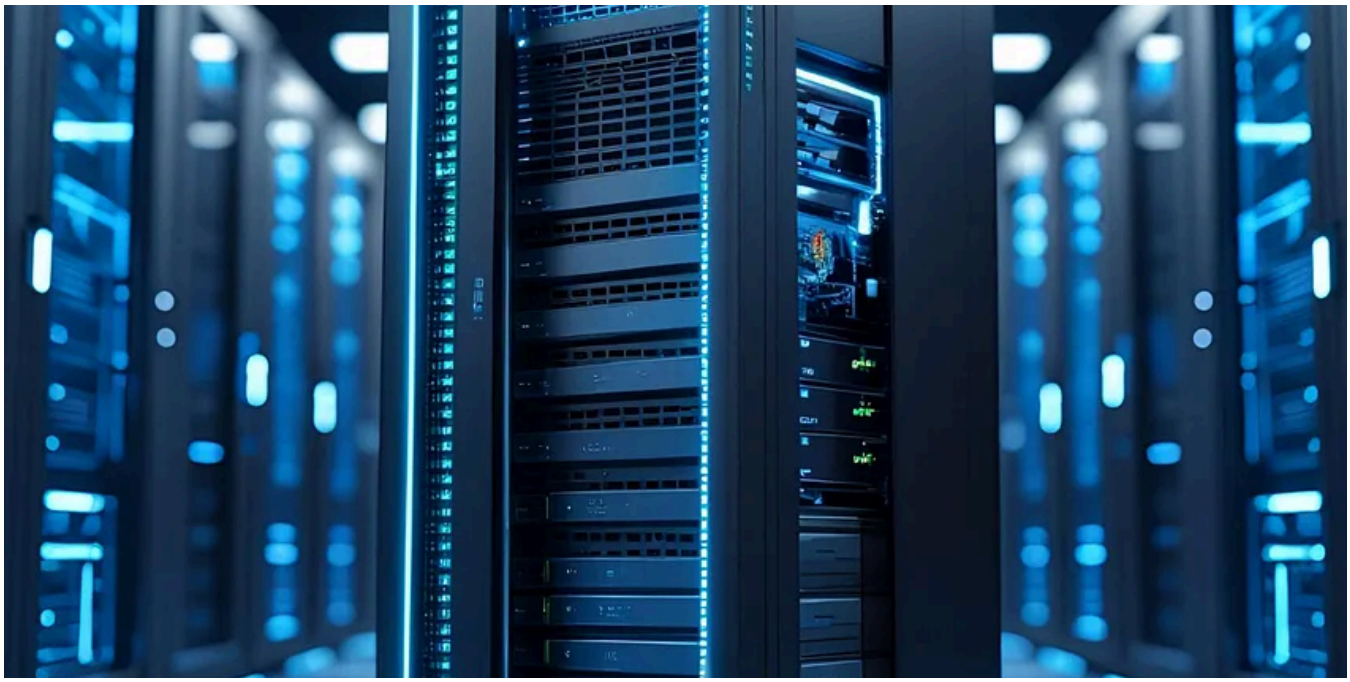
## 🚀 Streaming Replication in PostgreSQL 17 on EC2 with RHEL 10—A Deep Dive

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

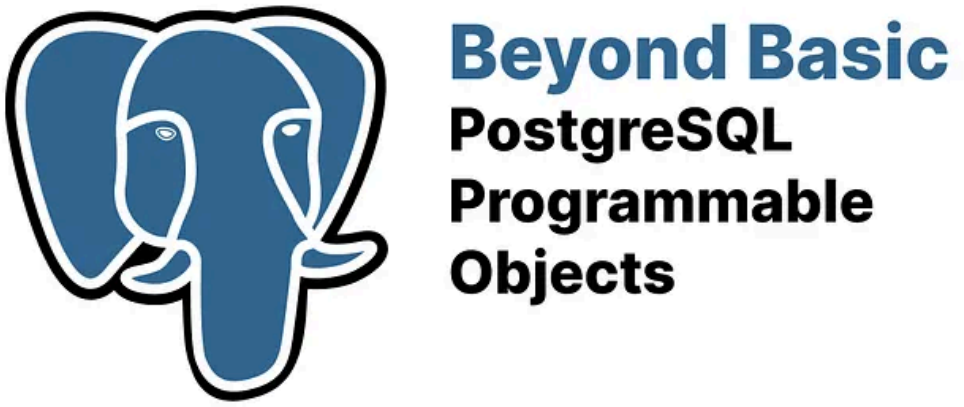
Sep 15 11 1


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

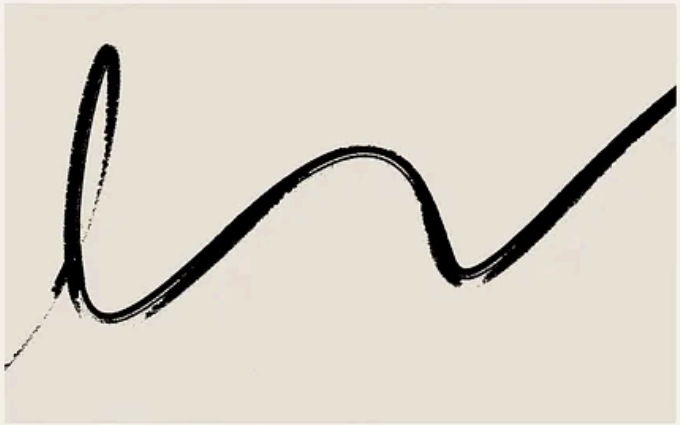



In Stackademic by bektiaw

## 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



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

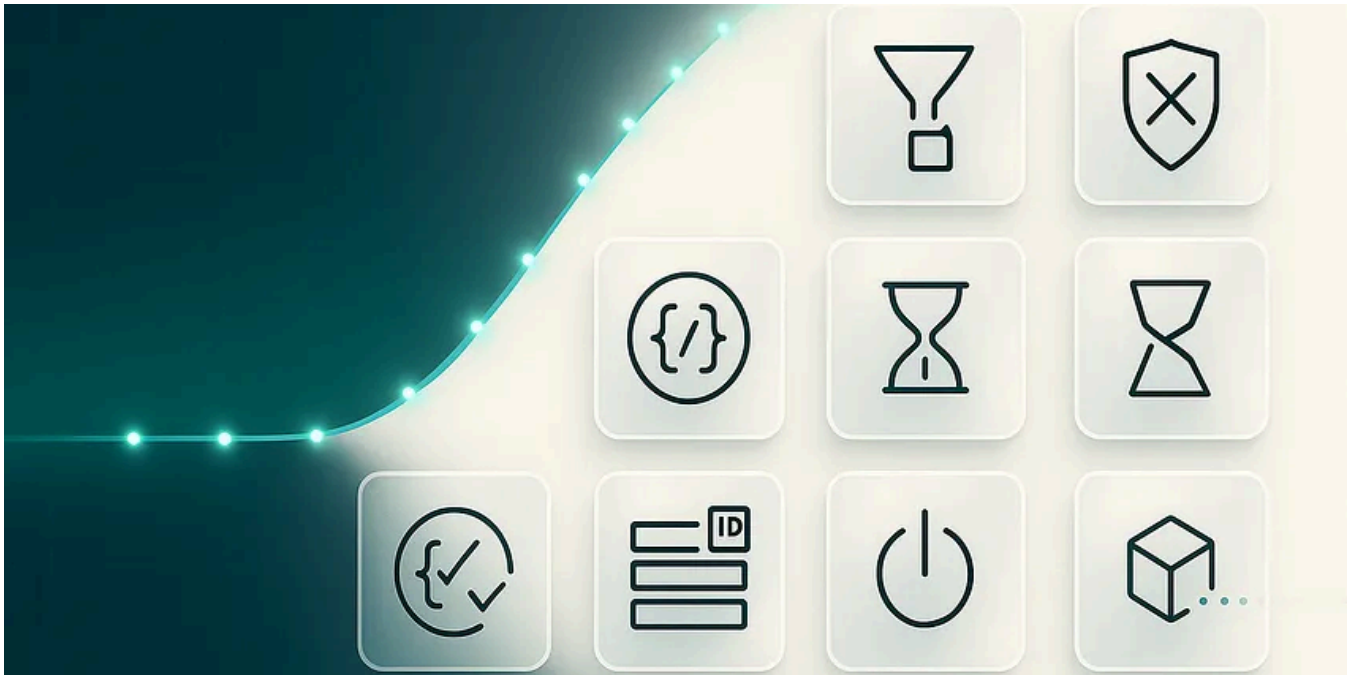
Jul 18 12 1


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



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

See more recommendations