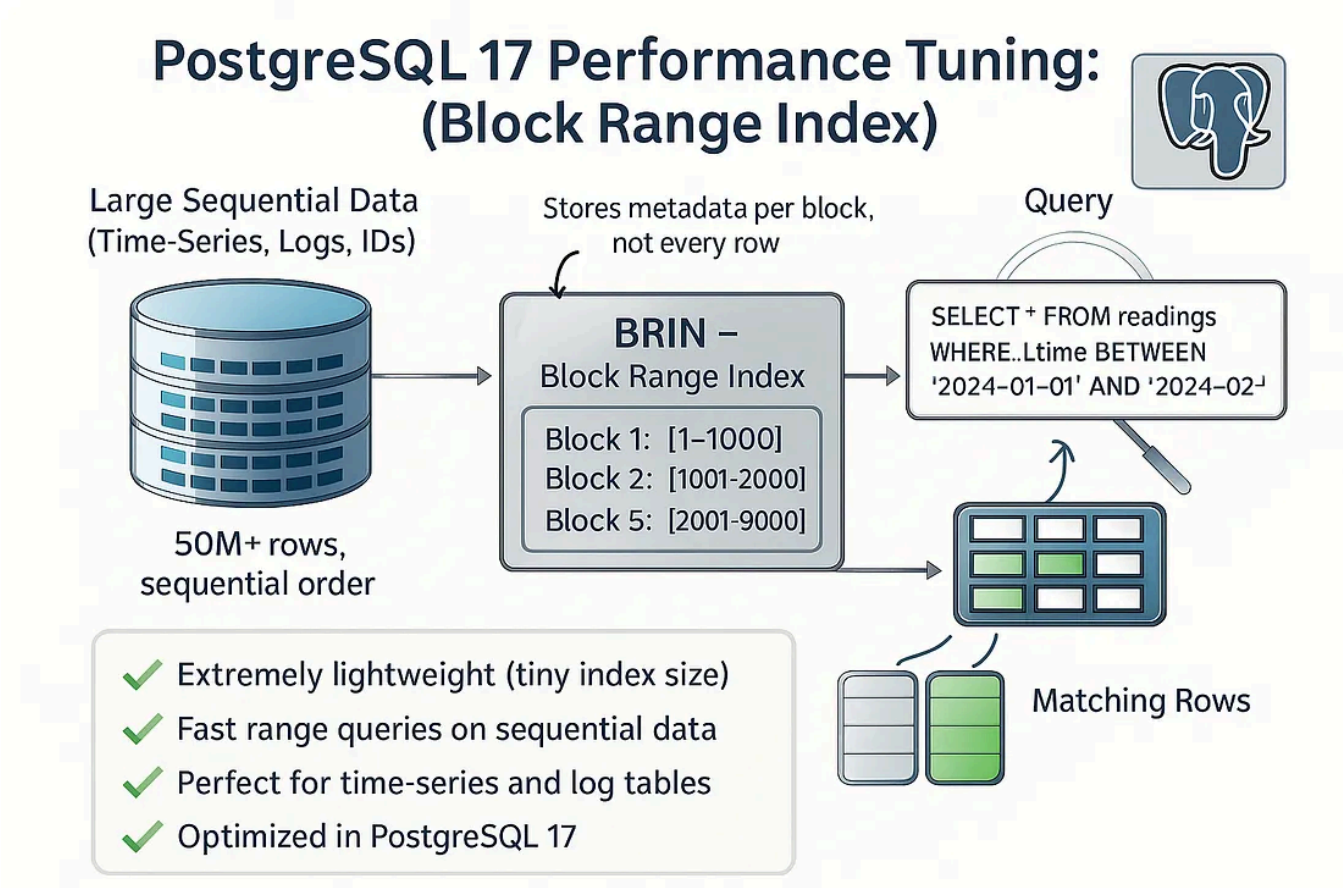
22 - PostgreSQL 17 Performance Tuning: BRIN (Block Range INdex)

5 min read · Sep 7, 2025

J Jeyaram Ayyalusamy Following



When working with **very large tables** in PostgreSQL, traditional indexes like B-Tree or GiST can grow extremely large and take significant time to maintain. For time-series or sequentially ordered data, PostgreSQL offers a smarter, more lightweight option: the **BRIN (Block Range Index)**.

BRIN indexes summarize data in block ranges instead of storing individual row entries, making them **compact, fast to build, and efficient for range queries** on ordered data.

Let's walk through a step-by-step example.

Step 1: Create the `weather_data` table

We'll simulate sensor readings (temperature and humidity) recorded over time.

```
CREATE TABLE weather_data (
    reading_id   BIGSERIAL,
    reading_time TIMESTAMP NOT NULL,
    temperature  NUMERIC(5,2),
    humidity     NUMERIC(5,2)
);
```

```
postgres=# CREATE TABLE weather_data (
    reading_id   BIGSERIAL,
    reading_time TIMESTAMP NOT NULL,
    temperature  NUMERIC(5,2),
    humidity     NUMERIC(5,2)
);
CREATE TABLE
postgres=#
```

- `reading_id` : A unique identifier for each reading.
- `reading_time` : Sequential timestamps, perfect for BRIN indexing.
- `temperature` and `humidity` : Numeric values for our weather data.

Step 2: Insert 50 million rows

Now let's load the table with **50 million rows** of sequential data.

```
-- Insert 50M rows of sequential readings
INSERT INTO weather_data (reading_time, temperature, humidity)
SELECT
   NOW() - (g || ' seconds')::INTERVAL,   -- sequential timestamps
   (20 + random()*15)::NUMERIC(5,2),      -- temperature between 20-35
   (40 + random()*30)::NUMERIC(5,2)       -- humidity between 40-70
FROM generate_series(1, 50000000) AS g;
```

```
postgres=# -- Insert 50M rows of sequential readings
INSERT INTO weather_data (reading_time, temperature, humidity)
SELECT
   NOW() - (g || ' seconds')::INTERVAL,   -- sequential timestamps
   (20 + random()*15)::NUMERIC(5,2),      -- temperature between 20-35
   (40 + random()*30)::NUMERIC(5,2)       -- humidity between 40-70
FROM generate_series(1, 50000000) AS g;
INSERT 0 50000000
postgres=#
```

Here's what's happening:

- We generate 50M readings spaced one second apart.
- Each row has a random temperature (20–35°C) and humidity (40–70%).
- Since timestamps are sequential, PostgreSQL stores them in ordered **block ranges**, making BRIN a great fit.

Step 3: Run ANALYZE

```
ANALYZE weather_data;
```

```
postgres=# ANALYZE weather_data;
ANALYZE
postgres=#
```

This updates PostgreSQL statistics so the planner can make informed choices.

Step 4: Query without index

Now, let's query for data in a **1-day window** without any index.

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM weather_data
WHERE reading_time BETWEEN '2024-01-01' AND '2024-01-02';
```

Output (simplified):

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM weather_data
WHERE reading_time BETWEEN '2024-01-01' AND '2024-01-02';
                                                                QUER
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..681148.90 rows=1 width=28) (actual time=26909.049..2691
   Workers Planned: 2
   Workers Launched: 2
```

```
      ->  Parallel Seq Scan on weather_data  (cost=0.00..680148.80 rows=1 width=28
            Filter: ((reading_time >= '2024-01-01 00:00:00'::timestamp without tim
            Rows Removed by Filter: 16666667
 Planning Time: 0.126 ms
 Execution Time: 26914.260 ms
(8 rows)

postgres=#
```

- PostgreSQL performs a **sequential scan** across all 50M rows.
- This takes around **26.914 seconds** because every row is checked.

Step 5: Create a BRIN index

Now, let's build a BRIN index on the `reading_time` column.

```
CREATE INDEX idx_weather_time_brin ON weather_data USING brin (reading_time);
```

```
postgres=# CREATE INDEX idx_weather_time_brin ON weather_data USING brin (readi
CREATE INDEX
postgres=#
```

- Unlike a B-Tree, this index stores **summaries of block ranges**, not individual row entries.
- This makes it **extremely lightweight**, even on massive datasets.

Run ANALYZE

```
ANALYZE weather_data;
```

```
postgres=# ANALYZE weather_data;
ANALYZE
postgres=#
```

This updates PostgreSQL statistics so the planner can make informed choices.

Step 6: Query with BRIN index

Let's run the same query again.

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM weather_data
WHERE reading_time BETWEEN '2024-01-01' AND '2024-01-02';
```

Output (simplified):

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM weather_data
WHERE reading_time BETWEEN '2024-01-01' AND '2024-01-02';
                                                                        QU
-------------------------------------------------------------------------------
 Bitmap Heap Scan on weather_data  (cost=50.03..57347.34 rows=1 width=28) (actu
   Recheck Cond: ((reading_time >= '2024-01-01 00:00:00'::timestamp without tim
   ->  Bitmap Index Scan on idx_weather_time_brin  (cost=0.00..50.03 rows=17403
```

```
          Index Cond: ((reading_time >= '2024-01-01 00:00:00'::timestamp without
 Planning Time: 0.155 ms
 Execution Time: 0.341 ms
(6 rows)

postgres=#
```

- PostgreSQL now uses the **BRIN index**.
- Instead of scanning all 50M rows, it jumps directly to block ranges likely containing the data.
- Execution time dropped from **26.914 seconds to 0.341 ms**.

That's nearly a 79,000x speed improvement with a very small index footprint.

Step 7: Compare index sizes

You can check the index size with:

```
postgres=# \di+ idx_weather_time_brin;
```

```
postgres=# \di+ idx_weather_time_brin;
                                                    List of relations
 Schema |          Name          | Type  |  Owner   |    Table     | Persistence
--------+------------------------+-------+----------+--------------+------------
 public | idx_weather_time_brin | index | postgres | weather_data | permanent
(1 row)

postgres=#
```

- For 50M rows, the BRIN index is typically **just a few MB**.
- A B-Tree index on the same column could easily take up **multiple GB**.

This makes BRIN especially attractive when working with **huge time-series or log tables.**

Why BRIN Works Well

- ✅ **Efficient for sequential data:** Time, IDs, or ordered values.
- ✅ **Compact:** Tiny footprint compared to B-Trees.
- ✅ **Fast to create and maintain:** Great for massive tables.
- ⚠️ **Not ideal for point lookups:** If you frequently search for a single exact value, B-Tree is better.

Final Thoughts

In PostgreSQL 17, **BRIN indexes remain a powerful choice for time-series and big-data workloads.** If your dataset is huge and your queries mostly involve ranges on sequential data, BRIN provides a near-perfect balance of speed and efficiency.

Instead of letting your indexes consume gigabytes of storage, try BRIN — a small index that delivers big performance improvements. 🚀

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

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

Responses (1)

Gvadakte

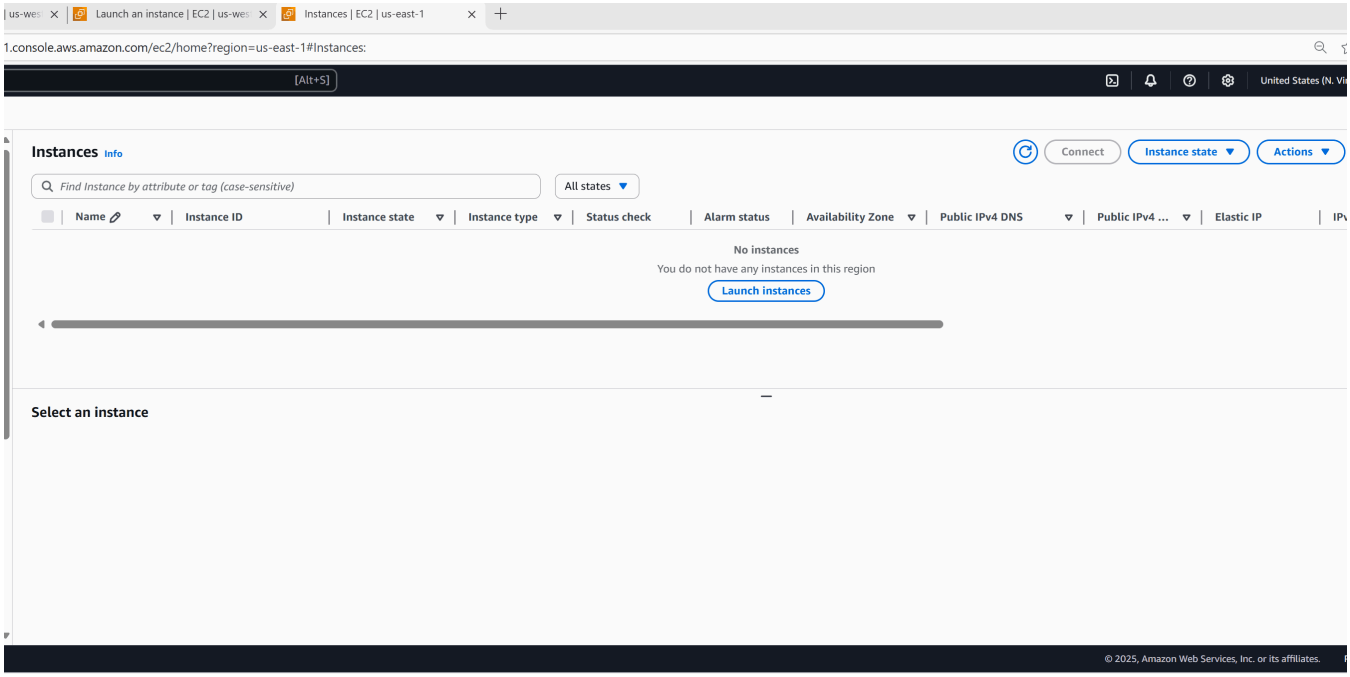
What are your thoughts?

Chigozie Damian Okali
20 hours ago

Quite insightful

Reply

More from Jeyaram Ayyalusamy

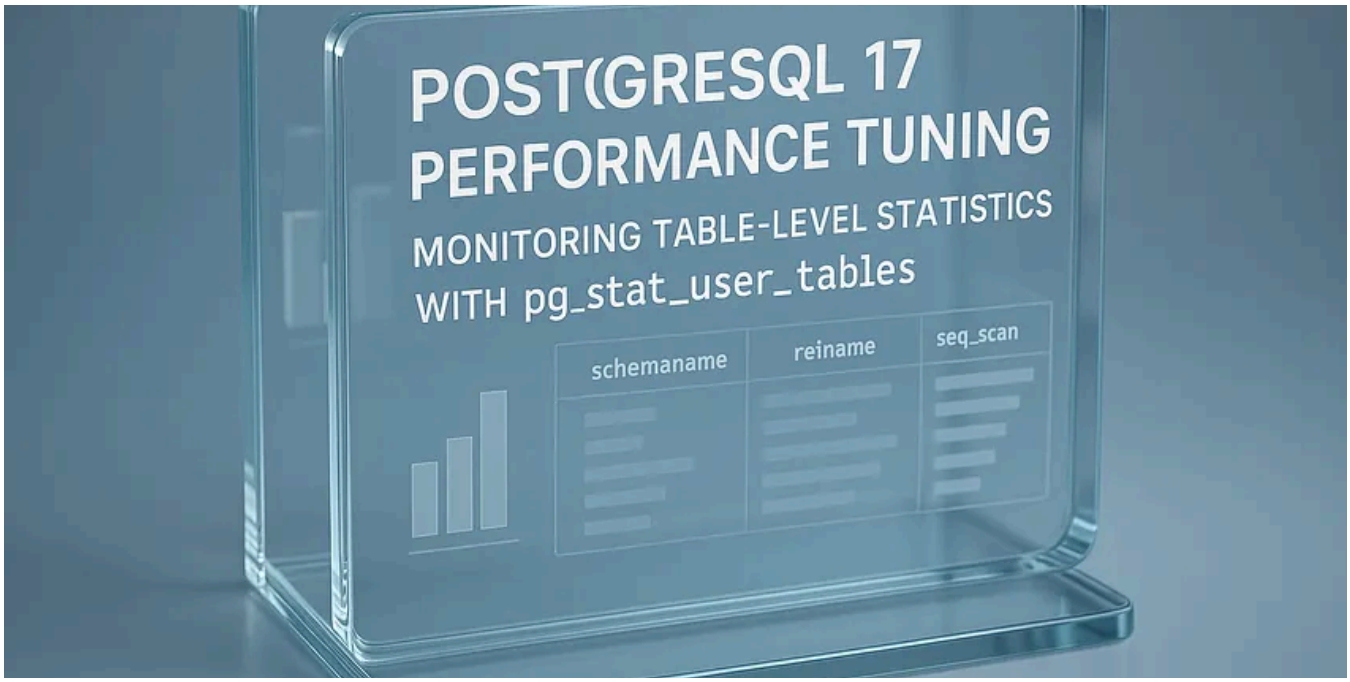


Jeyaram Ayyalusamy

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

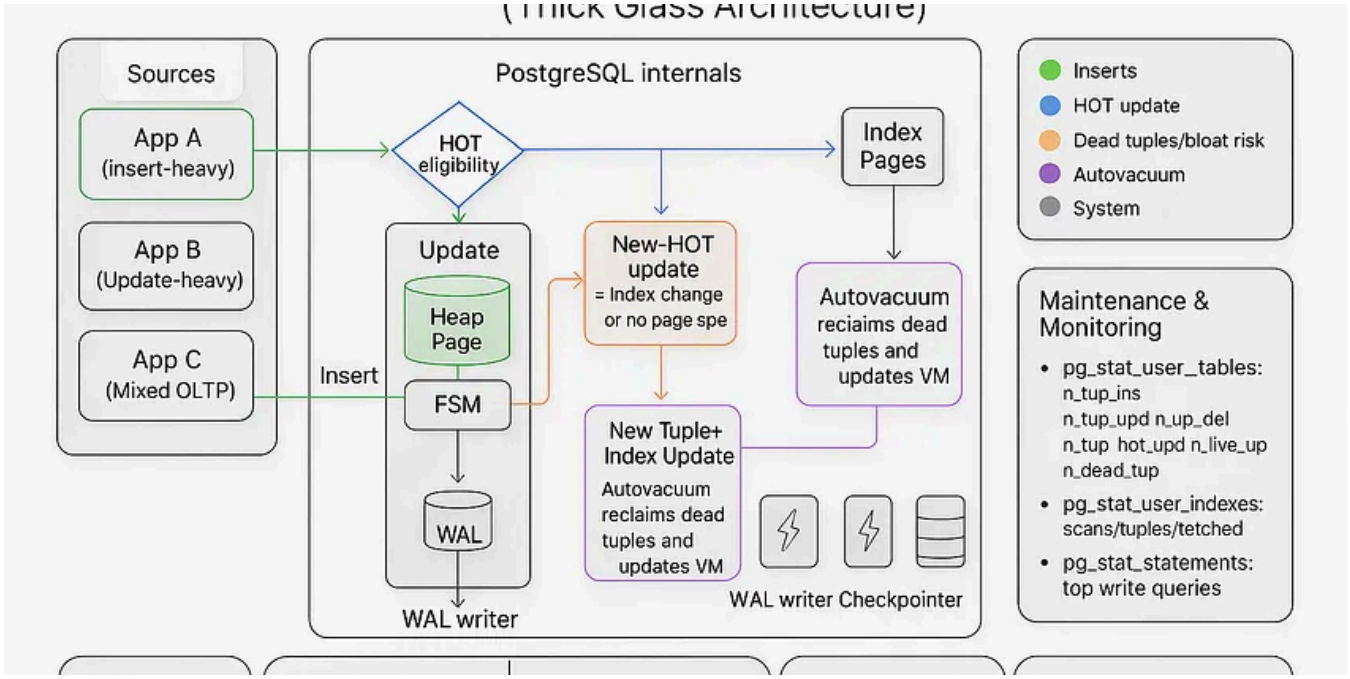


Jeyaram Ayyalusamy

24 - PostgreSQL 17 Performance Tuning: Monitoring Table-Level Statistics with pg_stat_user_tables

When tuning PostgreSQL, one of the most important steps is to observe table-level statistics. You cannot optimize what you cannot measure...

Sep 7 19 1

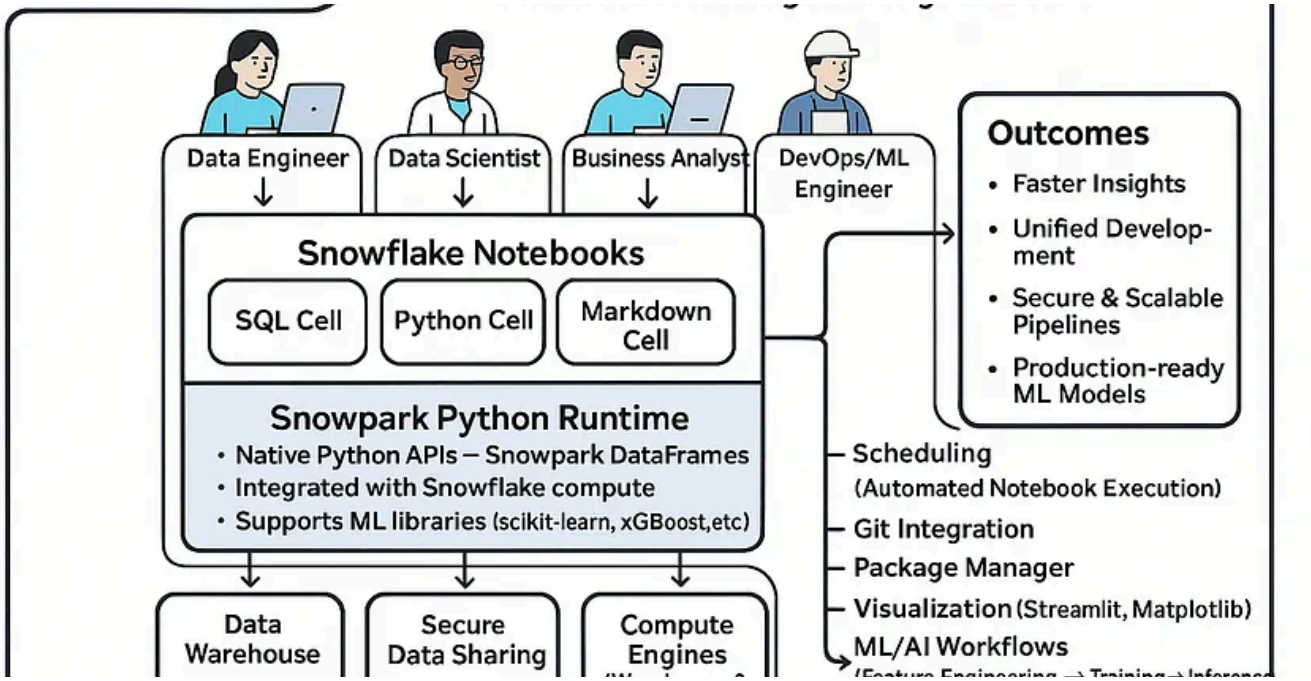


Jeyaram Ayyalusamy

23 - PostgreSQL 17 Performance Tuning: Monitoring Inserts, Updates, and HOT Updates

When tuning PostgreSQL, it is very important to understand the INSERT, UPDATE, and DELETE patterns of your tables. Different workloads...

Sep 7 11



Jeyaram Ayyalusamy

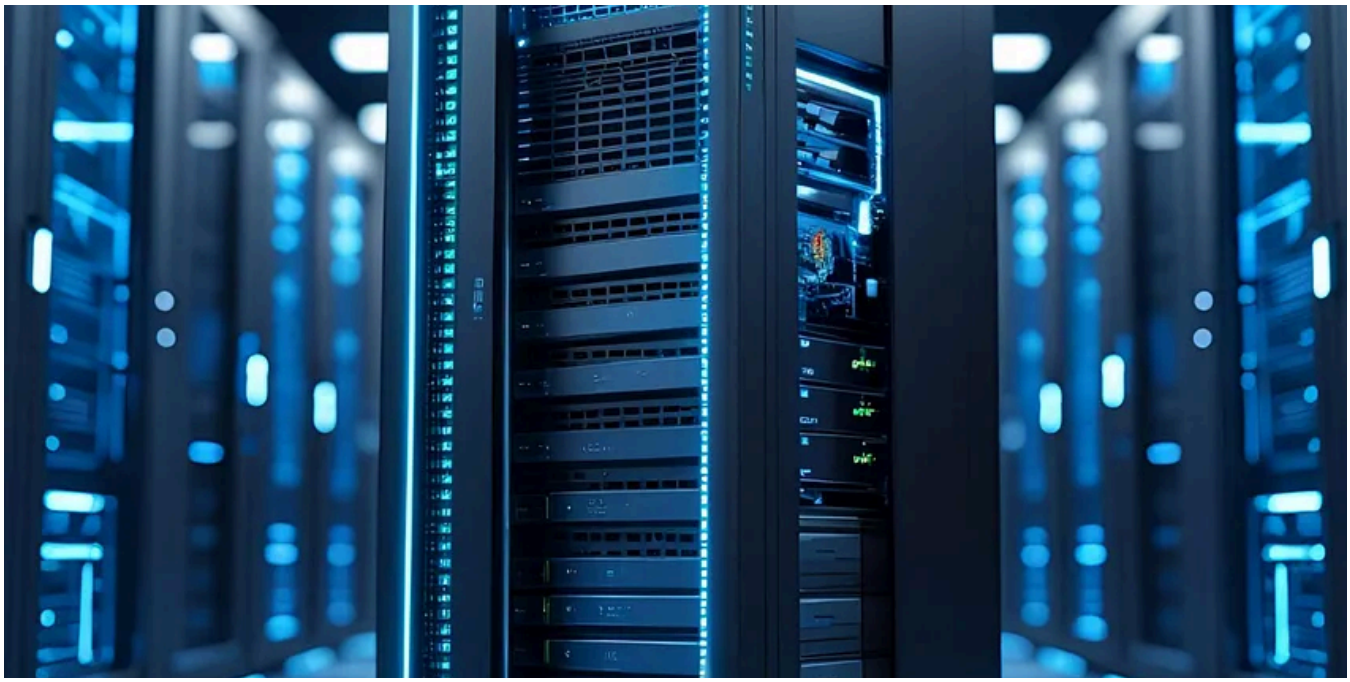
16—Experience Snowflake with Notebooks and Snowpark Python: A Unified Data Engineering Platform

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

Jul 13

See all from Jeyaram Ayyalusamy

Recommended from Medium

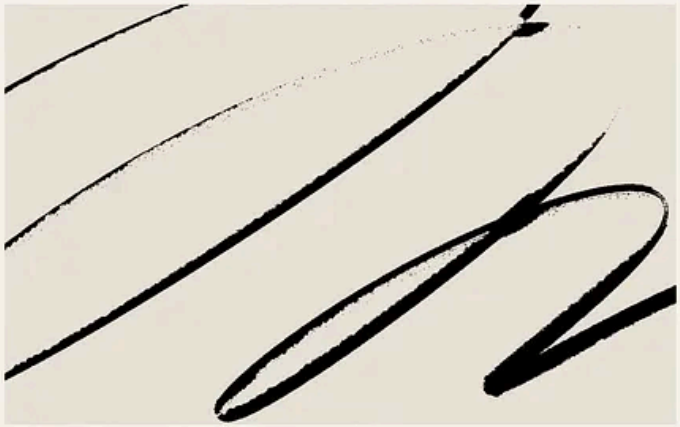


Rizqi Mulki

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

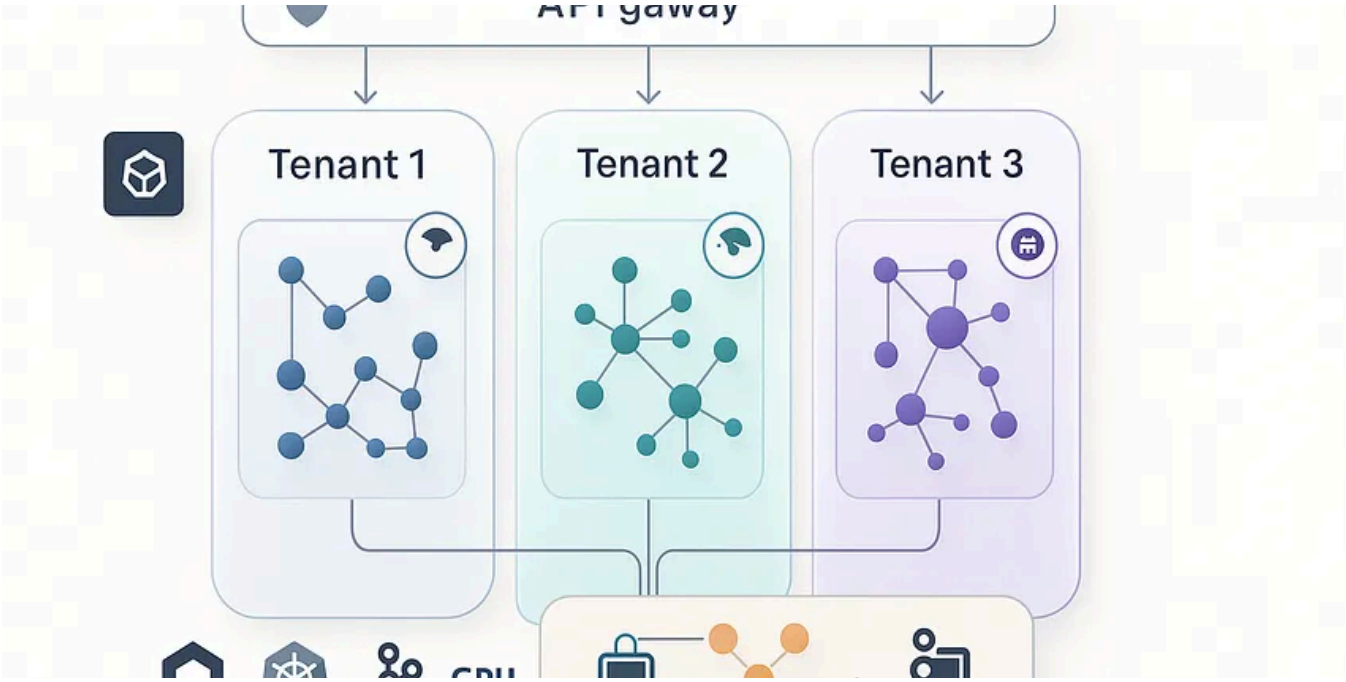


Vahid Yousefzadeh

Real-Time Statistics in Oracle 19c

Until Oracle version 12c, executing DML statements on a table (either using the conventional method or direct-path) would not update the...

Aug 13

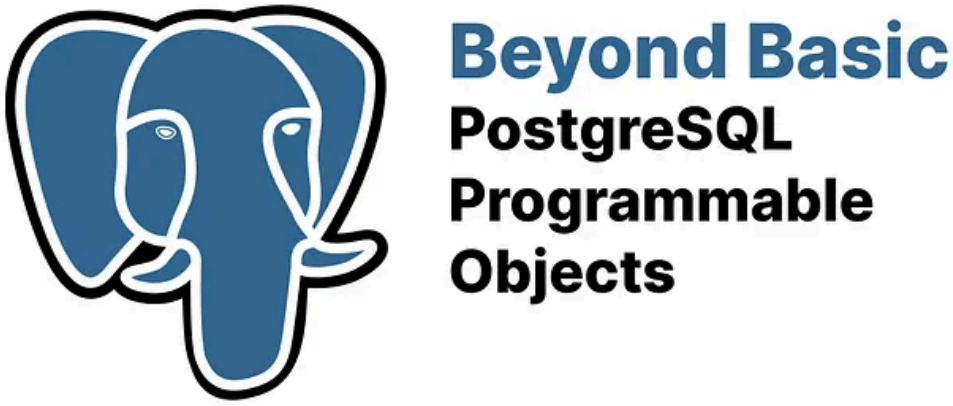


Thinking Loop

5 Ironclad Rules for Multi-Tenant Vector Isolation

Keep noisy neighbors out and SLAs intact—without overprovisioning your vector cluster.

Sep 15 17

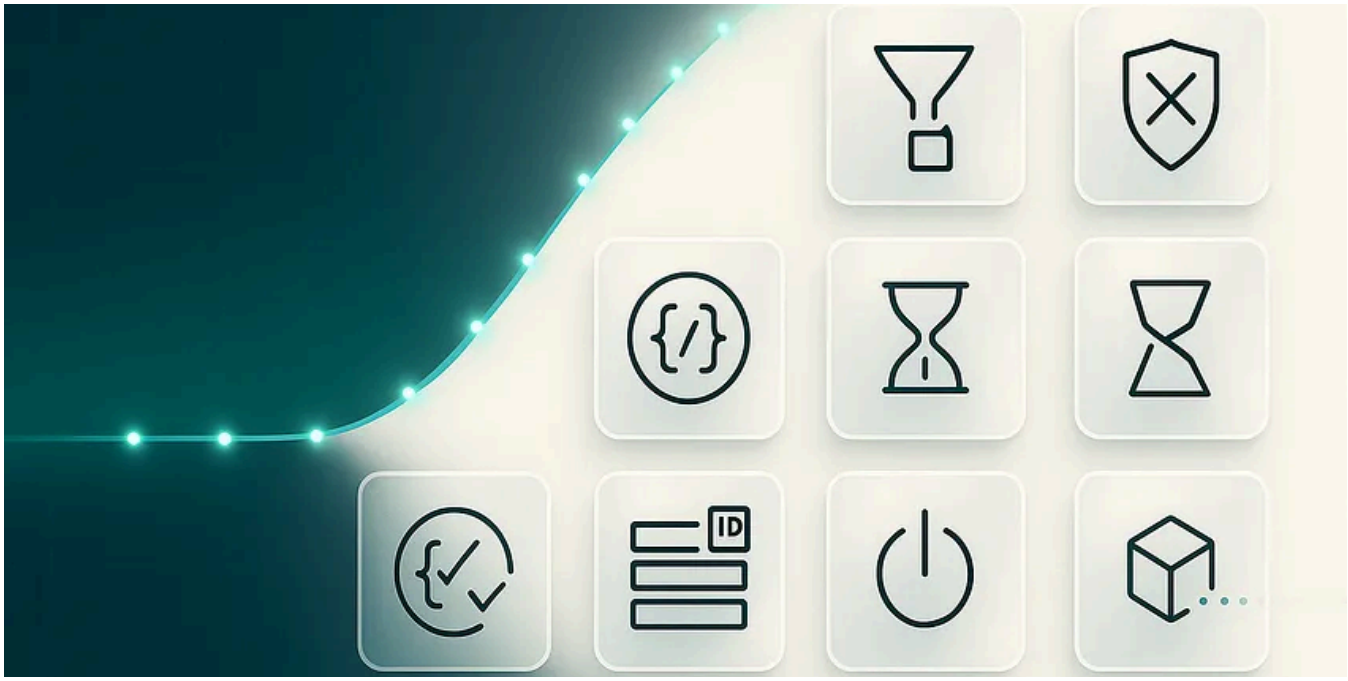


In Stackademic by bektiaw

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



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


Azlan Jamal

Mastering PostgreSQL Array Types: When, Why, and How to Use Them

PostgreSQL's array type is a powerful feature that lets you store multiple values of the same data type in a single column. It's one of...

Aug 11

See more recommendations