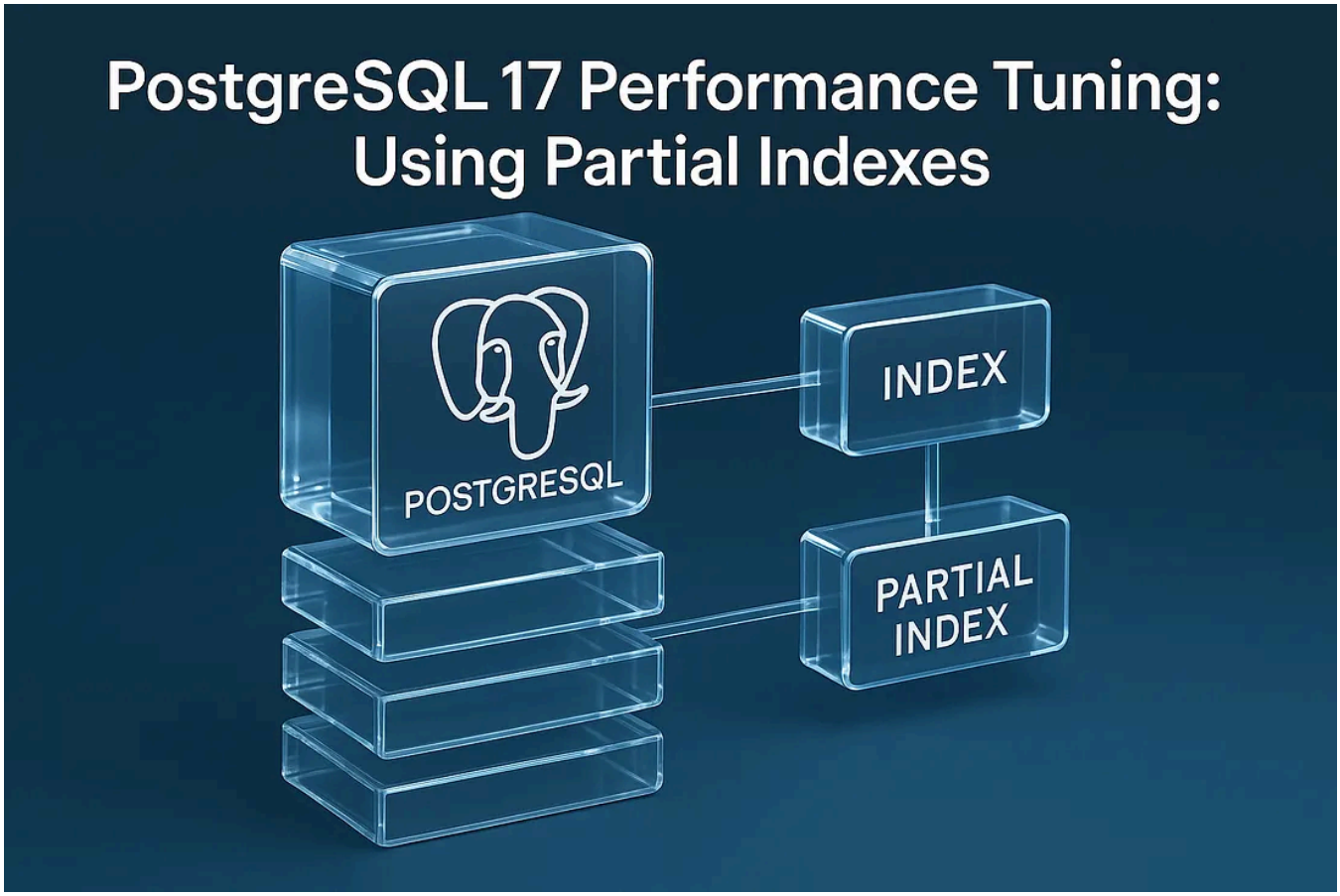
17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes

6 min read · Sep 4, 2025

Jeyaram Ayyalusamy Following



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.** This type of index is called a **partial index.**

A partial index is smaller than a full index because it only covers rows that match a condition. Since there's less data in the index, it is faster to scan and cheaper to maintain. This is especially useful when queries consistently filter for a **small subset** of rows.

Why Use Partial Indexes?

Imagine a **task management system** stored in a `products`-like table. Rows representing tasks still pending are flagged as **`open`**, while completed ones are marked as **`done`**.

Over time, most rows will be marked **`done`**, and only a small fraction will be **`open`**. But in practice, users will frequently query the **`open`** tasks.

👉 If we create a full index on the `status` column, PostgreSQL will index all 10 million rows (both `open` and `done`). This consumes space and slows down writes, even though most queries only care about the small subset (`open`).

👉 Instead, if we create a **partial index only for `status = 'open'`**, PostgreSQL can use a much smaller, faster index, while ignoring rows that don't matter.

Step 1: Create Products Table with 10 Million Rows

```
CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    status       TEXT,         -- 'open' or 'done'
    price        NUMERIC,
    stock_qty    INT
);
```

```
postgres=# CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    status       TEXT,         -- 'open' or 'done'
    price        NUMERIC,
    stock_qty    INT
);
CREATE TABLE
postgres=#
```

```
-- 1M rows with status = 'open' (~10%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'open'
FROM generate_series(1, 1000000) g;

-- 2.5M rows with status = 'done' (~25%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'done'
FROM generate_series(1000001, 3500000) g;

-- 3M rows with status = 'denied' (~30%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'denied'
FROM generate_series(3500001, 6500000) g;
```

```
-- 3.5M rows with status = 'new' (~35%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'new'
FROM generate_series(6500001, 10000000) g;
```

```
postgres=# -- 1M rows with status = 'open' (~10%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'open'
FROM generate_series(1, 1000000) g;

-- 2.5M rows with status = 'done' (~25%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'done'
FROM generate_series(1000001, 3500000) g;

-- 3M rows with status = 'denied' (~30%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'denied'
FROM generate_series(3500001, 6500000) g;

-- 3.5M rows with status = 'new' (~35%)
INSERT INTO products (product_name, category, price, stock_qty, status)
SELECT
    'Product_' || g,
    'Category_' || (g % 50),
```

```
    round((random() * 500)::numeric, 2),
    (random() * 1000)::int,
    'new'
FROM generate_series(6500001, 10000000) g;

INSERT 0 1000000
INSERT 0 2500000
INSERT 0 3000000
INSERT 0 3500000
postgres=#
postgres=# ANALYZE products;
ANALYZE
postgres=#
```

```
postgres=# SELECT status, COUNT(*)
FROM products
GROUP BY status;
 status |  count
--------+---------
 denied | 3000000
 done   | 2500000
 new    | 3500000
 open   | 1000000
(4 rows)
```

Step 2: Full Index vs Partial Index

Full Index on Status

```
CREATE INDEX idx_products_status ON products(status);
```

```
postgres=# CREATE INDEX idx_products_status ON products(status);
CREATE INDEX
postgres=#
```

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

Covers **all 10M rows.**

- Size is large (hundreds of MBs).
- Every insert/update must maintain this index, even for rows where `status = 'done'`.

Query Example

Now run a query for open tasks:

```
EXPLAIN ANALYZE
SELECT * FROM products WHERE status = 'open';
```

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM products WHERE status = 'open';
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on products  (cost=10979.37..203257.63 rows=985669 width=49)
```

```
   Recheck Cond: (status = 'open'::text)
   Heap Blocks: exact=10310
   ->  Bitmap Index Scan on idx_products_status  (cost=0.00..10732.95 rows=9856
         Index Cond: (status = 'open'::text)
 Planning Time: 1.067 ms
 Execution Time: 6343.627 ms
(7 rows)

postgres=#
```

Partial Index on Status = ‘open’

```
DROP INDEX idx_products_status;
```

```
postgres=# DROP INDEX idx_products_status;
DROP INDEX
postgres=#
```

```
CREATE INDEX idx_products_open ON products(status) WHERE status = 'open';
```

```
postgres=# CREATE INDEX idx_products_open ON products(status) WHERE status = 'c
CREATE INDEX
postgres=#
```

```
analyze products;
```

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

- Covers only **1M rows** (10% of total).
- Much smaller, cheaper to maintain.
- Perfect for queries that only look at `open` rows.

Step 3: Query Example

Now run a query for open tasks:

```
EXPLAIN ANALYZE
```

```
SELECT * FROM products WHERE status = 'open';
```

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM products WHERE status = 'open';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Bitmap Heap Scan on products  (cost=8417.18..200578.87 rows=956335 width=49) (
   Recheck Cond: (status = 'open'::text)
   Heap Blocks: exact=10310
   ->  Bitmap Index Scan on idx_products_open  (cost=0.00..8178.10 rows=956335
 Planning Time: 1.010 ms
 Execution Time: 1294.439 ms
(6 rows)

postgres=#
```

👉 PostgreSQL will use `idx_products_open`, skipping all the `done` rows. The query is much faster because the index is tiny.

If you query for `status = 'done'`, PostgreSQL won't use this index — it will either do a sequential scan or use another index, depending on the query.

Step 4: Measuring Index Space

Check how much space indexes use compared to the table:

```
SELECT
    pg_size_pretty(pg_relation_size('products')) AS table_only,
    pg_size_pretty(pg_indexes_size('products')) AS indexes_only,
    pg_size_pretty(pg_total_relation_size('products')) AS total_with_indexes;
```

```
postgres=# SELECT
    pg_size_pretty(pg_relation_size('products')) AS table_only,
    pg_size_pretty(pg_indexes_size('products')) AS indexes_only,
    pg_size_pretty(pg_total_relation_size('products')) AS total_with_indexes;
 table_only | indexes_only | total_with_indexes
------------+--------------+--------------------
 805 MB     | 221 MB       | 1027 MB
(1 row)

postgres=#
```

It's common to see indexes taking **more space than the table itself.** That's why it's so important to measure index usage vs cost. Partial indexes help reduce wasted space.

When to Use Partial Indexes

Partial indexes make sense when:

- Your table has **a few very common values** (e.g., `status = 'done'`) and queries mostly target the **less frequent values** (e.g., `status = 'open'`).
- The excluded values cover at least **25% or more** of the table.
- You know your data patterns well (e.g., gender, nationality, active/inactive users).

Partial indexes trade **bigger space savings** and **faster lookups** for less flexibility, but when applied wisely, they can make queries significantly faster while reducing index maintenance costs.

Key Takeaways

- A full index on 10M rows can be huge and expensive to maintain.
- A partial index only on frequently queried rows (e.g., `status = 'open'`) is **smaller, cheaper, and faster.**
- Always measure **space usage vs query benefit** before adding indexes.
- PostgreSQL 17 makes partial indexes just as powerful as full ones — but far more efficient in the right scenarios.

✅ Partial indexes are a powerful tuning tool when your workload is focused on a **small subset of values** inside a very large table.

📢 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 Mongodb Open Source Oracle AWS

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

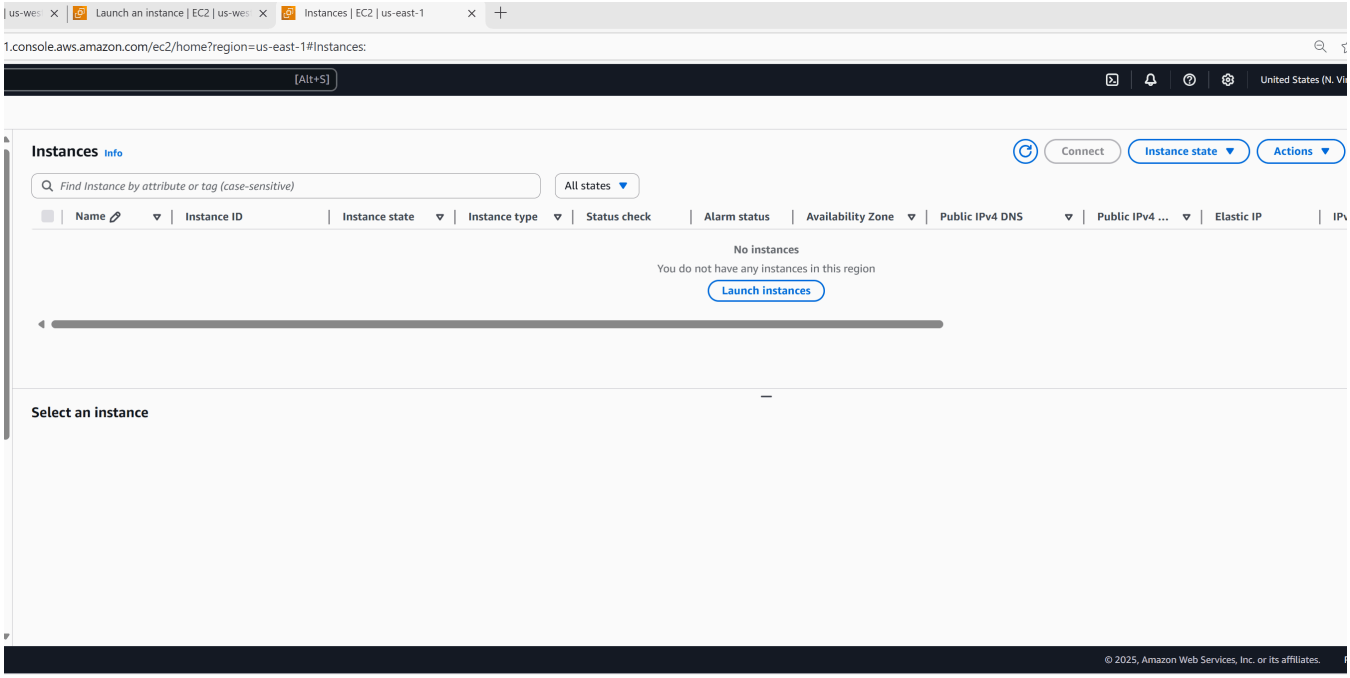
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

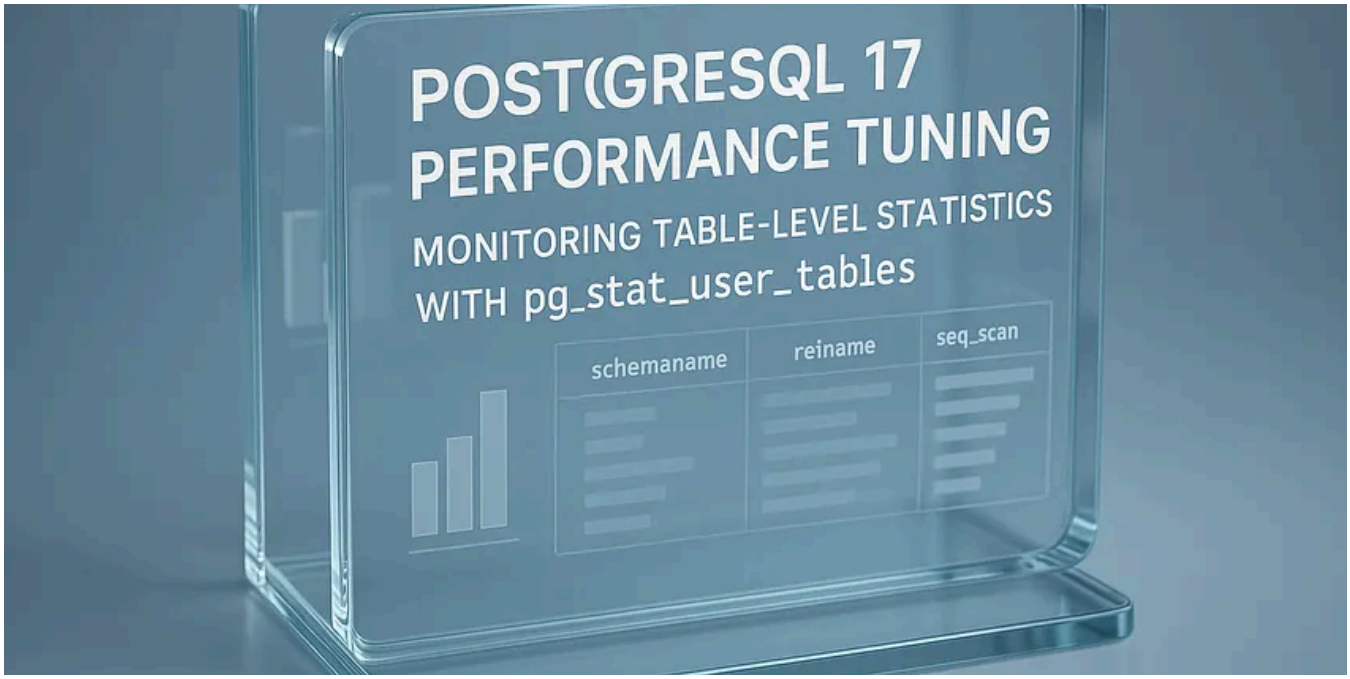


Jeyaram Ayyalusamy

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

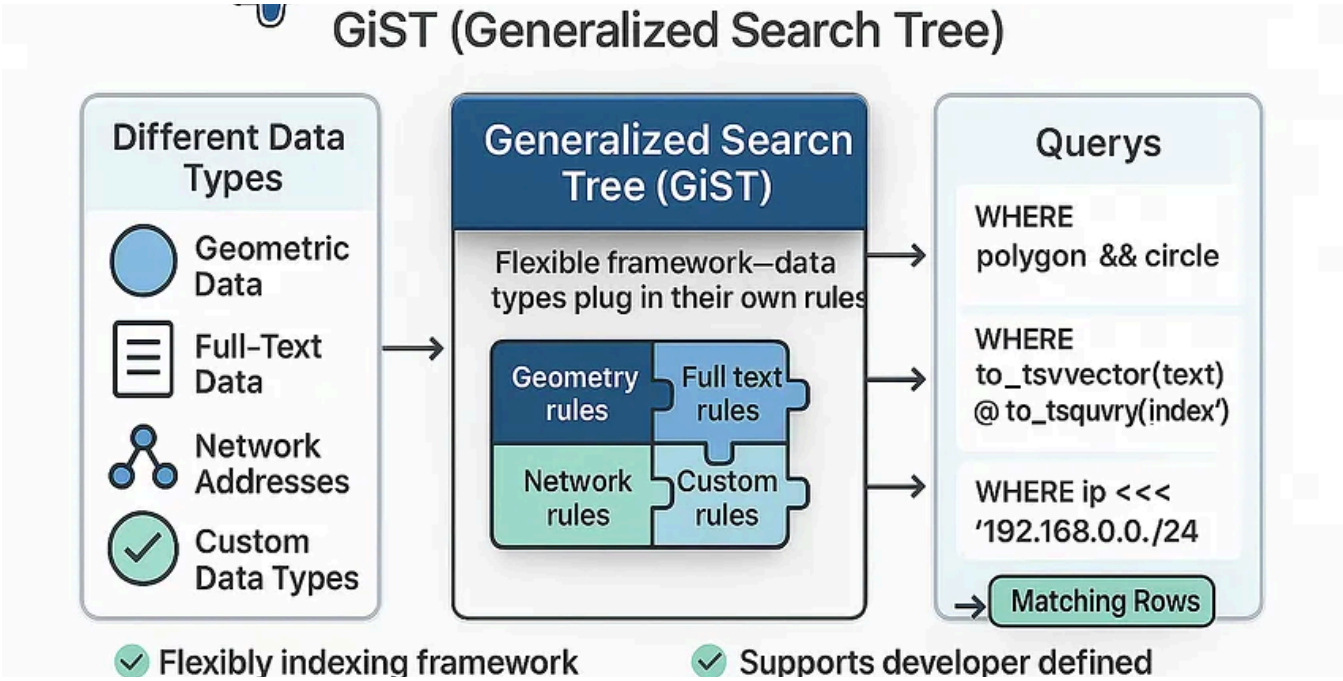


Jeyaram Ayyalusamy

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

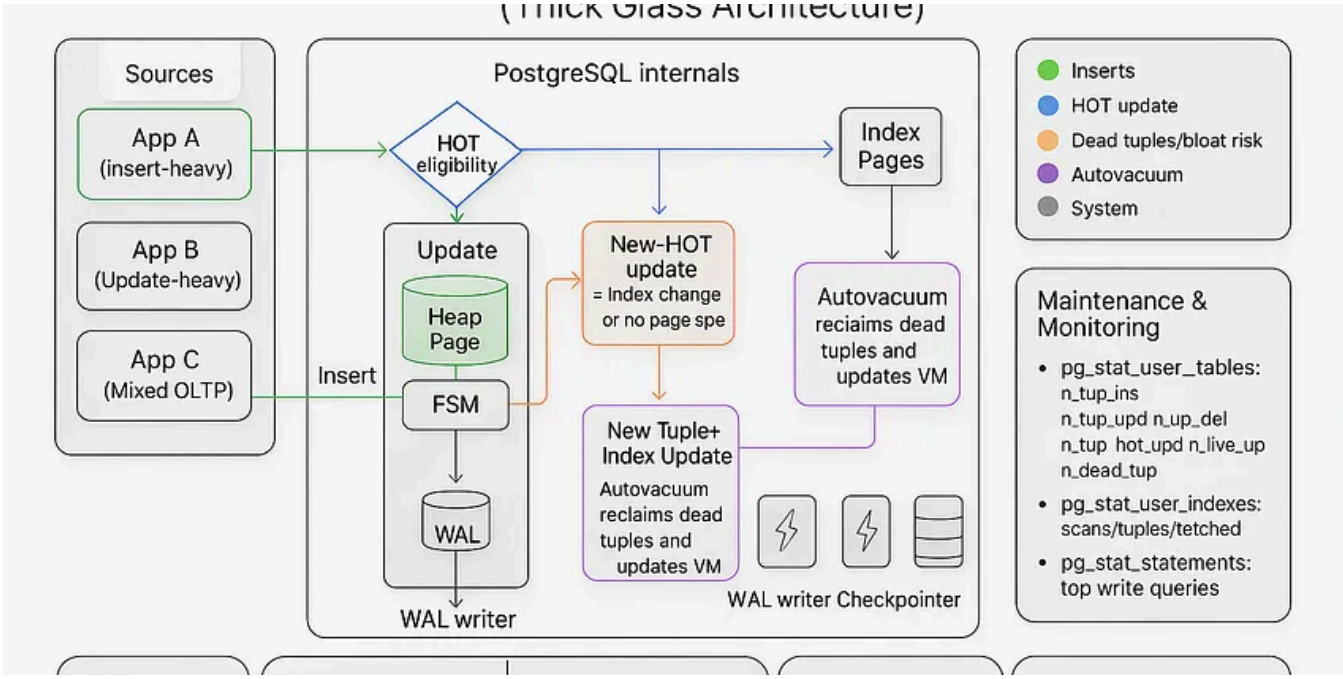


Jeyaram Ayyalusamy

21—PostgreSQL 17 Performance Tuning: GiST (Generalized Search Tree)

GiST (Generalized Search Tree) in PostgreSQL is a flexible indexing framework that allows developers to build indexes for many different...

Sep 5 1



Jeyaram Ayyalusamy

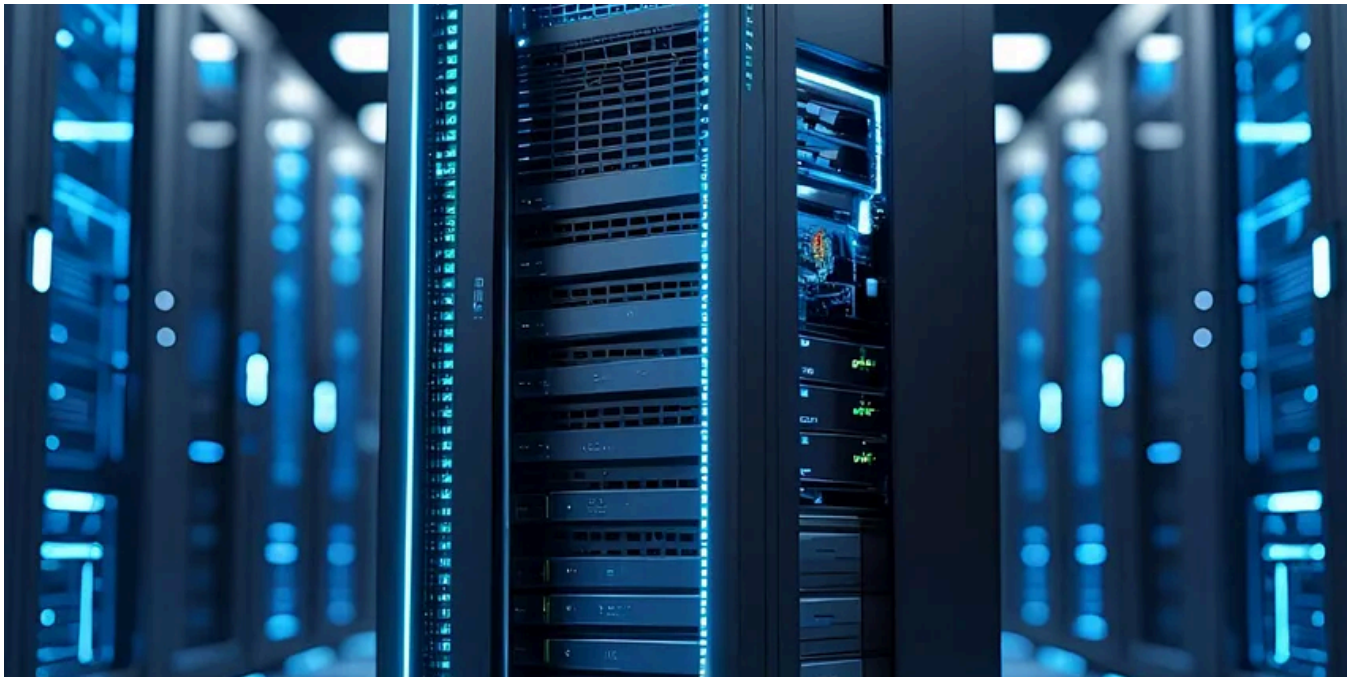
23 - PostgreSQL 17 Performance Tuning: Monitoring Inserts, Updates, and HOT Updates

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

Sep 7 11

See all from Jeyaram Ayyalusamy

Recommended from Medium



Rizqi Mulki

The Shocking Truth About PostgreSQL Locks (PostgreSQL documentation won't tell you)

Most developers are unknowingly creating database nightmares. Here's what the PostgreSQL documentation won't tell you.

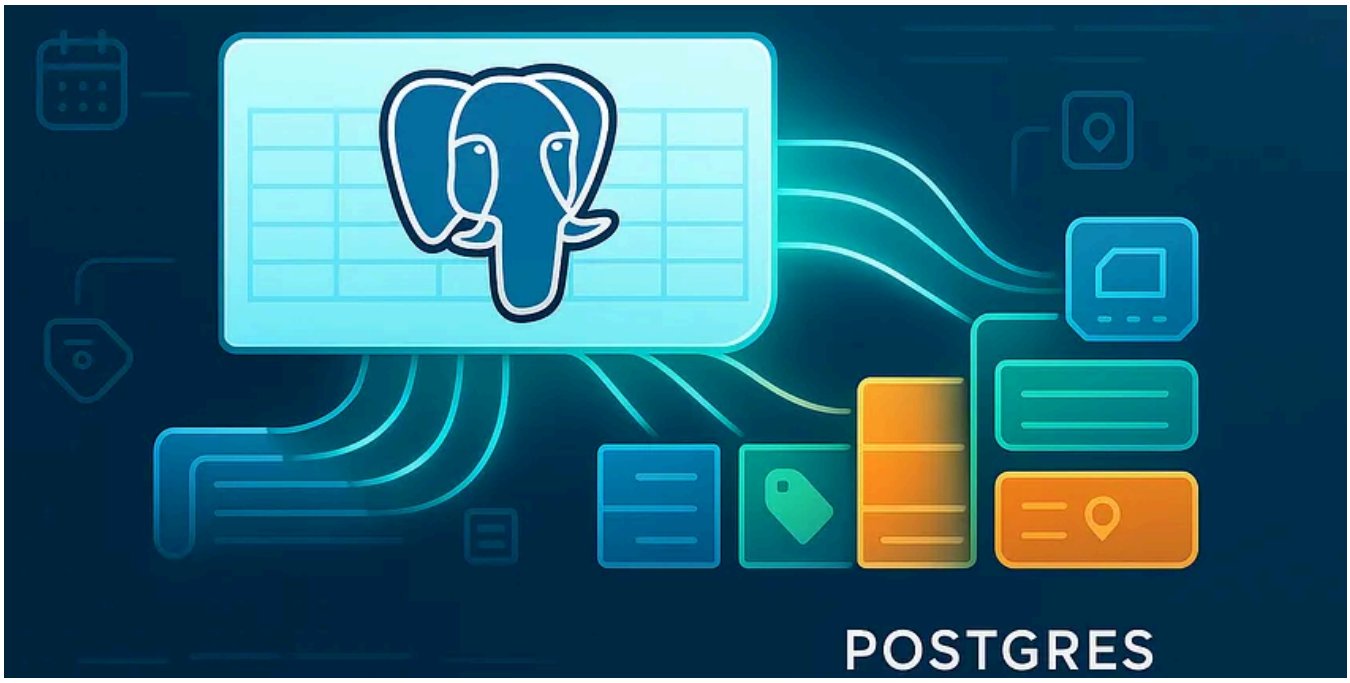
Sep 13 10 2

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

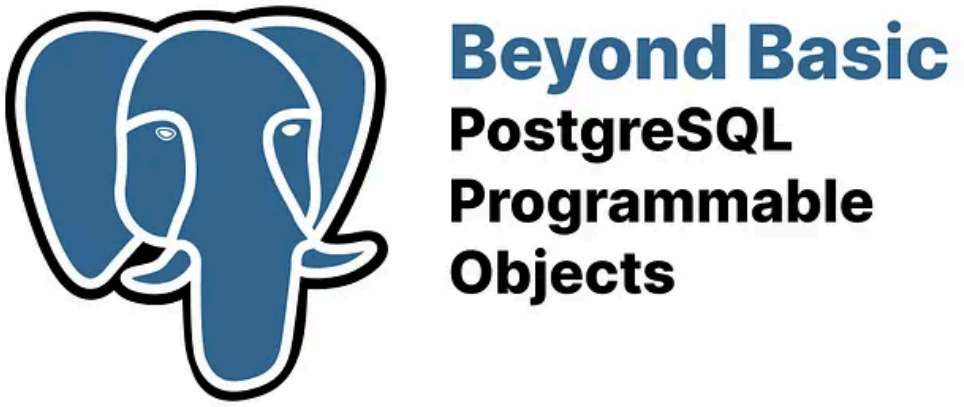


Thinking Loop

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

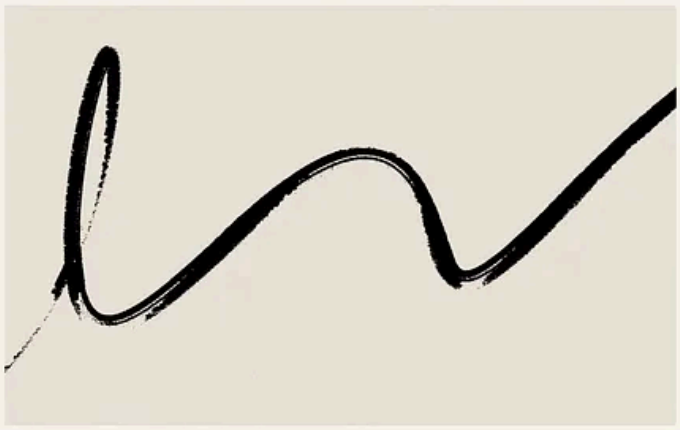


In Stackademic by bektiaw

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

Oz

Understanding PostgreSQL Page (Block) Structure

PostgreSQL stores data on disk in fixed-size units called pages or blocks. In this post, we'll explore how PostgreSQL organizes data...

May 14 60 1

See more recommendations