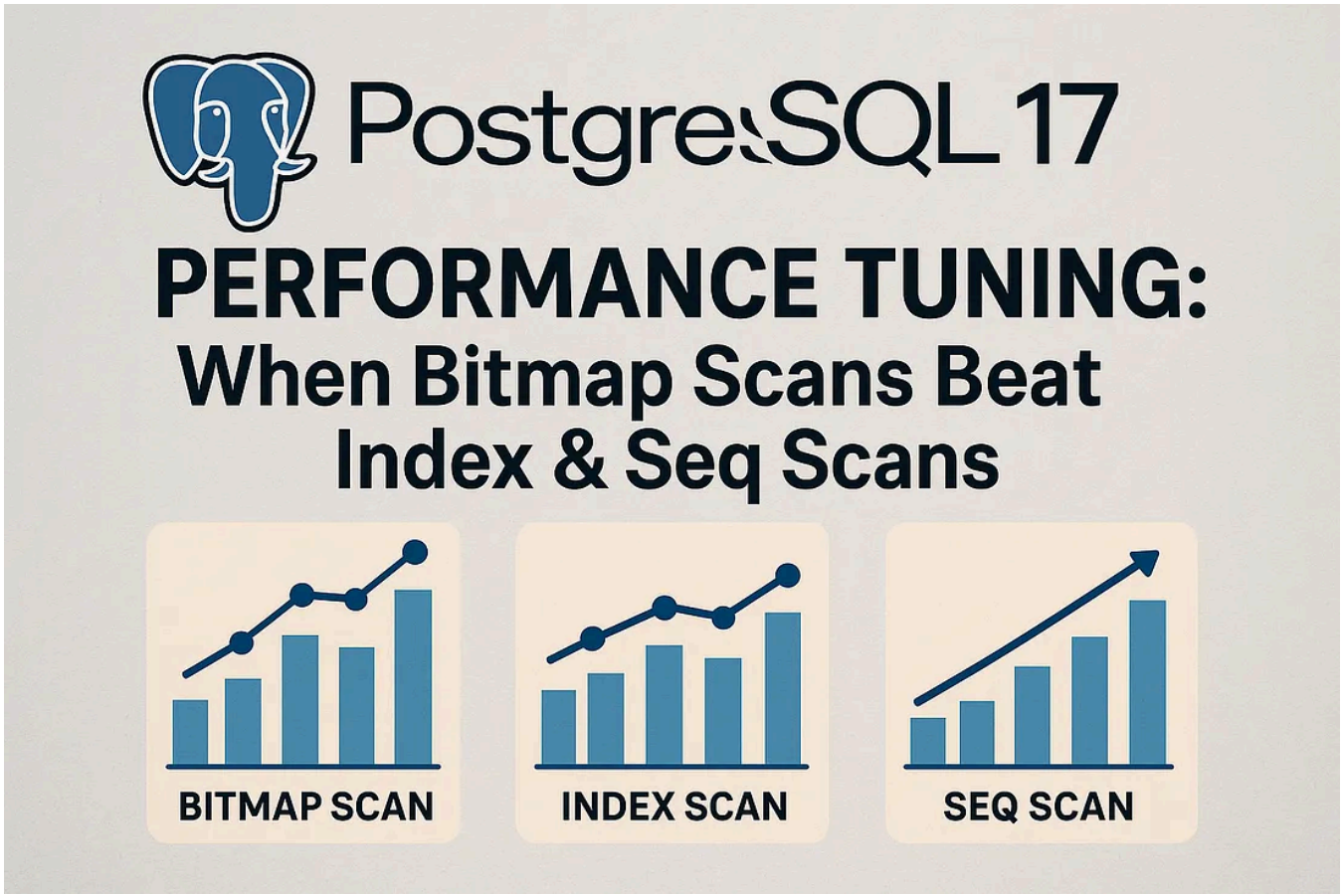
14 - PostgreSQL 17 Performance Tuning: When Bitmap Scans Beat Index & Seq Scans

7 min read · Sep 2, 2025

Jeyaram Ayyalusamy Following



PostgreSQL chooses among three broad strategies for filtering rows:

- **Index Scan** when the result set is tiny (needle in a haystack).
- **Sequential Scan** when most rows match (just read the whole table).

- **Bitmap (Index + Heap) Scan** for the *in-between*: too many rows for a fast Index Scan, too few to justify a full Seq Scan, or when **combining multiple indexes**.

Bitmap scans build an in-memory **bitmap of row locations (TIDs)** from one or more indexes, then visit each heap page **once**, pulling all needed rows per page. This dramatically reduces random I/O compared to a plain Index Scan over many matching rows.

Below is a reproducible demo you can paste into `psql`.

PostgreSQL 17 Performance Tuning: Why Bitmap Heap Scan Can Be Slower Than Seq Scan — and How `CLUSTER` Fixes It

When tuning queries in PostgreSQL, indexes don't always guarantee faster execution. In fact, sometimes a **Bitmap Heap Scan with an index** can be significantly slower than a simple **Sequential Scan**. In this section, we'll walk through a hands-on test with **10 million rows** and explain the query plans in detail.

Step 1: Creating the `products` table

```
CREATE TABLE products (
    product_id   BIGINT,
    product_name TEXT,
    category     TEXT,
    price        NUMERIC,
    stock_qty    INT
);
```

This table will hold product information such as name, category, price, and stock quantity.

Step 2: Insert 10M rows

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

- 10 evenly distributed categories (`Category_0 … Category_9`).
- Random prices between `0.00` and `500.00` .
- Stock quantities between `0` and `100` .

Step 3: Analyze the table

```
ANALYZE products;
```

This updates statistics for the planner.

Step 4: Query plan without index (Seq Scan)

```
EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
```

PostgreSQL 17 Performance Tuning: Understanding the Parallel Seq Scan

When analyzing query performance in PostgreSQL, looking at the execution plan is crucial. Let's break down the following plan in detail:

```
postgres=# EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
                                                          QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..146183.08 rows=97665 width=44) (actual time=3.077..3272
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on products  (cost=0.00..135416.58 rows=40694 width=44
         Filter: (stock_qty = 76)
         Rows Removed by Filter: 3299940
 Planning Time: 0.046 ms
 Execution Time: 3363.344 ms
(8 rows)
```

- **What it means:**
  The `Gather` node collects results from multiple parallel workers. PostgreSQL planned for 2 workers, and both were successfully launched.
- **Why it matters:**
  This is the coordination point where PostgreSQL merges all rows found by parallel workers and returns them as a single result set.

2. Parallel Seq Scan on `products`

- **What it means:**
  Each worker performed a **Sequential Scan** across a different portion of the `products` table. A sequential scan means PostgreSQL reads rows in physical order directly from the heap.
- **Rows processed:**

- Each worker scanned about **3.3 million rows** (`rows=33393` actually returned per worker).
- The filter `stock_qty = 76` discarded the vast majority of rows, as shown in `Rows Removed by Filter: 3,299,940` .
- **Why it matters:**
  Even though every row in the table was touched, parallel workers allowed the job to be divided, making it faster than if a single process had done all the scanning.

3. Costs vs. Actual Times

- **Cost estimates:**
- `cost=1000.00..146183.08` means the planner estimated this query would have a base startup cost of 1000 and a total cost of ~146k.
- **Actual execution:**
- The query started returning rows after ~3 ms.
- The entire scan finished in about **3.27 seconds** (`actual time=3.077..3272.108`).

👉 The actual time aligns reasonably with planner expectations, which means PostgreSQL chose an appropriate strategy.

4. Planning and Execution Time

- **Planning Time:** `0.046 ms` — PostgreSQL needed almost no time to decide on this plan.
- **Execution Time:** `3363.344 ms` — The real cost of scanning 10 million rows and filtering them.

Why the Sequential Scan Was Efficient

Even though scanning the whole table sounds expensive, it performed well here because:

1. **Parallelism:** Multiple workers shared the workload.
2. **Sequential I/O:** Reading pages in order is much cheaper than jumping around randomly.
3. **Data selectivity:** The condition matched about 1% of rows, but scanning was still competitive because sequential reads are highly optimized.

✅ **Key Takeaway:**
In PostgreSQL, a sequential scan (especially parallelized) can be faster than an index-based plan when the query touches a large percentage of the table or when rows are spread across many pages.

Step 5: Create index on `stock_qty`

```
CREATE INDEX idx_products_stock ON products(stock_qty);
```

Step 6: Query plan with index (Bitmap Heap Scan)

```
EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
```

Query Plan Output

```
postgres=# EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------
 Bitmap Heap Scan on products  (cost=1089.35..89823.31 rows=97667 width=44) (ac
   Recheck Cond: (stock_qty = 76)
   Heap Blocks: exact=58440
   ->  Bitmap Index Scan on idx_products_stock  (cost=0.00..1064.94 rows=97667
         Index Cond: (stock_qty = 76)
 Planning Time: 4.174 ms
 Execution Time: 42559.194 ms
(7 rows)

postgres=#
```

Explanation:

- **Bitmap Index Scan:** Quickly finds ~100k row pointers (TIDs) matching `stock_qty = 76`.
- **Bitmap Heap Scan:** Fetches those rows from the heap but must touch **58,440 pages** scattered across the table.
- **Execution Time (~42.6s):** Much slower than the Seq Scan because of **random I/O.**

👉 Lesson: An index doesn't help if matching rows are scattered everywhere.

Step 7: Cluster the Table

```
CLUSTER products USING idx_products_stock;
```

When you run the **CLUSTER** command in PostgreSQL, the database physically **rewrites the table** so that its rows are stored on disk in the same order as the specified index (in this case, `idx_products_stock`).

In our example, rows are reordered so that all entries with the same **`stock_qty`** value are stored **contiguously** on disk.

Why Does This Improve Performance?

1. **Data Locality**

- Normally, even if you have an index, the corresponding rows may be scattered across different disk pages.
- After clustering, rows with the same `stock_qty` are stored together.
- This means fewer page reads are needed when scanning rows that share the same value, reducing I/O.

2. **Faster Sequential Access**

- PostgreSQL can fetch grouped rows in **sequential blocks** instead of random disk access.
- Sequential reads are much faster than random reads, especially on large tables.

3. **Improved Index Efficiency**

- The index `idx_products_stock` now directly maps to data that is already in physical order.
- This reduces the number of heap fetches, since PostgreSQL doesn't have to jump around to find matching tuples.

4. **Better Cache Utilization**

- When rows are stored together, PostgreSQL can load an entire block of related rows into memory.
- This improves buffer cache hit ratio, making repeated queries faster.

5. **Query Patterns Benefit Most**

- Queries like:

```
SELECT * FROM products WHERE stock_qty = 76;
```

- or

```
SELECT * FROM products WHERE stock_qty BETWEEN 70 AND 80;
```

- run significantly faster because the database engine reads **fewer scattered pages.**

Key Considerations

- **One-time Reorganization:**
  CLUSTER is not automatic. If new rows are inserted or updated, the physical order can get out of sync again.
- **Use Periodically:**
  For frequently queried tables where data distribution changes over time, consider re-clustering during maintenance windows.
- **Table Locking:**
  CLUSTER requires an **exclusive lock**, meaning the table is not available for reads/writes during the operation.

✅ In summary, clustering improves performance by **reducing I/O, improving sequential access, and aligning physical storage with logical query patterns.** This leads to **faster lookups and more efficient scans** for queries that filter or group by the clustered column.

Step 8: Query plan after clustering

```
EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
```

Query Plan Output

```
postgres=# EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------
 Bitmap Heap Scan on products  (cost=1089.35..89823.31 rows=97667 width=44) (ac
   Recheck Cond: (stock_qty = 76)
   Heap Blocks: exact=835
   ->  Bitmap Index Scan on idx_products_stock  (cost=0.00..1064.94 rows=97667
         Index Cond: (stock_qty = 76)
 Planning Time: 2.360 ms
 Execution Time: 430.219 ms
(7 rows)

postgres=#
```

Explanation:

- The Bitmap Index Scan still identifies 100k rows.
- But this time, only **835 heap pages** need to be fetched (instead of 58k).
- **Execution Time (~0.43s):** Dramatic improvement because rows are now **physically contiguous.**

Step 9: Analyze after clustering

```
ANALYZE products;
```

Step 10: Query plan after Analyze (Index Scan)

```
EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
```

Query Plan Output

```
postgres=# EXPLAIN ANALYZE SELECT * FROM products WHERE stock_qty = 76;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------
 Index Scan using idx_products_stock on products  (cost=0.43..3170.28 rows=1083
   Index Cond: (stock_qty = 76)
 Planning Time: 0.148 ms
 Execution Time: 128.913 ms
(4 rows)

postgres=#
```

Explanation:

- PostgreSQL switches to a **direct Index Scan.**
- Since rows are tightly packed and statistics are updated, the planner knows the index scan is optimal.
- **Execution Time (~0.13s):** Fastest plan so far.

Why Bitmap Heap Scan Was Slower Than Seq Scan

- **Without clustering:** The Bitmap Heap Scan had to jump to **58,440 scattered pages** → costly random I/O.
- **Seq Scan** read the table linearly with parallel workers → faster despite reading more data.

Why `CLUSTER` Improved Execution Time

- **Physically orders data** according to the index.
- Rows with `stock_qty = 76` became adjacent → only **835 pages** to fetch.
- Execution time dropped from **42.6s → 0.43s → 0.13s** after Analyze.

Final Takeaways

- An index doesn't always guarantee better performance.
- Bitmap Heap Scans can be very slow if data is **scattered across the heap**.
- Sequential scans may outperform them because of efficient linear I/O and parallelism.
- `CLUSTER` dramatically improves performance by grouping related rows together.
- After clustering and analyzing, PostgreSQL can switch to the fastest **Index Scan**.

👉 **Lesson for tuning:** Always look at the query plan, not just the presence of indexes. Data organization matters just as much as indexes.

📢 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

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

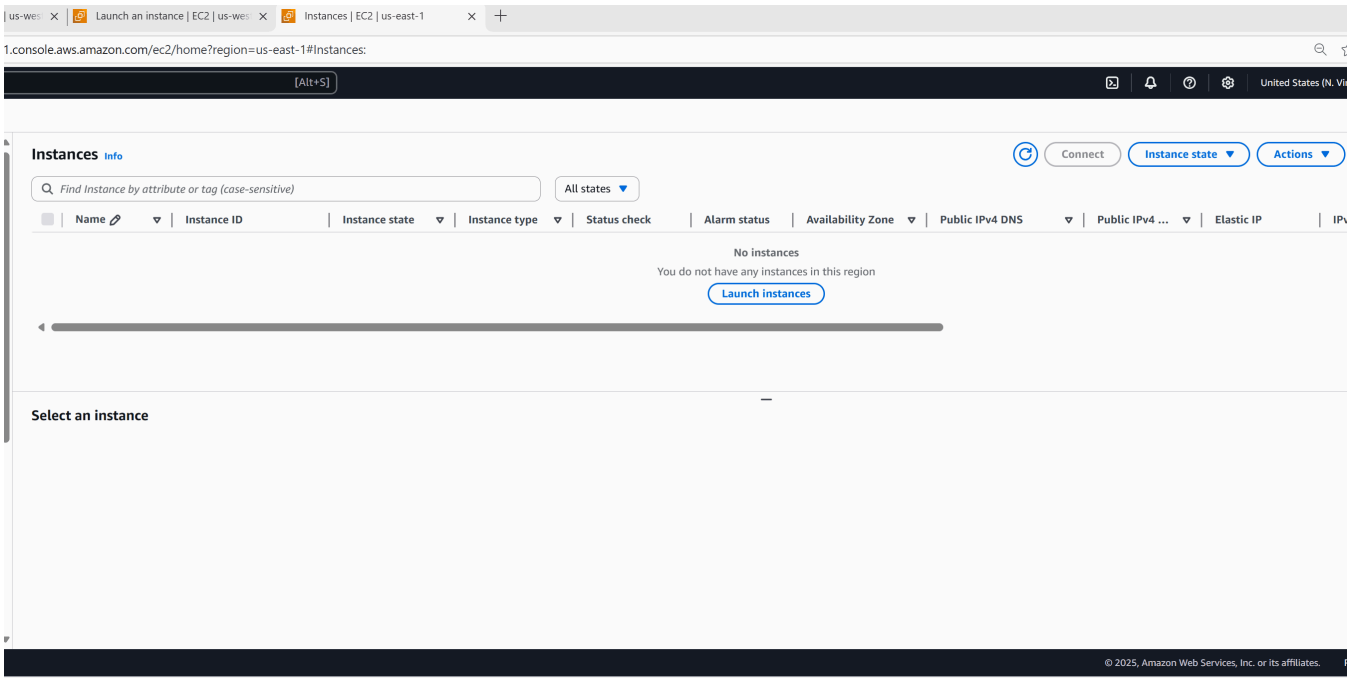
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

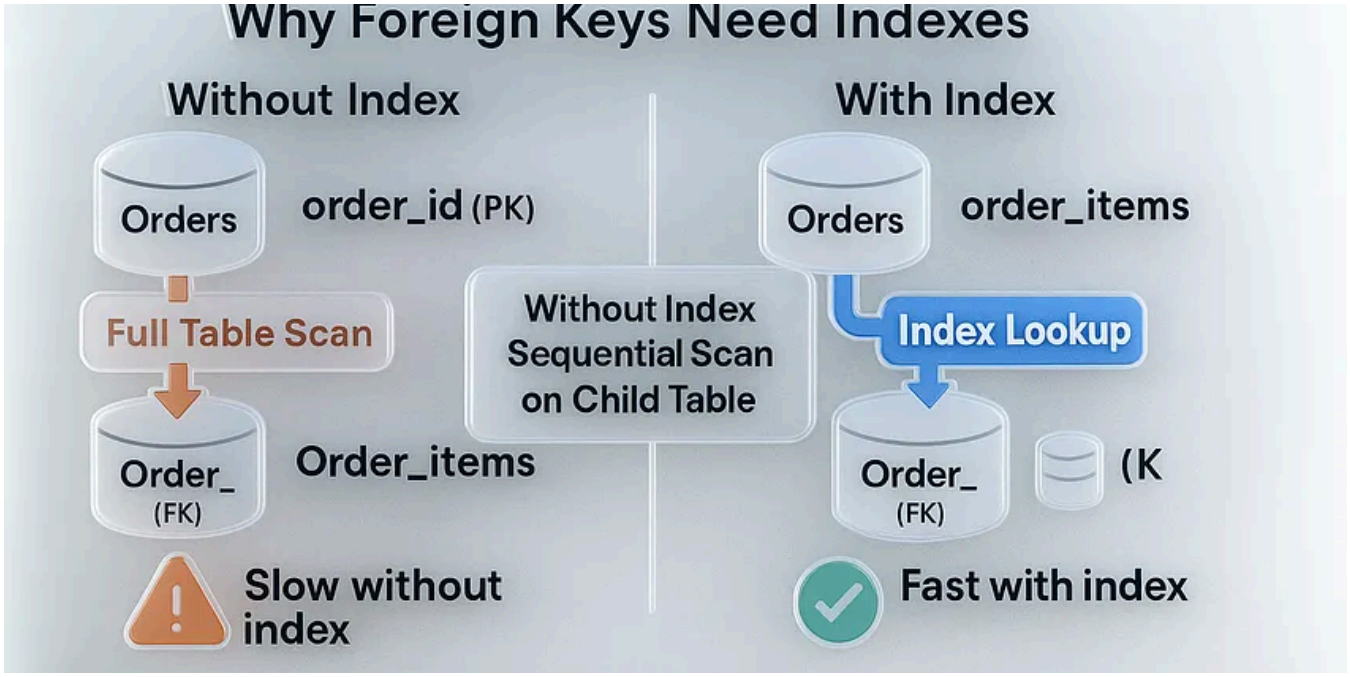


Jeyaram Ayyalusamy

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

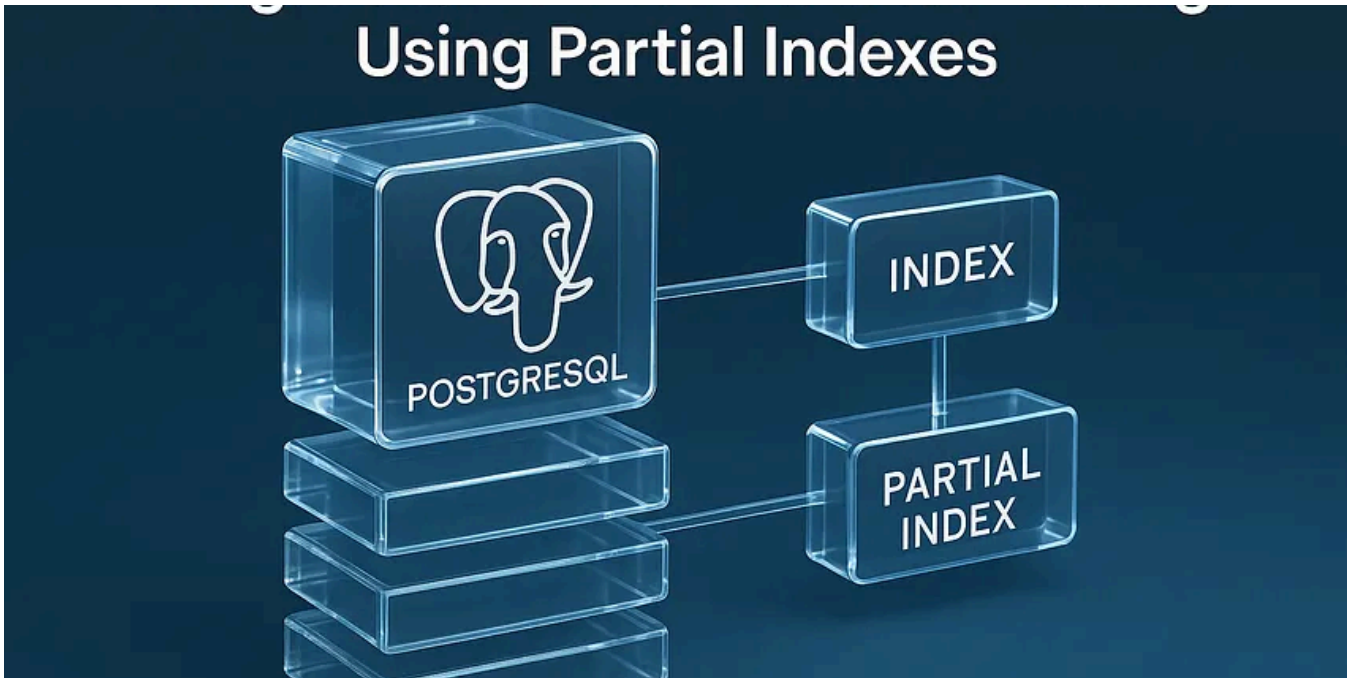


Jeyaram Ayyalusamy

16 -PostgreSQL 17 Performance Tuning: Why Foreign Keys Need Indexes

When designing relational databases, parent-child relationships are common. A classic case is the Orders table (parent) and the Items table...

Sep 3 3 2

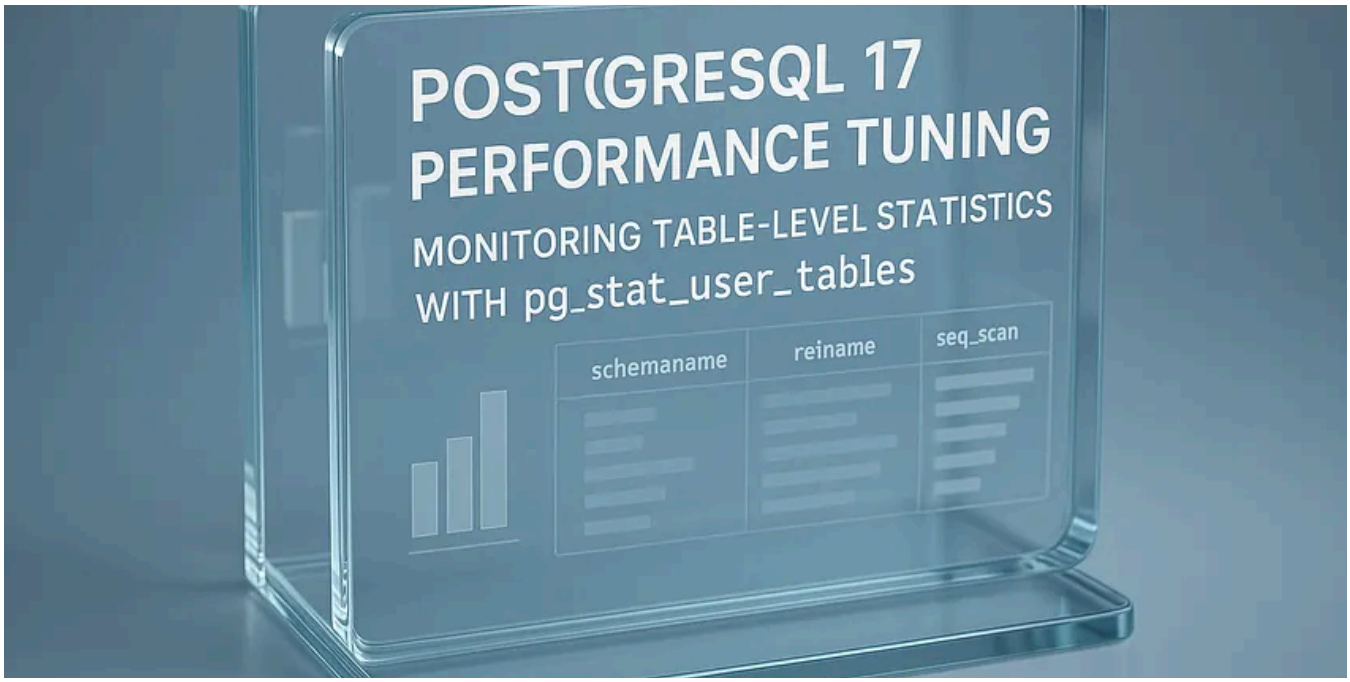


Jeyaram Ayyalusamy

17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes

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...

Sep 4 3



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

See all from Jeyaram Ayyalusamy

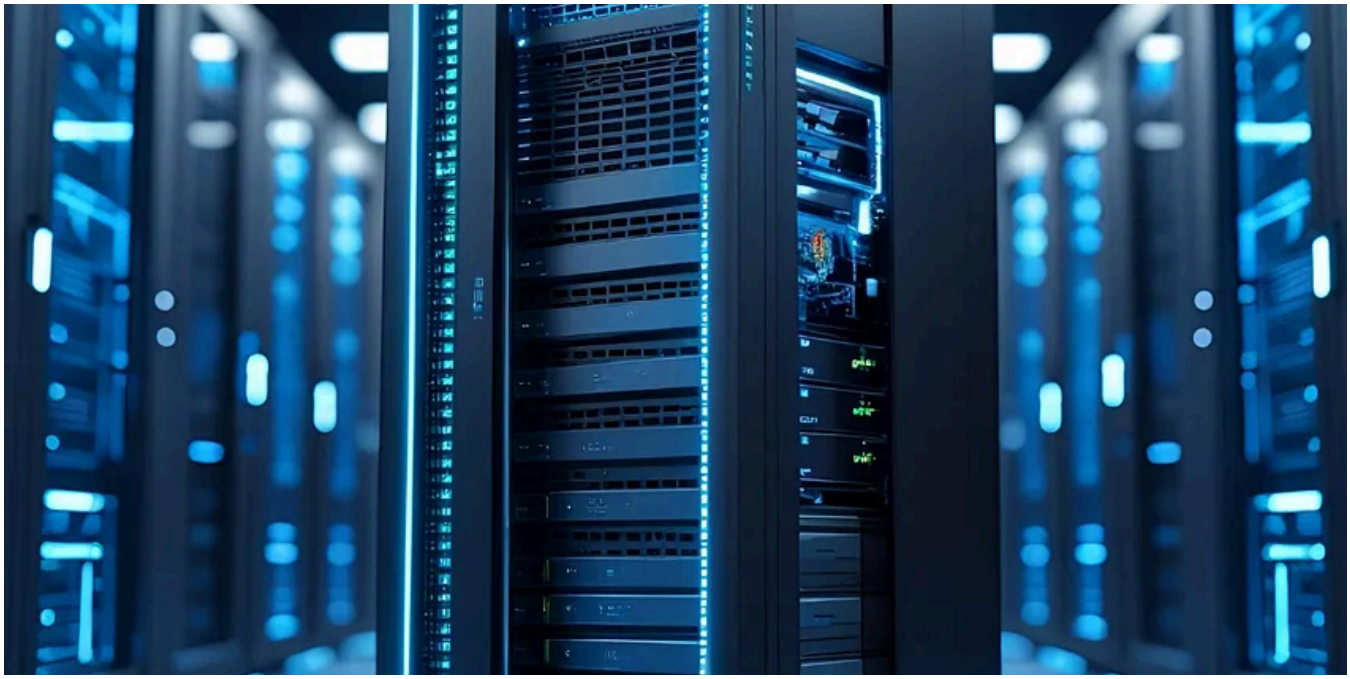
Recommended from Medium

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

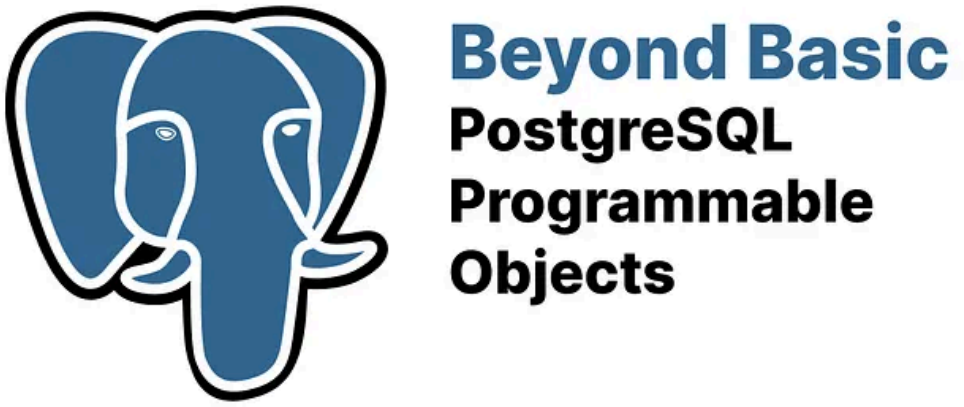


Rizqi Mulki

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

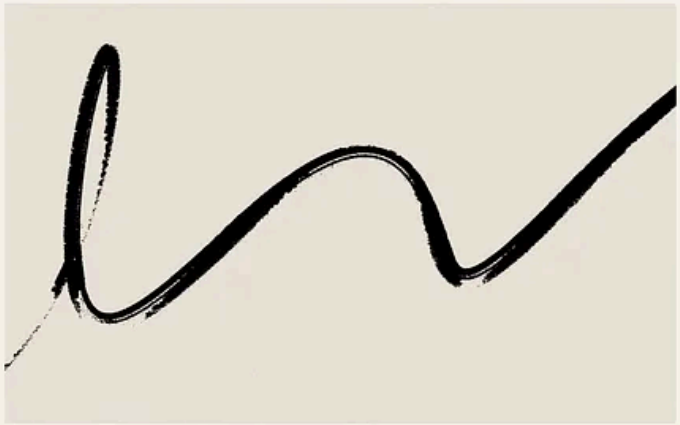


In Stackademic by bektiaw

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

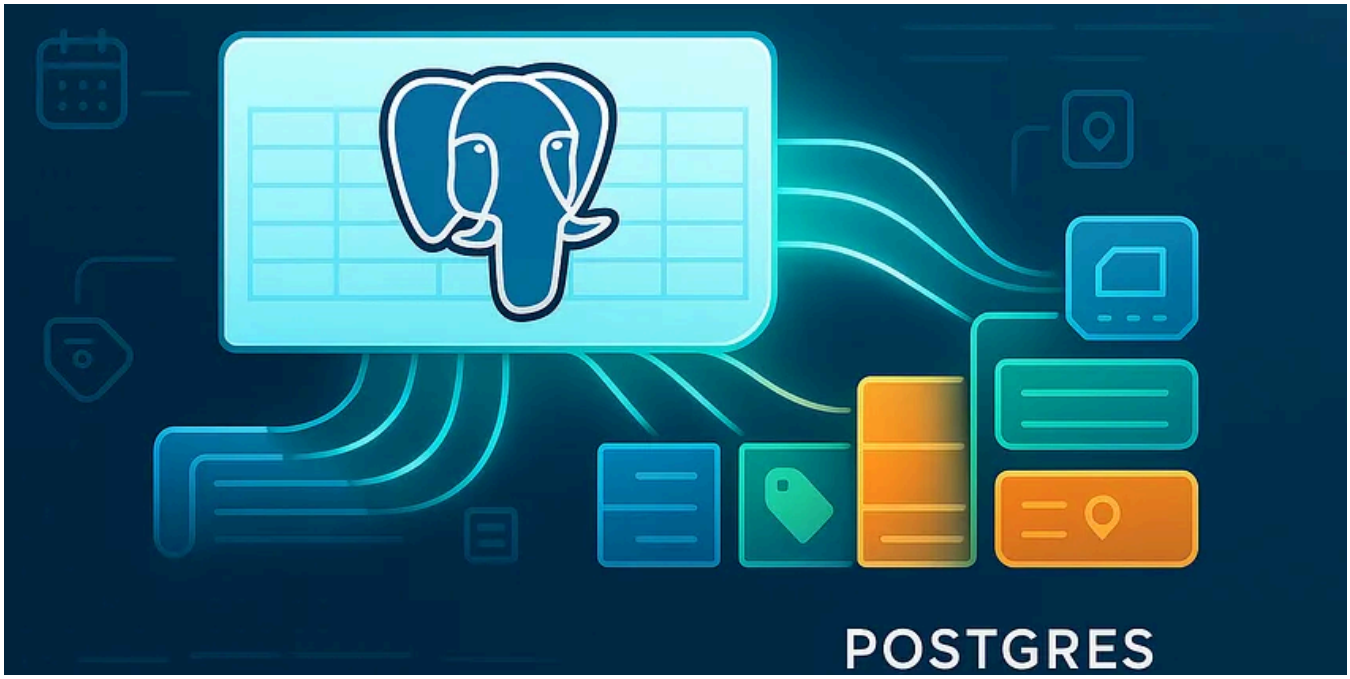
Jul 18 12 1

Thread Whisperer

Postgres 18 Arrives: Async I/O You Should Turn On First

Turn disk waits into throughput with a few safe switches

Sep 15 77




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