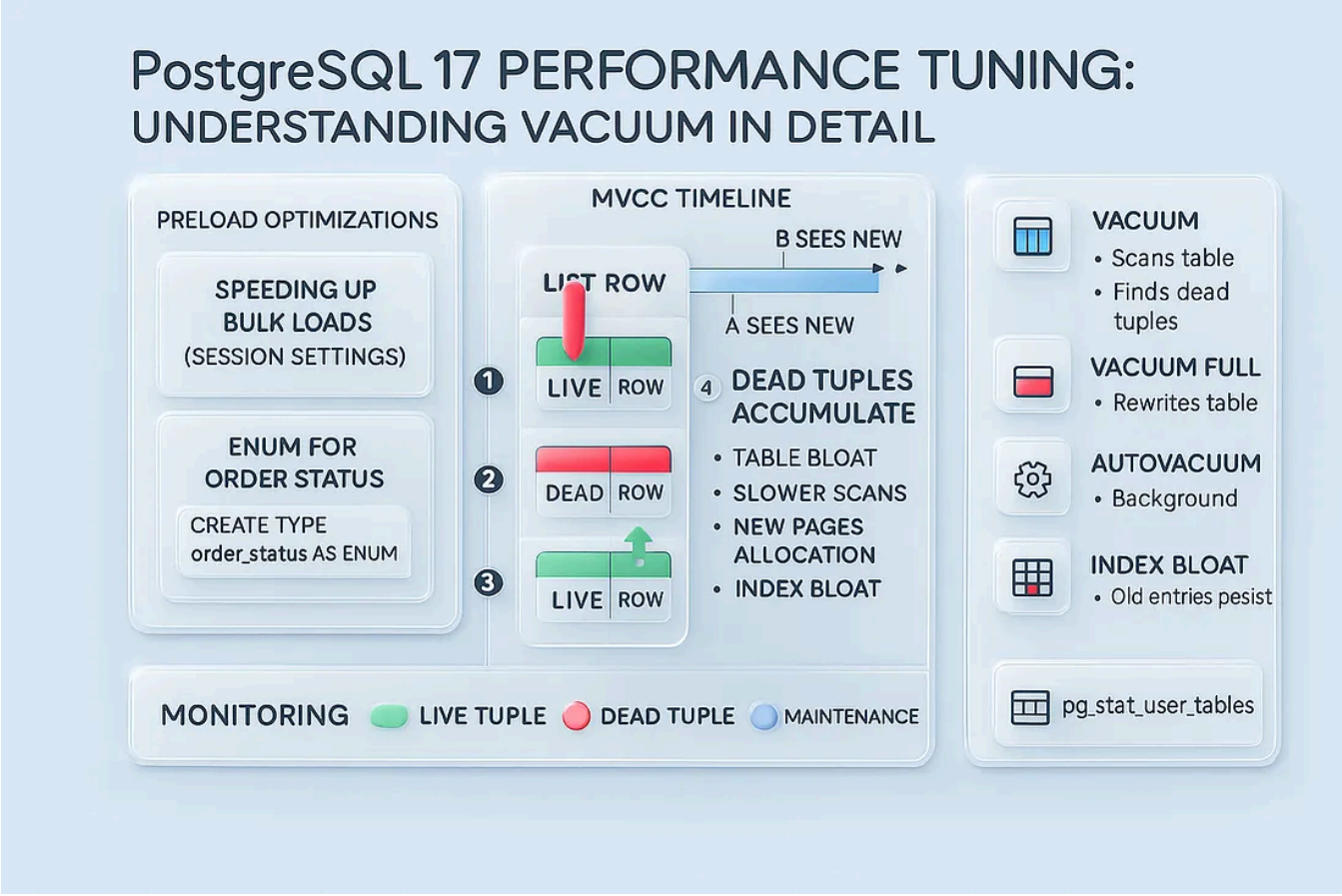
07 - PostgreSQL 17 Performance Tuning: Understanding VACUUM in Detail

10 min read · Sep 1, 2025

Jeyaram Ayyalusamy Following



When it comes to PostgreSQL performance tuning, one of the most important maintenance tasks is **VACUUM**. Without it, tables grow unnecessarily large, queries slow down, and indexes become inefficient. Many people hear about vacuuming but

don't fully understand why it is needed or how it works. Let's explore this concept step by step, in an easy-to-understand format, with examples.

Bulk Loading 3 Million Orders in PostgreSQL

When working with PostgreSQL, one of the most common tasks for testing and benchmarking is generating large datasets quickly. Let's walk through how to create an **orders table with 3 million rows**, complete with a `status` column and realistic values.

1. Speeding Up Bulk Loads (Session Settings)

Before inserting millions of rows, we can temporarily adjust some PostgreSQL settings for faster performance:

```
SET synchronous_commit = OFF;
SET maintenance_work_mem = '1GB';
SET temp_buffers = '256MB';
```

Output:

```
postgres=# SET synchronous_commit = OFF;
SET maintenance_work_mem = '1GB';
SET temp_buffers = '256MB';
SET
SET
SET
postgres=#
```

- **synchronous_commit = OFF** → avoids waiting for WAL flush after each commit.
- **maintenance_work_mem** → gives more memory for maintenance tasks like index creation.

- **temp_buffers** → increases the memory used for temporary tables.

⚠️ These are **session-only changes.** Once done, reset them back to safe defaults.

2. Creating an Enum for Order Status

Instead of using free-text, we define an enum to keep `status` values clean and consistent:

```
DO $$
BEGIN
  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'order_status') THEN
    CREATE TYPE order_status AS ENUM ('PENDING','PROCESSING','SHIPPED','DELIVER
  END IF;
END$$;
```

Output

```
postgres=# DO $$
BEGIN
  IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'order_status') THEN
    CREATE TYPE order_status AS ENUM ('PENDING','PROCESSING','SHIPPED','DELIVER
  END IF;
END$$;
DO
postgres=#
```

3. Creating the Orders Table

We'll create the table as **UNLOGGED** first. This skips WAL logging for faster inserts. After the bulk load, we'll switch it back to `LOGGED` for durability.

```
DROP TABLE IF EXISTS orders CASCADE;
```

```
postgres=#
postgres=# DROP TABLE IF EXISTS orders CASCADE;
NOTICE:  table "orders" does not exist, skipping
DROP TABLE
postgres=#
```

```
CREATE UNLOGGED TABLE orders (
  order_id      BIGSERIAL PRIMARY KEY,
  customer_id   BIGINT NOT NULL,
  status        order_status NOT NULL,
  amount_usd    NUMERIC(10,2) NOT NULL,
  item_count    INT NOT NULL,
  created_at    TIMESTAMPTZ NOT NULL,
  updated_at    TIMESTAMPTZ NOT NULL
);
```

```
postgres=# CREATE UNLOGGED TABLE orders (
  order_id      BIGSERIAL PRIMARY KEY,
  customer_id   BIGINT NOT NULL,
  status        order_status NOT NULL,
  amount_usd    NUMERIC(10,2) NOT NULL,
  item_count    INT NOT NULL,
  created_at    TIMESTAMPTZ NOT NULL,
  updated_at    TIMESTAMPTZ NOT NULL
);
CREATE TABLE
postgres=#
```

4. Disable Autovacuum (Optional)

When bulk-loading millions of rows, PostgreSQL's **autovacuum** process might kick in and slow things down. To avoid unnecessary background cleanup during the initial load, you can temporarily disable autovacuum on the table:

```
ALTER TABLE orders SET (autovacuum_enabled = off);
```

```
postgres=# ALTER TABLE orders SET (autovacuum_enabled = off);
ALTER TABLE
postgres=#
```

This ensures that PostgreSQL doesn't try to vacuum the table while you're still inserting massive amounts of data.

⚠️ **Important:** Don't leave autovacuum off permanently. After your bulk load and indexing are done, it's good practice to re-enable it:

```
ALTER TABLE orders SET (autovacuum_enabled = on);
```

5. Bulk-Loading 3 Million Rows

We generate rows using `generate_series` and `random()`. Each row simulates:

- A **customer_id** (random between 1 and ~1,000,000).
- A **status** distributed realistically:

- PENDING ≈ 12%
- PROCESSING ≈ 18%
- SHIPPED ≈ 30%
- DELIVERED ≈ 35%
- CANCELLED ≈ 5%
- An **amount_usd** between $5 and $500.
- An **item_count** between 1 and 5.
- **created_at** spread over the past two years.
- **updated_at** within two weeks of `created_at`.

```
WITH src AS (
  SELECT
    (1 + (random()*999999)::BIGINT)              AS customer_id,
    random()                                     AS r,   -- used for status dis
    ROUND((5 + random()*495)::NUMERIC, 2)        AS amount_usd,
    GREATEST(1, (random()*5)::INT)               AS item_count,
    now() - ((random()*730)::INT || ' days')::INTERVAL  AS created_at_rand,
    random()                                     AS ro   -- used for updated_at
  FROM generate_series(1, 3000000) AS gs
)
INSERT INTO orders (customer_id, status, amount_usd, item_count, created_at, up
SELECT
  customer_id,
  CASE
    WHEN r < 0.12 THEN 'PENDING'::order_status
    WHEN r < 0.30 THEN 'PROCESSING'::order_status
    WHEN r < 0.60 THEN 'SHIPPED'::order_status
    WHEN r < 0.95 THEN 'DELIVERED'::order_status
    ELSE 'CANCELLED'::order_status
  END                                            AS status,
  amount_usd,
  item_count,
  created_at_rand,
  created_at_rand + ((ro*14)::INT || ' days')::INTERVAL
FROM src;
```

```
postgres=# WITH src AS (
  SELECT
    (1 + (random()*999999)::BIGINT)              AS customer_id,
    random()                                     AS r,   -- used for status dis
    ROUND((5 + random()*495)::NUMERIC, 2)        AS amount_usd,
    GREATEST(1, (random()*5)::INT)               AS item_count,
    now() - ((random()*730)::INT || ' days')::INTERVAL  AS created_at_rand,
    random()                                     AS ro   -- used for updated_at
  FROM generate_series(1, 3000000) AS gs
)
INSERT INTO orders (customer_id, status, amount_usd, item_count, created_at, up
SELECT
  customer_id,
  CASE
    WHEN r < 0.12 THEN 'PENDING'::order_status
    WHEN r < 0.30 THEN 'PROCESSING'::order_status
    WHEN r < 0.60 THEN 'SHIPPED'::order_status
    WHEN r < 0.95 THEN 'DELIVERED'::order_status
    ELSE 'CANCELLED'::order_status
  END                                            AS status,
  amount_usd,
  item_count,
  created_at_rand,
  created_at_rand + ((ro*14)::INT || ' days')::INTERVAL
FROM src;

INSERT 0 3000000
postgres=#
```

```
SELECT pg_size_pretty(pg_relation_size('orders'));
```

```
postgres=# SELECT pg_size_pretty(pg_relation_size('orders'));
 pg_size_pretty
----------------
 219 MB
```

```
(1 row)

postgres=#
```

6. Switch Table Back to Logged

After the bulk load, switch the table back to normal logging:

```
ALTER TABLE orders SET LOGGED;
```

```
postgres=# ALTER TABLE orders SET LOGGED;
ALTER TABLE
postgres=#
```

7. Create Indexes After the Load

Always create indexes after bulk inserts — it's much faster.

```
CREATE INDEX orders_status_idx        ON orders (status);
CREATE INDEX orders_created_at_idx    ON orders (created_at);
CREATE INDEX orders_customer_id_idx   ON orders (customer_id);
```

```
postgres=# CREATE INDEX orders_status_idx        ON orders (status);
CREATE INDEX
```

```
postgres=# CREATE INDEX orders_created_at_idx    ON orders (created_at);
CREATE INDEX
postgres=# CREATE INDEX orders_customer_id_idx   ON orders (customer_id);
CREATE INDEX
postgres=#
```

8. Analyze for the Query Planner

Run `ANALYZE` so PostgreSQL collects fresh statistics:

```
ANALYZE orders;
```

```
postgres=# ANALYZE orders;
ANALYZE
postgres=#
```

9. Reset Session Settings

Put things back to normal:

```
RESET synchronous_commit;
RESET maintenance_work_mem;
RESET temp_buffers;
```

```
postgres=# ANALYZE orders;
ANALYZE
postgres=# RESET synchronous_commit;
RESET
```

```
postgres=# RESET maintenance_work_mem;
RESET
postgres=# RESET temp_buffers;
RESET
postgres=#
```

Quick Sanity Checks

```
SELECT COUNT(*) AS total_rows FROM orders;
SELECT status, COUNT(*) AS c FROM orders GROUP BY status ORDER BY c DESC;
```

```
postgres=# SELECT COUNT(*) AS total_rows FROM orders;
 total_rows
------------
    3000000
(1 row)

postgres=# SELECT status, COUNT(*) AS c FROM orders GROUP BY status ORDER BY c
   status   |    c
------------+---------
 DELIVERED  | 1051338
 SHIPPED    |  899705
 PROCESSING |  540512
 PENDING    |  358959
 CANCELLED  |  149486
(5 rows)

postgres=#
```

✅ And that's it! You now have a fully populated `orders` table with **3 million realistic records**, perfect for performance tuning, query optimization tests, or demo dashboards.

Step 1: What Happens When You Delete a Record?

In PostgreSQL, deleting a row does **not** immediately remove it from disk.

- Instead, the row is simply **marked as deleted.**
- It is no longer visible to queries, but the storage space is still occupied.
- Such rows are called **dead tuples.**

👉 **Example:**

```
DELETE FROM orders WHERE status = 'DELIVERED';
```

```
postgres=# DELETE FROM orders WHERE status = 'DELIVERED';
DELETE 1050792
postgres=#
```

```
SELECT pg_size_pretty(pg_relation_size('orders'));
```

```
postgres=# SELECT pg_size_pretty(pg_relation_size('orders'));
 pg_size_pretty
----------------
 219 MB
(1 row)

postgres=#
```

- Suppose the `orders` table had 3,000,000 rows.

- After deleting 1,050,792, PostgreSQL still stores space for 3,000,000 rows.
- 1,949,208 rows are live; 1,050,792 are dead tuples.

📌 **Analogy:** Imagine writing in a notebook and crossing out a paragraph. The text is still there on the page, but you've decided to ignore it.

Step 2: Updates Work Like Delete + Insert

In PostgreSQL, **UPDATE = DELETE + INSERT** under the hood.

1. The old row version is marked as deleted.
2. A new row version is inserted.

👉 **Example:**

```
UPDATE orders
SET customer_id = '7967210'
WHERE order_id = 3000010;
```

```
postgres=# UPDATE orders
SET customer_id = '7967210'
WHERE order_id = 3000010;
UPDATE 1
postgres=#
```

- If you update the same customer 5 times, PostgreSQL keeps 5 versions of that row.
- Only the newest version is visible; the other 4 are dead tuples.

📌 **Analogy:** Think of sticky notes. Every time you change something, you paste a new sticky note on top. The old ones are still stuck under there, piling up.

Step 3: Why PostgreSQL Keeps Old Versions

This happens because PostgreSQL uses **MVCC (Multi-Version Concurrency Control)**.

- Multiple transactions may run at the same time.
- One transaction might need to see the old version while another uses the new version.

👉 Example scenario:

- Transaction A starts and queries a row.
- Transaction B updates that row.
- Transaction A still sees the old row, while Transaction B and later transactions see the updated row.

📌 **Analogy:** Imagine a shared Google Doc. You opened it at 2:00 PM. Even if your friend makes edits at 2:05 PM, you still see the older version you opened.

Step 4: Dead Tuples — The Hidden Problem

Over time, dead tuples accumulate. This causes:

- **Table bloat:** tables get bigger on disk.
- **Slower queries:** scans take longer.
- **Inefficient storage:** PostgreSQL allocates new pages even though space is available.
- **Index bloat:** indexes also keep references to deleted rows.

👉 **Check dead tuples:**

```
SELECT relname AS table_name,
       n_live_tup AS live_rows,
       n_dead_tup AS dead_rows
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
```

```
postgres=# SELECT relname AS table_name,
       n_live_tup AS live_rows,
       n_dead_tup AS dead_rows
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;
   table_name    | live_rows | dead_rows
-----------------+-----------+-----------
 orders          |   1949208 |   1050793
 project_tasks_1 |   1000000 |         0
 customers       |         5 |         0
 audit_log       |         3 |         0
 project_tasks_2 |   1000000 |         0
 employee        |         1 |         0
(6 rows)

postgres=#
```

📌 **Analogy:** Imagine a warehouse filled with empty boxes. New shipments arrive, but instead of reusing space, the warehouse keeps expanding — wasting resources.

Monitor Dead Tuples in Your Tables

After loading millions of rows, it's a good idea to **check how many dead tuples** (deleted/expired row versions) exist in your tables. PostgreSQL tracks this information in the `pg_stat_user_tables` view.

You can run the following query:

```
SELECT relname, n_dead_tup, n_live_tup,
       round(100.0*n_dead_tup/GREATEST(n_live_tup,1),2) AS dead_pct
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;
```

```
postgres=# SELECT relname, n_dead_tup, n_live_tup,
       round(100.0*n_dead_tup/GREATEST(n_live_tup,1),2) AS dead_pct
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;
     relname     | n_dead_tup | n_live_tup | dead_pct
-----------------+------------+------------+----------
 orders          |    1050793 |    1949208 |    53.91
 project_tasks_1 |          0 |    1000000 |     0.00
 customers       |          0 |          5 |     0.00
 audit_log       |          0 |          3 |     0.00
 project_tasks_2 |          0 |    1000000 |     0.00
 employee        |          0 |          1 |     0.00
(6 rows)

postgres=#
```

This will:

- Show the **table name** (**`relname`**).
- Display the number of **dead tuples** and **live tuples**.
- Calculate the percentage of dead rows (`dead_pct`).
- Order the results so the tables with the most dead tuples appear first.

This query helps you quickly identify which tables are suffering the most from bloat and may need **VACUUM** or **VACUUM FULL**.

Step 5: How VACUUM Fixes It

The **VACUUM** command solves this.

- It scans the table.
- Finds dead tuples.
- Marks their space as **available for reuse**.

👉 **Example:**

```
VACUUM orders;
```

```
postgres=# VACUUM orders;
VACUUM
postgres=#
```

```
postgres=# SELECT pg_size_pretty(pg_relation_size('orders'));
 pg_size_pretty
----------------
 219 MB
(1 row)

postgres=#
```

- The 1,050,792 deleted rows in `orders` don't vanish physically.
- But PostgreSQL now knows it can reuse that space for future inserts.

✅ Key Points:

- Normal `VACUUM` does not shrink the physical size of the table.
- It runs without locking the table — reads and writes can continue.

📌 **Analogy:** It's like clearing shelves in a warehouse so new products can be stored in the same space.

Step 6: VACUUM FULL — The Heavy Cleaner

Sometimes you don't just want reusable space — you want to **shrink the table file** on disk. That's when you use **VACUUM FULL.**

- It rewrites the entire table.
- Physically removes dead tuples.
- Compacts the rows, reducing file size.

👉 **Example:**

```
VACUUM FULL orders;
```

```
postgres=# VACUUM FULL orders;
VACUUM
postgres=#
```

```
SELECT pg_size_pretty(pg_relation_size('orders'));
```

```
postgres=# SELECT pg_size_pretty(pg_relation_size('orders'));
 pg_size_pretty
----------------
 142 MB
(1 row)
```

```
postgres=#
```

- If your `orders` table grew to 219 MB but only had 142 MB of real data, `VACUUM FULL` could shrink it back to 142 MB.

📌 **Analogy:** Regular vacuuming is like sweeping the floor. `VACUUM FULL` is like moving the furniture, scrubbing, and reorganizing the room completely.

Step 7: Drawbacks of VACUUM FULL

While powerful, `VACUUM FULL` has two major downsides:

1. **Exclusive Lock**

- The table is locked for the entire duration.
- No queries can run during this time.

2. **Resource Intensive**

- On large tables, it can take hours.
- Uses a lot of CPU and disk I/O.

👉 **Bad scenario:** Running `VACUUM FULL` on a 100-million-row production table during peak hours. Users will see downtime.

Step 8: Autovacuum — PostgreSQL's Automatic Cleaner

Manually vacuuming isn't practical. PostgreSQL has **autovacuum**, a background process that:

- Automatically vacuums tables when dead tuples exceed a threshold.
- Runs continuously to keep tables healthy.

👉 **Configuration Example (`postgresql.conf`):**

```
autovacuum = on
autovacuum_naptime = 60
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
```

📌 In PostgreSQL 17, autovacuum improvements include:

- Better prioritization of busy tables.
- Smarter scheduling.
- More detailed logging.

📌 **Analogy:** Autovacuum is like a robotic vacuum cleaner that quietly runs in the background, keeping your house from getting too messy.

Step 9: Index Bloat — Another Challenge

Even after vacuuming, indexes can remain bloated.

- Updates and deletes leave behind old index entries.
- This slows down searches and joins.

👉 **Solution — REINDEX:**

```
REINDEX TABLE orders;
```

📌 **Analogy:** If vacuuming is cleaning shelves, reindexing is like reorganizing the product catalog to make searches faster and more efficient.

Step 10: Best Practices in PostgreSQL 17

1. Keep **autovacuum enabled** and tuned for your workload.
2. Run **manual VACUUM** on high-update tables when needed.
3. Avoid frequent `VACUUM FULL` — only use it for extreme bloat.
4. Regularly **monitor dead tuples** with `pg_stat_user_tables`.
5. Handle **index bloat separately** with `REINDEX`.

👉 **Quick check for dead tuples:**

```
SELECT relname, n_dead_tup, n_live_tup
FROM pg_stat_user_tables
WHERE n_dead_tup > 100000;
```

Final Thoughts

- **Deletes and updates** don't immediately free space — they create dead tuples.
- **VACUUM** reclaims space for reuse, keeps performance stable, and is safe to run while queries execute.
- **VACUUM FULL** shrinks tables but locks them and is expensive.
- **Autovacuum** is your ongoing housekeeper.
- **Indexes** need separate care with `REINDEX`.
- In **PostgreSQL 17**, vacuuming is smarter, faster, and easier to monitor, but still requires attention from DBAs.

✅ Think of vacuuming as **housekeeping for your database**. Ignore it, and performance suffers. Manage it well, and your PostgreSQL 17 instance will run efficiently for years to come.

📢 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 MySQL Open Source Oracle AWS

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

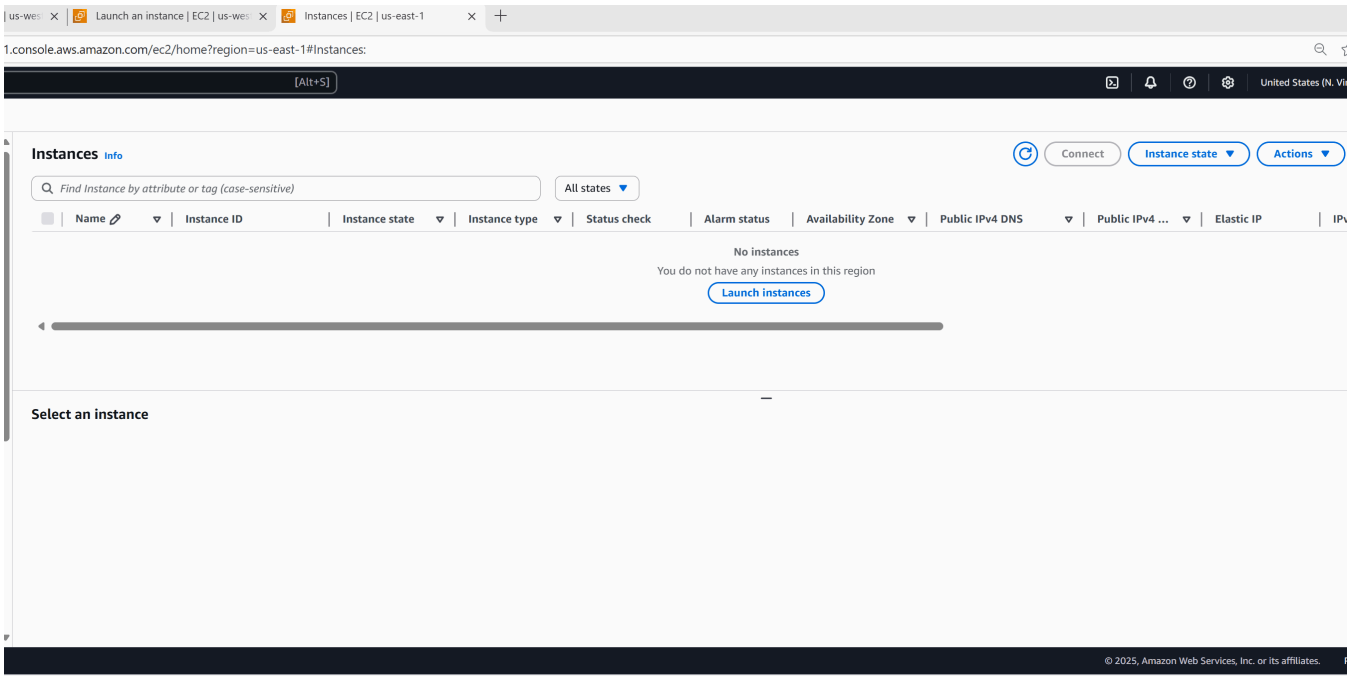
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

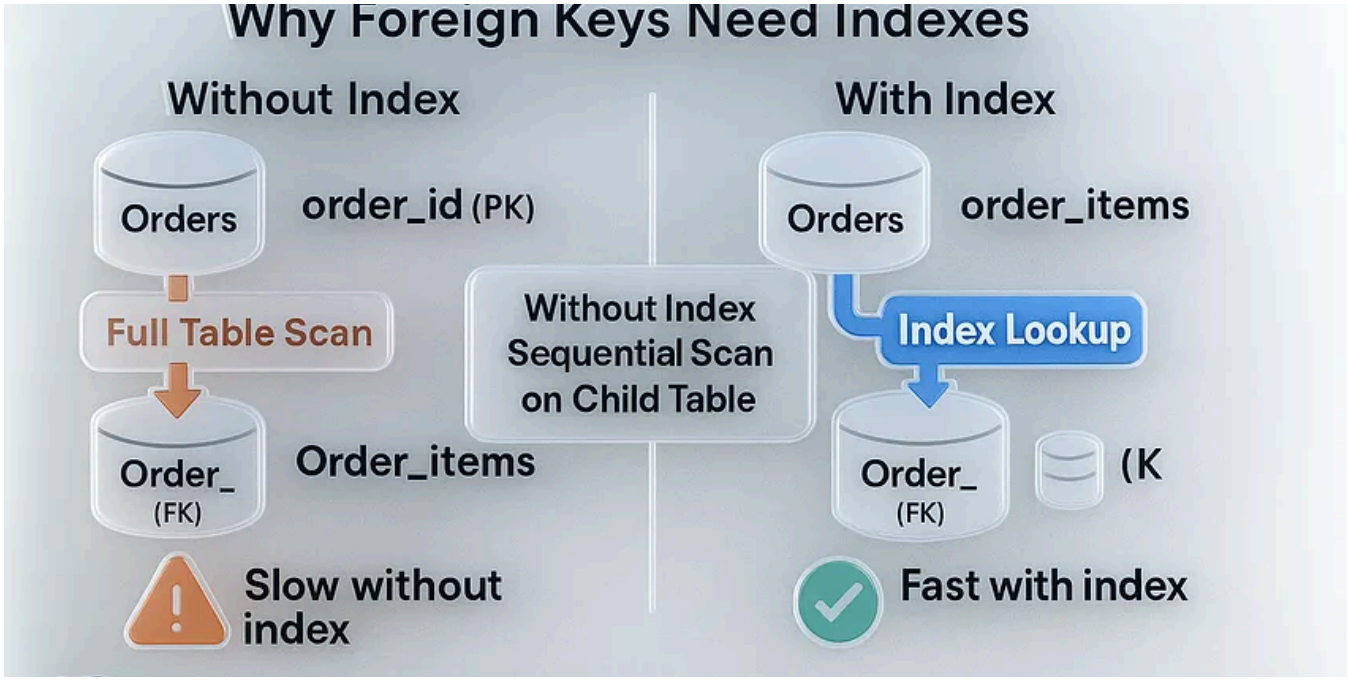


Jeyaram Ayyalusamy

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

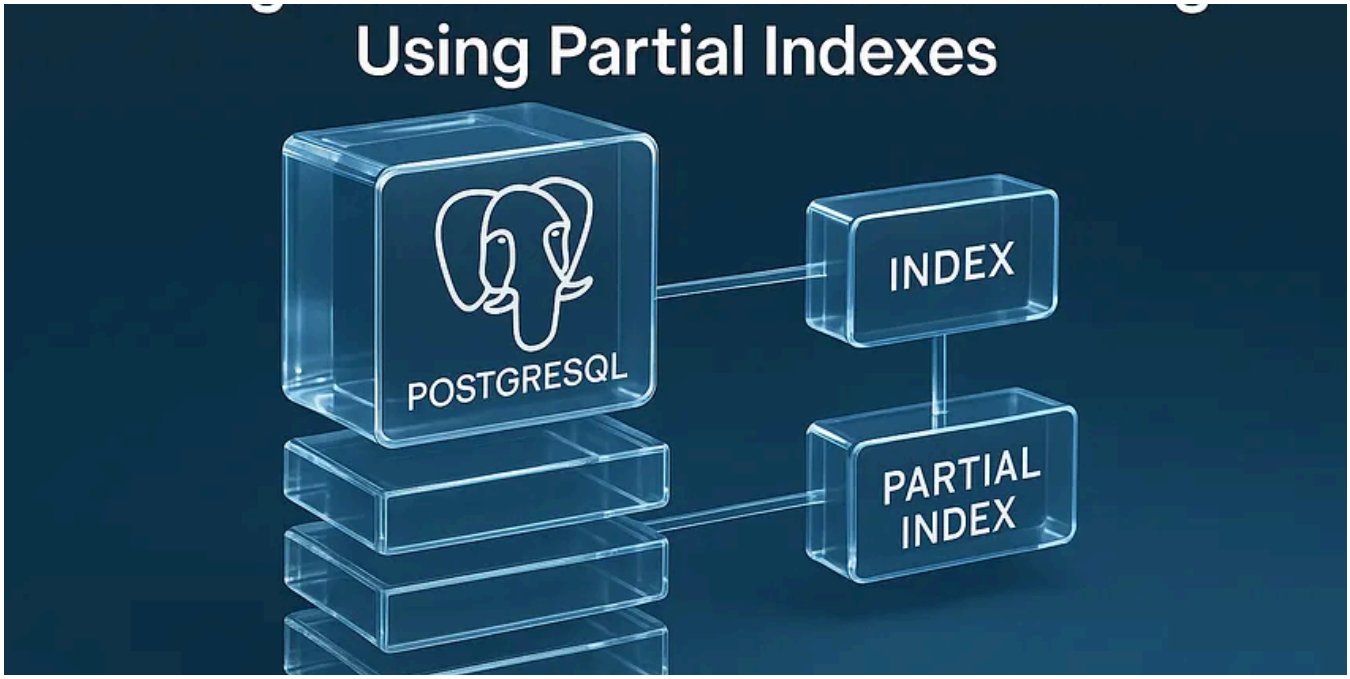


Jeyaram Ayyalusamy

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

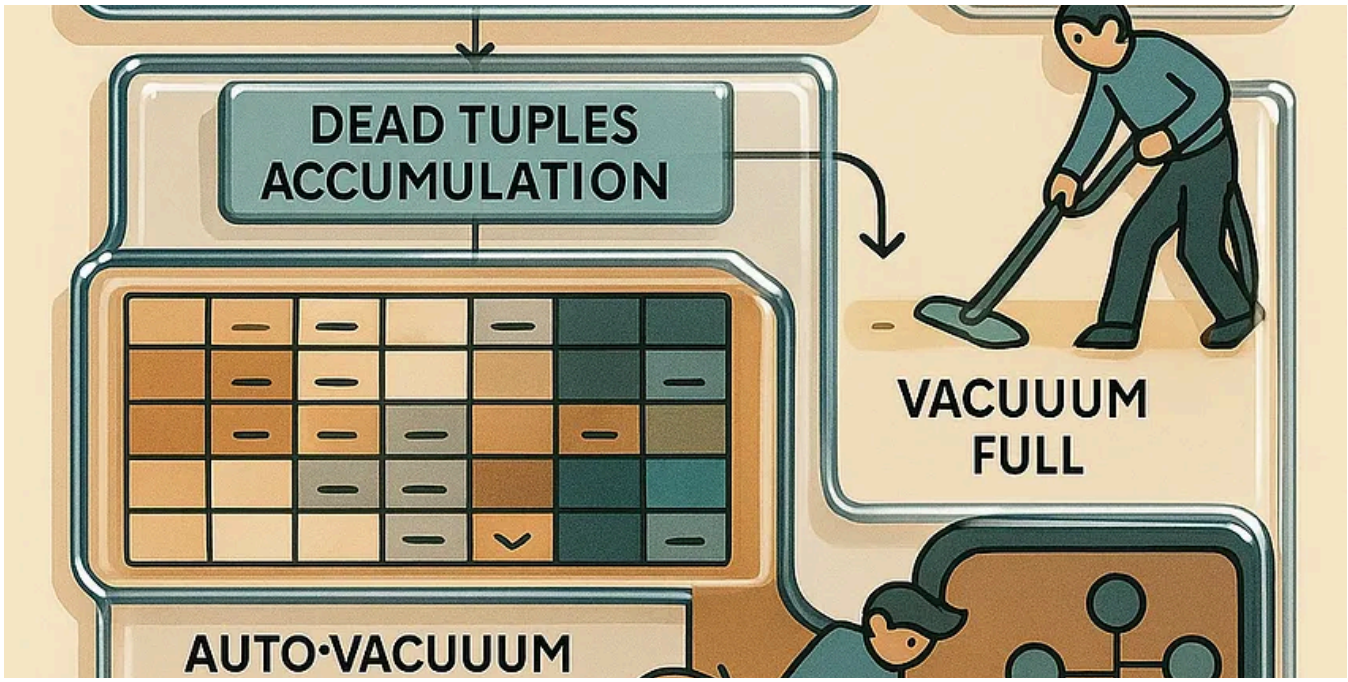


Jeyaram Ayyalusamy

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

08-PostgreSQL 17: Complete Tuning Guide for VACUUM & AUTOVACUUM

PostgreSQL's MVCC design creates dead tuples during UPDATE/DELETE. VACUUM reclaims them; AUTOVACUUM schedules that work. Get these knobs...

Sep 1 26

See all from Jeyaram Ayyalusamy

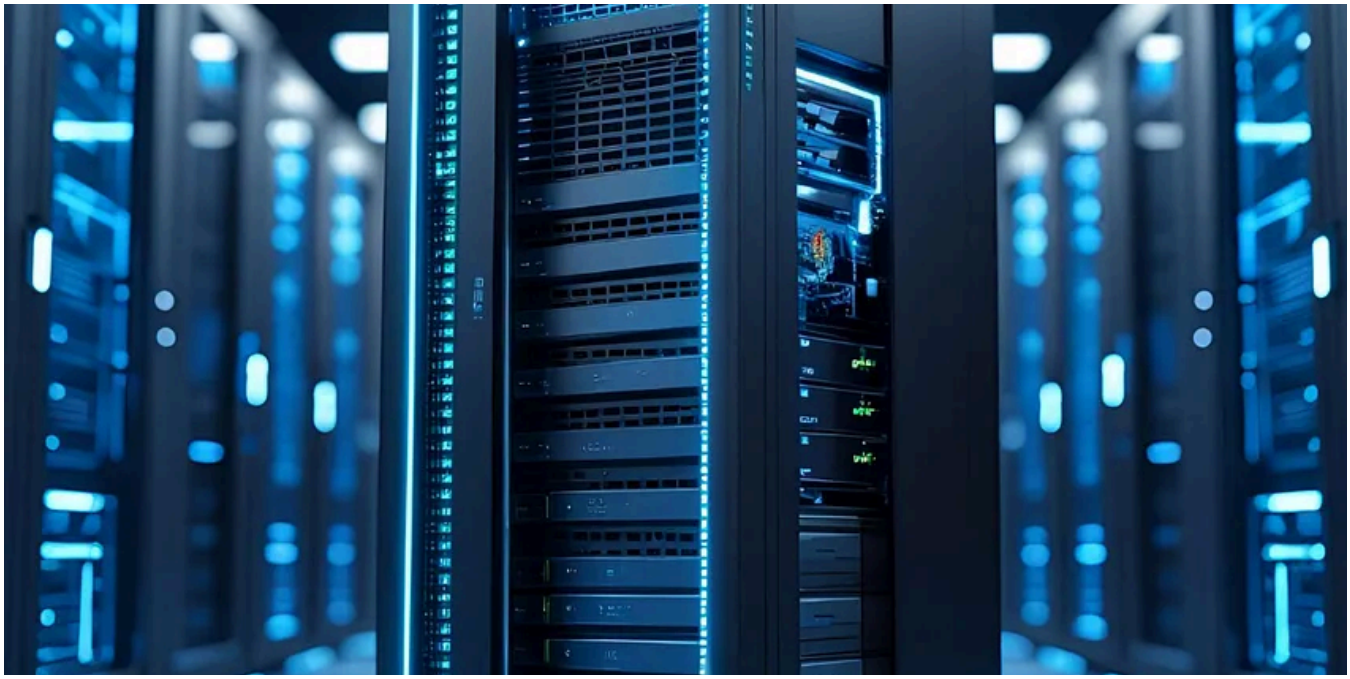
Recommended from Medium

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

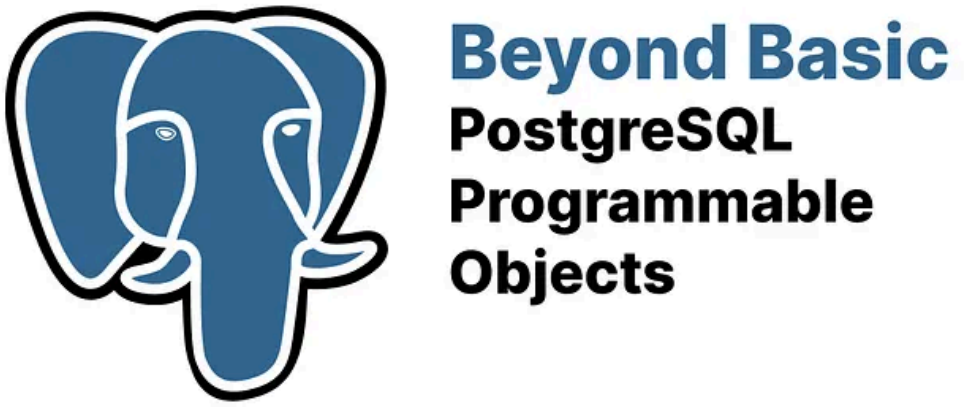


Rizqi Mulki

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.

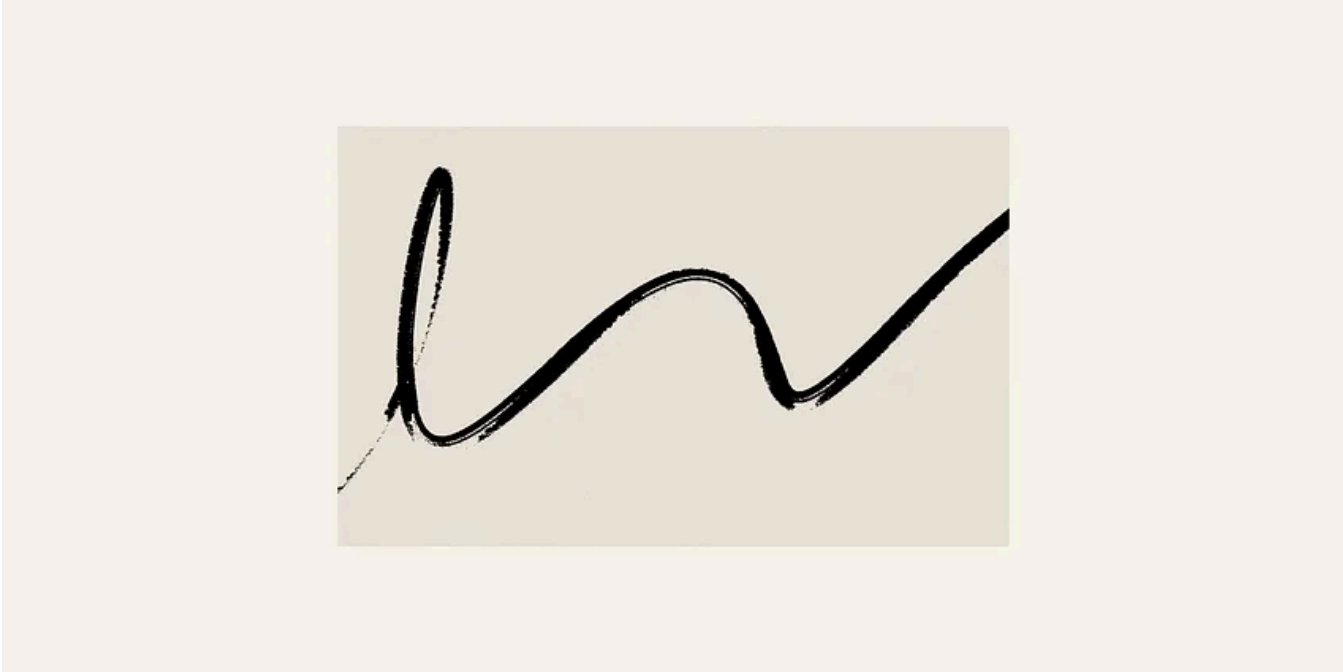
Sep 1 68 1

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



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