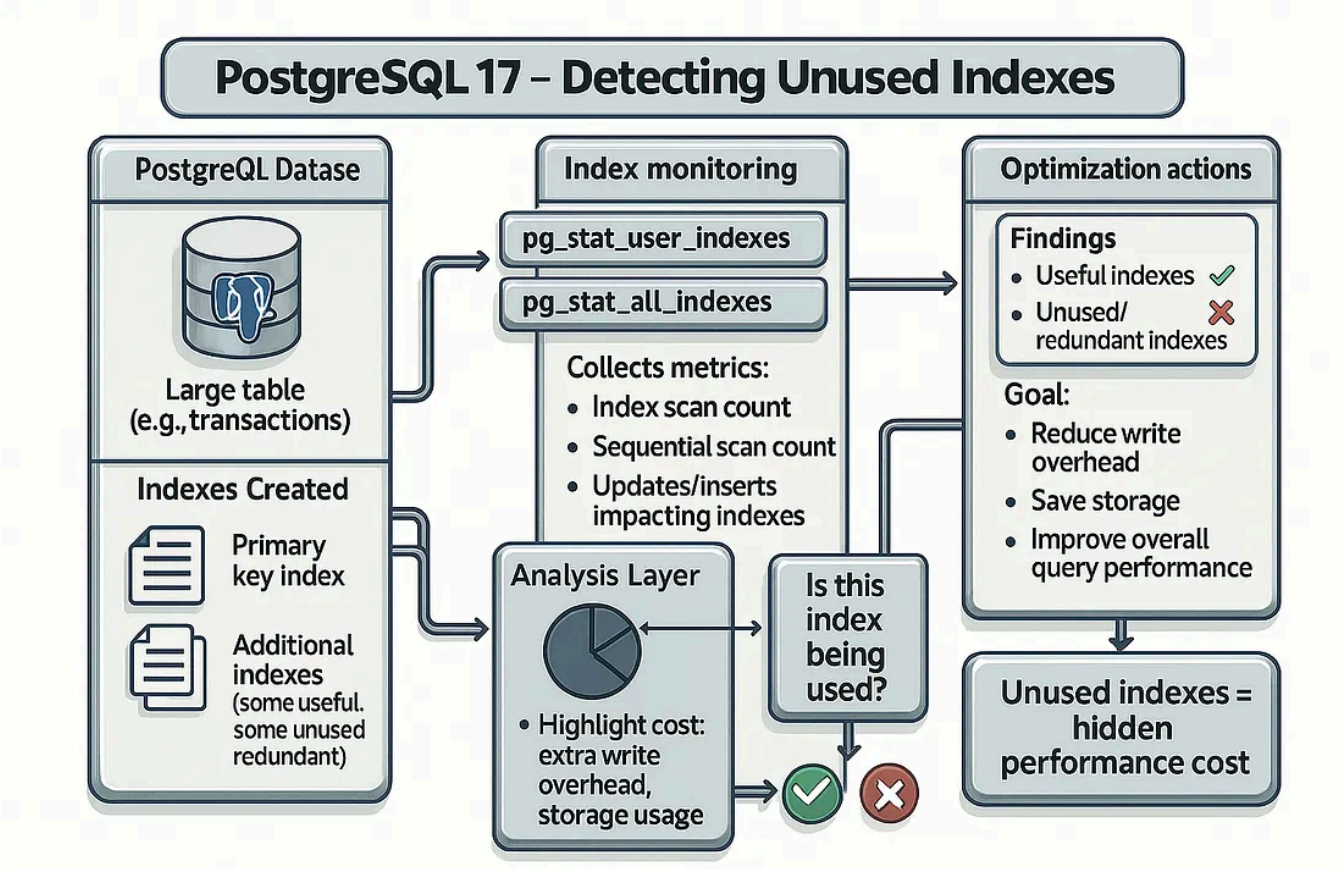
28 - PostgreSQL 17 Performance Tuning: Creating Tables, Populating 10M Records, and Detecting Unused Indexes

7 min read · 3 days ago

Jeyaram Ayyalusamy Following



When tuning PostgreSQL, most developers and DBAs focus on finding **missing indexes** to accelerate queries. That's an important step, but there's another side to the story: sometimes you need to identify **indexes that should not exist at all.**

Unused or redundant indexes are often overlooked, yet they can silently degrade system performance. In this article, we'll explore why this happens, and walk through a practical example of creating a large dataset, building indexes, and then analyzing which ones are actually useful.

Why Unused Indexes Can Hurt Performance

Indexes exist to make queries faster, but they come with trade-offs. While they improve read performance, they add extra work to every write operation. Here's why:

- **Disk Space Waste** — Every index consumes physical storage. If your database holds tens or hundreds of gigabytes of unused indexes, you're wasting space that could be used for valuable data.
- **Slower Write Operations** — Each `INSERT`, `UPDATE`, or `DELETE` must also update every index on the table. The more indexes you have, the more overhead is added to each transaction.
- **Maintenance Overhead** — Background tasks like **VACUUM**, **autovacuum**, and **analyze** have to maintain every index. This makes them slower and consumes additional CPU and I/O.

In short: **an unused index is not free**. In large production systems, a few unused indexes can be costly, while dozens or hundreds can seriously harm performance.

Step 1: Creating a Test Table

Let's simulate a real-world scenario with a product catalog table.

```
CREATE TABLE products (
    product_id   BIGINT GENERATED BY DEFAULT AS IDENTITY,
    product_name TEXT NOT NULL,
    category     TEXT NOT NULL,
    price        NUMERIC(10,2) NOT NULL,
```

```
    stock_qty    INT NOT NULL,
    status       TEXT NOT NULL,
    created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
    updated_at   TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

```
postgres=# CREATE TABLE products (
postgres(#     product_id   BIGINT GENERATED BY DEFAULT AS IDENTITY,
postgres(#     product_name TEXT NOT NULL,
postgres(#     category     TEXT NOT NULL,
postgres(#     price        NUMERIC(10,2) NOT NULL,
postgres(#     stock_qty    INT NOT NULL,
postgres(#     status       TEXT NOT NULL,
postgres(#     created_at   TIMESTAMPTZ NOT NULL DEFAULT now(),
postgres(#     updated_at   TIMESTAMPTZ NOT NULL DEFAULT now()
postgres(# );
CREATE TABLE
postgres=#
```

This table includes common fields you'd find in an e-commerce or inventory management system: product names, categories, prices, stock levels, and status flags.

Step 2: Inserting 10 Million Rows

To properly test index usage, we need a large dataset. Using PostgreSQL's `generate_series()` and random functions, we can quickly populate 10 million rows:

```
INSERT INTO products (product_name, category, price, stock_qty, status, created
SELECT
    'Product_' || gs::text,
    'Category_' || (1 + (gs % 20))::text,
    round((random()*999 + 1)::numeric, 2),
    (random()*1000)::int,
    CASE
```

```
        WHEN random() < 0.8 THEN 'open'
        WHEN random() < 0.15 THEN 'archived'
        ELSE 'backorder'
    END,
    now() - ((random()*365)::int || ' days')::interval,
    now()
FROM generate_series(1, 10000000) AS gs;
```

```
postgres=# INSERT INTO products (product_name, category, price, stock_qty, stat
SELECT
    'Product_' || gs::text,
    'Category_' || (1 + (gs % 20))::text,
    round((random()*999 + 1)::numeric, 2),
    (random()*1000)::int,
    CASE
        WHEN random() < 0.8 THEN 'open'
        WHEN random() < 0.15 THEN 'archived'
        ELSE 'backorder'
    END,
    now() - ((random()*365)::int || ' days')::interval,
    now()
FROM generate_series(1, 10000000) AS gs;

INSERT 0 10000000
postgres=#
```

This script generates:

- **20 categories** (`Category_1` … `Category_20`)
- Prices between **$1.00 and $999.00**
- Random stock levels from **0 to 1000**
- Status values, 80% `open` , 15% `archived` , 5% `backorder`

- Creation dates spread across the past year

By the end, we have a **realistic 10M row table** — large enough to illustrate the performance impact of indexes.

Step 3: Creating Indexes

Next, we'll create several indexes. Some of these will be useful for queries, while others may rarely get used.

```
-- Primary key
ALTER TABLE products ADD CONSTRAINT products_pkey PRIMARY KEY (product_id);
```

```
-- Index on status
CREATE INDEX idx_products_status ON products (status);

-- Partial index for "open" status only
CREATE INDEX idx_products_status_open ON products (status)
WHERE status = 'open';

-- Multi-column index on category and price
CREATE INDEX idx_products_category_price ON products (category, price);

-- Index on stock quantity (may or may not be useful)
CREATE INDEX idx_products_stock ON products (stock_qty);

-- Expression index for case-insensitive searches
CREATE INDEX idx_products_lower_name ON products (lower(product_name));

-- Index on creation timestamp for time-range queries
CREATE INDEX idx_products_created_at ON products (created_at);
```

```
postgres=# ALTER TABLE products ADD CONSTRAINT products_pkey PRIMARY KEY (produ
ALTER TABLE
postgres=#
postgres=# CREATE INDEX idx_products_status ON products (status);
CREATE INDEX
postgres=# CREATE INDEX idx_products_status_open ON products (status)
postgres-# WHERE status = 'open';
CREATE INDEX
postgres=#
postgres=# CREATE INDEX idx_products_category_price ON products (category, pric
CREATE INDEX
postgres=#
postgres=# CREATE INDEX idx_products_stock ON products (stock_qty);
CREATE INDEX
postgres=# CREATE INDEX idx_products_lower_name ON products (lower(product_name
CREATE INDEX
postgres=# CREATE INDEX idx_products_created_at ON products (created_at);
CREATE INDEX
postgres=#
```

Now, our table has **seven indexes** in addition to the primary key. Some of these will be heavily used, others may end up as dead weight.

Step 4: Detecting use Indexes

PostgreSQL tracks index usage statistics in the `pg_stat_all_indexes` system view. To identify which indexes are actively used, you can filter for those with non-zero scan counts (`idx_scan > 0`). This helps distinguish valuable indexes from those that may be candidates for review or removal.

Use the following query to list only the indexes that have been used at least once:

```
SELECT
    s.schemaname,
    c.relname AS table_name,
    s.indexrelname AS index_name,
    s.idx_scan,
    pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size
FROM
    pg_stat_all_indexes s
JOIN
    pg_class c ON c.oid = s.indexrelid
WHERE
    s.schemaname NOT IN ('pg_catalog', 'information_schema')
    AND s.idx_scan > 0  -- ✅ Only include indexes that have been used
ORDER BY
    s.idx_scan ASC,
    pg_relation_size(s.indexrelid) DESC;
```

```
postgres=# SELECT
postgres-#     s.schemaname,
postgres-#     c.relname AS table_name,
postgres-#     s.indexrelname AS index_name,
postgres-#     s.idx_scan,
postgres-#     pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size
postgres-# FROM
postgres-#     pg_stat_all_indexes s
postgres-# JOIN
postgres-#     pg_class c ON c.oid = s.indexrelid
postgres-# WHERE
postgres-#     s.schemaname NOT IN ('pg_catalog', 'information_schema')
postgres-#     AND s.idx_scan > 0  -- ✅ Only include indexes that have been us
postgres-# ORDER BY
postgres-#     s.idx_scan ASC,
postgres-#     pg_relation_size(s.indexrelid) DESC;
 schemaname |         table_name          |         index_name          | idx_s
------------+-----------------------------+-----------------------------+------
 public     | idx_products_status         | idx_products_status         |
 pg_toast   | pg_toast_2618_index         | pg_toast_2618_index         |
 public     | idx_products_category_price | idx_products_category_price |
(3 rows)

postgres=#
```

Step 4: Detecting Unused Indexes

PostgreSQL provides index usage statistics in `pg_stat_all_indexes`. With a simple query, you can see how often an index has been used and how much space it consumes:

```
SELECT
    s.schemaname,
    c.relname AS table_name,
    s.indexrelname AS index_name,
    s.idx_scan,
    pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size
FROM
    pg_stat_all_indexes s
JOIN
    pg_class c ON c.oid = s.indexrelid
WHERE
    s.schemaname NOT IN ('pg_catalog', 'information_schema')
ORDER BY
    s.idx_scan ASC,
    pg_relation_size(s.indexrelid) DESC;
```

```
postgres=# SELECT
postgres-#     s.schemaname,
postgres-#     c.relname AS table_name,
postgres-#     s.indexrelname AS index_name,
postgres-#     s.idx_scan,
postgres-#     pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size
postgres-# FROM
postgres-#     pg_stat_all_indexes s
postgres-# JOIN
postgres-#     pg_class c ON c.oid = s.indexrelid
postgres-# WHERE
postgres-#     s.schemaname NOT IN ('pg_catalog', 'information_schema')
postgres-# ORDER BY
postgres-#     s.idx_scan ASC,
postgres-#     pg_relation_size(s.indexrelid) DESC;
 schemaname |      table_name      |      index_name      | idx_scan | index_si
```

```
------------+----------------------+----------------------+----------+---------
 pg_toast   | pg_toast_2618_index  | pg_toast_2618_index  |        0 | 16 kB
 pg_toast   | pg_toast_2619_index  | pg_toast_2619_index  |        0 | 16 kB
 pg_toast   | pg_toast_1255_index  | pg_toast_1255_index  |        0 | 16 kB
```

What the Columns Mean:

- **`idx_scan`** → number of times the index was scanned since the last stats reset
- **`index_size`** → how much disk space the index uses

If `idx_scan = 0` and the index is several gigabytes in size, it's a candidate for review and possibly removal.

Step 5: Why Manual Review Still Matters

Dropping indexes isn't always straightforward. There are edge cases:

- Some indexes are rarely used but **critical for month-end reporting**.
- Unique indexes enforce constraints, even if not scanned.
- Certain application features might rely on an index only under specific conditions.

That's why it's wise to:

- Perform **manual checking** before dropping.
- Use monitoring tools like **pgBadger**, **pg_stat_statements**, or enterprise observability dashboards.
- Reset statistics (`SELECT pg_stat_reset();`) in a staging environment to measure fresh usage.

Step 6: Best Practices for Index Monitoring

1. **Regularly check index statistics** — Run the `pg_stat_all_indexes` query during maintenance cycles.
2. **Automate monitoring** — Integrate with tools that track index usage over time.
3. **Avoid over-indexing** — Create indexes only for queries that genuinely benefit from them.
4. **Benchmark before dropping** — Test query performance with and without the index.

Final Thoughts

Unused indexes are a **hidden performance tax.** They don't throw errors or show up in query plans, but they quietly consume storage, slow down writes, and increase maintenance overhead.

By creating a table, populating it with millions of rows, and monitoring index usage in PostgreSQL 17, you can:

- Identify indexes that truly add value.
- Spot those that waste space and hurt performance.
- Keep your database lean, efficient, and scalable.

Monitoring activity and statistics lookups isn't just a nice-to-have — it's a **must-have for serious performance tuning**.

📢 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 AWS MySQL Oracle Open Source

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

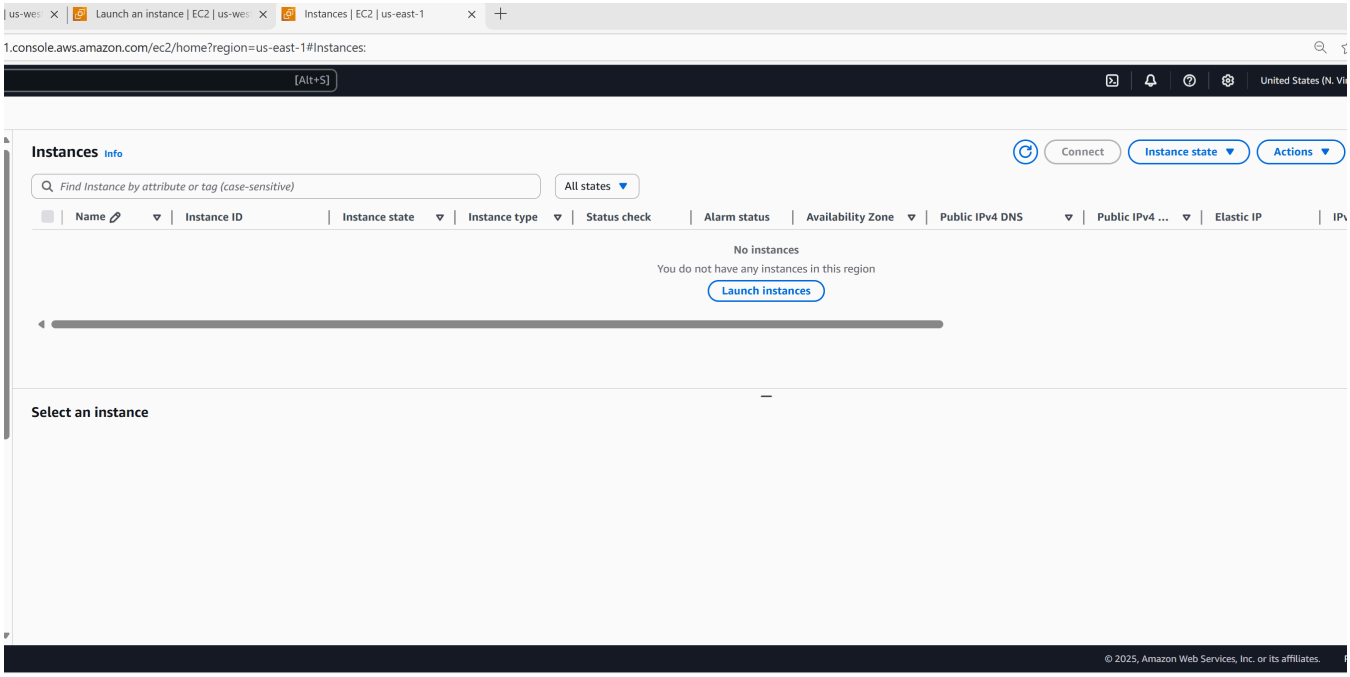
What are your thoughts?

Addy Clement
2 days ago

Quite detailed and easy to follow !

1 Reply

More from Jeyaram Ayyalusamy

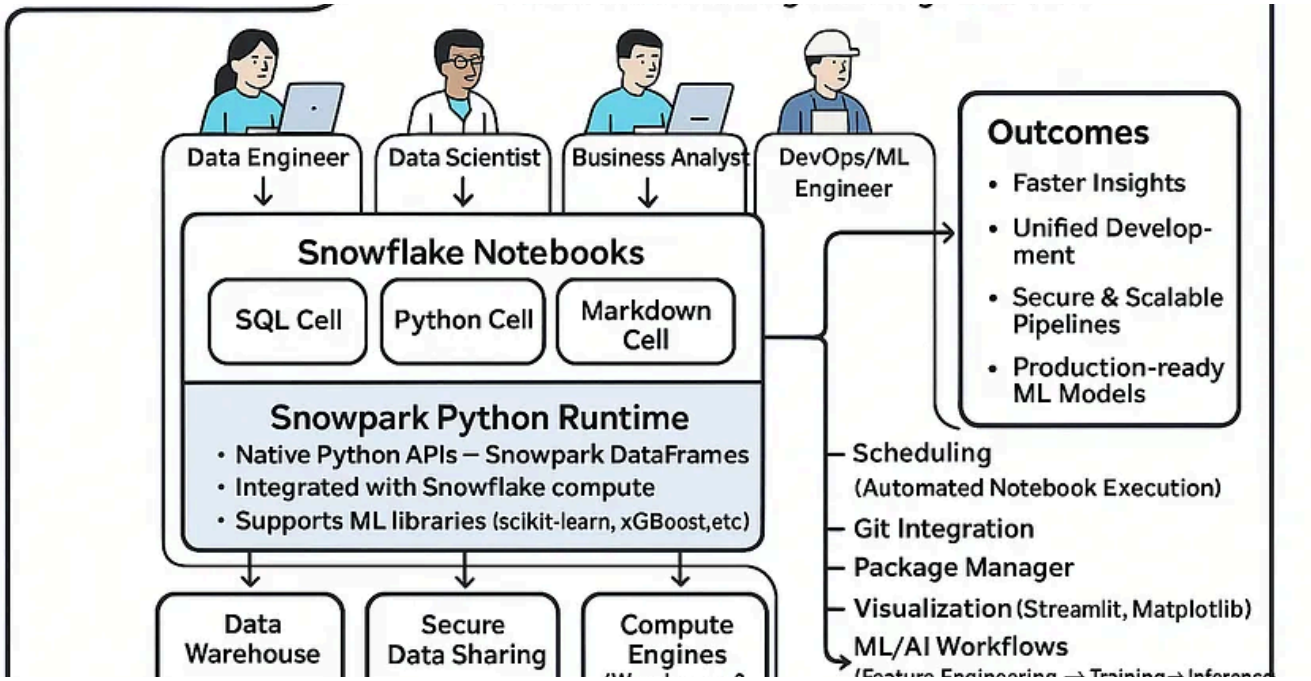


Jeyaram Ayyalusamy

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

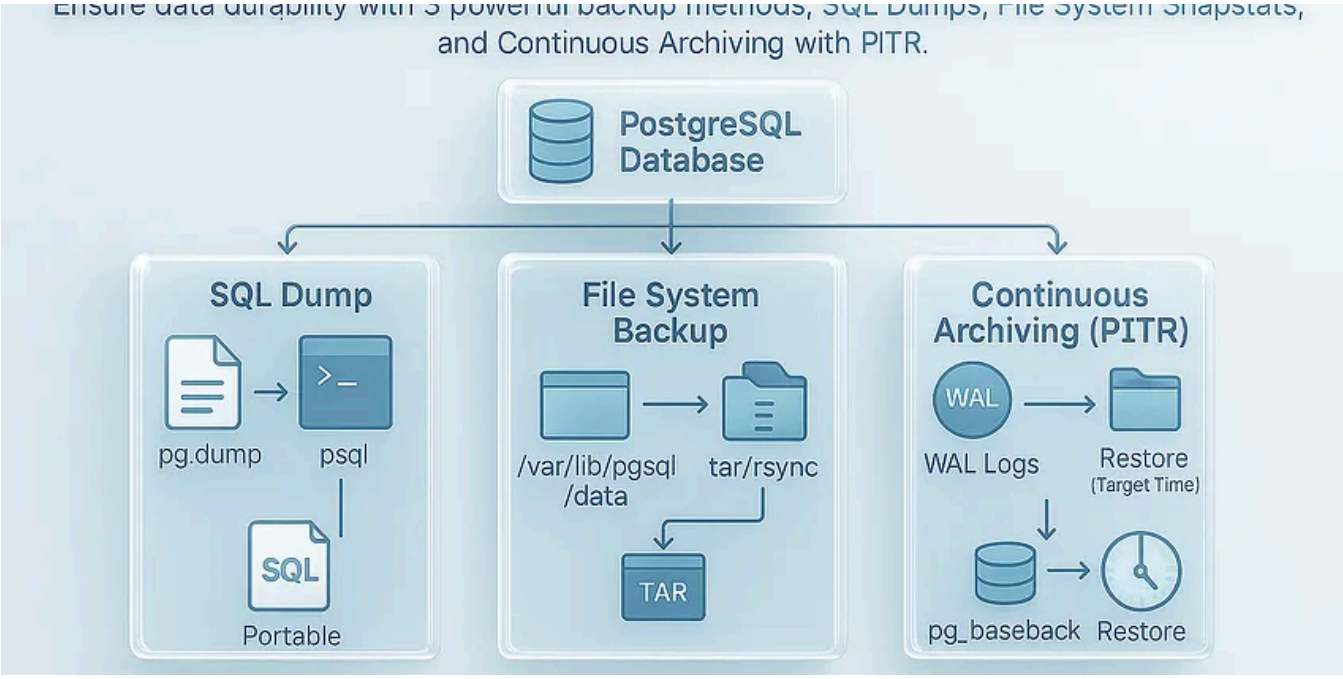


Jeyaram Ayyalusamy

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

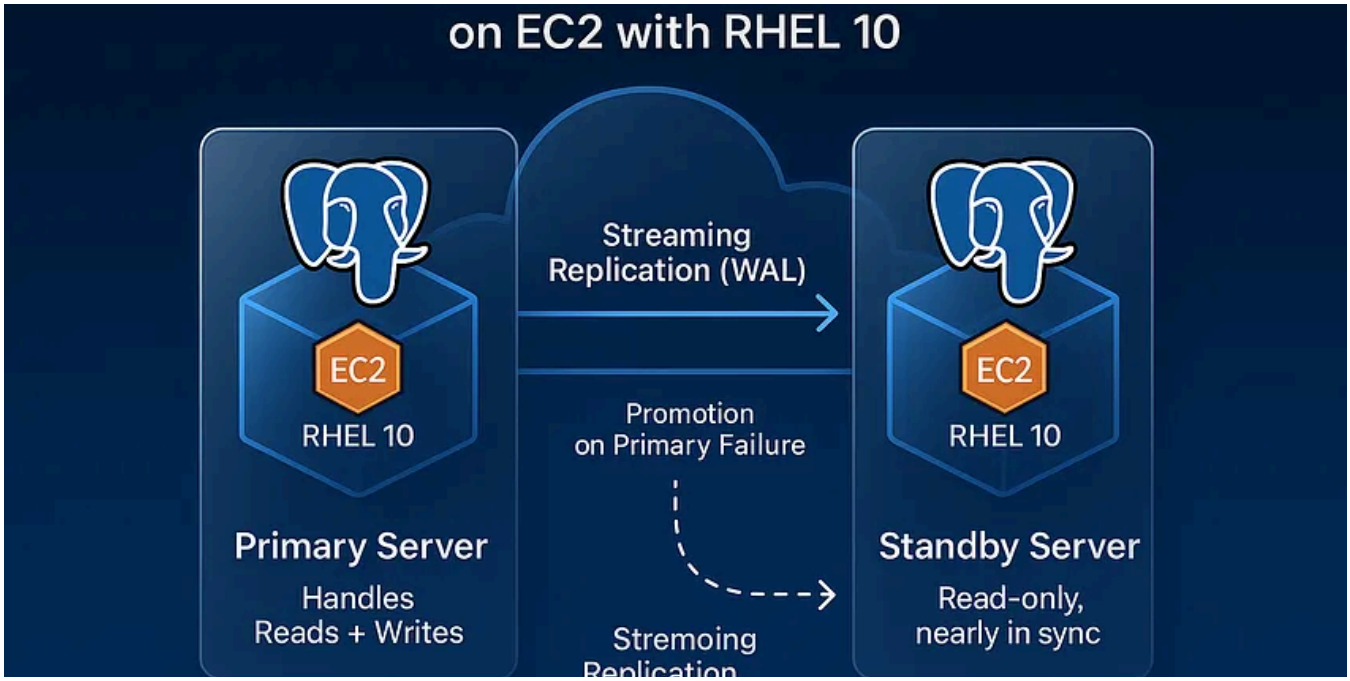


Jeyaram Ayyalusamy

🛡️ 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

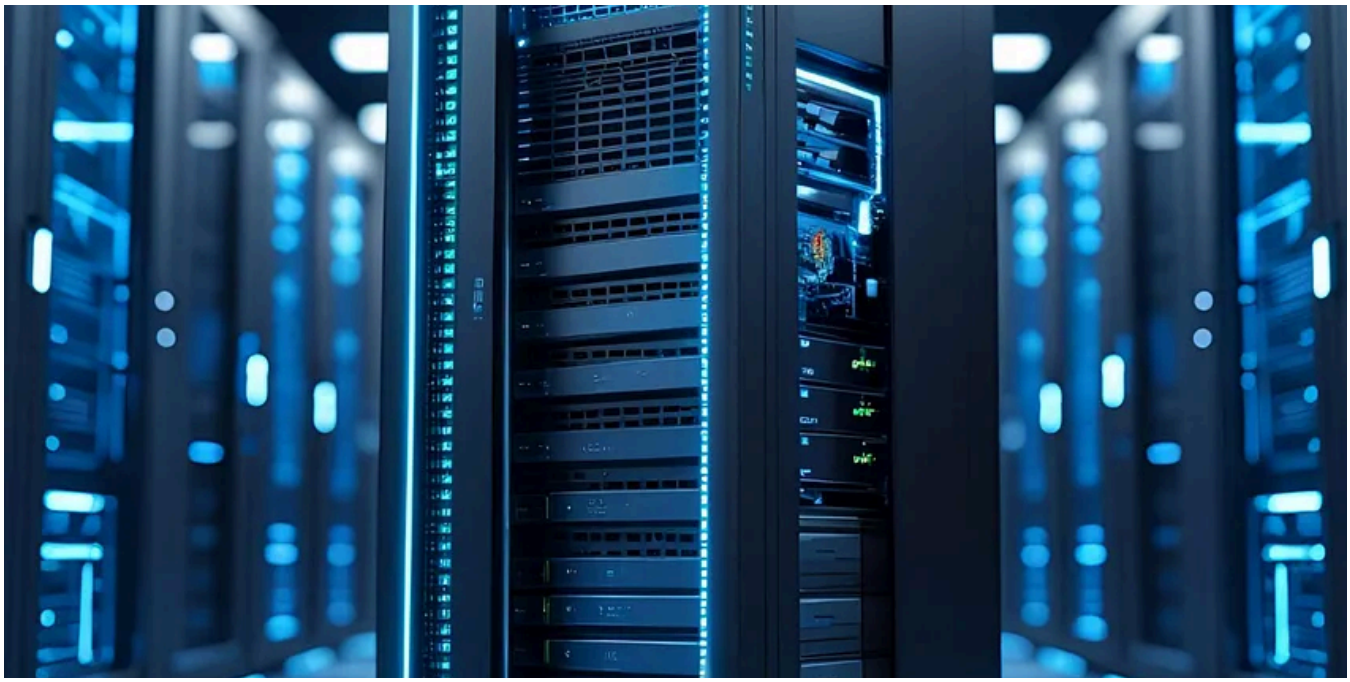
🚀 Streaming Replication in PostgreSQL 17 on EC2 with RHEL 10—A Deep Dive

🐘 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...

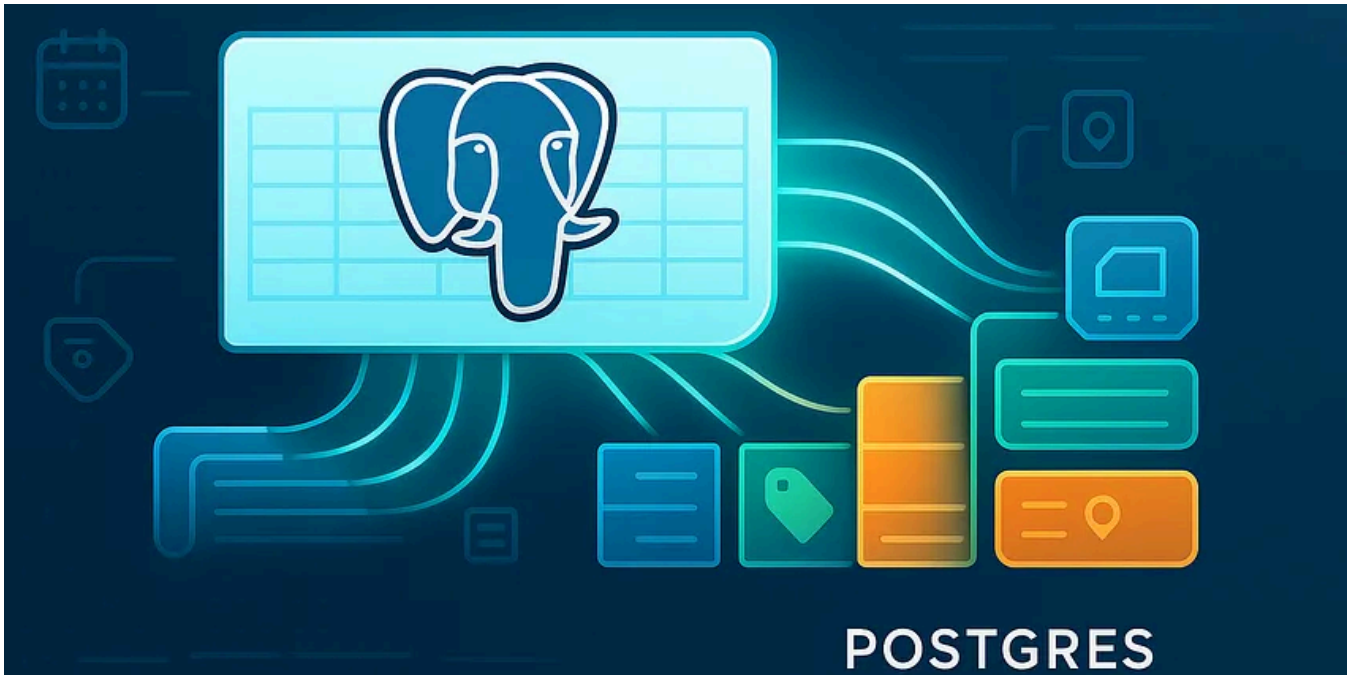
6d ago 5

Vijay Gadhave

Master SQL's QUALIFY Clause for Cleaner, Faster Window Queries

Note: If you're not a medium member, CLICK HERE

May 20 12 1

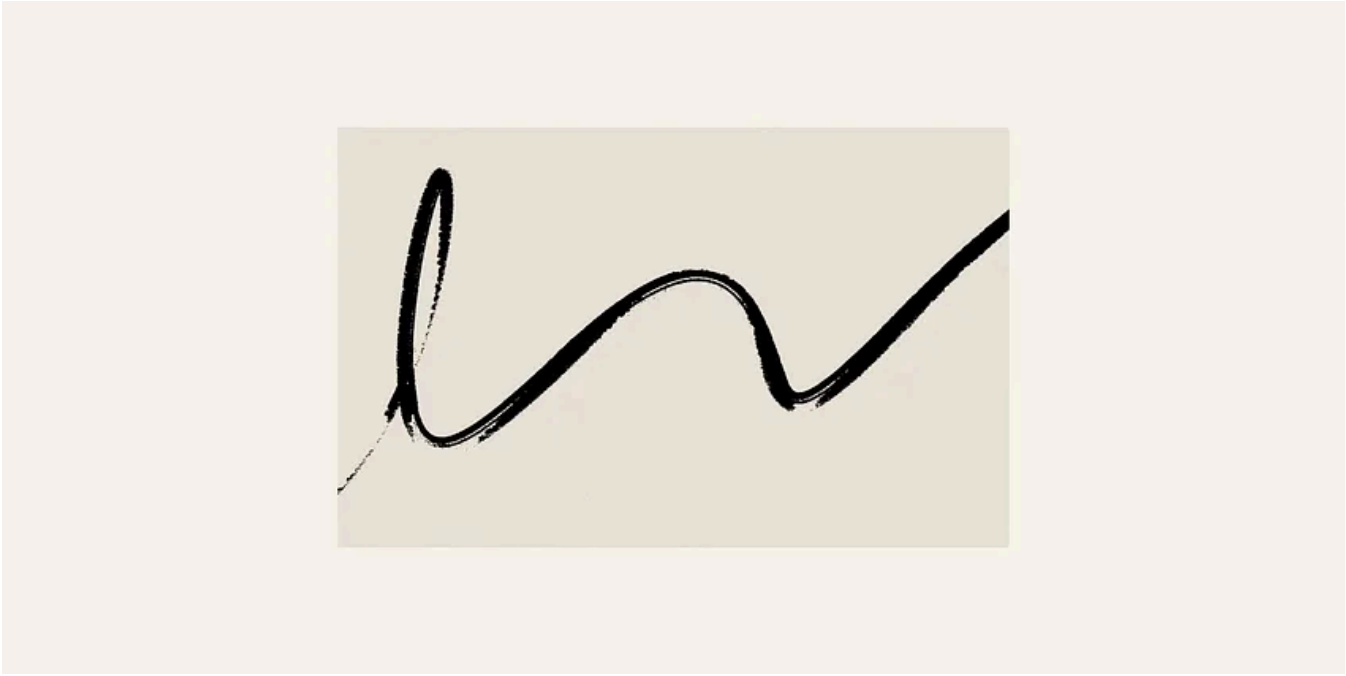


Thinking Loop

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

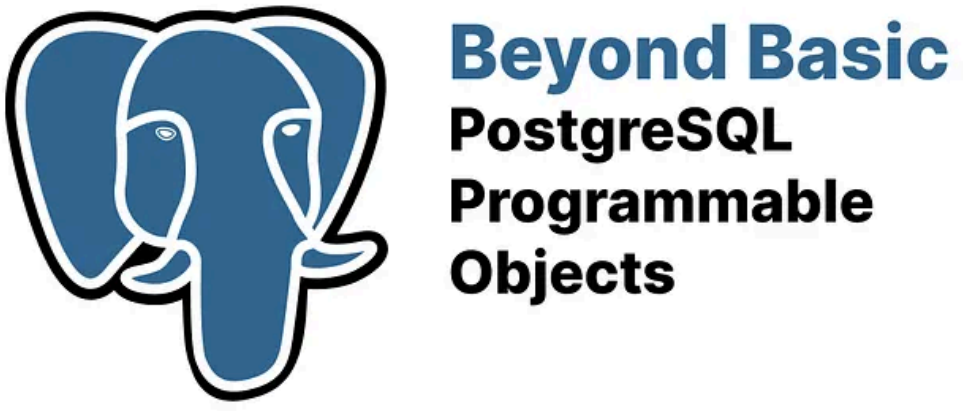


Rohan

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



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