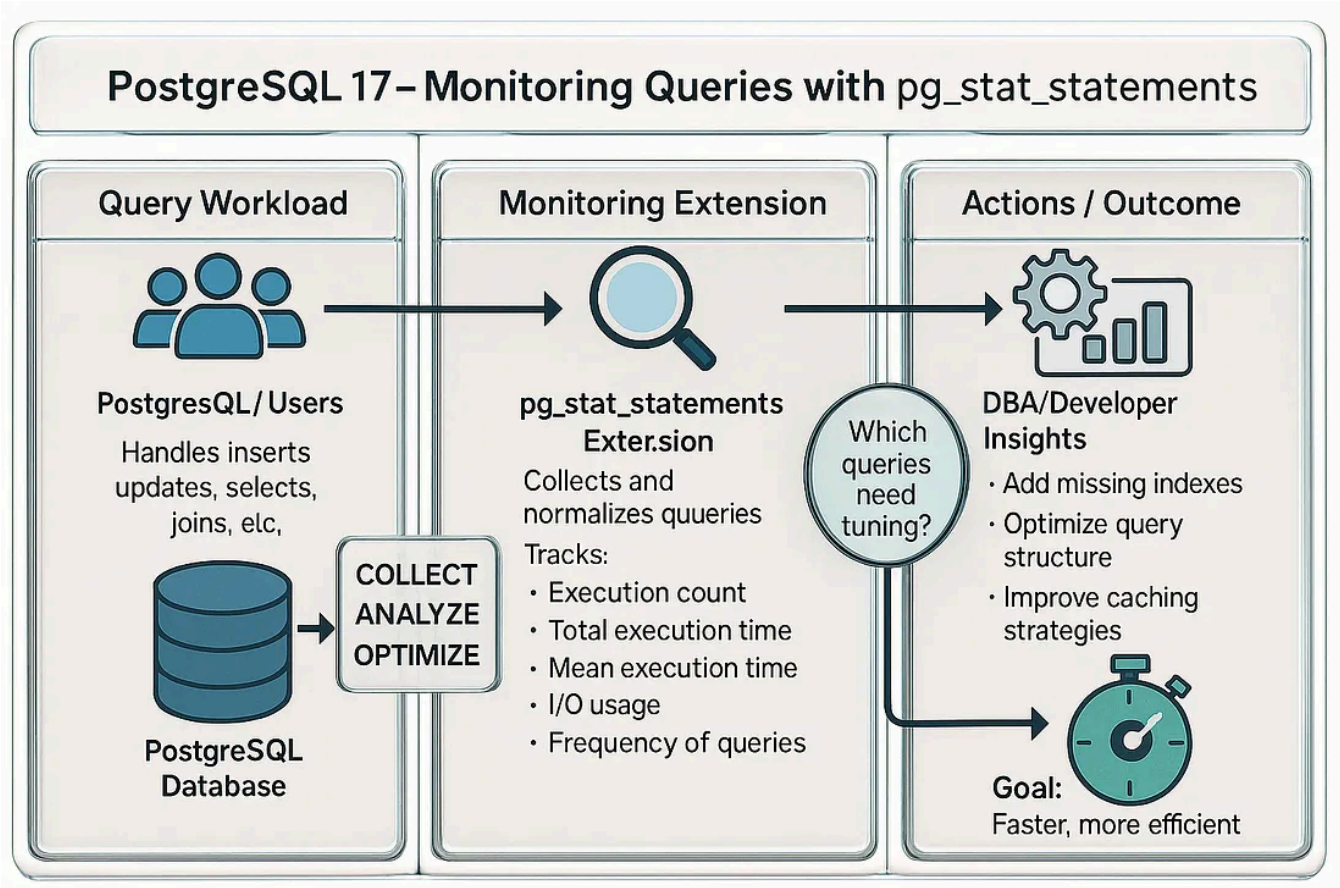
27 - PostgreSQL 17 Performance Tuning: Monitoring Queries with `pg_stat_statements`

11 min read · 3 days ago

Jeyaram Ayyalusamy Following



You cannot improve database performance without first collecting the right data. PostgreSQL provides a wide range of **system views** that let administrators and developers understand what's really happening inside their database.

One of the most powerful tools for this purpose is the **`pg_stat_statements`** extension. It helps you identify:

- Which queries are slow.
- How often queries are executed.
- Whether queries have stable or fluctuating runtimes.

Instead of guessing or parsing log files, `pg_stat_statements` provides a clear, aggregated view of query behavior.

Step 1: 🧩 Installing PostgreSQL Extension Modules

To enable advanced features like `pg_stat_statements`, you need to install the PostgreSQL contrib package. This package includes a collection of officially supported extensions that enhance PostgreSQL's functionality.

On Red Hat-based systems (RHEL, CentOS, Amazon Linux), run:

```
sudo yum install postgresql17-contrib
```

```
[ec2-user@ip-172-31-29-78 ~]$ sudo yum install postgresql17-contrib
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use "rhc" or

Last metadata expiration check: 2:44:15 ago on Fri Sep 19 18:47:07 2025.
Dependencies resolved.
================================================================================
 Package                                 Architecture                  Ve
================================================================================
Installing:
 postgresql17-contrib                    x86_64                        17
```

```
Installing dependencies:
 libxslt                                    x86_64                          1.

Transaction Summary
================================================================================
Install  2 Packages

Total download size: 920 k
Installed size: 3.2 M
Is this ok [y/N]: y
Downloading Packages:
(1/2): libxslt-1.1.39-8.el10_0.x86_64.rpm
(2/2): postgresql17-contrib-17.6-1PGDG.rhel10.x86_64.rpm
--------------------------------------------------------------------------------
Total
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
  Installing       : libxslt-1.1.39-8.el10_0.x86_64
  Installing       : postgresql17-contrib-17.6-1PGDG.rhel10.x86_64
  Running scriptlet: postgresql17-contrib-17.6-1PGDG.rhel10.x86_64
Installed products updated.

Installed:
  libxslt-1.1.39-8.el10_0.x86_64

Complete!
[ec2-user@ip-172-31-29-78 ~]$
```

This command:

- Installs the **contrib module** for PostgreSQL 17
- Adds support for extensions like `pg_stat_statements`, `tablefunc`, `uuid-ossp`, and more
- Is required before enabling certain features in `postgresql.conf` (e.g., `shared_preload_libraries = 'pg_stat_statements'`)

> 🛠️ After installation, don't forget to restart the PostgreSQL service and run `CREATE EXTENSION pg_stat_statements;` inside your database to activate it.

Step 2: Enable `pg_stat_statements`

By default, this extension is not active. You need to enable it in `postgresql.conf`.

1. Open your PostgreSQL config file (e.g., `postgresql.conf`).
2. Find the line:

```
#shared_preload_libraries = ''
```

```
postgres=# show shared_preload_libraries;
 shared_preload_libraries
--------------------------

(1 row)

postgres=#
```

3. Uncomment it and set the value to include `pg_stat_statements`:

```
shared_preload_libraries = 'pg_stat_statements'
```

```
[postgres@ip-172-31-29-78 data]$ cat postgresql.conf | grep -i shared_preload_l
shared_preload_libraries = 'pg_stat_statements'         # (change requires rest
[postgres@ip-172-31-29-78 data]$
```

```
postgres=# show shared_preload_libraries;
 shared_preload_libraries
--------------------------
 pg_stat_statements
(1 row)

postgres=#
```

4. Restart PostgreSQL for the change to take effect.

```
[ec2-user@ip-172-31-29-78 ~]$ sudo systemctl restart postgresql-17
[ec2-user@ip-172-31-29-78 ~]$
```

Finally, install the extension in the database you want to monitor:

```
CREATE EXTENSION pg_stat_statements;
```

This will create the system view `pg_stat_statements`.

```
postgres=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
postgres=#
```

Step 3: Generate Some Demo Load

For demonstration, let's create a `products` table with 10 million rows and run a few queries against it.

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

```
postgres=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
postgres=#
postgres=# drop table products;
DROP TABLE
postgres=# CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    price        NUMERIC,
    stock_qty    INT
);
CREATE TABLE
postgres=#
```

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

```
postgres=# INSERT INTO products (product_name, category, price, stock_qty)
SELECT
    'Product_' || g,
    'Category_' || (g % 10),
    (random()*500)::NUMERIC,
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
INSERT 0 10000000
ANALYZE
postgres=#
```

Now run queries such as:

```
SELECT * FROM products WHERE product_id = 100;
SELECT * FROM products WHERE category = 'Category_3';
UPDATE products SET stock_qty = stock_qty + 1 WHERE category = 'Category_5';
```

```
postgres=# SELECT * FROM products WHERE product_id = 100;
 product_id | product_name |  category  |      price       | stock_qty
------------+--------------+------------+------------------+-----------
        100 | Product_100  | Category_0 | 228.865334176435 |        36
(1 row)

postgres=#
```

```
postgres=# SELECT * FROM products WHERE category = 'Category_3';
 product_id |  product_name   |  category  |        price         | stock_qty
------------+-----------------+------------+----------------------+-----------
          3 | Product_3       | Category_3 |    216.496292091521 |         9
         13 | Product_13      | Category_3 |    158.801528884299 |        50
         23 | Product_23      | Category_3 |    402.177798888444 |        60
         33 | Product_33      | Category_3 |    188.638645203066 |        26
         43 | Product_43      | Category_3 |    217.520085735964 |        95
         53 | Product_53      | Category_3 |    204.260688438616 |        43
         63 | Product_63      | Category_3 |    4.38792827933232 |        36
         73 | Product_73      | Category_3 |    179.192418212635 |        13
         83 | Product_83      | Category_3 |    177.750125505008 |        33
         93 | Product_93      | Category_3 |    406.374932408794 |        32
        103 | Product_103     | Category_3 |    47.2825437678309 |         5
        113 | Product_113     | Category_3 |    347.047753193479 |        34
        123 | Product_123     | Category_3 |    243.453650026469 |        46
        133 | Product_133     | Category_3 |    400.878856999775 |        99
        143 | Product_143     | Category_3 |    110.710054872779 |        84
        153 | Product_153     | Category_3 |    422.519471685829 |        43
        163 | Product_163     | Category_3 |     418.99101277994 |        10
        173 | Product_173     | Category_3 |    330.138006726809 |        66
        183 | Product_183     | Category_3 |    266.622715912463 |        60
```

```
postgres=# UPDATE products SET stock_qty = stock_qty + 1 WHERE category = 'Cate
UPDATE 1000000
postgres=#
```

Step 4: Query `pg_stat_statements`

Now check what PostgreSQL recorded:

```
SELECT query, calls, total_exec_time, mean_exec_time, rows, stddev_exec_time
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

👉 Each row in this view represents one **normalized query**. Queries with different parameter values are aggregated together. For example:

- `SELECT * FROM products WHERE product_id = 100;`
- `SELECT * FROM products WHERE product_id = 200;`

Both are shown as:

```
postgres=# SELECT query, calls, total_exec_time, mean_exec_time, rows, stddev_e
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
                              query                                | calls |
-------------------------------------------------------------------+-------+--
 INSERT INTO products (product_name, category, price, stock_qty)   +|     1 | 3
 SELECT                                                            +|       |
     $1 || g,                                                      +|       |
     $2 || (g % $3),                                               +|       |
     (random()*$4)::NUMERIC,                                       +|       |
```

```
      (random()*$5)::INT                                                +|       |
  FROM generate_series($6, $7) g                                         |       |
  UPDATE products SET stock_qty = stock_qty + $1 WHERE category = $2      |     1 |
  ANALYZE products                                                       |     1 |
  SELECT * FROM products WHERE category = $1                             |     1 |
  CREATE EXTENSION pg_stat_statements                                    |     1 |
  CREATE TABLE products (                                               +|     1 |
      product_id   BIGSERIAL PRIMARY KEY,                               +|       |
      product_name TEXT,                                                +|       |
      category     TEXT,                                                +|       |
      price        NUMERIC,                                             +|       |
      stock_qty    INT                                                  +|       |
  )                                                                      |       |
  drop table products                                                    |     1 |
  SELECT * FROM products WHERE product_id = $1                           |     1 |
  SELECT query, calls, total_exec_time, mean_exec_time, rows            +|     1 |
  FROM pg_stat_statements                                               +|       |
  ORDER BY total_exec_time DESC                                         +|       |
  LIMIT $1                                                               |       |
  show shared_preload_libraries                                          |     2 |
 (10 rows)

 postgres=#
```

This makes it easier to measure query patterns, not just individual runs.

Step 5: Useful Columns in `pg_stat_statements`

- **`calls`** → how often the query ran.
- **`total_exec_time`** → total time spent across all executions.
- **`mean_exec_time`** → average execution time.
- **`rows`** → average number of rows returned.
- **`stddev_exec_time`** → (standard deviation) shows if query times are stable or fluctuate heavily.

Why is this important?

- If runtimes are unstable, it could mean the query sometimes hits the cache and sometimes reads from disk.

- **Shared blocks vs read blocks** columns help identify whether data came from PostgreSQL's buffer cache (`shared_blks_hit`) or from disk (`shared_blks_read`).

```
SELECT
    query,
    calls,
    shared_blks_hit,
    shared_blks_read,
    ROUND(shared_blks_hit * 100.0 / GREATEST(shared_blks_hit + shared_blks_reac
FROM
    pg_stat_statements
ORDER BY
    hit_ratio ASC
LIMIT 20;
```

```
    query,
    calls,
    shared_blks_hit,
    shared_blks_read,
    ROUND(shared_blks_hit * 100.0 / GREATEST(shared_blks_hit + shared_blks_reac
FROM
    pg_stat_statements
ORDER BY
    hit_ratio ASC
LIMIT 20;
                                    query                                     |
------------------------------------------------------------------------------+
 show shared_preload_libraries                                                |
 SELECT query, calls, total_exec_time, mean_exec_time, rows, stddev_exec_time+|
 FROM pg_stat_statements                                                     +|
 ORDER BY total_exec_time DESC                                               +|
 LIMIT $1                                                                     |
 SELECT * FROM products WHERE category = $1                                   |
 ANALYZE products                                                             |
 SELECT * FROM products WHERE product_id = $1                                 |
 CREATE TABLE products (                                                     +|
     product_id   BIGSERIAL PRIMARY KEY,                                     +|
     product_name TEXT,                                                      +|
     category     TEXT,                                                      +|
     price        NUMERIC,                                                   +|
```

```
      stock_qty    INT                                                    +|
 )                                                                        |
 CREATE EXTENSION pg_stat_statements                                      |
 drop table products                                                      |
 UPDATE products SET stock_qty = stock_qty + $1 WHERE category = $2       |
 SELECT query, calls, total_exec_time, mean_exec_time, rows               +|
 FROM pg_stat_statements                                                  +|
 ORDER BY total_exec_time DESC                                            +|
 LIMIT $1                                                                 |
 INSERT INTO products (product_name, category, price, stock_qty)          +|
 SELECT                                                                   +|
     $1 || g,                                                             +|
     $2 || (g % $3),                                                      +|
     (random()*$4)::NUMERIC,                                              +|
     (random()*$5)::INT                                                   +|
 FROM generate_series($6, $7) g                                           |
(11 rows)

postgres=#
```

Step 6: Tracking Temporary File Usage

Queries that use too much memory can spill to disk, creating **temporary files**. In OLTP systems (transaction-heavy workloads), this is usually a red flag because it slows everything down.

`pg_stat_statements` tracks this too. High temporary file I/O may indicate:

- Work memory (`work_mem`) is too low.
- Large sorts, hash joins, or index builds are happening.

```
SELECT
    query,
    calls,
    temp_blks_written,
    temp_blks_read,
    ROUND(temp_blks_written * 8 / 1024.0, 2) AS temp_written_MB,
    ROUND(temp_blks_read * 8 / 1024.0, 2) AS temp_read_MB
FROM
    pg_stat_statements
WHERE
```

```
    temp_blks_written > 0
ORDER BY
    temp_blks_written DESC
LIMIT 20;
```

```
postgres=# SELECT
    query,
    calls,
    temp_blks_written,
    temp_blks_read,
    ROUND(temp_blks_written * 8 / 1024.0, 2) AS temp_written_MB,
    ROUND(temp_blks_read * 8 / 1024.0, 2) AS temp_read_MB
FROM
    pg_stat_statements
WHERE
    temp_blks_written > 0
ORDER BY
    temp_blks_written DESC
LIMIT 20;
                              query                               | calls | temp
------------------------------------------------------------------+-------+-----
 INSERT INTO products (product_name, category, price, stock_qty)+|     1 |
 SELECT                                                         +|       |
     $1 || g,                                                   +|       |
     $2 || (g % $3),                                            +|       |
     (random()*$4)::NUMERIC,                                    +|       |
     (random()*$5)::INT                                         +|       |
 FROM generate_series($6, $7) g                                  |       |
(1 row)

postgres=#
```

Step 7: Enable I/O Timing

By default, I/O timing is not recorded because it adds some overhead. To track it, enable:

```
track_io_timing = on
```

```
postgres=# show track_io_timing;
 track_io_timing
-----------------
 off
(1 row)

postgres=#
```

```
[postgres@ip-172-31-29-78 data]$ cat postgresql.conf | grep -i track_io_timing
track_io_timing = on
[postgres@ip-172-31-29-78 data]$
```

```
[ec2-user@ip-172-31-29-78 ~]$ sudo systemctl restart postgresql-17
[ec2-user@ip-172-31-29-78 ~]$
```

```
postgres=# show track_io_timing;
 track_io_timing
-----------------
 on
(1 row)
```

```
postgres=#
```

This will let you see how much query time was spent on actual disk I/O.

🔍 Using `EXPLAIN (ANALYZE, BUFFERS)` to Diagnose Query Performance

PostgreSQL's `EXPLAIN (ANALYZE, BUFFERS)` command provides deep visibility into how a query interacts with memory and disk. Here's a real-world example:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM products WHERE category = 'Category_5';
```

Sample Output:

```
postgres=# EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM products WHERE category = 'electronics';
                                                              QUERY PLAN
---------------------------------------------------------------------------------------
 Gather  (cost=1000.00..187116.59 rows=1 width=49) (actual time=5987.833..5998.
   Workers Planned: 2
   Workers Launched: 2
   Buffers: shared hit=12025 read=111592 dirtied=19987 written=13943
   I/O Timings: shared read=13342.256 write=1658.516
   ->  Parallel Seq Scan on products  (cost=0.00..186116.49 rows=1 width=49) (a
         Filter: (category = 'electronics'::text)
         Rows Removed by Filter: 3333333
         Buffers: shared hit=12025 read=111592 dirtied=19987 written=13943
         I/O Timings: shared read=13342.256 write=1658.516
 Planning:
   Buffers: shared hit=6 read=3
   I/O Timings: shared read=1.824
 Planning Time: 1.928 ms
 Execution Time: 6000.232 ms
(15 rows)

postgres=#
```

🧠 What This Tells Us

- **Sequential Scan:** PostgreSQL scanned the entire `products` table—this is often a red flag in OLTP workloads.
- **Rows Removed by Filter:** 9 million rows were scanned but didn't match the filter, indicating poor selectivity.
- **Buffers:**
- `shared hit=14584` : These blocks were found in memory (PostgreSQL's buffer cache).
- `shared read=109033` : These blocks were read from disk—this is expensive and slows down performance.
- **I/O Timings:**
- `shared read=4033.907 ms` : Over 4 seconds were spent just reading from disk.
- **Execution Time:** The query took over 8 seconds to complete.

⚠️ Optimization Opportunity

This query would benefit from:

- A **B-tree index** on `category` :

```
CREATE INDEX idx_products_category ON products (category);
```

```
postgres=# CREATE INDEX idx_products_category ON products (category);
CREATE INDEX
```

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM products WHERE category = 'Category_5';
```

```
postgres=# EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM products WHERE category = 'Category_5';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Bitmap Heap Scan on products  (cost=11229.68..230035.48 rows=1003000 width=49)
   Recheck Cond: (category = 'Category_5'::text)
   Heap Blocks: exact=10321
   Buffers: shared hit=468 read=10711 written=59
   I/O Timings: shared read=4543.616 write=0.454
   ->  Bitmap Index Scan on idx_products_category  (cost=0.00..10978.93 rows=10
         Index Cond: (category = 'Category_5'::text)
         Buffers: shared read=858 written=25
         I/O Timings: shared read=6.589 write=0.144
 Planning:
   Buffers: shared hit=43 read=2
   I/O Timings: shared read=0.945
 Planning Time: 1.156 ms
 Execution Time: 5955.813 ms
(14 rows)

postgres=#
```

- Increasing `work_mem` if sorts or joins are involved.
- Rewriting the query to reduce the result set or improve filtering.

> 💡 Always pair `EXPLAIN (ANALYZE, BUFFERS)` with `track_io_timing = on` in `postgresql.conf` to capture disk latency.

Key Takeaways

- `pg_stat_statements` is the go-to extension for query monitoring in PostgreSQL 17.
- It aggregates queries with parameters, helping spot patterns instead of noise.
- Key insights:
- **Most expensive queries** (by time).
- **Most frequent queries** (by calls).
- **Queries with unstable runtimes** (cache vs disk).
- **Queries causing temp file I/O** (tune `work_mem`).
- Without monitoring, tuning is guesswork — with `pg_stat_statements`, it's **data-driven optimization.**

✅ Before adjusting indexes, memory, or autovacuum, always check what queries are actually causing the load.

📢 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 Open Source Mongodb Sql

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

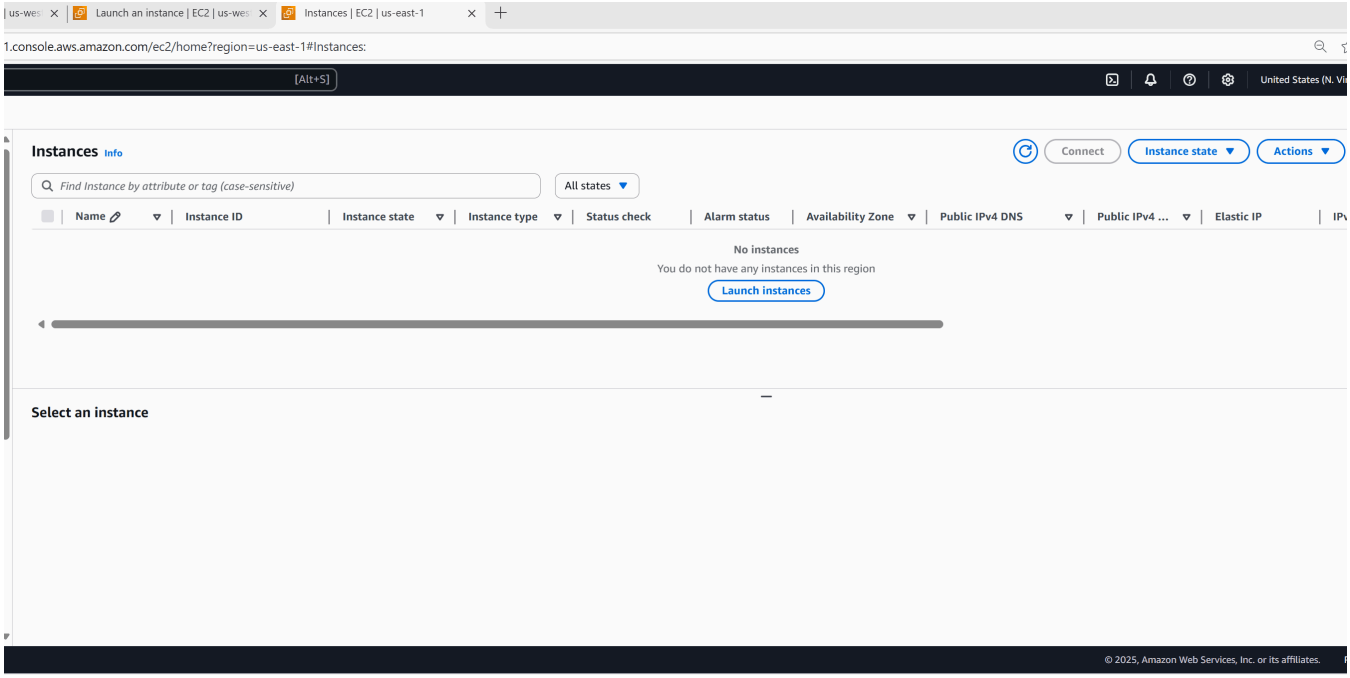
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

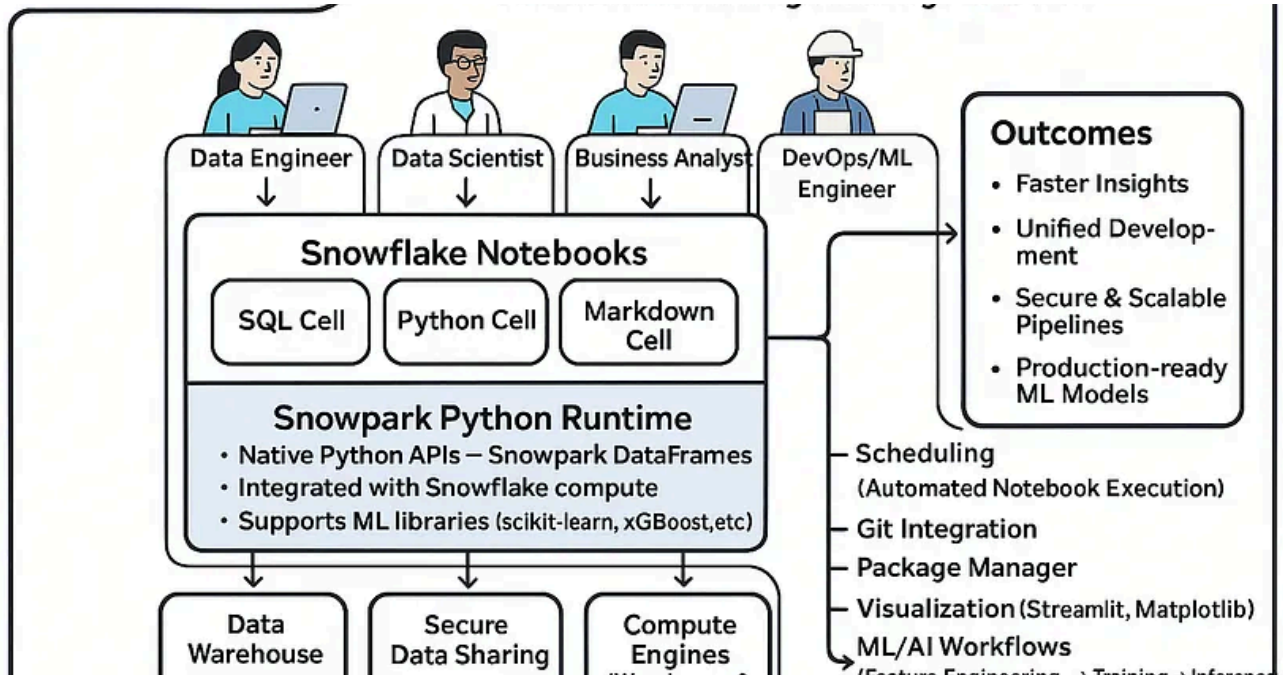


Jeyaram Ayyalusamy

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

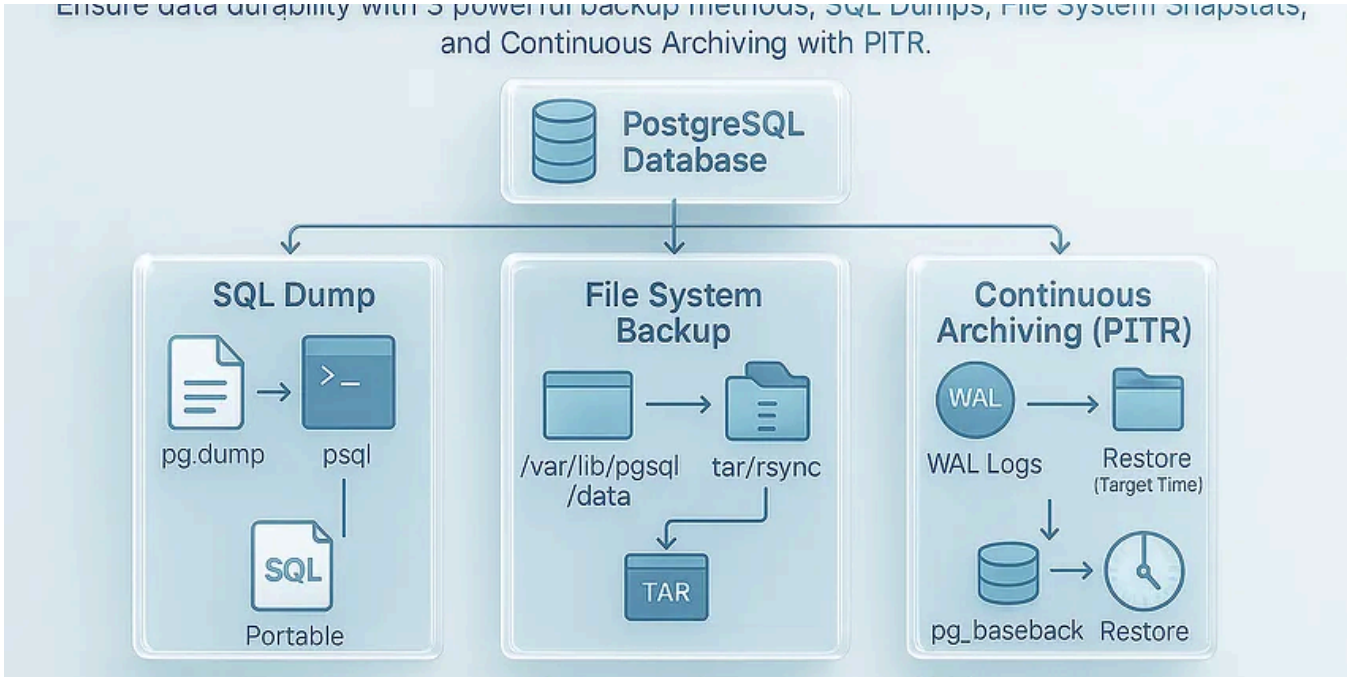


Jeyaram Ayyalusamy

16—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

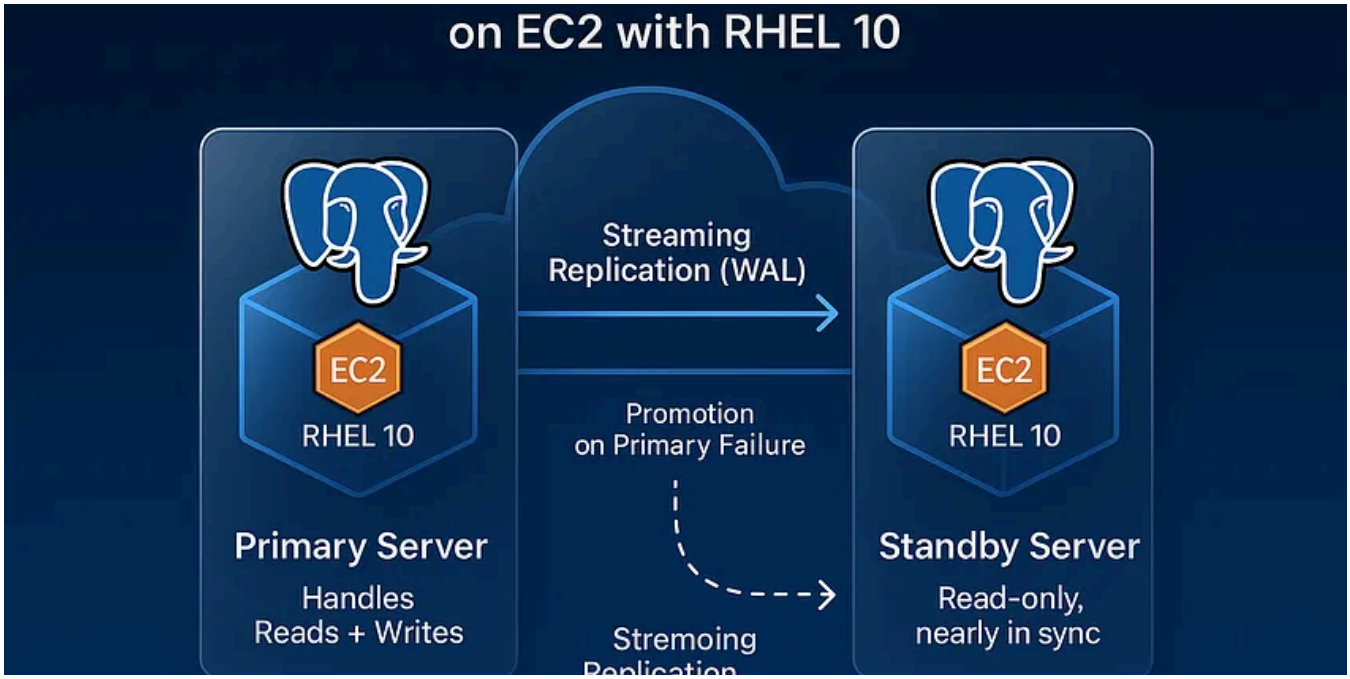


Jeyaram Ayyalusamy

🛡️ 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

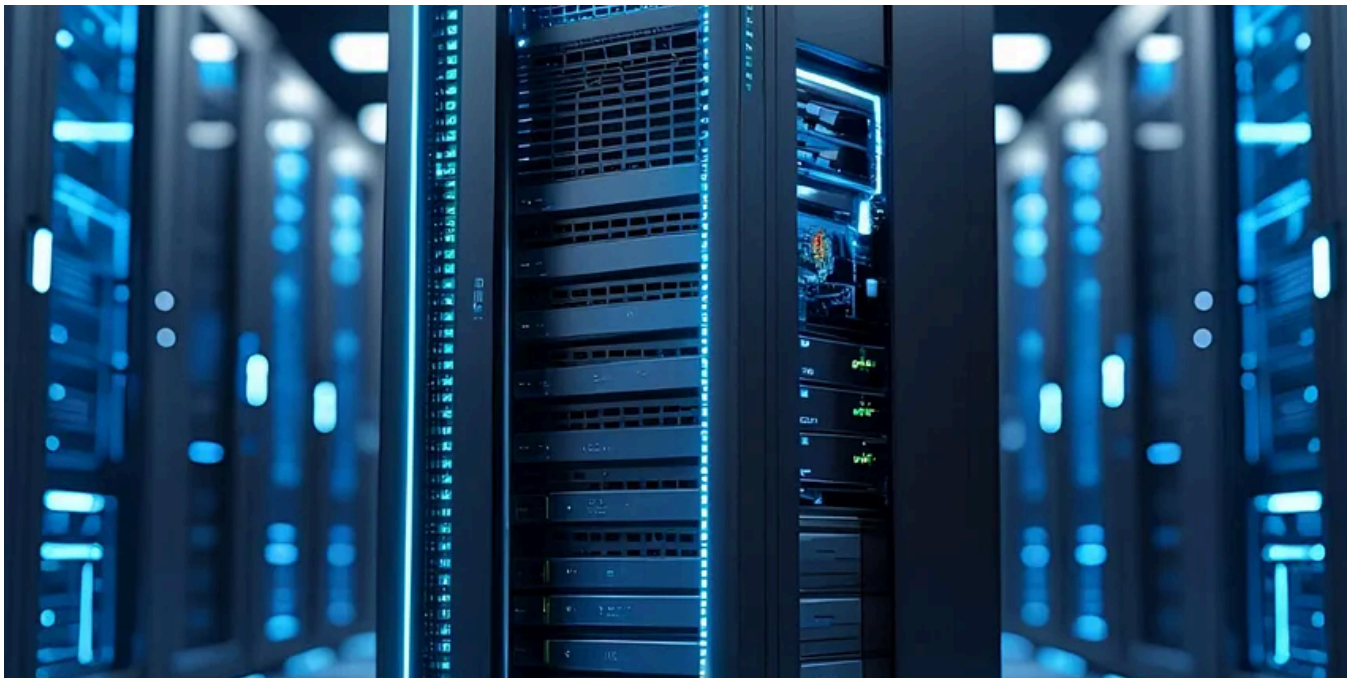
🚀 Streaming Replication in PostgreSQL 17 on EC2 with RHEL 10—A Deep Dive

🐘 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

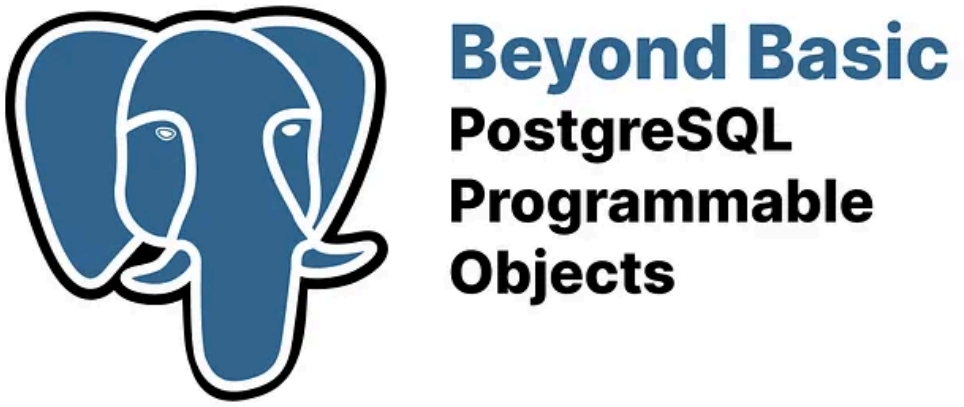
Sep 15 11 1

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

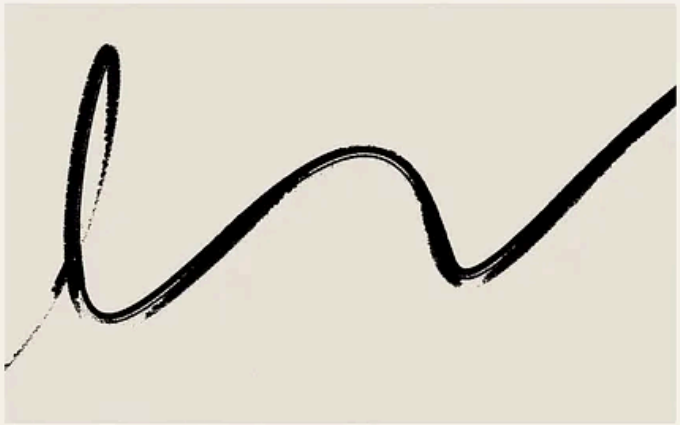


In Stackademic by bektiaw

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

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

5 DuckDB–Arrow–Polars Workflows in Minutes

Turn day-long pipelines into small, local, reproducible runs without clusters or drama.

5d ago 14