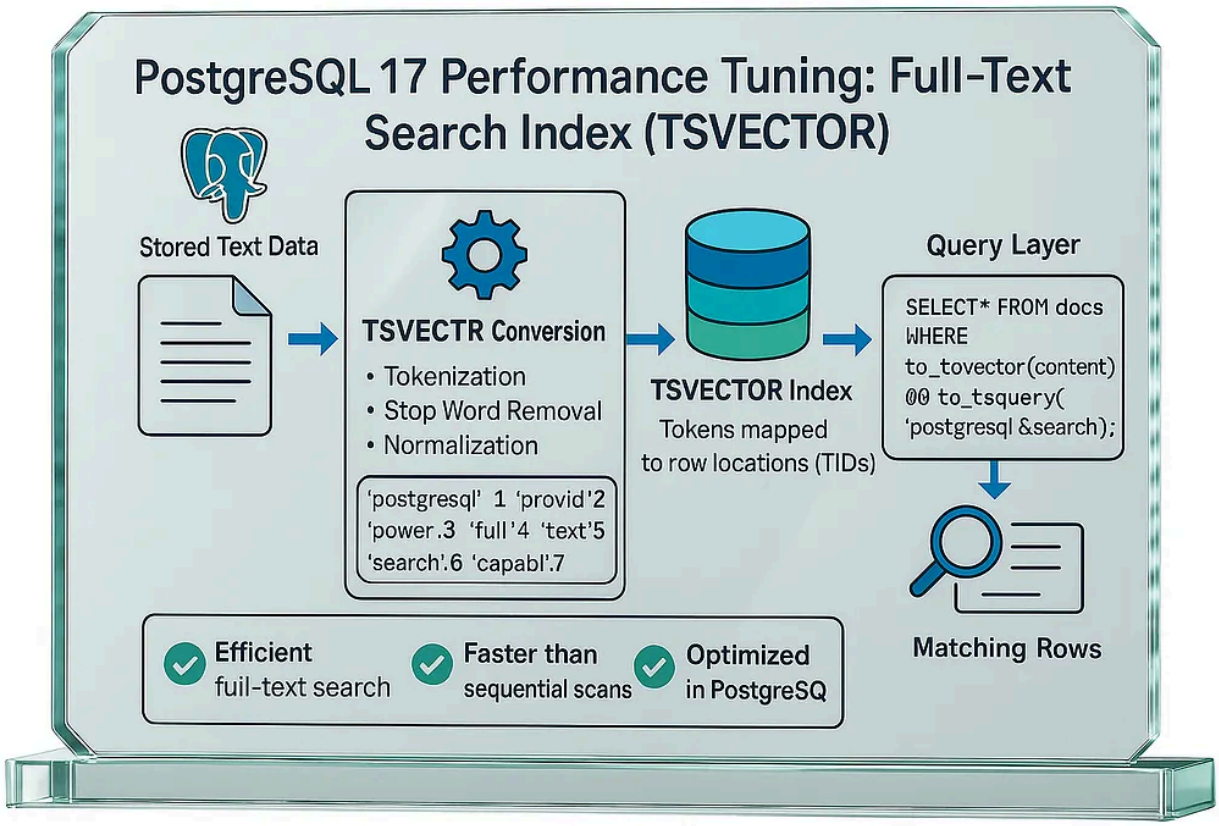
20 - PostgreSQL 17 Performance Tuning: Full-Text Search Index (TSVECTOR

4 min read · Sep 5, 2025

Jeyaram Ayyalusamy Following



A **Full-Text Search Index (TSVECTOR)** in PostgreSQL is a special way of indexing text data so that it can be searched quickly and efficiently. Instead of storing raw text, PostgreSQL converts the text into a `tsvector`, which is a preprocessed form

that breaks the text into tokens (words), removes stop words (like “a”, “the”), and normalizes words to their root form (e.g., “running”, “ran” → “run”).

This `tsvector` is then indexed, usually with a **GIN (Generalized Inverted Index)** or **GiST (Generalized Search Tree)** index, which allows PostgreSQL to match search queries (`tsquery`) against documents at high speed.

By using `tsvector` , PostgreSQL supports full-text search features like:

- Finding documents that contain specific words or phrases.
- Ranking results based on relevance.
- Highlighting matched words in search results.

In short, a Full-Text Search Index with `tsvector` transforms text into a searchable format and, when combined with an index, provides fast and accurate full-text search inside PostgreSQL.

Step 1 — Create the table

```
CREATE TABLE articles (
    id           BIGSERIAL PRIMARY KEY,
    title        TEXT,
    body         TEXT,
    published_at TIMESTAMPTZ DEFAULT now()
);
```

```
postgres=#
postgres=# CREATE TABLE articles (
    id           BIGSERIAL PRIMARY KEY,
    title        TEXT,
    body         TEXT,
```

```
    published_at TIMESTAMPTZ DEFAULT now()
);
CREATE TABLE
postgres=#
```

Step 2 — Load 1,000,000 rows (synthetic corpus)

We'll sprinkle common tech keywords so the search actually matches.

```
INSERT INTO articles (title, body)
SELECT
  'Post ' || g || ' about ' ||
    (ARRAY['database','performance','indexing','postgresql','tuning',
           'sql','json','cache','query','optimizer'])
      [1 + (random()*9)::int] AS title,
  'This article discusses ' ||
    (ARRAY['database','performance tuning','indexing strategies','PostgreSQL 17
           'GIN indexes','full-text search','planning and costing','caching','
      [1 + (random()*9)::int] ||
  ' with practical tips on ' ||
    (ARRAY['tsvector','tsquery','plainto_tsquery','to_tsquery','ranking','highl
      [1 + (random()*5)::int] || '.' AS body
FROM generate_series(1, 1000000) g;
```

```
postgres=# INSERT INTO articles (title, body)
SELECT
  'Post ' || g || ' about ' ||
    (ARRAY['database','performance','indexing','postgresql','tuning',
           'sql','json','cache','query','optimizer'])
      [1 + (random()*9)::int] AS title,
  'This article discusses ' ||
    (ARRAY['database','performance tuning','indexing strategies','PostgreSQL 17
           'GIN indexes','full-text search','planning and costing','caching','
      [1 + (random()*9)::int] ||
  ' with practical tips on ' ||
    (ARRAY['tsvector','tsquery','plainto_tsquery','to_tsquery','ranking','highl
```

```
      [1 + (random()*5)::int] || '.' AS body
FROM generate_series(1, 1000000) g;
```

Step 3 — Analyze statistics

```
ANALYZE articles;
```

```
postgres=# ANALYZE articles;
ANALYZE
postgres=#
```

Step 4 — Baseline query without an index

We'll search for "**database performance**". Computing the vector on the fly forces a full scan.

```
EXPLAIN ANALYZE
SELECT count(*)
FROM articles
WHERE to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,'')) @@
      plainto_tsquery('english','database performance');
```

```
postgres=# EXPLAIN ANALYZE
SELECT count(*)
FROM articles
WHERE to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,'')) @@
      plainto_tsquery('english','database performance');

------------------------------------------------------------------------------
 Finalize Aggregate  (cost=130449.57..130449.58 rows=1 width=8) (actual time=13
   ->  Gather  (cost=130449.36..130449.57 rows=2 width=8) (actual time=13184.60
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=129449.36..129449.37 rows=1 width=8) (act
               ->  Parallel Seq Scan on articles  (cost=0.00..129449.33 rows=10
                     Filter: (to_tsvector('english'::regconfig, ((COALESCE(titl
                     Rows Removed by Filter: 329211
 Planning Time: 0.090 ms
 Execution Time: 13191.826 ms
(10 rows)
```

```
postgres=#
```

```
Time: 13191.826 ms (00:13.826)
postgres=#
```

> With no index, PostgreSQL scans all 1M rows (~*13.826* s).

Step 5 — Add a generated `tsvector` column (stored)

This keeps document vectors always up-to-date without triggers.

```
ALTER TABLE articles
ADD COLUMN tsv tsvector
GENERATED ALWAYS AS (
```

```
  to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''))
) STORED;
```

```
postgres=# postgres=# ALTER TABLE articles
ADD COLUMN tsv tsvector
GENERATED ALWAYS AS (
  to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''))
) STORED;
ALTER TABLE
postgres=#
```

Step 6 — Create a GIN index on the `tsvector`

```
CREATE INDEX idx_articles_tsv_gin ON articles USING gin (tsv);
```

```
postgres=# CREATE INDEX idx_articles_tsv_gin ON articles USING gin (tsv);
CREATE INDEX
postgres=#
```

(If building online in production, use `CREATE INDEX CONCURRENTLY`.)

Analyze statistics

```
ANALYZE articles;
```

```
postgres=# ANALYZE articles;
ANALYZE
postgres=#
```

Step 7 — Search again (now using the index)

```
postgres=# EXPLAIN ANALYZE
SELECT count(*)
FROM articles
WHERE tsv @@ plainto_tsquery('english','database performance');
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=34497.68..34497.69 rows=1 width=8) (actual time=39.425..39.42
   ->  Bitmap Heap Scan on articles  (cost=190.83..34440.90 rows=22714 width=0)
         Recheck Cond: (tsv @@ '''databas'' & ''perform'''::tsquery)
         Heap Blocks: exact=10602
         ->  Bitmap Index Scan on idx_articles_tsv_gin  (cost=0.00..185.15 rows
               Index Cond: (tsv @@ '''databas'' & ''perform'''::tsquery)
 Planning Time: 0.101 ms
 Execution Time: 39.453 ms
(8 rows)
```

```
postgres=#
```

```
Time: 39.453 ms
postgres=#
```

> GIN flips the plan to **Bitmap Index Scan**, dropping runtime from ***Time reduction:*** *(13,191.826 − 39.453) =* ***13,152.373 ms*** *saved (≈* ***99.701%*** *reduction)*

📢 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 Open Source Mongodb MySQL

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

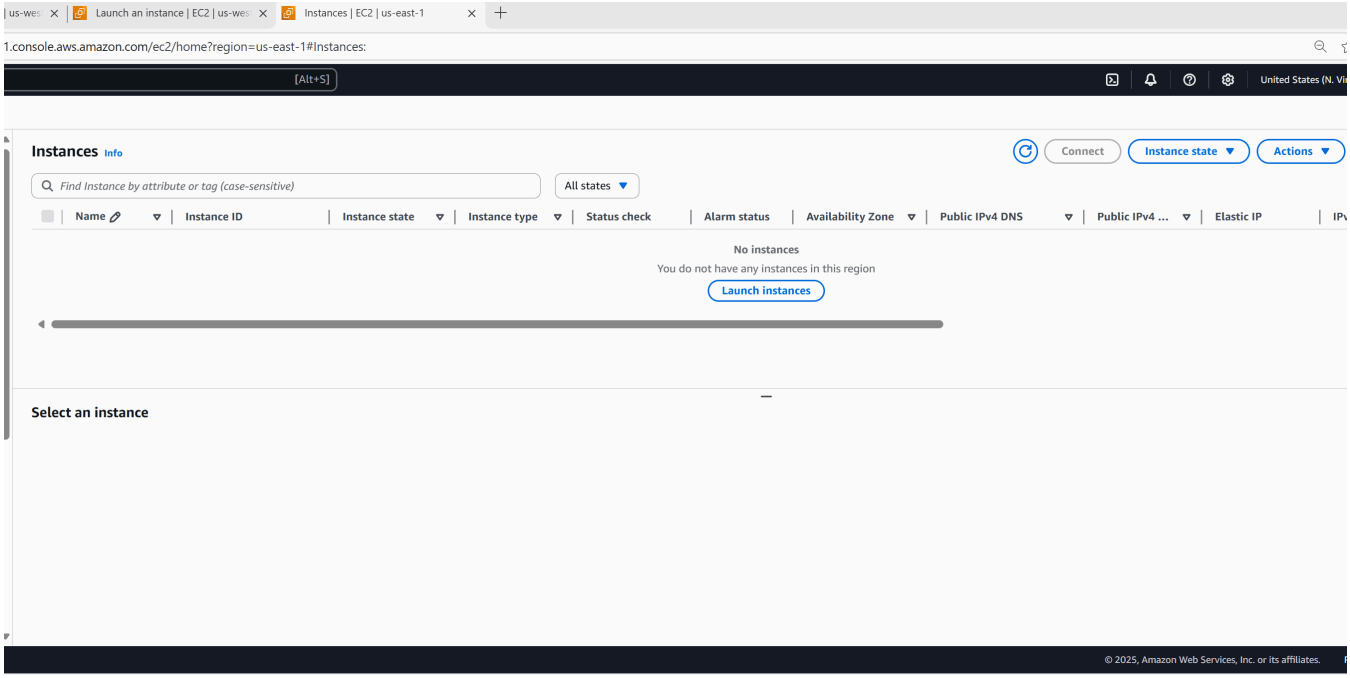
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

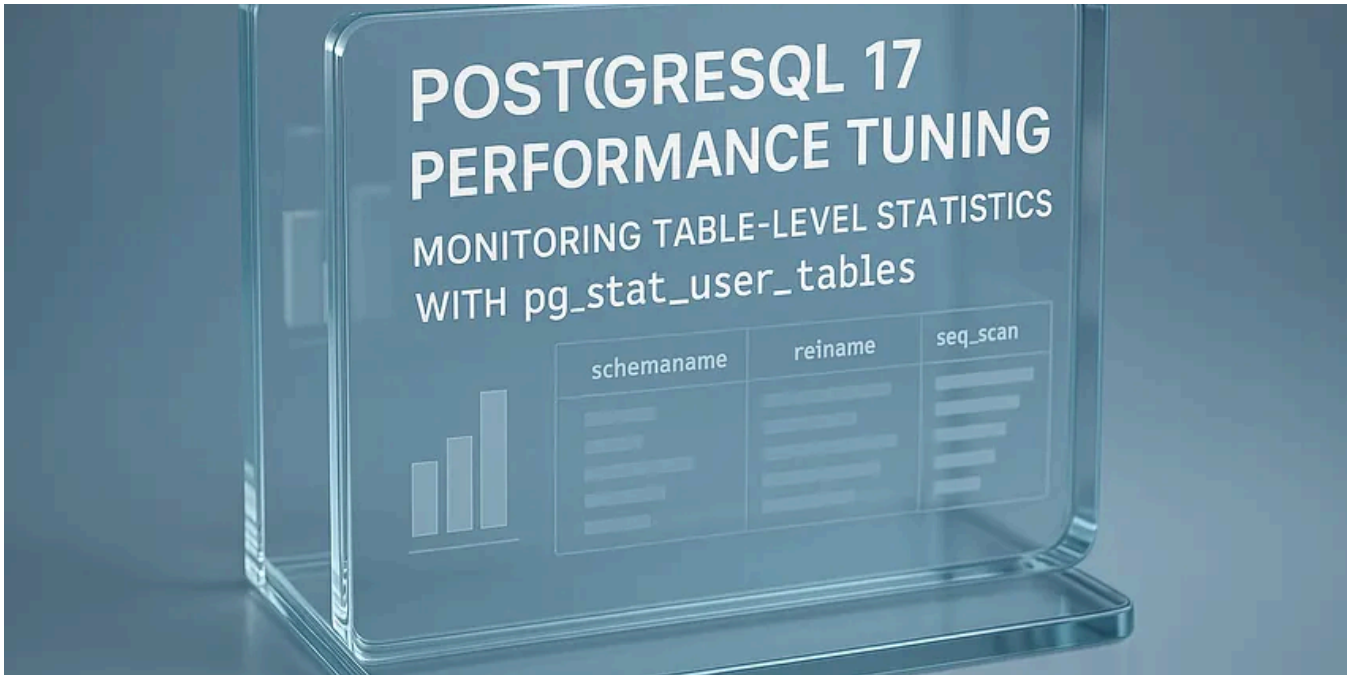


Jeyaram Ayyalusamy

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

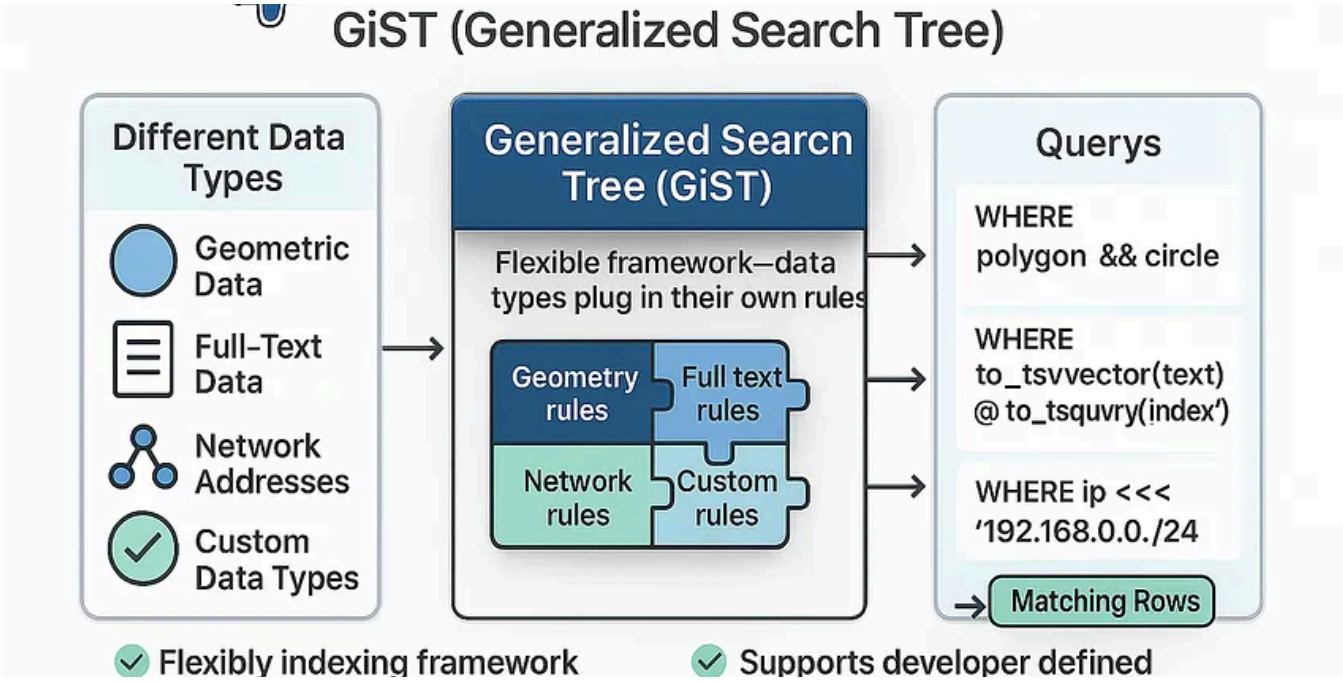


Jeyaram Ayyalusamy

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

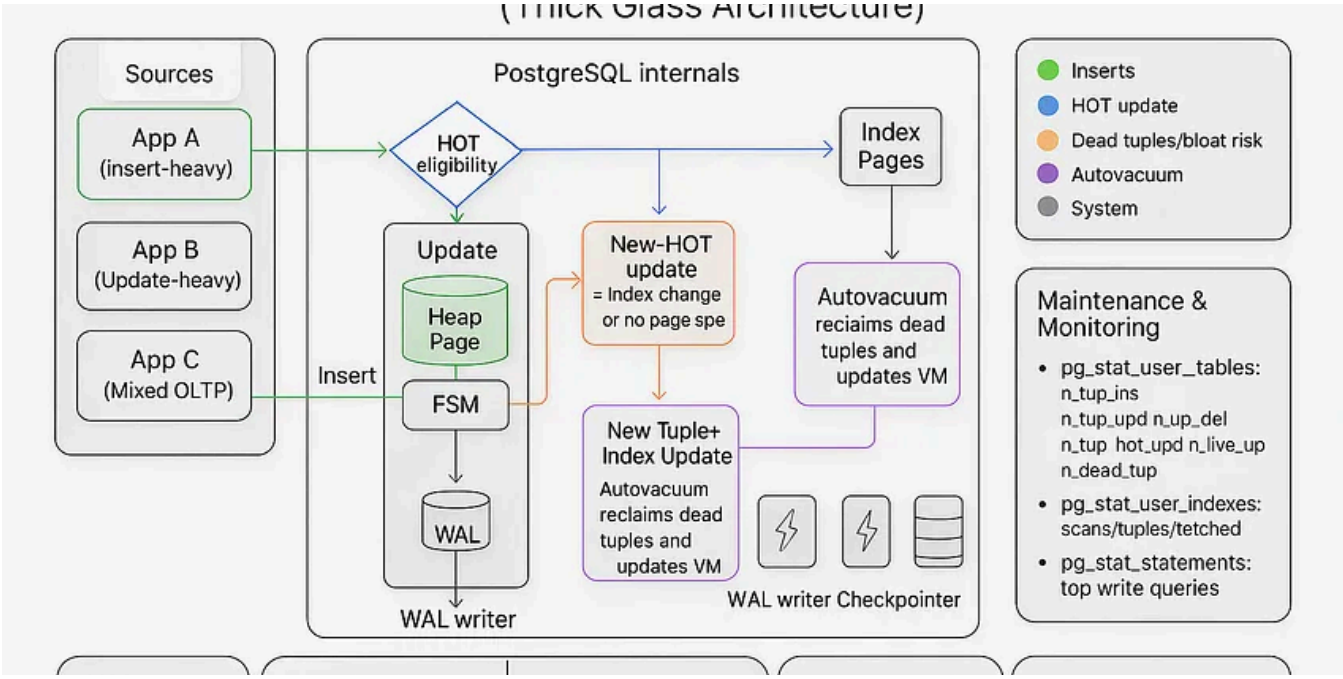


Jeyaram Ayyalusamy

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

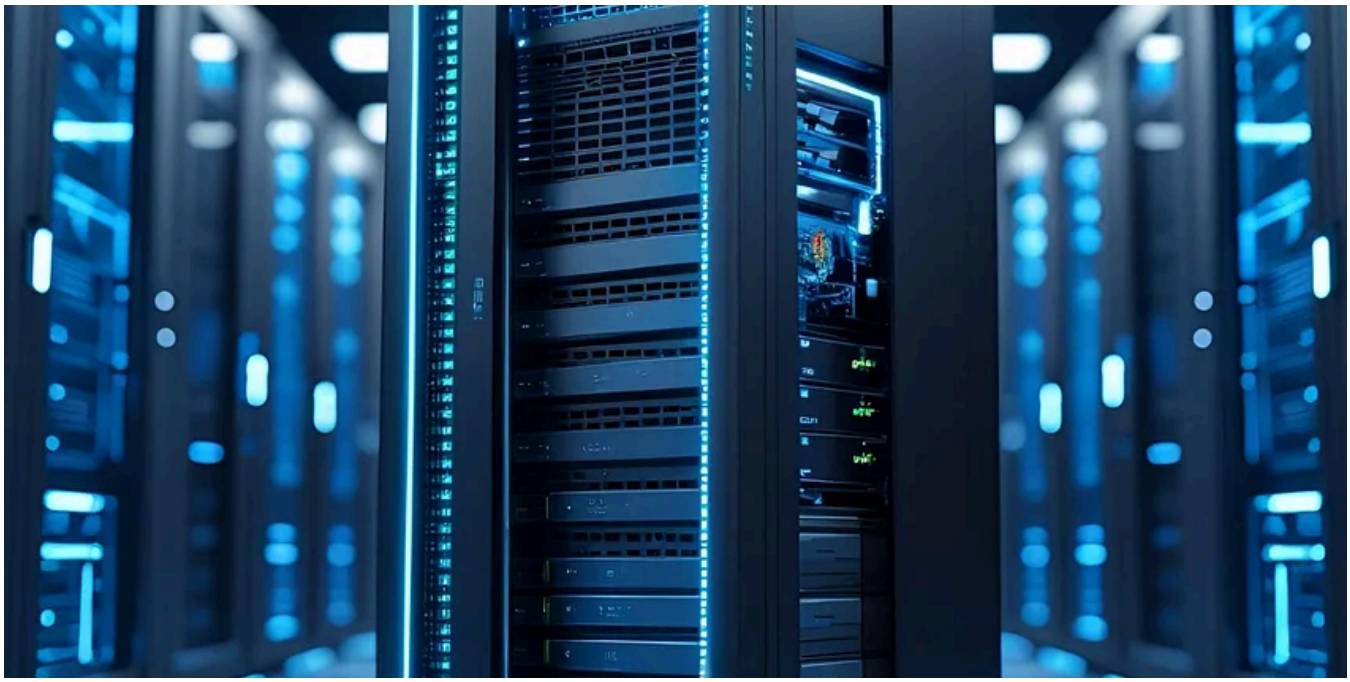
23 - PostgreSQL 17 Performance Tuning: Monitoring Inserts, Updates, and HOT Updates

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

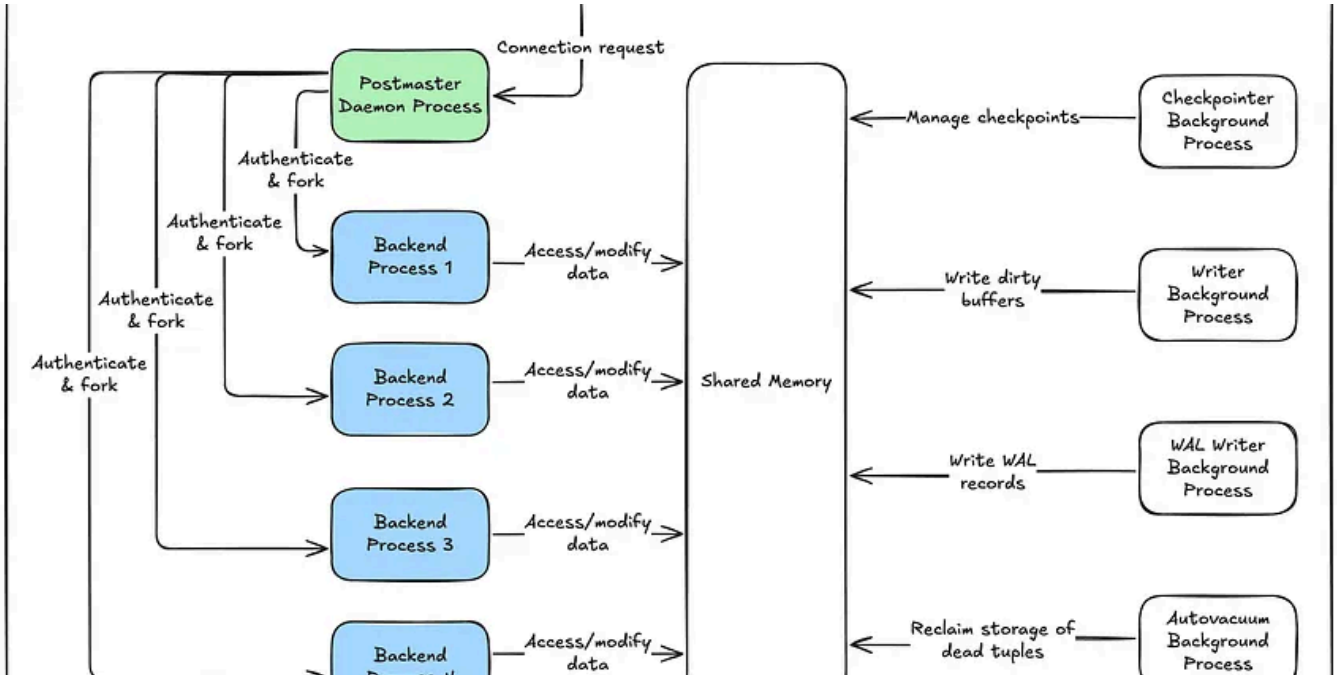


Rizqi Mulki

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

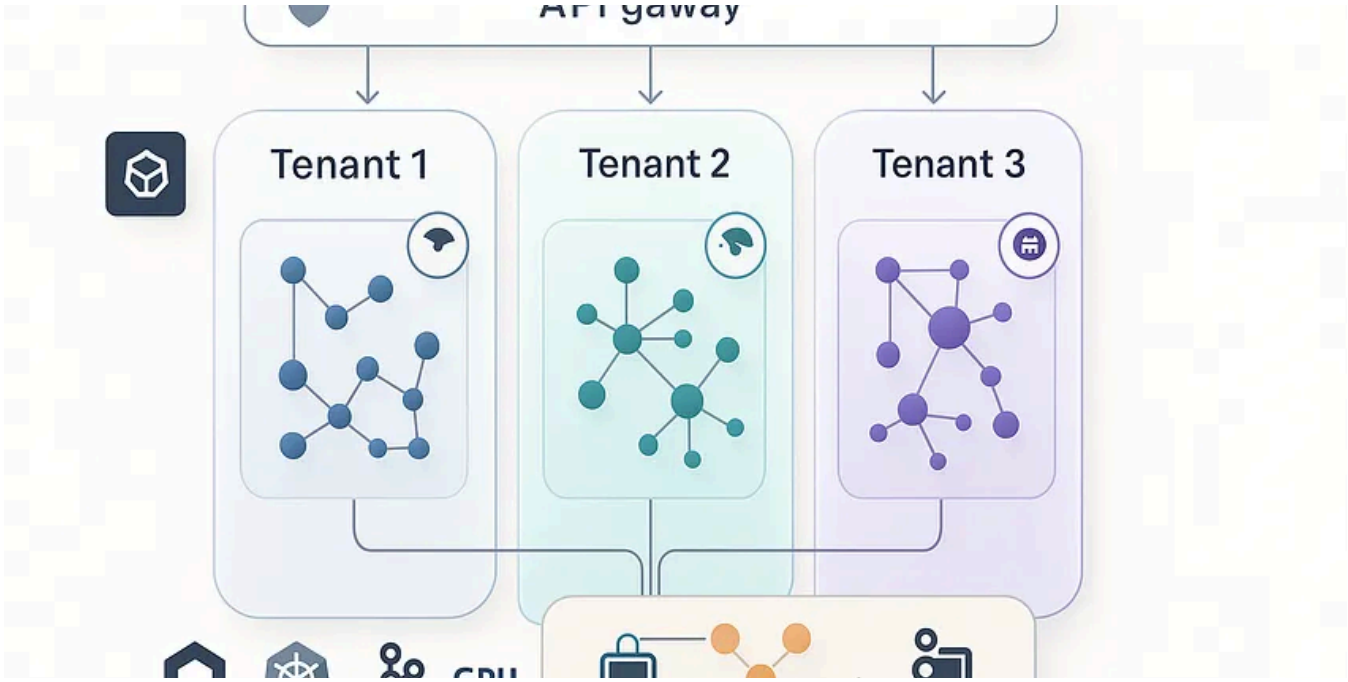


Kareem Mohllal

Database Connection Pooling with PgBouncer

Database connections are expensive, let's explore why and how PgBouncer can save your database from drowning in connections.

Aug 5 10

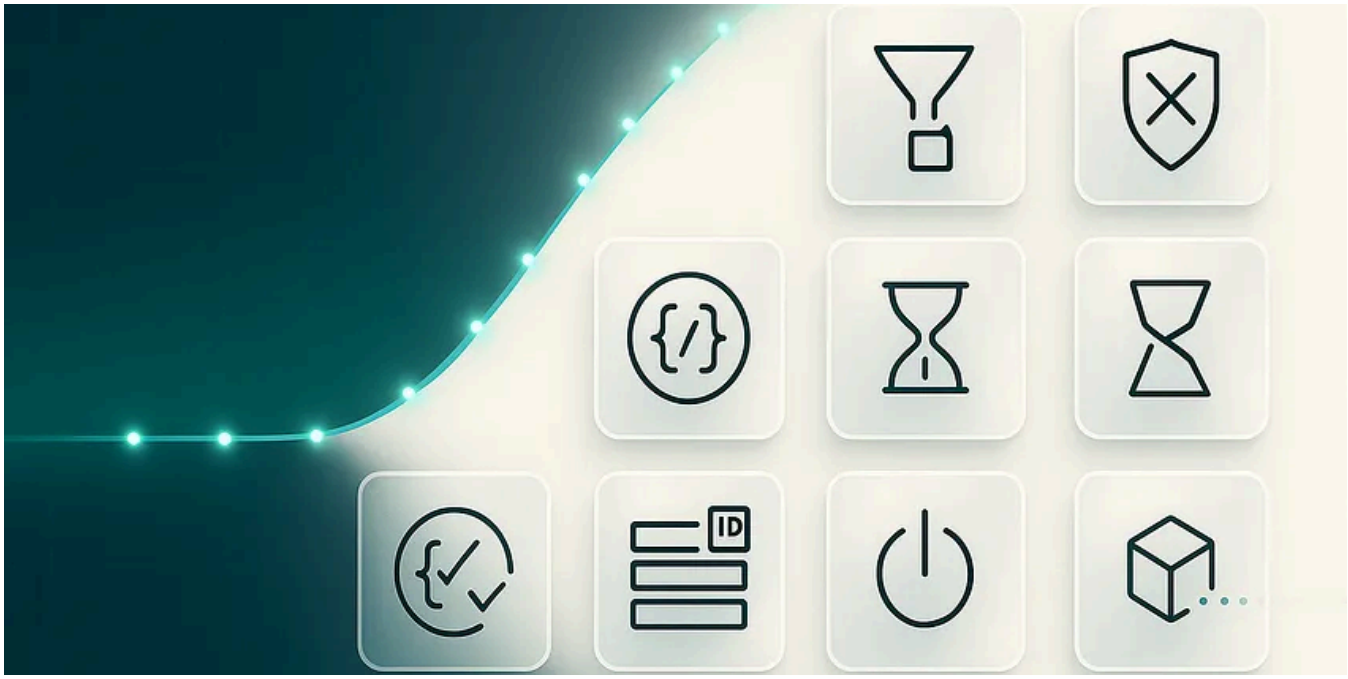


Thinking Loop

5 Ironclad Rules for Multi-Tenant Vector Isolation

Keep noisy neighbors out and SLAs intact—without overprovisioning your vector cluster.

Sep 15 17

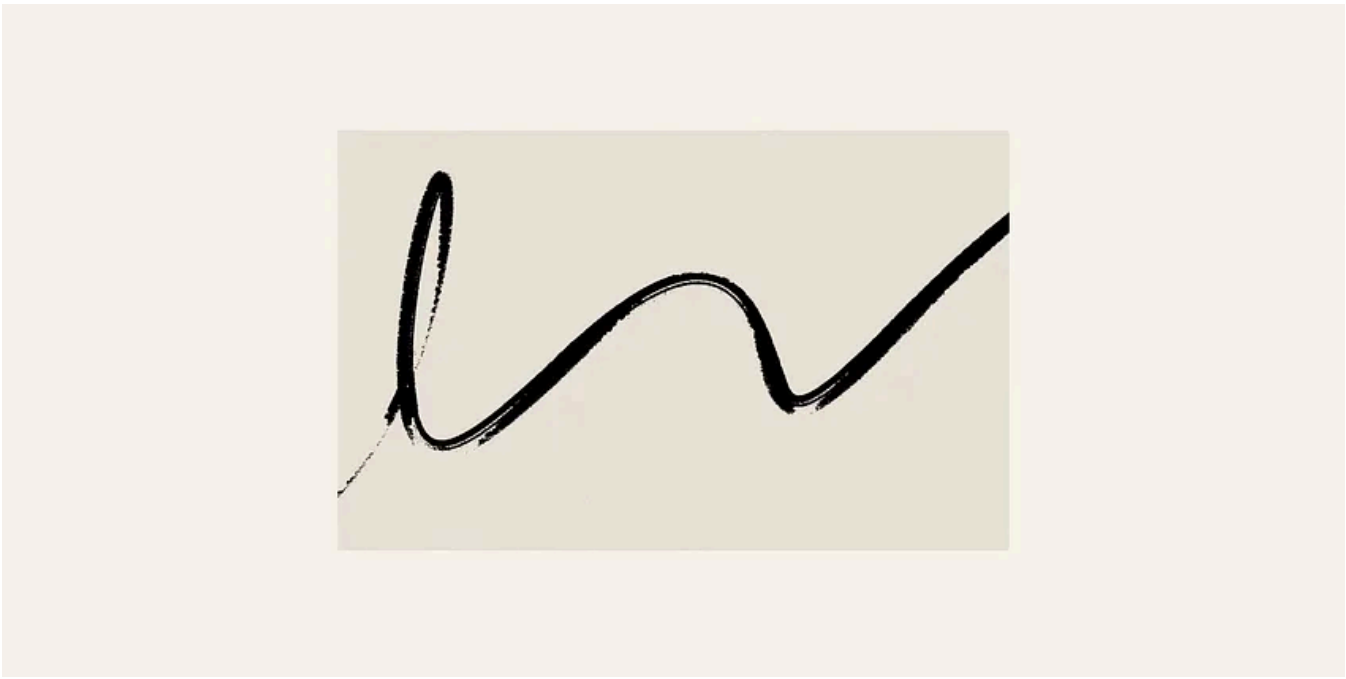


Hash Block

10 Node.js Error-Handling Tricks That Saved Me

Practical patterns to catch bugs early, keep services alive, and turn chaos into readable logs.

4d ago 17



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

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

See more recommendations