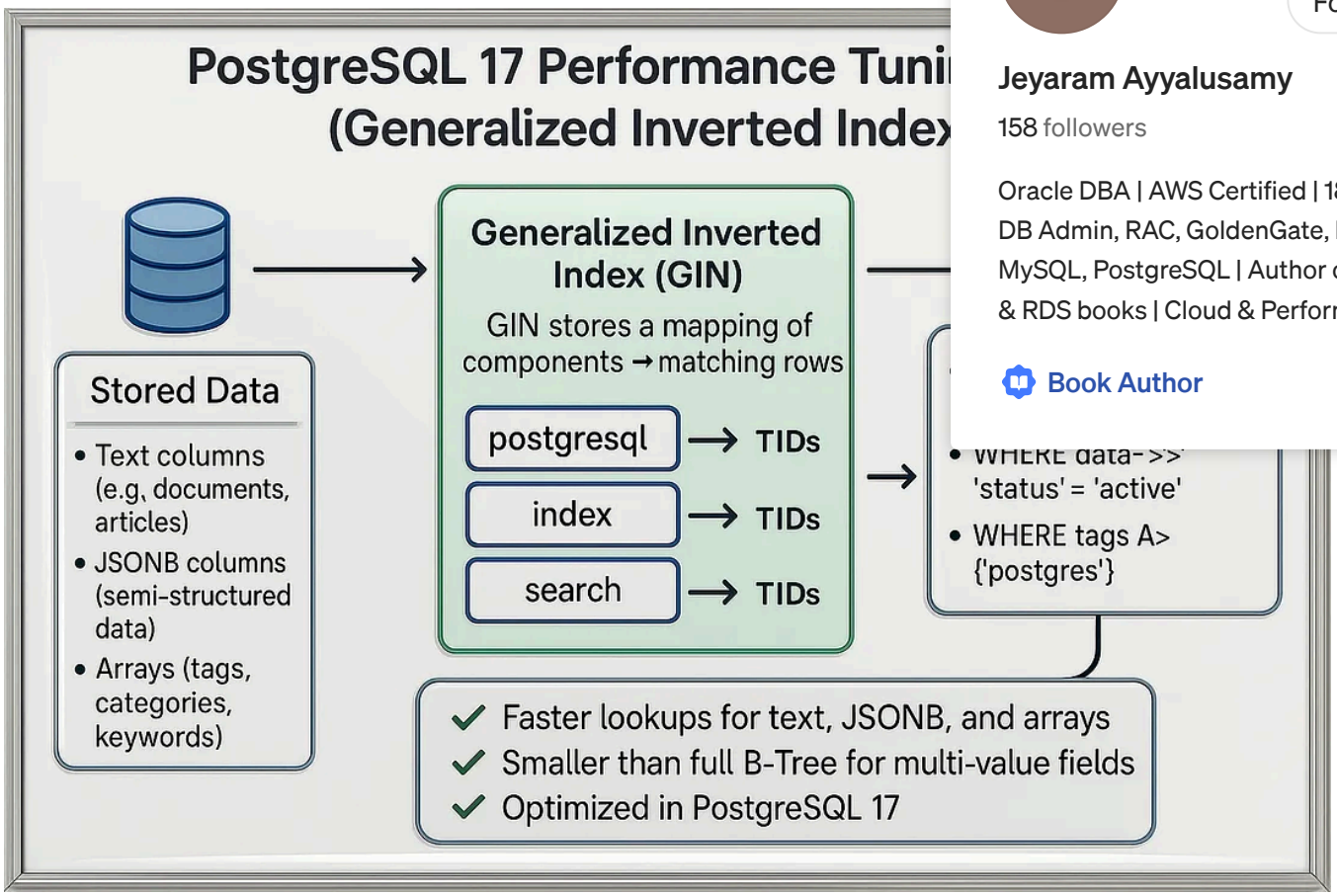
19 - PostgreSQL 17 Performance Tuning: GIN (Generalized Inverted Index)

5 min read · Sep 4, 2025

Jeyaram Ayyalusamy Following



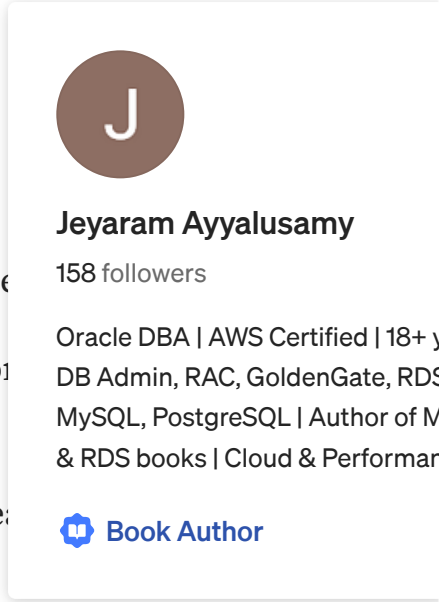
PostgreSQL is famous for its indexing capabilities. Beyond the common **B-Tree index**, PostgreSQL supports powerful index types for specialized workloads. One of the most important is the **GIN (Generalized Inverted Index)**.

What is GIN in PostgreSQL?

GIN (Generalized Inverted Index) is a special type of index in PostgreSQL designed to handle complex data types where a single column can contain **multiple values.** Instead of storing data in a sorted order like a B-Tree, a GIN index works by creating an **inverted map** — it records *which rows contain a particular value.*

This makes it ideal for searching inside:

- **Arrays** → e.g., find all employees who have `'PostgreSQL'` in their skills array.
- **JSONB documents** → e.g., find all records where `"status": "active"`.
- **Full-text search** → e.g., find all articles containing the word `"database"`.



How GIN Works

Think of a GIN index like the index at the back of a book:

- For each keyword, it lists the pages where that word appe
- Similarly, for each value in an array, JSONB document, o
  the rows where that value exists.
- When you query, PostgreSQL just looks up the map inste
  whole table.

Key Features of GIN

- **Optimized for containment checks (`@>`, `?`, `@@`)**
  Example: `skills @> ARRAY['PostgreSQL']` → fast with a GIN index.
- **Great for multi-valued fields**
  Handles arrays, JSONB, full-text documents.
- **Not useful for range queries**
  Unlike B-Tree, GIN indexes can't efficiently process `<`, `>`, `BETWEEN`.

- **Larger and slower to update**
  GIN indexes take more storage and can be slower on inserts/updates compared to B-Tree. But they give huge speedups for searches.

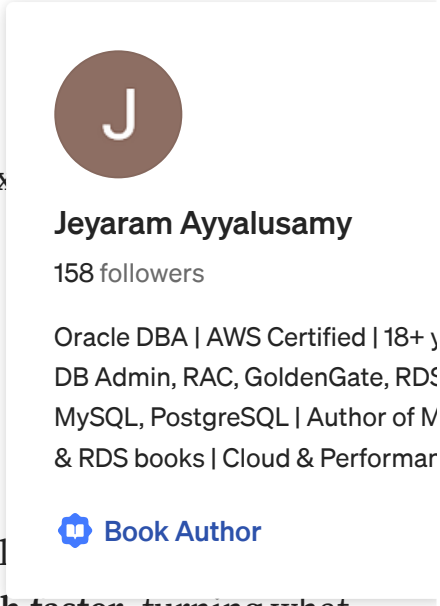
When to Use GIN

✅ Use when:

- You frequently query inside arrays, JSONB documents, or text fields.
- Your queries use operators like `@>`, `?`, or full-text search.

❌ Avoid when:

- You need range queries (`<`, `>`, `BETWEEN`).
- You need frequent updates on large datasets where index
  high.



👉 In short:
A **GIN index in PostgreSQL** is like a lookup map for multi-val
data. It makes queries on arrays, JSONB, and text fields **much faster**, turning what would be full table scans into quick index lookups.

Step 1: Creating the employees table

Let's create a table where each employee has a list of skills. Instead of storing skills as separate rows, we'll store them in an **array column**.

```
CREATE TABLE employees (
    emp_id   BIGINT,
    emp_name TEXT,
```

```
    skills   TEXT[]        -- array of skills for each employee
);
```

```
postgres=#
postgres=# CREATE TABLE employees (
    emp_id   BIGINT,
    emp_name TEXT,
    skills   TEXT[]        -- array of skills for each employee
);
CREATE TABLE
postgres=#
```

📌 Why arrays?
Arrays are useful when each entity (like an employee) can ha
single column (like skills). This design avoids join overhead l
indexing for fast searches.

Step 2: Insert 1 million rows with random skills

We'll load one million rows where each employee has at least one skill, randomly assigned.

```
-- Insert 1M rows with random skills
INSERT INTO employees (emp_name, skills)
SELECT
  'Employee_' || g,
  ARRAY[
    (ARRAY['SQL','Python','PostgreSQL','Java','C++','Tableau',
           'AWS','Docker','Kubernetes','PowerBI'])
      [1 + (random()*9)::int]
  ]
FROM generate_series(1, 1000000) g;
```

```
postgres=# -- Insert 1M rows with random skills
INSERT INTO employees (emp_name, skills)
SELECT
  'Employee_' || g,
  ARRAY[
    (ARRAY['SQL','Python','PostgreSQL','Java','C++','Tableau',
           'AWS','Docker','Kubernetes','PowerBI'])
      [1 + (random()*9)::int]
  ]
FROM generate_series(1, 1000000) g;
INSERT 0 1000000
postgres=#
```

✅ Now we have:

- 1 million employees
- Each with one randomly assigned skill
- A realistic dataset for testing queries

Step 3: Analyze the table

PostgreSQL relies on statistics to choose query plans. Running `ANALYZE` ensures the planner has up-to-date information.

```
ANALYZE employees;
```

```
postgres=# ANALYZE employees;
ANALYZE
postgres=#
```

Step 4: Query without an index

Suppose we want to find all employees who know **PostgreSQL**:

```
EXPLAIN ANALYZE
SELECT * FROM employees WHERE skills @> ARRAY['PostgreS
```



```
postgres=# EXPLAIN ANALYZE
SELECT * FROM employees WHERE skills @> ARRAY['PostgreS
                                                QU
------------------------------------------------------------------------------
 Seq Scan on employees  (cost=0.00..22413.00 rows=115367 width=55) (actual time
   Filter: (skills @> '{PostgreSQL}'::text[])
   Rows Removed by Filter: 888294
 Planning Time: 0.084 ms
 Execution Time: 311.761 ms
(5 rows)

postgres=#
```

🔎 What happened:

- PostgreSQL did a **sequential scan**, checking every row.
- Almost **1 million rows were examined**, with ~99,873 matches.

- Execution took ~**311.761 ms** — too slow for large-scale workloads.

Step 5: Create a GIN index

Now let's add a GIN index on the `skills` array column:

```
CREATE INDEX idx_employees_skills_gin ON employees USING gin (skills);
```

```
postgres=# CREATE INDEX idx_employees_skills_gin ON emp
CREATE INDEX
postgres=#
```


📌 Why GIN?

- A **B-Tree index** is not suitable for array membership che
- A **GIN index** stores an **inverted map**: it knows which rows contain each skill.
- This makes queries like `skills @> ARRAY['PostgreSQL']` lightning fast.

Step 6: Query with the GIN index

Re-run the same query:

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM employees WHERE skills @> ARRAY['PostgreSQL'];
```

📊 Query Plan (with GIN index):

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM employees WHERE skills @> ARRAY['PostgreSQL'];
                                                              QUERY PLAN
-----------------------------------------------------------------------------------
 Bitmap Heap Scan on employees  (cost=783.95..12130.70 rows=114700 width=55) (a
   Recheck Cond: (skills @> '{PostgreSQL}'::text[])
   Heap Blocks: exact=9913
   ->  Bitmap Index Scan on idx_employees_skills_gin  (cost=0.00..755.27 rows=1
         Index Cond: (skills @> '{PostgreSQL}'::text[])
 Planning Time: 0.068 ms
 Execution Time: 169.715 ms
(7 rows)

postgres=#
```

🔥 Performance boost:

- The sequential scan was replaced with a **Bitmap Index Sc**
  index.
- Instead of checking every row, PostgreSQL jumped straig
- Execution dropped from **311.761 ms to ~169.715 ms.**

Why GIN Indexes Matter

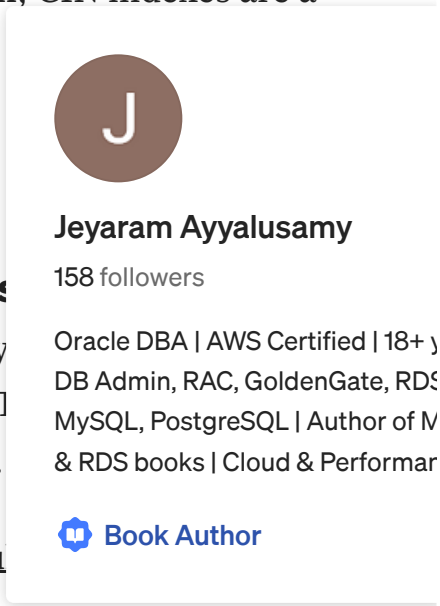
- **Designed for complex data:** GIN indexes make it possible to search **inside arrays, JSONB documents, and full-text search fields.**
- **Highly optimized:** Instead of storing full rows, they store a mapping of “value → row IDs.”
- **Use cases:**
- Arrays: Find employees with a specific skill.
- JSONB: Find documents containing a specific key or value.

- Full-text: Fast search across large text fields.

Key Takeaways

- Without indexes, PostgreSQL falls back to **sequential scans** — very expensive on large datasets.
- GIN indexes make **membership queries** (`@>` , `?` , `@@` , etc.) extremely fast.
- In PostgreSQL 17, GIN indexes are fully production-ready and widely used in modern apps handling **semi-structured data.**

👉 If your application uses **arrays, JSONB, or full-text search**, GIN indexes are a must-have for performance tuning.



📢 Stay Updated with Daily PostgreSQL & Cloud Tips

If you've been finding my blog posts helpful and want to stay
insights on PostgreSQL, Cloud Infrastructure, Performance T
Practices — I invite you to subscribe to my Medium account.

🔔 Subscribe here 👉 https://medium.com/@jramcloud1/su

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?

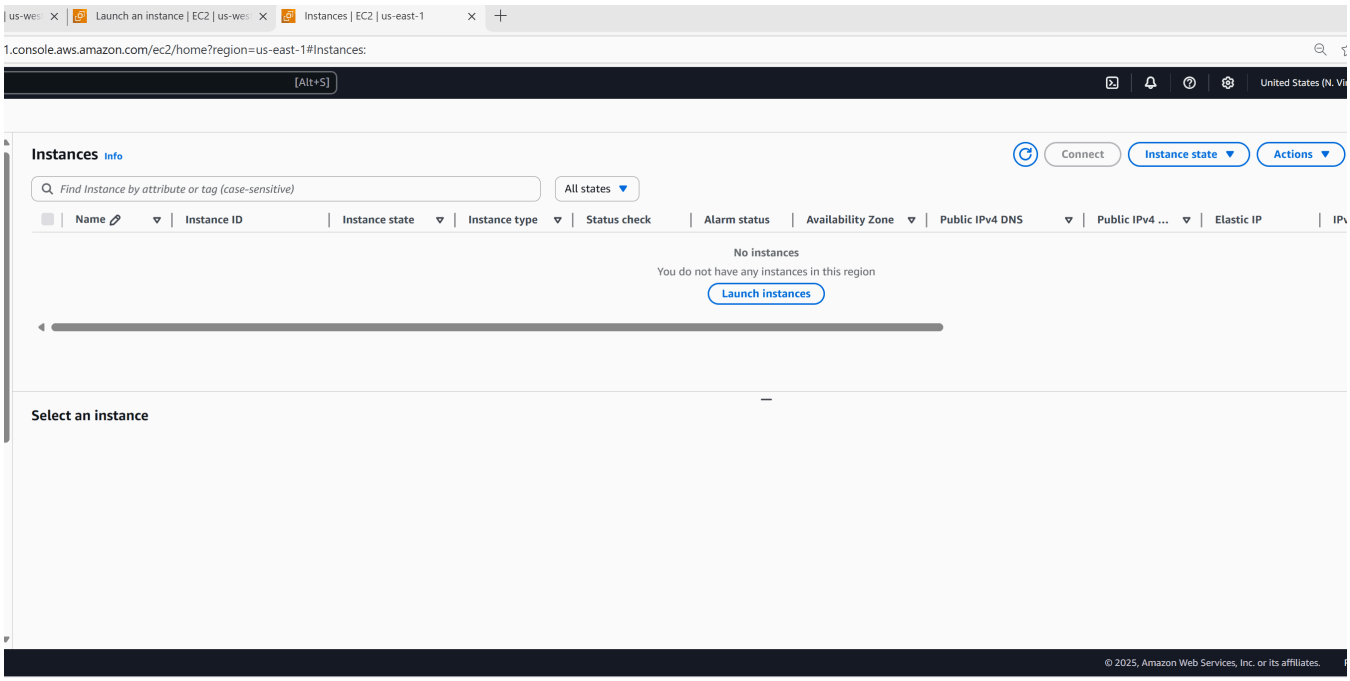
Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

Book Author

More from Jeyaram Ayyalusamy



Jeyaram Ayyalusamy

Upgrading PostgreSQL from Version 16 to Version 17 on a Linux Server AWS EC2...

🛠️ A Complete Step-by-Step Guide to Installing PostgreSQL 16 on a Explanations)

Aug 4 👏 40

J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

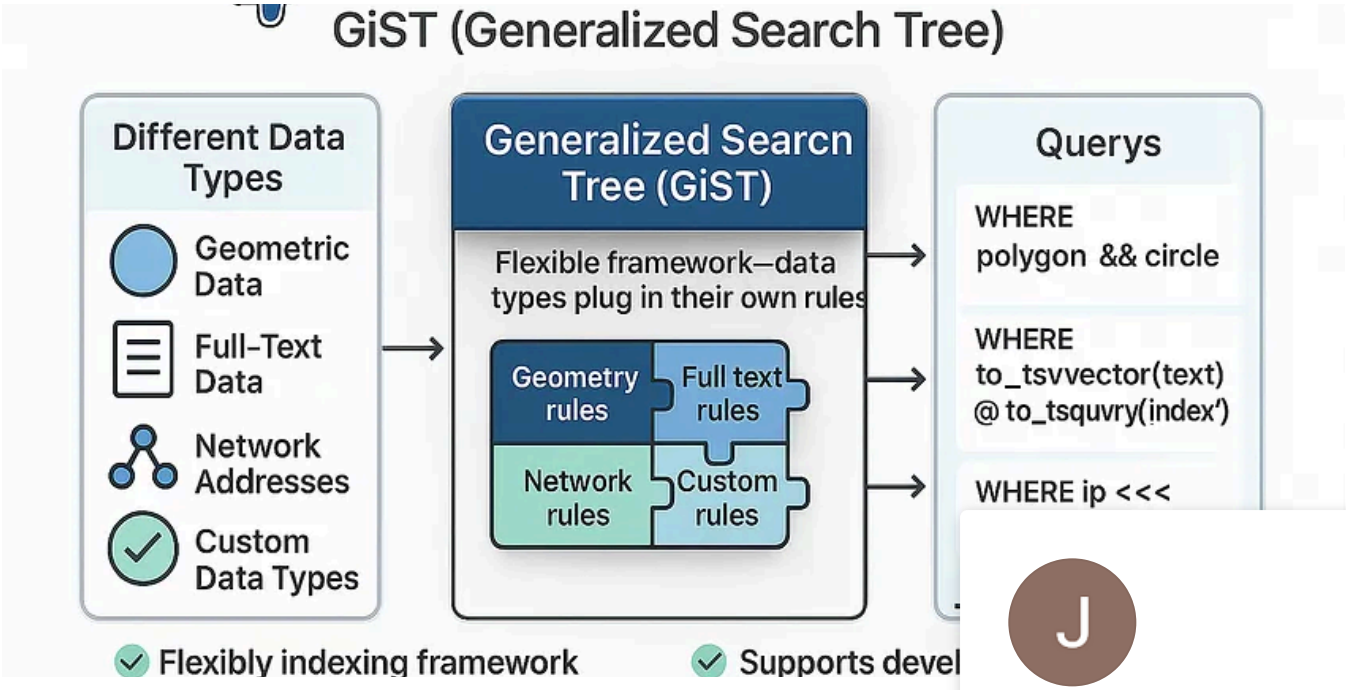
Book Author

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 (Gene

GiST (Generalized Search Tree) in PostgreSQL is a flexible indexing f developers to build indexes for many different...

Sep 5 1

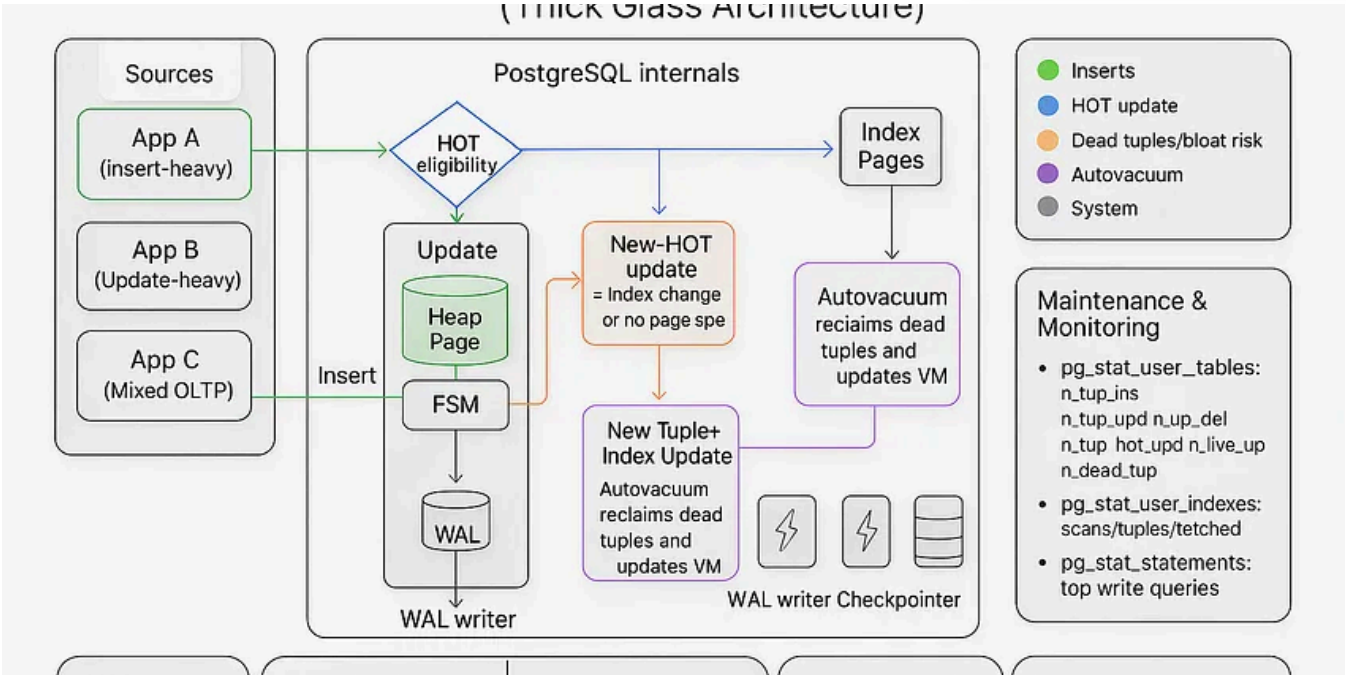
J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y DB Admin, RAC, GoldenGate, RDS MySQL, PostgreSQL | Author of M & RDS books | Cloud & Performar

Book Author



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

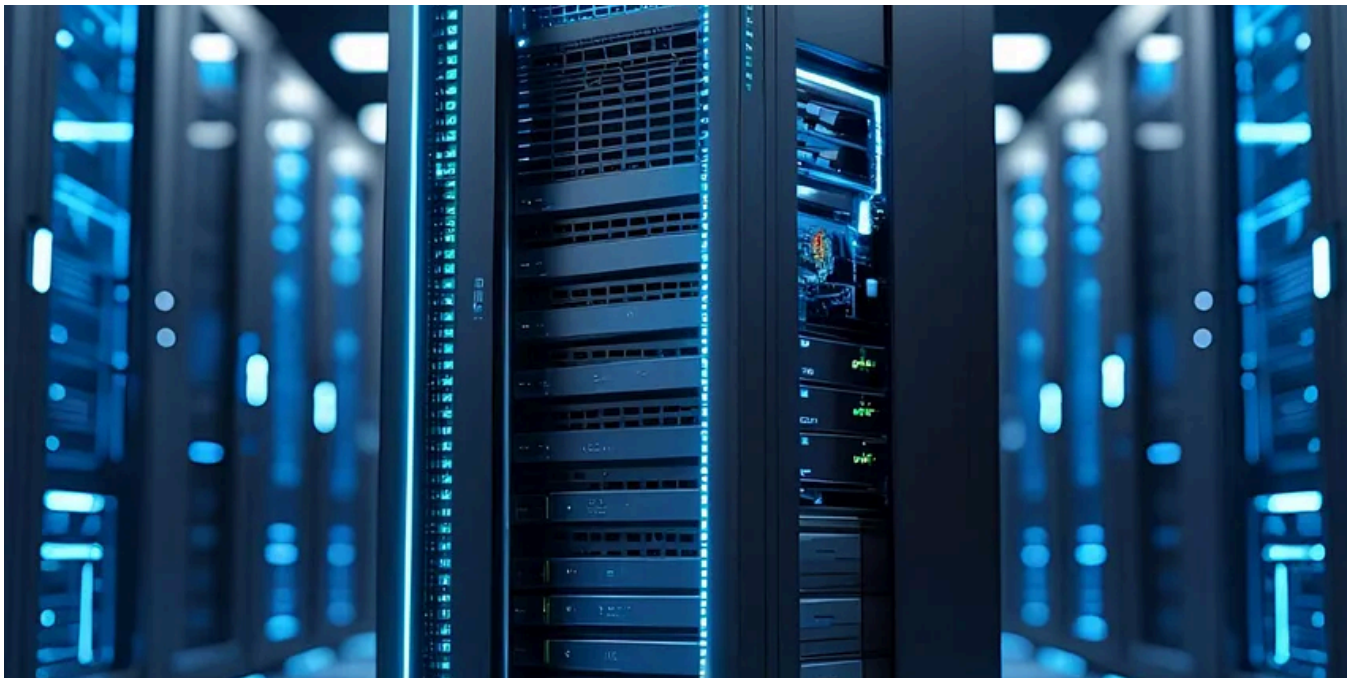
Recommended from Medium

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

Book Author



Rizqi Mulki

PostgreSQL Index Bloat: The Silent Killer of Performa

How a 10TB database became 3x faster by fixing the invisible proble
production PostgreSQL deployments

Sep 15 11 1

J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

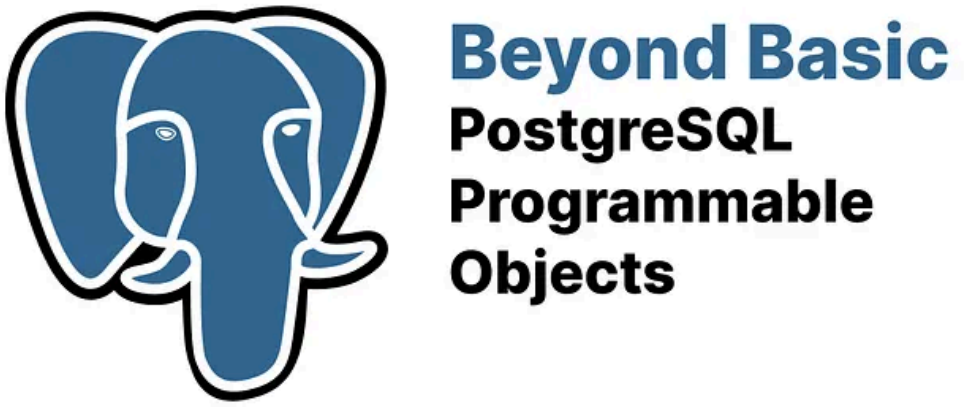
Book Author

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 (
Optimize, Control)

Tired of repeating the same SQL logic? Learn how PostgreSQL can c

Sep 1 68 1

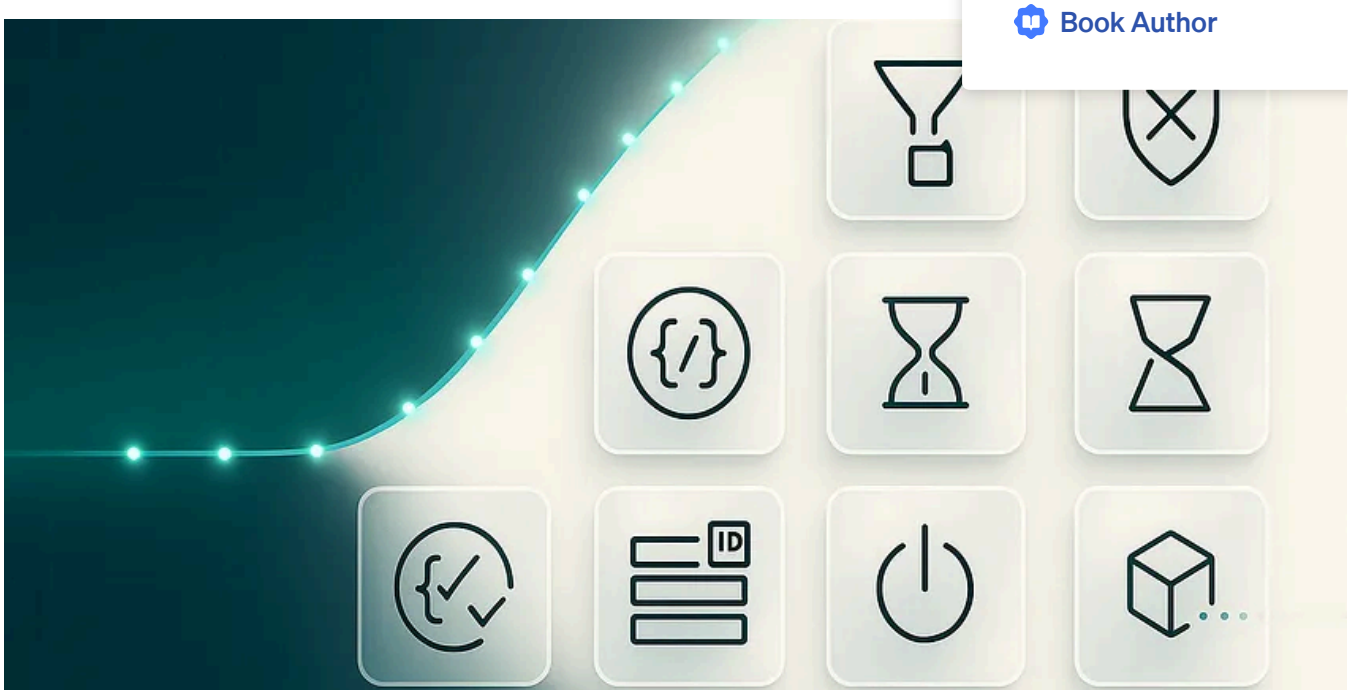
J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

Book Author

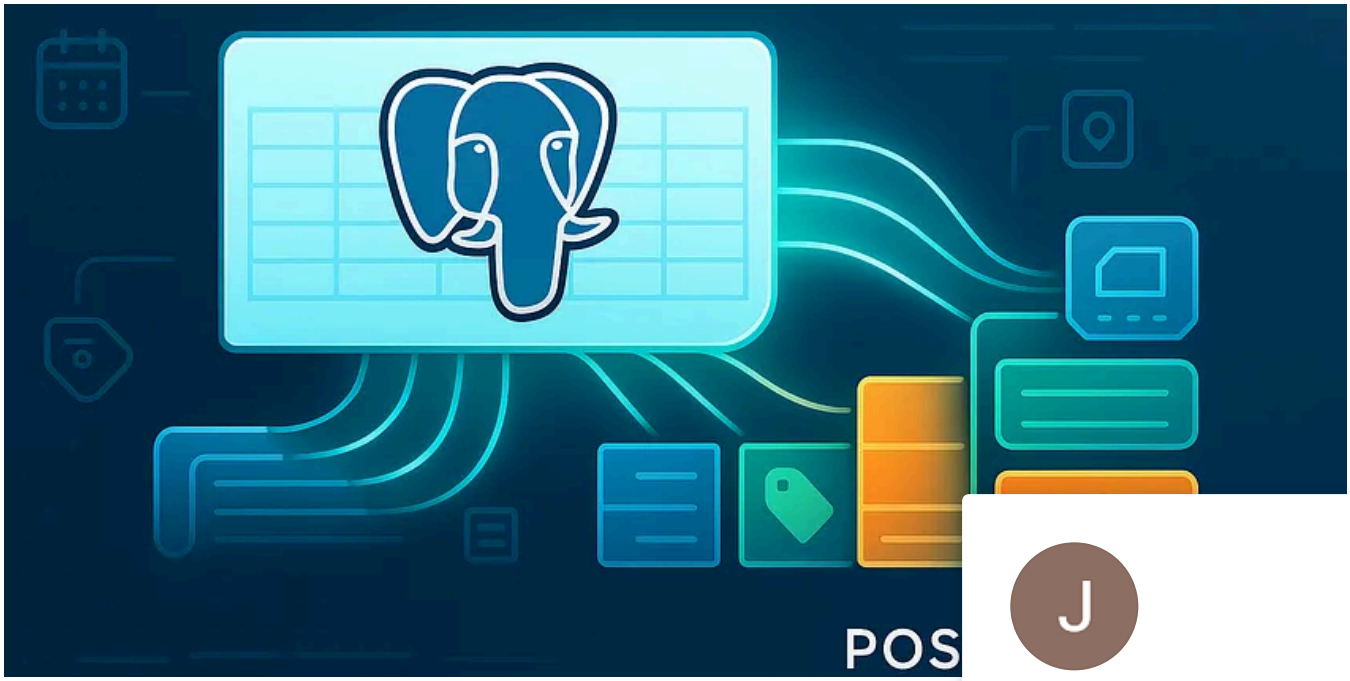


Hash Block

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

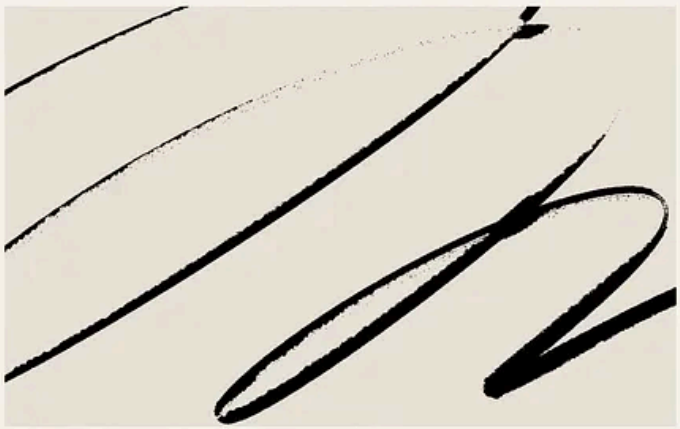


Thinking Loop

10 Postgres Partitioning Patterns for Internet-Scale A

Keep Postgres fast past a billion rows with real patterns and ready-to

Aug 13 88 2



J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

Book Author

Vahid Yousefzadeh

Real-Time Statistics in Oracle 19c

Until Oracle version 12c, executing DML statements on a table (either using the conventional method or direct-path) would not update the...

Aug 13

See more recommendations

J

Jeyaram Ayyalusamy

158 followers

Oracle DBA | AWS Certified | 18+ y
DB Admin, RAC, GoldenGate, RDS
MySQL, PostgreSQL | Author of M
& RDS books | Cloud & Performar

Book Author