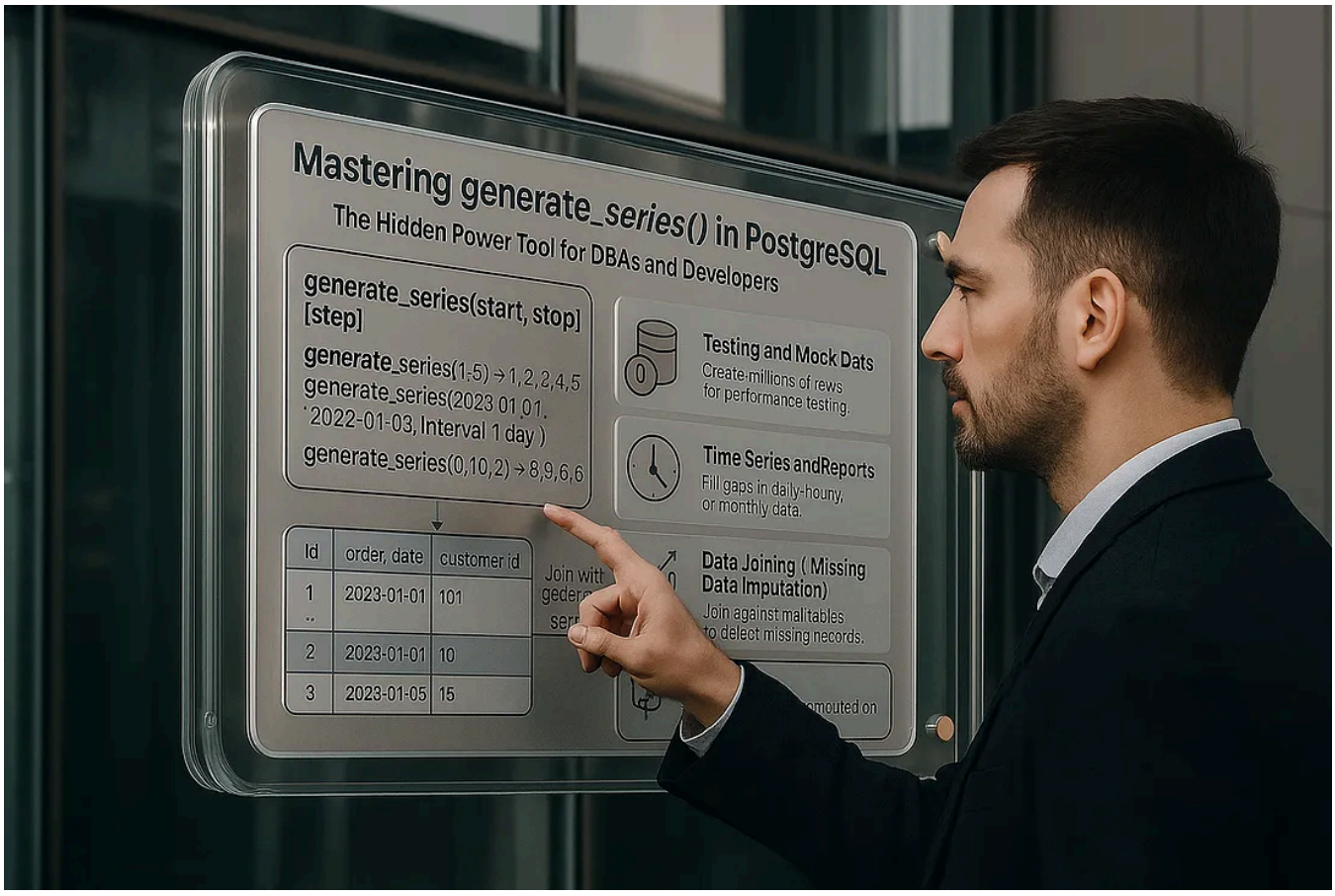
Mastering `generate_series()` in PostgreSQL: The Hidden Power Tool for DBAs and Developers

6 min read · Jul 12, 2025

J Jeyaram Ayyalusamy Following



PostgreSQL is one of the most advanced open-source relational databases, known for its extensibility, standards compliance, and rich built-in functionality. Among its

many powerful features lies a seemingly simple, often underestimated function: `generate_series()`.

This function is not just syntactic sugar; it's a **virtual data generator**, a performance-friendly replacement for looping constructs, and an indispensable tool for time-series analysis, data simulation, and reporting tasks. Whether you are a database administrator (DBA), backend developer, or data analyst, **mastering `generate_series()` will enhance the way you write SQL.**

This article presents a complete, in-depth guide on `generate_series()` —including syntax, use cases, and best practices—to help you unlock its full potential in real-world scenarios.

🔧 What Is `generate_series()`?

`generate_series()` is a **set-returning function (SRF)** in PostgreSQL. It returns a sequence of values that can be **integers, dates, timestamps**, or any other incrementable type, depending on the parameters you provide.

The function **dynamically produces a virtual dataset** — meaning it does not rely on any underlying physical table or stored data — and can be used directly in queries, joins, and views.

✅ Syntax

```
generate_series(start, stop [, step])
```

- `start` : Required. The starting value of the series.
- `stop` : Required. The upper/lower limit of the sequence.
- `step` : Optional. The increment between values (default is `1`).

🔍 Supported Types

- Integer types: `int`, `bigint`

- Temporal types: `date`, `timestamp`, `interval`

The step must be of the same type as the difference between `start` and `stop`.

🔢 Example 1: Generating a Simple Integer Series

Use Case: You need a list of numbers from 1 to 10

```
 generate_series
-----------------
               1
               2
               3
               4
               5
               6
               7
               8
               9
              10
(10 rows)
```

Output:

```
generate_series
-----------------
 1
 2
 3
 ...
10
```

🧰 Practical Applications:

- Quickly generate IDs for synthetic test data
- Use in loops, cursors, or procedural constructs

- Replace recursive CTEs for bounded numeric sequences

➕ Example 2: Custom Step Values

Use Case: You need every second number (odd numbers between 1 and 10)

```
 generate_series
-----------------
               1
               3
               5
               7
               9
(5 rows)
```

Output:

```
generate_series
-----------------
 1
 3
 5
 7
 9
```

You can also go **backward** using a negative step:

```
SELECT generate_series(10, 1, -2);
```

Output:

```
 generate_series
-----------------
              10
               8
               6
               4
               2
(5 rows)
```

🧰 Practical Applications:

- Create arithmetic progressions
- Simulate pagination or sharded keys
- Generate index-like sequences for test cases

🗓️ Example 3: Generating Date and Timestamp Series

Temporal data manipulation is a core part of modern applications, and `generate_series()` excels at producing date/time intervals.

Use Case: Generate daily dates for a report range

```
SELECT generate_series(
  '2024-01-01'::date,
  '2024-01-10'::date,
  '1 day'::interval
);
```

Output:

```
    generate_series
------------------------
 2024-01-01 00:00:00+00
 2024-01-02 00:00:00+00
 2024-01-03 00:00:00+00
 2024-01-04 00:00:00+00
```

```
 2024-01-05 00:00:00+00
 2024-01-06 00:00:00+00
 2024-01-07 00:00:00+00
 2024-01-08 00:00:00+00
 2024-01-09 00:00:00+00
 2024-01-10 00:00:00+00
(10 rows)
```

Other Variants:

- Hourly intervals:

```
SELECT generate_series(
  '2024-01-01 00:00'::timestamp,
  '2024-01-01 23:00'::timestamp,
  '1 hour'::interval
);
```

```
   generate_series
---------------------
 2024-01-01 00:00:00
 2024-01-01 01:00:00
 2024-01-01 02:00:00
 2024-01-01 03:00:00
 2024-01-01 04:00:00
 2024-01-01 05:00:00
 2024-01-01 06:00:00
 2024-01-01 07:00:00
 2024-01-01 08:00:00
 2024-01-01 09:00:00
 2024-01-01 10:00:00
 2024-01-01 11:00:00
 2024-01-01 12:00:00
 2024-01-01 13:00:00
 2024-01-01 14:00:00
 2024-01-01 15:00:00
 2024-01-01 16:00:00
 2024-01-01 17:00:00
 2024-01-01 18:00:00
 2024-01-01 19:00:00
```

```
 2024-01-01 20:00:00
 2024-01-01 21:00:00
 2024-01-01 22:00:00
 2024-01-01 23:00:00
(24 rows)

postgres=#
```

- **Monthly intervals:**

```
SELECT generate_series(
  '2023-01-01'::date,
  '2023-12-01'::date,
  '1 month'::interval
);
```

```
    generate_series
------------------------
 2023-01-01 00:00:00+00
 2023-02-01 00:00:00+00
 2023-03-01 00:00:00+00
 2023-04-01 00:00:00+00
 2023-05-01 00:00:00+00
 2023-06-01 00:00:00+00
 2023-07-01 00:00:00+00
 2023-08-01 00:00:00+00
 2023-09-01 00:00:00+00
 2023-10-01 00:00:00+00
 2023-11-01 00:00:00+00
 2023-12-01 00:00:00+00
(12 rows)

postgres=#
```

🧰 Practical Applications:

- Build time dimension tables

- Fill gaps in time-series reports
- Drive scheduled processes like backups or batch jobs

🔎 Example 4: Use as a Virtual Table in SELECT Statements

You can treat `generate_series()` like any other table in your query's `FROM` clause.

Use Case: Generate and transform data on-the-fly

```
SELECT gs AS number, gs * gs AS square
FROM generate_series(1, 5) AS gs;
```

Output:

```
 number | square
--------+--------
      1 |      1
      2 |      4
      3 |      9
      4 |     16
      5 |     25
(5 rows)
```

🧰 Practical Applications:

- Compute derived fields without preexisting data
- Simulate loops and nested datasets
- Build custom lookup or mapping tables

🔗 Example 5: Join `generate_series()` with Real Tables

One of the most compelling uses of `generate_series()` is to **join it with existing tables** to enrich or complete datasets.

Use Case: Join with an `authors` table to extract rows matching ID ranges

```
SELECT a.id, a.name, s.n
FROM authors a
JOIN generate_series(1, 5) AS s(n)
  ON a.id = s.n;
```

```
 id |      name      | n
----+----------------+---
  1 | Leo Tolstoy    | 1
  2 | Jane Austen    | 2
  3 | Mark Twain     | 3
  4 | Virginia Woolf | 4
  5 | George Orwell  | 5
(5 rows)
```

Use Case: Identify missing dates in order records

```
WITH date_range AS (
  SELECT generate_series(
    '2024-01-01'::date,
    '2024-01-07'::date,
    '1 day'::interval
  ) AS day
)
SELECT d.day, COUNT(o.id) AS order_count
FROM date_range d
LEFT JOIN orders o
  ON o.order_date = d.day
GROUP BY d.day
ORDER BY d.day;
```

Result: Dates with zero orders are included with a `count = 0`, providing accurate analytics.

```
          day           | order_count
------------------------+-------------
 2024-01-01 00:00:00+00 |           2
 2024-01-02 00:00:00+00 |           0
 2024-01-03 00:00:00+00 |           2
 2024-01-04 00:00:00+00 |           0
 2024-01-05 00:00:00+00 |           1
 2024-01-06 00:00:00+00 |           0
 2024-01-07 00:00:00+00 |           1
(7 rows)
```

🧰 Practical Applications:

- Detecting missing transactions in logs
- Filling sparse time data in visualizations
- Simulating matrices or product combinations

💡 Advanced Use Cases and Benefits

Use Case Description **Mock/test data generation** Populate large datasets without inserts **Time-series dashboards** Create all required time points, even if data is missing **Data warehouse backfilling** Identify and fill missing partitions **Gap analysis** Find skipped IDs, dates, or intervals **Matrix-style cross joins** Simulate multidimensional relationships

🛡️ Performance Considerations

- `generate_series()` is evaluated **on-the-fly** and does **not incur I/O overhead.**

- It can produce **millions of rows** quickly, but watch out for unbounded queries in production.
- Always use **LIMIT** or restrict ranges when working in exploratory SQL.

🧠 Best Practices

1. **Use descriptive aliases**
   Always alias the result:

```
FROM generate_series(1, 10) AS s(id)
```

2. **Wrap in CTEs for modular queries**
Break out series generation logic for readability:

```
WITH series AS (   SELECT generate_series(1, 100) AS id ) SELECT * FROM series
```

```
 id
-----
   5
  10
  15
  20
  25
  30
  35
  40
  45
  50
  55
  60
```

```
 65
 70
 75
 80
 85
 90
 95
100
(20 rows)
```

3. Use in reporting layers or views
Combine with joins to create reusable views in BI systems.

4. Avoid unnecessary high-volume generation
Always filter or LIMIT for safety.

🏁 Conclusion

`generate_series()` is one of PostgreSQL's most elegant and efficient tools. It abstracts complexity, saves time, and provides unmatched flexibility in SQL workflows. Its ability to produce virtual datasets—ranging from numeric sequences to sophisticated time intervals—makes it indispensable for modern database use.

Whether you're:

- Designing **test scenarios**
- Building **time-aware dashboards**
- Simulating **data structures**
- Detecting **missing values**

— this function allows you to do so without schema changes, disk usage, or procedural logic.

🔑 Key Takeaways:

- It's fast, lightweight, and memory-efficient.
- It works natively with PostgreSQL — no extensions needed.

- It reduces reliance on procedural SQL or dummy tables.

Once you integrate `generate_series()` into your daily SQL practice, you'll start solving common problems with **less code and greater elegance.**

📢 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 | Open Source | Oracle | AWS | Google

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

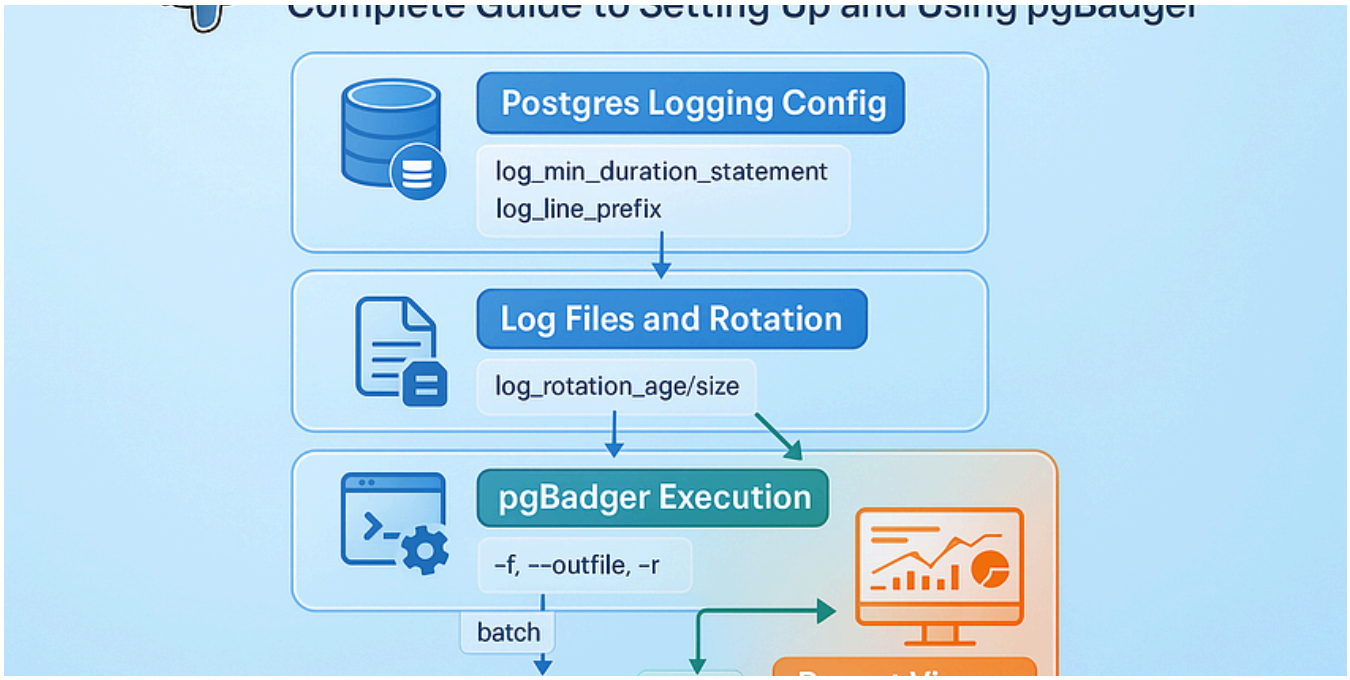
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

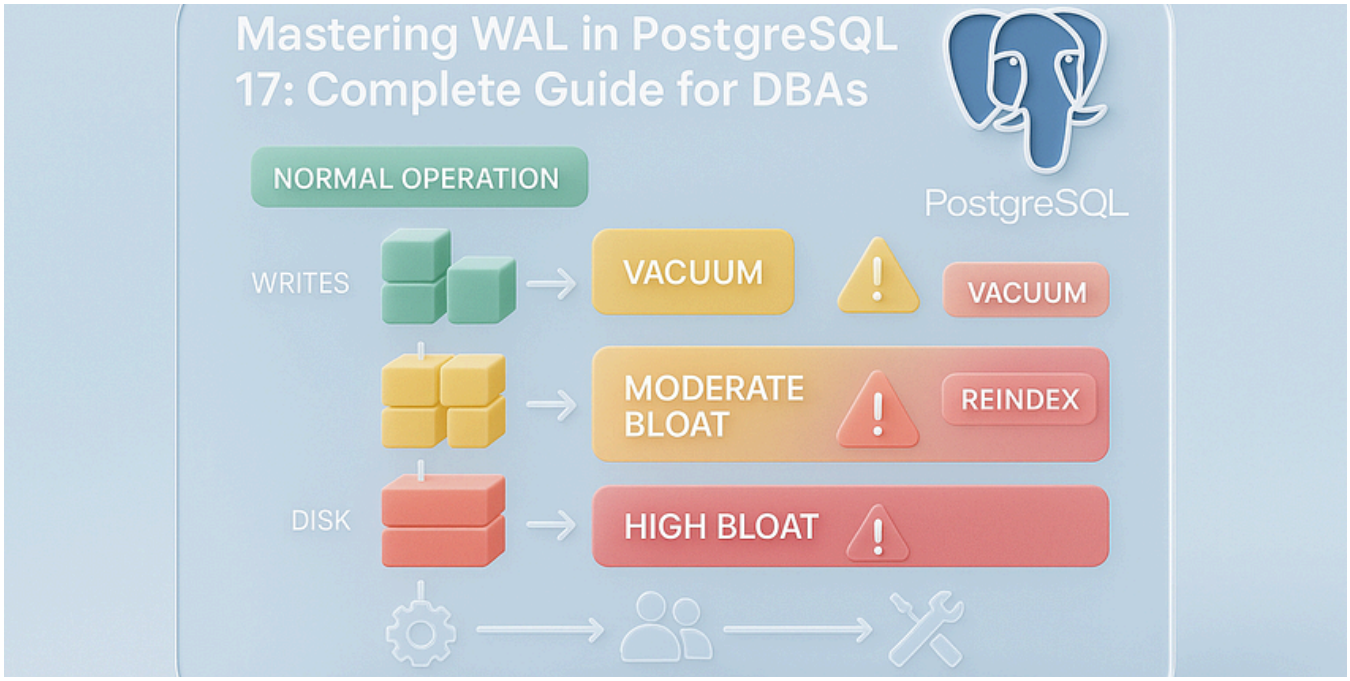


Jeyaram Ayyalusamy

PostgreSQL 17 Log Analysis Made Easy: Complete Guide to Setting Up and Using pgBadger

Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

Jun 23 52

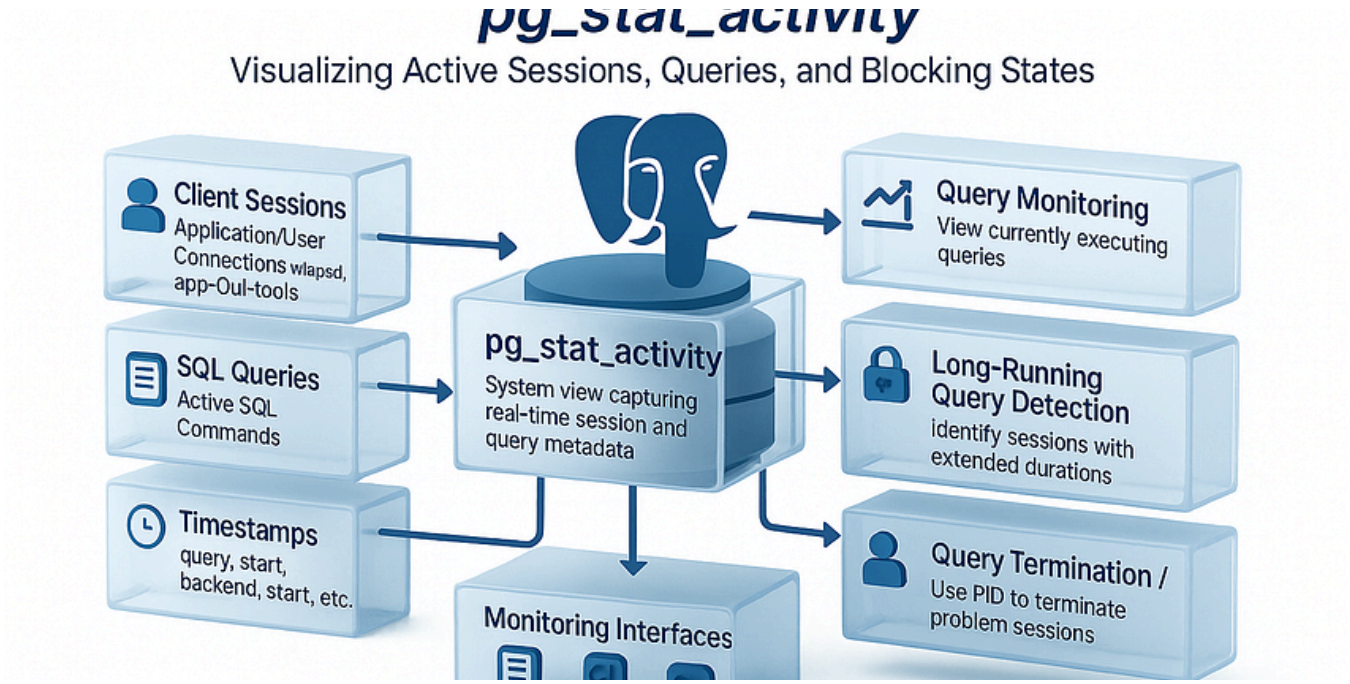


Jeyaram Ayyalusamy

Understanding and Managing Bloating in PostgreSQL: A Complete Guide for DBAs

PostgreSQL is a powerful, reliable, and feature-rich open-source relational database system. It's praised for its extensibility, ACID...

Jun 25 52

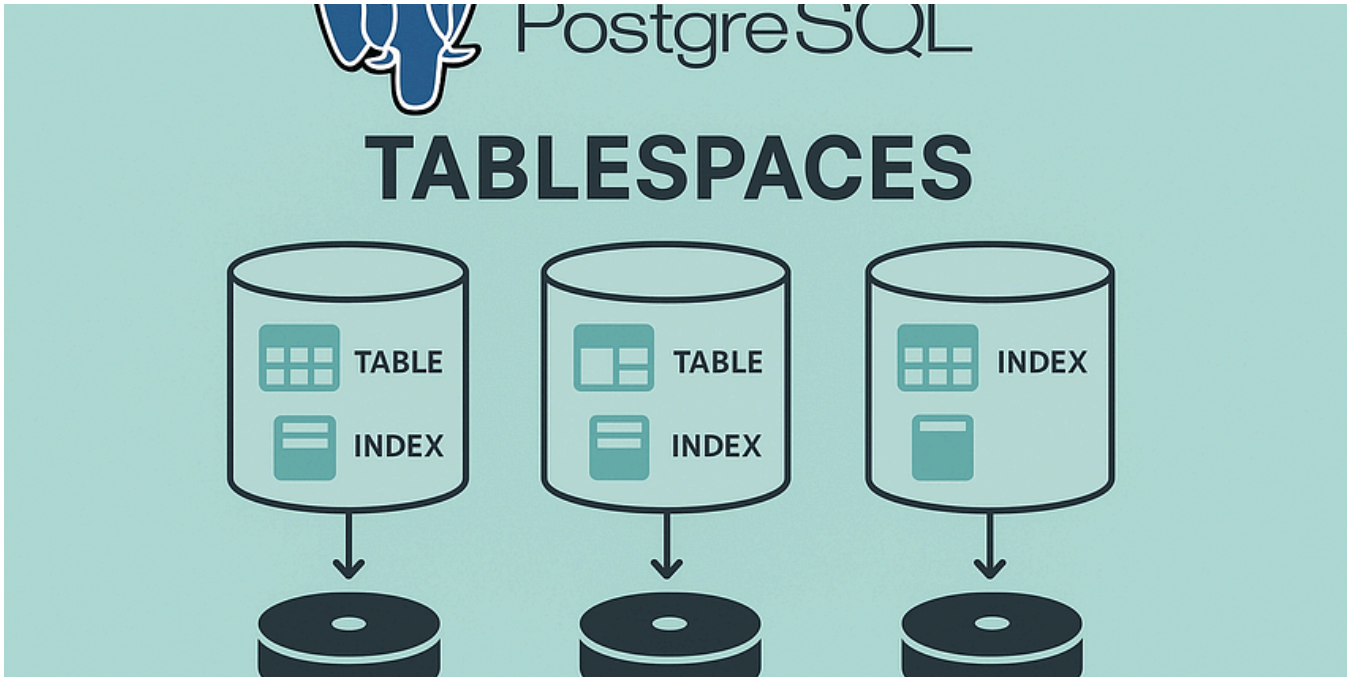


Jeyaram Ayyalusamy

Monitoring Active Queries in PostgreSQL: Real-Time Performance Diagnostics Using pg_stat_activity

PostgreSQL administrators often face the challenge of diagnosing slow performance, managing long-running queries, and identifying blocked...

Jul 12 50



Jeyaram Ayyalusamy

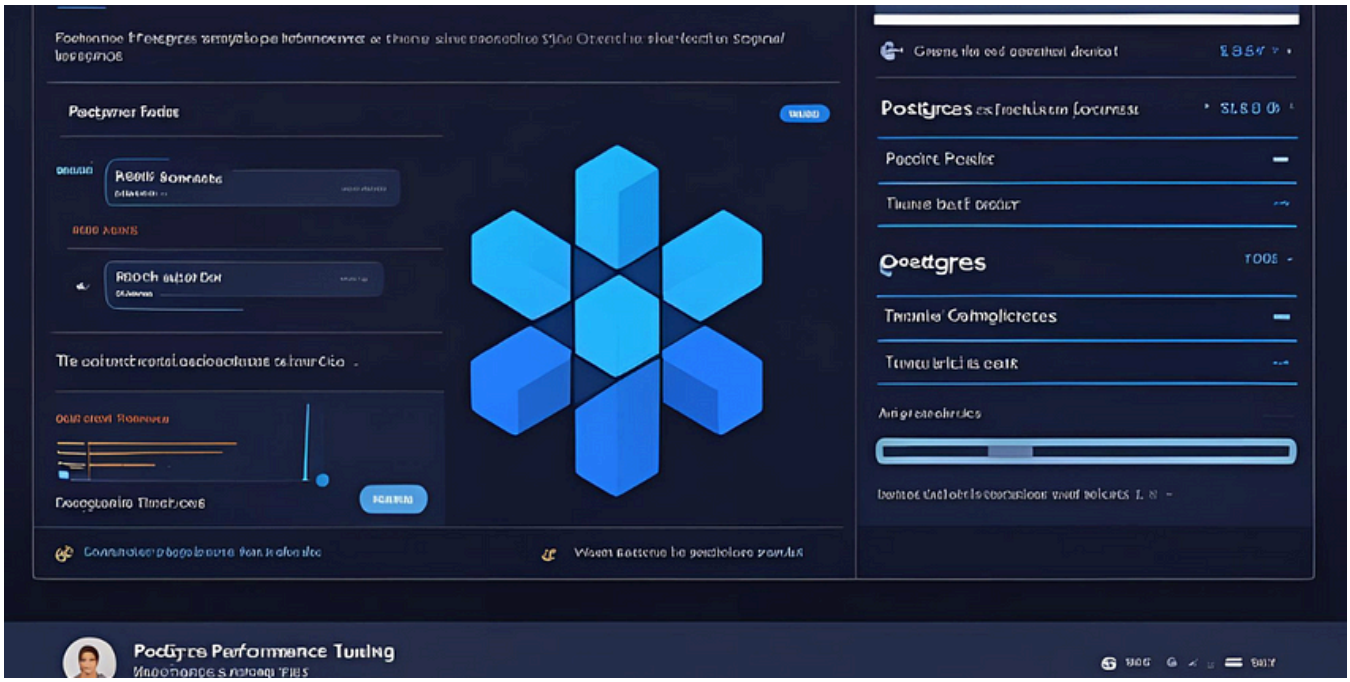
PostgreSQL Tablespaces Explained: Complete Guide for PostgreSQL 17 DBAs

PostgreSQL offers a powerful feature called tablespaces, allowing database administrators to take control of how and where data is...

Jun 12 8

See all from Jeyaram Ayyalusamy

Recommended from Medium

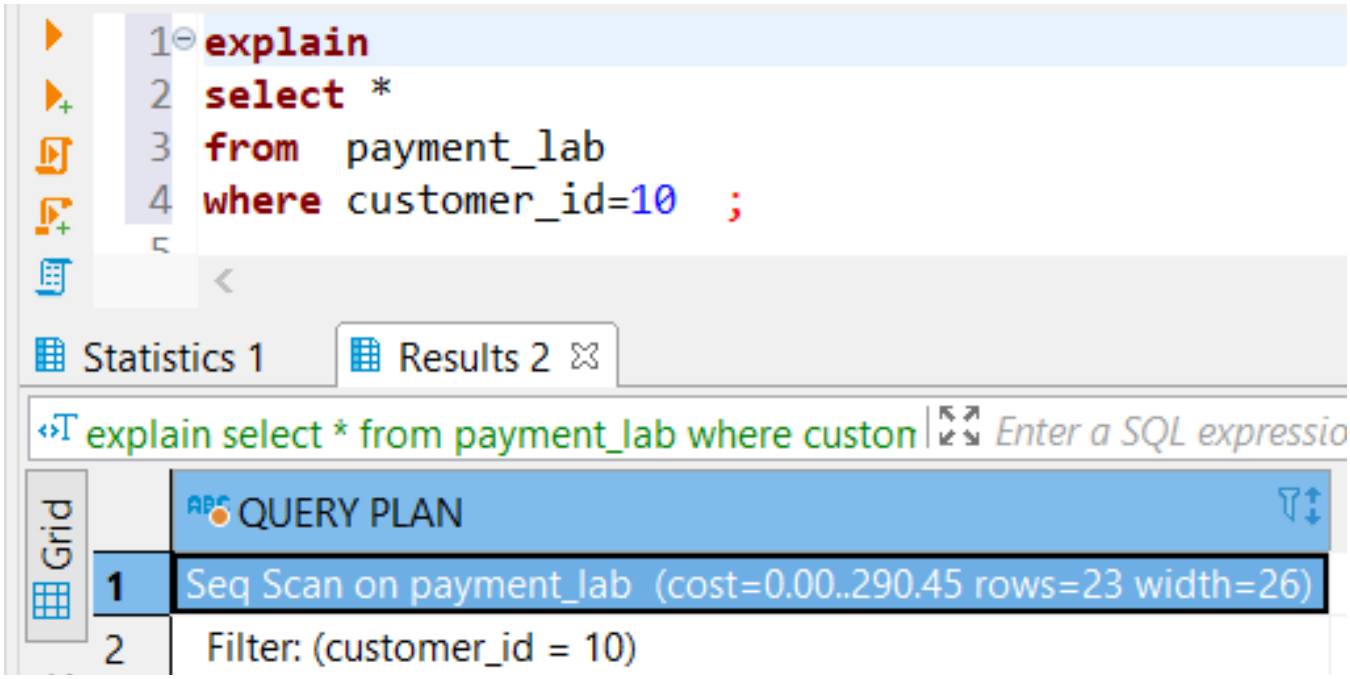


Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55



Muhammet Kurtoglu

Postgresql Query Performance Analysis

SQL tuning is critically important for ensuring database performance, reliability, and scalability. In most cases, performance issues in a...

6d ago 10

Azlan Jamal

Stop Using SERIAL in PostgreSQL: Here's What You Should Do Instead

In PostgreSQL, there are several ways to create a PRIMARY KEY for an id column, and they usually involve different ways of generating the...

Jul 12 33

techWithNeeru

This SQL Query Took 20 Seconds. Now It Takes 20ms. Here's What I Changed

Last week, I was troubleshooting a performance issue that was driving our team crazy. Our analytics dashboard was timing out, users were...

Jul 10 66

Harishsingh

PostgreSQL 18 in Microservices: You Don't Need a Separate DB for Everything

Introduction: The Myth of Database-Per-Service

Jul 13 11 1

Oz

Understanding PostgreSQL Page (Block) Structure

PostgreSQL stores data on disk in fixed-size units called pages or blocks. In this post, we'll explore how PostgreSQL organizes data...

May 14 58 1