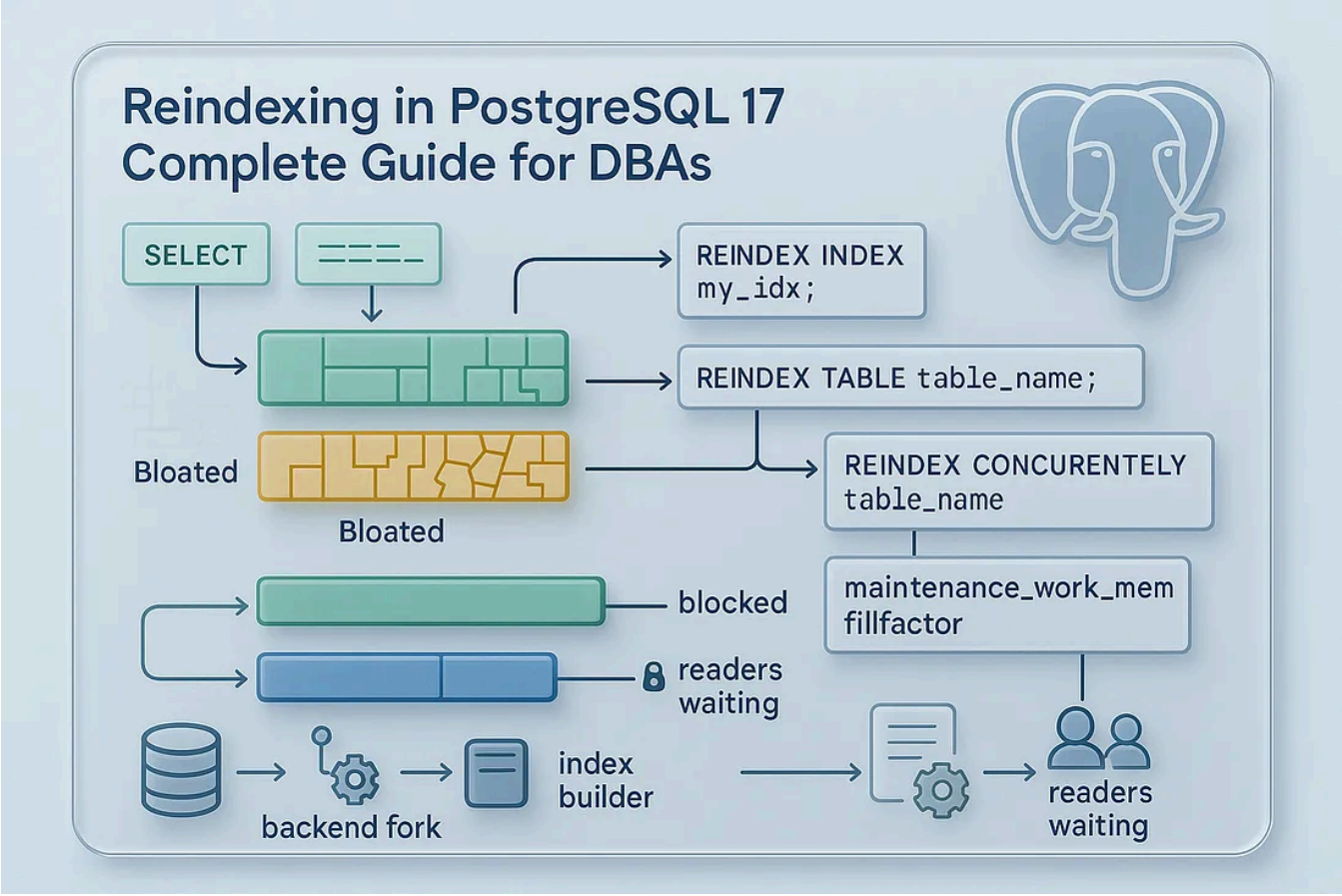
Reindexing in PostgreSQL 17: Complete Guide for DBAs

12 min read · Jun 25, 2025

Jeyaram Ayyalusamy Following



Indexes are crucial for PostgreSQL performance — but like any structure, they can degrade over time. This degradation, known as **index fragmentation**, can significantly slow down query performance and increase disk space usage.

Fortunately, PostgreSQL provides robust tools for **reindexing**, allowing administrators to rebuild indexes and restore optimal performance.

In this post, we'll explore everything you need to know about reindexing in PostgreSQL 17 — including a full hands-on demo using real-world scripts.

🚩 Why Reindexing Is Important

PostgreSQL uses indexes to speed up data access, especially for large datasets. Indexes work like pointers that help the database engine find the required rows quickly without scanning the entire table. But over time, as the data in a table changes (through inserts, updates, and deletes), the indexes that support this data can become **inefficient**, **bloated**, and **fragmented**.

Let's break down what that means and why it matters.

🔍 What Happens to Indexes Over Time?

As your PostgreSQL database gets used, it handles a lot of transactions that modify the data. For example:

- **INSERT** adds new rows.
- **UPDATE** creates a new version of a row and marks the old one as obsolete.
- **DELETE** marks rows as deleted but doesn't remove them immediately.

These actions also impact the **indexes**. PostgreSQL doesn't clean up the index space right away. Instead, over time, it leads to:

1. **Index Fragmentation**
   Empty or partially filled index pages are created, especially after many updates or deletes. These fragmented pages are still read during queries, adding overhead.
2. **Index Bloat**
   The index grows larger than necessary. This increases disk usage and memory

pressure, as more data must be cached or read from storage.

3. **Reduced Index Efficiency**
   When an index has too many levels (called a deeper B-tree), searches take longer. PostgreSQL might need to read more blocks to find a row, slowing down queries.

📉 What Are the Consequences?

When indexes are bloated and fragmented:

- **Query performance suffers**
  PostgreSQL must scan more pages, which increases read time and CPU usage.
- **Disk space is wasted**
  A bloated index can consume gigabytes of storage unnecessarily, especially in frequently updated tables.
- **Inefficient caching**
  Since bloated indexes take up more memory, PostgreSQL might evict useful data from its cache to make room.

🔧 How Reindexing Helps

The PostgreSQL command `REINDEX` is used to rebuild indexes from scratch. This means PostgreSQL creates a new index using the current, live data and then replaces the old one.

Benefits of reindexing include:

- ✅ **Removes fragmentation**
  All dead or empty pages are discarded.

- ✅ **Reduces disk usage**
  The index size is compacted to reflect only the active data.
- ✅ **Improves query speed**
  Search operations become faster because the index is optimized and may require fewer levels.
- ✅ **Restores index health**
  A fresh, clean index ensures that the PostgreSQL planner makes accurate decisions.

💡 When Should You Reindex?

You should consider reindexing in the following scenarios:

- After **bulk updates or deletes**
- When you notice **index size is larger** than the table itself
- If queries are becoming **slower despite VACUUM and ANALYZE**
- On **heavily updated transactional tables**
- As a part of **routine database maintenance**

Example SQL commands:

```
-- Reindex a specific index
REINDEX INDEX idx_customer_name;
```

```
-- Reindex all indexes on a table
REINDEX TABLE customers;

-- Reindex the entire database
REINDEX DATABASE pgdb;
```

🧠 Summary

Index bloat is a hidden performance killer in PostgreSQL. Without regular cleanup, indexes become sluggish and space-hungry. Reindexing is a **safe**, **effective**, and **simple** operation that removes this bloat and restores index efficiency. If your system deals with a lot of data changes, scheduling periodic reindexing is one of the smartest ways to keep PostgreSQL running at peak performance.

🔧 Key Concepts of Reindexing in PostgreSQL

Reindexing in PostgreSQL is more than just a cleanup operation — it's a crucial performance maintenance tool. To apply it effectively and safely, it's important to understand the **purpose**, the **locking behavior**, and the **types of indexes** supported. Let's break down each of these key concepts in detail.

✅ Purpose of Reindexing

Reindexing serves multiple performance and storage goals. Over time, due to data modifications like `INSERT`, `UPDATE`, and `DELETE`, PostgreSQL indexes can become bloated or fragmented. This leads to performance issues and unnecessary disk usage.

The **main purposes** of reindexing are:

- **Reclaiming Space**
  As dead tuples and unused pages accumulate, indexes grow in size. Reindexing reclaims this wasted space, reducing the index footprint on disk.
- **Rebuilding Efficient Index Structures**
  A freshly built index is clean, compact, and balanced. It removes internal fragmentation and restores optimal structure, which helps PostgreSQL locate data faster.
- **Improving Query Response Times**
  A healthy index allows faster lookups, joins, and sorting operations. Reindexing

can significantly reduce query latency for workloads that heavily rely on indexes.

Think of reindexing like defragmenting a hard drive — it reorganizes scattered index data for maximum efficiency.

✅ Locking Behavior

Understanding how reindexing impacts concurrency is essential, especially in production environments. PostgreSQL offers **two modes** of reindexing:

1. Exclusive Mode (Default)

- This is the traditional reindexing mode.
- It **fully locks** the table or index being rebuilt.
- **No reads or writes** are allowed on the table while reindexing is in progress.
- It is **faster** but more **disruptive**, making it suitable for maintenance windows or less critical systems.

```
REINDEX TABLE orders;
```

2. Concurrent Mode

- Introduced to minimize downtime, this mode allows reindexing **without blocking normal operations**.
- It does **not block SELECT, INSERT, UPDATE, or DELETE** operations.
- Takes **longer to complete** than exclusive mode due to extra work behind the scenes.
- Not supported for all index types (more on that below).

```
REINDEX INDEX CONCURRENTLY idx_customer_number;
```

> ⚠️ **Important:** In concurrent mode, the original and the new index exist side-by-side until the process completes, which temporarily increases disk usage.

✅ Supported Index Types

PostgreSQL supports multiple types of indexes, but not all of them can be reindexed concurrently. Here's a breakdown:

Index Type	Concurrent Reindex Support	Notes
B-tree	✅ Fully supported	Most common index type; safe for concurrent use
GiST	⚠️ Partial support	Used in geometric and full-text search; limited concurrency
GIN	❌ No concurrent support	Used for full-text search; must use exclusive mode
SP-GiST	❌ No concurrent support	Supports partitioned search spaces; exclusive only
BRIN	❌ No concurrent support	Compact for large tables; exclusive only

> 🔎 **Tip:** Always check the index type before using `REINDEX CONCURRENTLY` to avoid runtime errors. You can use `pg_indexes` or `pg_class` system views to inspect index definitions.

🧠 Final Thought

Reindexing is a powerful operation, but it should be applied with awareness of the **index type** and the **impact on running queries.** Using concurrent reindexing smartly can help keep production systems online while still reaping the benefits of optimized indexes.

🛠️ Full Demo: Reindexing in Action (PostgreSQL 17)

In this section, we'll walk through a complete hands-on example to demonstrate how **index bloat occurs**, how to **detect** it, and how to **resolve it using PostgreSQL's `REINDEX` functionality**. The scenario uses sample data to simulate a real-world case in which heavy data churn leads to index inefficiencies. You'll learn how to restore optimal index performance using `REINDEX CONCURRENTLY` —without significant downtime.

🔧 Step 1 — Create Tables & Load Sample Data

We begin by creating several large tables to simulate user activity logs. These tables will have millions of rows to mimic typical enterprise or analytics workloads.

Let's create three such tables:

```
psql -c "CREATE TABLE user_activity_log_1 AS SELECT id AS activity_id FROM gene
psql -c "CREATE TABLE user_activity_log_2 AS SELECT id AS activity_id FROM gene
psql -c "CREATE TABLE user_activity_log_3 AS SELECT id AS activity_id FROM gene
```

In each table:

- The column `activity_id` represents a simulated unique activity identifier.
- We are using `generate_series()` to quickly populate data.

Below is the output:

```
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE TABLE user_activity_log_1 AS SEL
SELECT 1000000
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE TABLE user_activity_log_2 AS SEL
SELECT 2000000
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE TABLE user_activity_log_3 AS SEL
```

```
SELECT 3000000
[postgres@ip-172-31-20-155 ~]$
```

Create Indexes on Each Table

Next, we create a **B-tree index** on the `activity_id` column for each table. This simulates a real-world scenario where indexes are created to support fast lookups on frequently queried columns.

```
psql -c "CREATE INDEX idx_user_activity_log_1_id ON user_activity_log_1 (activi
psql -c "CREATE INDEX idx_user_activity_log_2_id ON user_activity_log_2 (activi
psql -c "CREATE INDEX idx_user_activity_log_3_id ON user_activity_log_3 (activi
```

At this point, we have large tables with indexes ready. Now, let's simulate fragmentation.

```
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE INDEX idx_user_activity_log_1_ic
CREATE INDEX
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE INDEX idx_user_activity_log_2_ic
CREATE INDEX
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "CREATE INDEX idx_user_activity_log_3_ic
CREATE INDEX
[postgres@ip-172-31-20-155 ~]$
```

🔧 Step 2 — Simulate Index Fragmentation

To mimic how index bloat occurs in real systems, we will **delete most of the rows** from each table. In PostgreSQL, deleting rows doesn't immediately remove them from disk or from the index; instead, they remain as **dead tuples**. This creates internal fragmentation within the index.

```
psql -c "DELETE FROM user_activity_log_1 WHERE activity_id >= 1;"
psql -c "DELETE FROM user_activity_log_2 WHERE activity_id >= 1;"
psql -c "DELETE FROM user_activity_log_3 WHERE activity_id >= 1;"
```

Below is the output:

```
[postgres@ip-172-31-20-155 ~]$ psql -c "DELETE FROM user_activity_log_1 WHERE a
DELETE 1000000
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "DELETE FROM user_activity_log_2 WHERE a
DELETE 2000000
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "DELETE FROM user_activity_log_3 WHERE a
DELETE 3000000
[postgres@ip-172-31-20-155 ~]$
```

After these deletes:

- The data is gone, but the index still contains entries for deleted rows.
- VACUUM can clean up the heap but **cannot rebuild the index** structure.
- Index pages are now bloated with empty space and fragmentation.

This is a common scenario in OLTP workloads with high update/delete frequency.

🔧 Step 3 — Generate Reindex Commands Automatically

Instead of manually reindexing each table, we can **automate the process** using a dynamic query. This query pulls table names and sizes from PostgreSQL's catalog and generates a list of `REINDEX TABLE CONCURRENTLY` commands, sorted by table size.

```
SELECT
  'REINDEX TABLE CONCURRENTLY ' || quote_ident(relname) ||
  ' /* ' || pg_size_pretty(pg_total_relation_size(C.oid)) || ' */;'
```

```
FROM pg_class C
JOIN pg_namespace N ON N.oid = C.relnamespace
WHERE nspname = 'public'
  AND relkind = 'r'
  AND nspname !~ '^pg_toast'
ORDER BY pg_total_relation_size(C.oid) ASC;
```

This query:

- Scans only **user-defined tables** in the `public` schema.
- Filters out system and TOAST tables.
- Outputs a ready-to-execute list of `REINDEX` commands with comments showing each table's size.

✅ **Benefits:**

- You can quickly identify which tables need reindexing the most.
- Easy to copy-paste the results into your terminal or script.

Sample output:

```
                          ?column?
-------------------------------------------------------------
 REINDEX TABLE CONCURRENTLY course /* 16 kB */;
 REINDEX TABLE CONCURRENTLY user_activity_log_1 /* 21 MB */;
 REINDEX TABLE CONCURRENTLY user_activity_log_2 /* 43 MB */;
 REINDEX TABLE CONCURRENTLY user_activity_log_3 /* 64 MB */;
(4 rows)

postgres=#
```

🔧 Step 4 — Execute Reindexing

Now, we take the output of the previous step and **execute each `REINDEX TABLE CONCURRENTLY` command** one by one.

Example:

```
psql -c "REINDEX TABLE CONCURRENTLY user_activity_log_1;"
psql -c "REINDEX TABLE CONCURRENTLY user_activity_log_2;"
psql -c "REINDEX TABLE CONCURRENTLY user_activity_log_3;"
```

🛠 What's happening under the hood:

- PostgreSQL builds a **new copy of the index** in parallel.
- Reads and writes continue to operate on the old index during this time.
- Once done, PostgreSQL swaps in the new index and drops the old one.

```
[postgres@ip-172-31-20-155 ~]$ psql -c "REINDEX TABLE CONCURRENTLY user_activit
REINDEX
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "REINDEX TABLE CONCURRENTLY user_activit
REINDEX
[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "REINDEX TABLE CONCURRENTLY user_activit
REINDEX
[postgres@ip-172-31-20-155 ~]$
```

This method is especially useful in **production environments** because it **avoids locking the table** and keeps your application online.

🔧 Step 5 — Validate Post-Reindex Improvements

After reindexing is complete, it's good practice to verify the results. PostgreSQL provides useful psql meta-commands to inspect tables and index sizes:

```
psql -c "\dt+"   -- Shows table size
```

```
psql -c "\di+"   -- Shows index size
```

Look for:

- Smaller index sizes (due to removal of dead tuples and fragmentation)
- Lower total disk usage for the table
- Improved query performance on the reindexed columns

You can also rerun the dynamic query from Step 3 to compare sizes **before and after** reindexing.

```
[postgres@ip-172-31-20-155 ~]$ psql -c "\dt+"
                                          List of relations
 Schema |        Name         | Type  |  Owner   | Persistence | Access method |  Size   | Description
--------+---------------------+-------+----------+-------------+---------------+---------+-------------
 public | course              | table | postgres | permanent   | heap          | 0 bytes |
 public | user_activity_log_1 | table | postgres | permanent   | heap          | 0 bytes |
 public | user_activity_log_2 | table | postgres | permanent   | heap          | 0 bytes |
 public | user_activity_log_3 | table | postgres | permanent   | heap          | 0 bytes |
(4 rows)

[postgres@ip-172-31-20-155 ~]$
[postgres@ip-172-31-20-155 ~]$ psql -c "\di+"
                                                    List of relations
 Schema |            Name             | Type  |  Owner   |        Table        |  Size   | Description
--------+-----------------------------+-------+----------+---------------------+---------+-------------
 public | idx_user_activity_log_1_id | index | postgres | user_activity_log_1 | 0 bytes |
 public | idx_user_activity_log_2_id | index | postgres | user_activity_log_2 | 0 bytes |
 public | idx_user_activity_log_3_id | index | postgres | user_activity_log_3 | 0 bytes |
(3 rows)

[postgres@ip-172-31-20-155 ~]$
```

📊 Bonus: Measuring the Performance Gain

Want to benchmark the difference?

Try running a simple query before and after reindexing:

```
EXPLAIN ANALYZE SELECT * FROM user_activity_log_3 WHERE activity_id = 1000;
```

You'll often see significant improvements in **execution time**, **index scan cost**, and **buffer access statistics** after reindexing.

```
postgres=# EXPLAIN ANALYZE SELECT * FROM user_activity_log_3 WHERE activity_id
                                                QUERY PLAN
-----------------------------------------------------------------------------------
 Seq Scan on user_activity_log_3  (cost=0.00..0.00 rows=1 width=4) (actual time
   Filter: (activity_id = 1000)
 Planning Time: 0.413 ms
 Execution Time: 0.111 ms
(4 rows)

postgres=#
```

🧠 Final Thoughts

This complete walkthrough demonstrates how easy it is to:

- Simulate index bloat
- Generate reindexing scripts dynamically
- Run reindexing without blocking access
- Validate improvements post-operation

Reindexing is a powerful and often overlooked performance tuning tool. By using `REINDEX CONCURRENTLY`, you can fix bloated indexes **safely and efficiently**, even in production.

🚩 When Should You Reindex?

Reindexing is a resource-intensive operation, so it's important to apply it strategically — only when needed. You don't want to run it on every table without reason, but ignoring it entirely can cause long-term performance issues. The key is to identify the **right scenarios** where reindexing adds value.

Let's look at common situations and whether reindexing is recommended:

Scenario	Should You Reindex?	Reason
High update/delete workloads	✅ Yes	Frequent row changes cause dead tuples and index fragmentation over time.
After bulk imports or large inserts	✅ Yes	Indexes may grow rapidly and become unbalanced, especially if not batched.
Disk space concerns	✅ Yes	Bloated indexes consume unnecessary storage that can be reclaimed.
Performance degradation observed	✅ Yes	Slow queries and increased I/O are common symptoms of bloated indexes.
Low-update, read-only/static tables	❌ Rarely needed	Indexes remain relatively clean and balanced since there are minimal changes.

🔍 How to Know It's Time?

Here are some indicators that reindexing may be beneficial:

- Index size is **larger than the table itself.**
- `EXPLAIN ANALYZE` shows poor performance on indexed queries.
- You observe **slow index scans** or **increased disk I/O.**
- `pg_stat_user_indexes` shows **low index usage** despite large size.
- Table is frequently vacuumed but **query performance hasn't improved.**

Running periodic checks using tools like `pgstattuple` or custom SQL scripts can help you detect index bloat before it causes problems.

🏁 Conclusion

Reindexing is one of PostgreSQL's **most effective yet frequently overlooked** maintenance techniques. While operations like `VACUUM` and `ANALYZE` are commonly automated, reindexing often gets ignored until performance issues emerge.

As your PostgreSQL database scales — with millions of rows, constant updates, and growing disk usage — indexes inevitably **degrade**, **fragment**, and **bloat.** Ignoring this silent buildup can lead to:

- Slower query performance
- Increased memory and disk pressure
- Poorer user experience and higher infrastructure costs

But with proper awareness and timing, **reindexing restores order and speed** to your database.

By understanding:

- 🕓 **When to reindex** (e.g., after heavy DML activity or when index bloat is detected)
- 🛡 **How to reindex safely** (using `REINDEX CONCURRENTLY` to avoid downtime)
- ⚖️ **When to choose concurrent vs exclusive reindexing** (based on system availability needs)

...you ensure your PostgreSQL environment stays:

✅ **Fast** — Queries return quickly and efficiently
✅ **Lean** — Storage is used optimally without unnecessary bloat
✅ **Stable** — Index corruption or performance degradation is prevented early

✅ Reindexing = Long-term PostgreSQL Health

Make reindexing a **proactive part of your database maintenance strategy**, not just a reactive fix. A well-maintained index structure is one of the keys to a high-performance, cost-efficient PostgreSQL deployment.

📢 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 Sql Oracle Open Source

Following

Written by Jeyaram Ayyalusamy

62 followers · 2 following

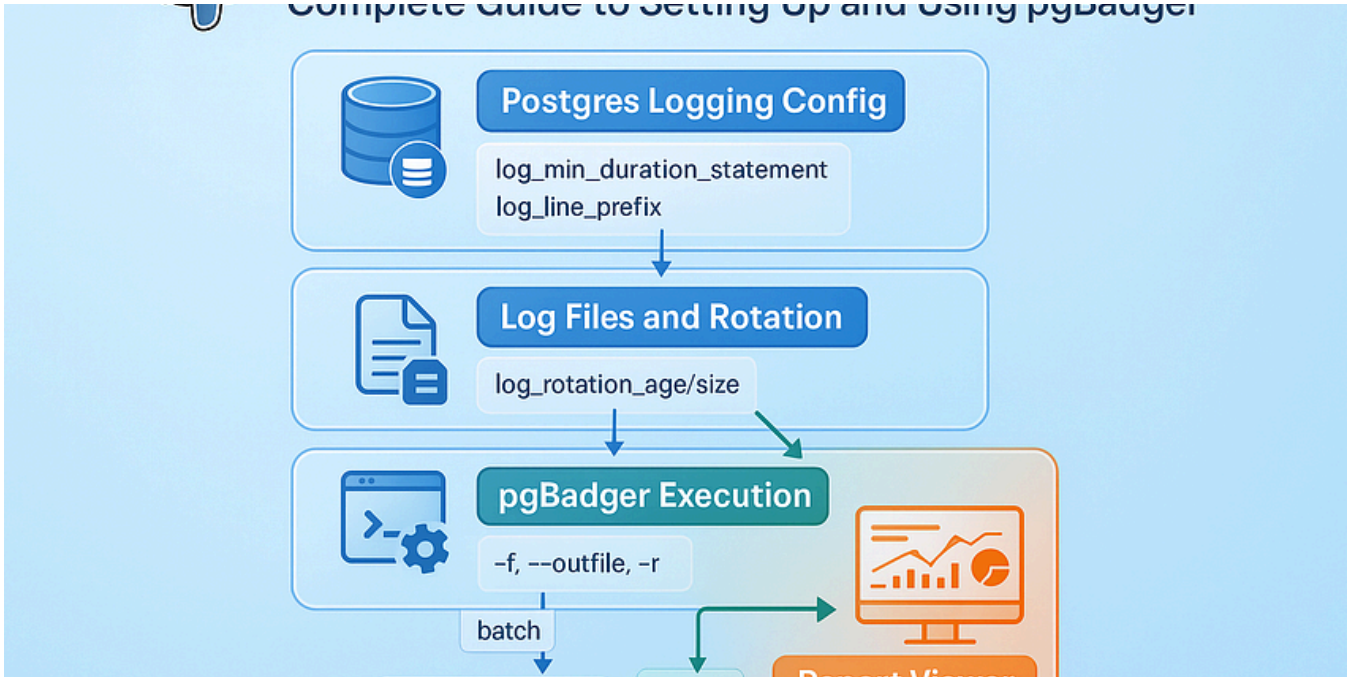
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

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...

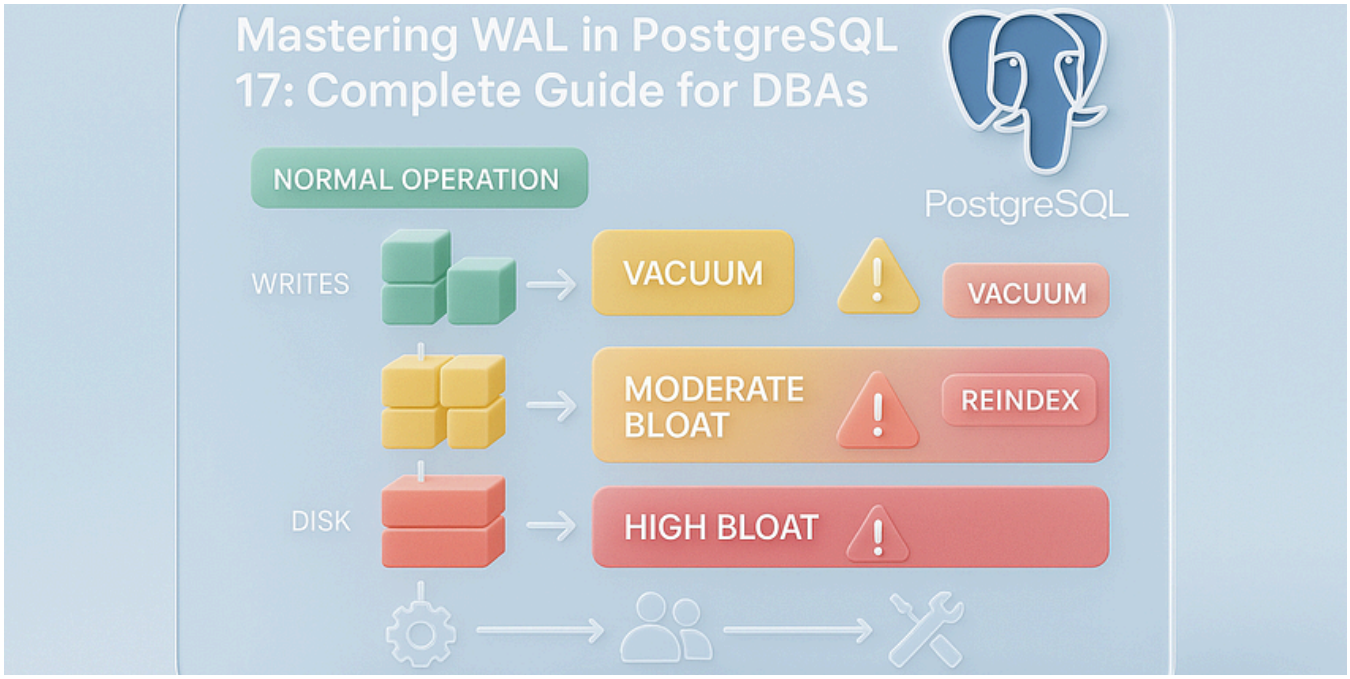
Jun 23 52

Jeyaram Ayyalusamy

PostgreSQL 17 Administration: Mastering Schemas, Databases, and Roles

PostgreSQL has evolved into one of the most feature-rich relational databases available today. With version 17, its administrative...

Jun 9 3



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 2

Jeyaram Ayyalusamy

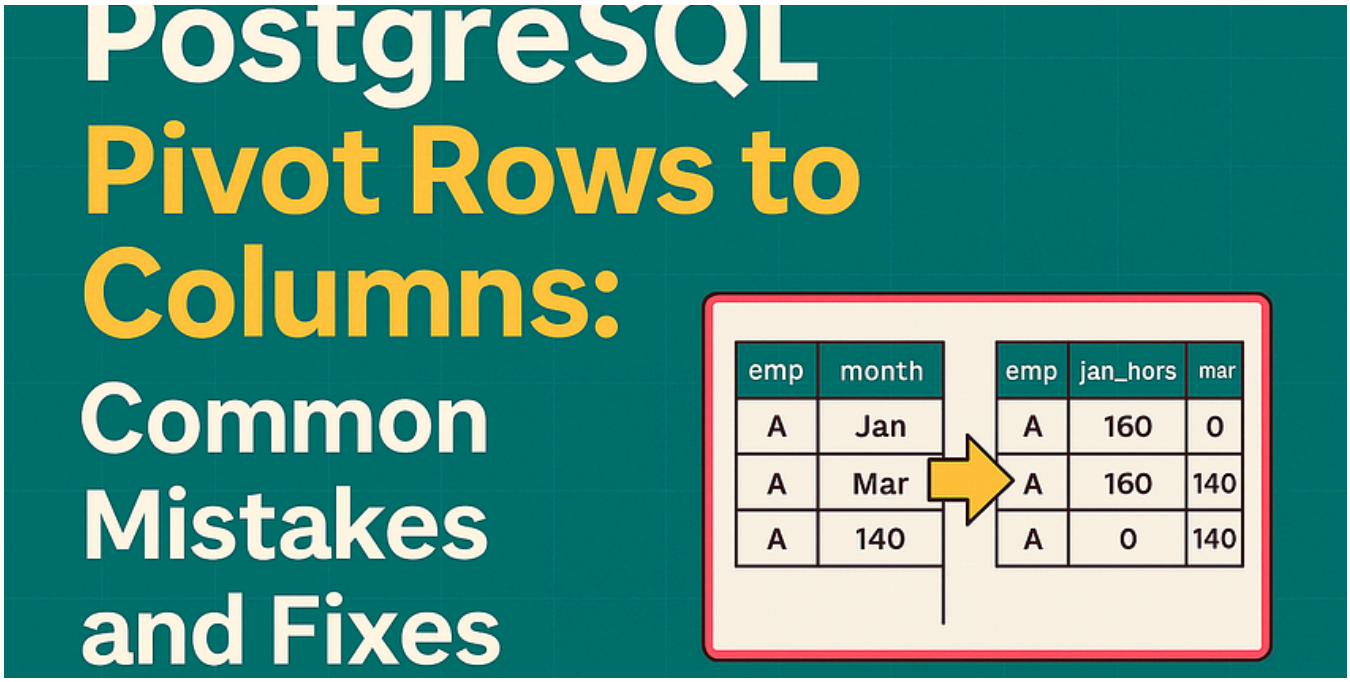
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

PostgreSQL 17 is the latest release of one of the world's most advanced open-source relational databases. If you're using a Red Hat-based...

Jun 3

See all from Jeyaram Ayyalusamy

Recommended from Medium



Ajaymaurya

PostgreSQL Pivot Rows to Columns: Common Mistakes and Fixes

When working with data in PostgreSQL, pivoting rows into columns can feel like a superpower—until something breaks. Whether you're...

Jun 27 15

ThreadSafe Diaries

PostgreSQL 18 Just Rewrote the Rulebook, Groundbreaking Features You Can't Ignore

From parallel ingestion to native JSON table mapping, this release doesn't just evolve Postgres, it future-proofs it.

Jun 27 477 4

In Level Up Coding by Daniel Craciun

Stop Using UUIDs in Your Database

How UUIDs Can Destroy SQL Database Performance

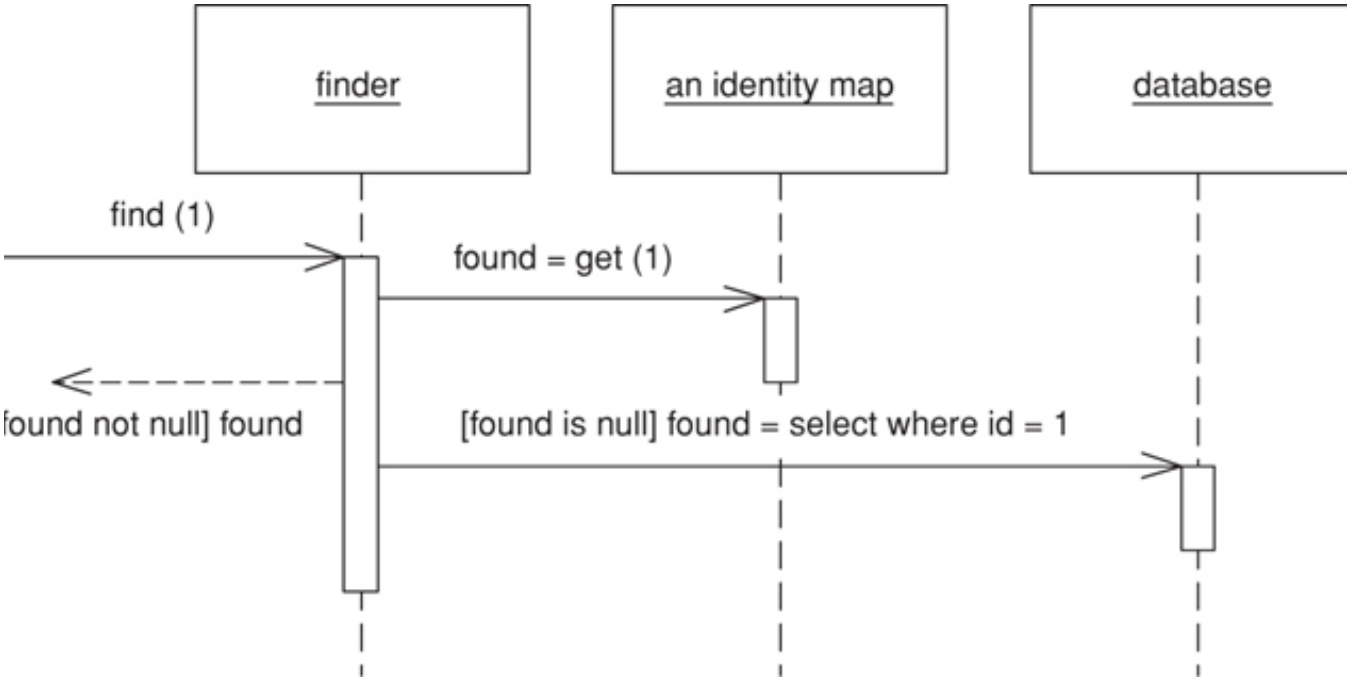
Jun 25 793 50

In AlgoMart by Yash Jain

SQL Server vs MySQL vs PostgreSQL — Picking the Right DB Like a Dev

If you've ever had to choose between SQL Server, MySQL, and PostgreSQL, you know it's not just about syntax or what the job post says. It's...

Jun 26 43



Harishsingh

The Forgotten SQL Pattern That Reduced Our Query Time by 90%

Like many teams, we assumed our SQL queries were "good enough" because they returned the correct results. But when a simple dashboard...

6d ago 4

Mojtaba Azad

How We Migrated a 100M Row PostgreSQL Table With Zero Downtime (and Survived)

Here's how we pulled off a 100-million-row table migration in PostgreSQL without a single second of downtime, using only native tools.

Jun 7 56 2

See more recommendations