

Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



Reindexing in PostgreSQL 17: The Complete DBA Guide to Keeping Your Indexes Healthy

16 min read · Jul 2, 2025



Jeyaram Ayyalusamy

Following



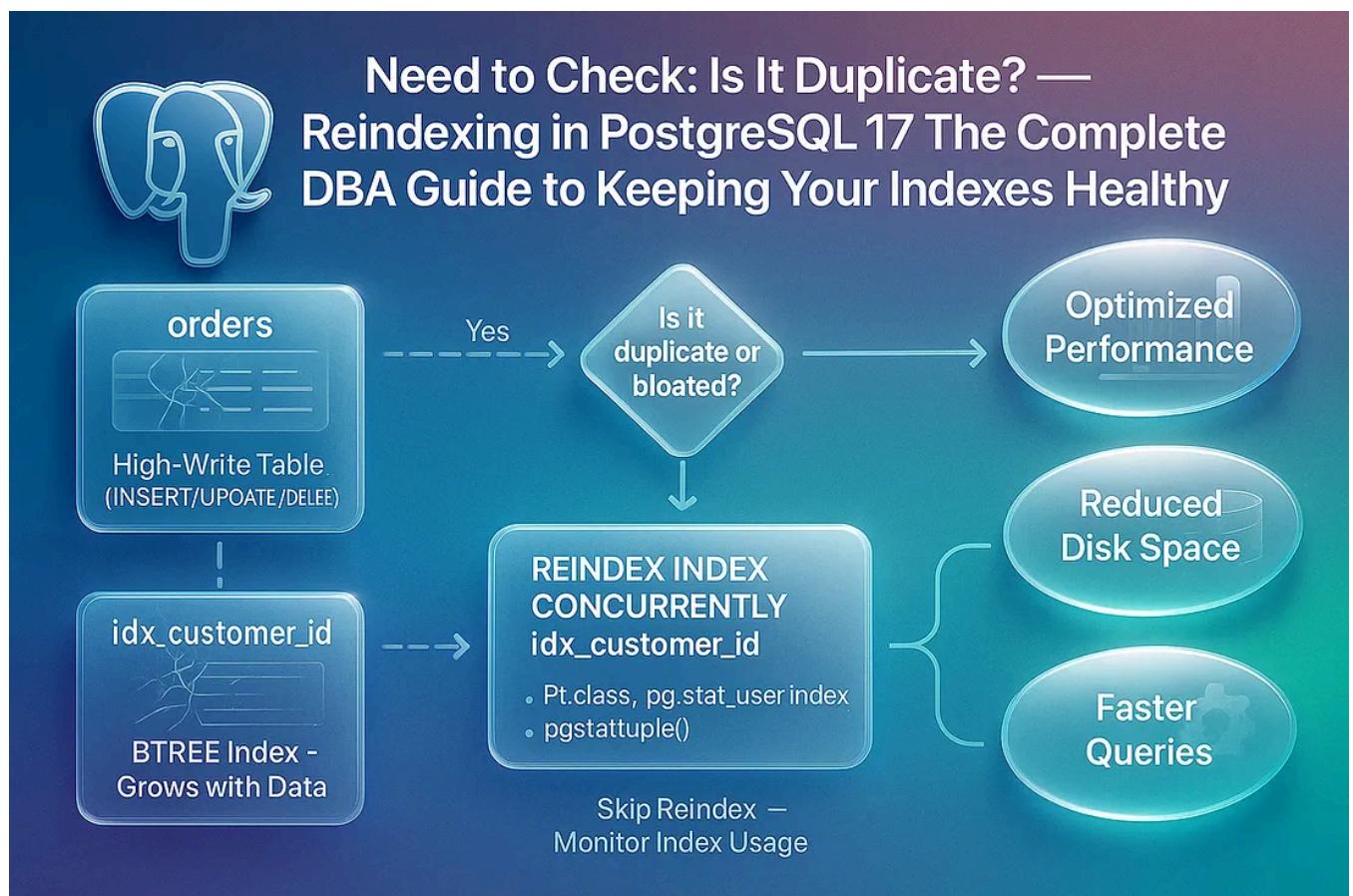
Listen



Share



More



PostgreSQL is one of the most trusted and performant open-source relational database systems available today. Its indexing capabilities — especially B-tree indexes — play a foundational role in ensuring fast and reliable data access for millions of workloads across industries. Whether you're building OLTP applications

or analytics dashboards, indexes are often what keep query performance sharp and efficient.

But there's a catch.

The Hidden Cost of Index Usage: Fragmentation

Behind the scenes, as your tables undergo **INSERTs, UPDATEs, and DELETEs**, their associated indexes are also constantly being modified. Over time, this leads to a problem known as **index fragmentation** — a state where the index structure becomes cluttered with obsolete entries and inefficient page splits.

Why does this happen?

- **UPDATEs** in PostgreSQL do not overwrite existing rows; instead, they create new versions of rows, while the old ones become **dead tuples**.
- **DELETEs** remove rows logically, but the space isn't reclaimed immediately in the index.
- **INSERTs** can cause page splits and misalignment, especially if data isn't inserted in sorted order.

Over time, these actions cause indexes to **grow larger than necessary**, even though the actual useful data remains the same.

Consequences of Ignoring Index Fragmentation

If left unchecked, bloated and fragmented indexes can severely impact database performance and operational costs. Here's what you risk:

-  **Slower query performance**

PostgreSQL has to traverse more pages and levels within the index, increasing CPU and disk I/O.

-  **Increased disk usage**

Bloated indexes consume more storage, which affects backup size, replication lag, and general resource utilization.

-  **Reduced memory and cache efficiency**

Large, inefficient indexes reduce the effectiveness of PostgreSQL's buffer cache, leading to more frequent disk reads.

-  **Skewed query planning**

Fragmented indexes can distort PostgreSQL's query planner statistics, resulting in suboptimal execution plans.

The Solution: Reindexing

To resolve this issue, PostgreSQL offers a straightforward but powerful solution: **REINDEX**.

Reindexing involves rebuilding the index from scratch based on the current table data. This eliminates fragmentation, compacts the index structure, and refreshes the internal metadata. Think of it like defragmenting your disk — but for database indexes.

The benefits of reindexing include:

-  **Faster queries** thanks to cleaner index trees and improved access paths
-  **Reduced bloat** by reclaiming space used by dead tuples and fragmented pages
-  **Improved planner accuracy** due to updated statistics
-  **Lower infrastructure cost** due to leaner index sizes

But reindexing isn't something you do blindly. It requires understanding of **index health**, **table access patterns**, and **locking behaviors**, especially in production systems.

What You'll Learn in This Guide

In this comprehensive PostgreSQL 17 reindexing guide, we'll cover everything a modern DBA or developer needs to know:

Why Reindexing is Essential

You'll learn why fragmentation builds up, how to detect bloated indexes, and what performance issues it causes.

Types of Reindexing in PostgreSQL

PostgreSQL offers multiple scopes of reindexing:

- **REINDEX INDEX:** Rebuild a specific index
- **REINDEX TABLE:** Rebuild all indexes on a table
- **REINDEX DATABASE:** Rebuild all user-defined indexes in the current database
- **REINDEX SYSTEM:** Rebuild catalog indexes — used for maintenance or corruption recovery

Each type serves a different use case. We'll explore when to use which.

Locking Behavior & Trade-Offs

Reindexing isn't free. Some modes require exclusive locks that block access, while others like **REINDEX CONCURRENTLY** allow reindexing without blocking reads and writes. We'll walk through:

- How concurrent reindexing works
- When to use it
- Trade-offs in speed vs availability

Full Demo in PostgreSQL 17

We'll provide a hands-on walkthrough using PostgreSQL 17:

- Creating large tables with simulated bloat
- Measuring bloat using PostgreSQL catalog views
- Running reindexing operations safely
- Verifying improvements in space and performance

Who Is This Guide For?

This article is ideal for:

-  PostgreSQL DBAs maintaining large production systems
-  Developers optimizing application-level query performance
-  Data engineers and analysts managing analytical workloads
-  DevOps and SRE teams monitoring database health

Whether you're managing a fast-moving transactional workload or analyzing terabytes of data, understanding how and when to reindex is a critical skill.

Let's Dive In

This isn't just a theoretical guide — it's packed with practical advice, real SQL examples, and visual explanations.

By the end of this post, you'll know exactly how to check for index duplication or bloat, how to choose the right reindexing strategy, and how to implement it safely and efficiently in PostgreSQL 17.

Let's get started. 

Key Concepts of Reindexing in PostgreSQL

Before diving into the implementation, it's essential to understand the **core principles** behind reindexing in PostgreSQL. Reindexing isn't just about freeing up disk space — it's about restoring the health, speed, and efficiency of your indexes. This section covers the **purpose, locking behavior, supported index types**, and the **performance trade-offs** you need to consider when reindexing.

1 Purpose of Reindexing

At its core, **reindexing** is the process of **rebuilding an index from scratch** using the current data in the table. This operation provides several key benefits:

-  **Eliminates Index Fragmentation**

Over time, frequent `INSERT`, `UPDATE`, and `DELETE` operations cause indexes to become fragmented and bloated. Reindexing discards obsolete entries and internal page splits, restoring a clean structure.

-  **Optimizes Access Paths for Faster Queries**

A well-maintained index speeds up data retrieval by reducing the number of disk pages PostgreSQL needs to scan. Reindexing ensures that the index tree is balanced and efficient, improving query performance.

-  **Reduces Disk Space Usage**

Bloated indexes can take up far more space than necessary. Reindexing compresses them back down to their ideal size, helping you manage storage more effectively — especially in high-write environments.

2 Locking Behavior

PostgreSQL offers two modes of reindexing, each with different implications for concurrency and availability.

Mode	Description	Locking Behavior
Exclusive (default)	Rebuilds the index by locking the table completely	Blocks all reads and writes until reindexing completes
Concurrent	Allows reindexing without full table lock	Minimal blocking, queries and transactions can continue

Exclusive Mode

- Fast and efficient.
- Suitable for small tables or maintenance windows.
- Can block users and applications — **not ideal for production workloads**.

Concurrent Mode

- Uses a background-safe strategy to rebuild the index.
- Keeps the old index in place until the new one is built.
- Slower but **much safer for production systems**, especially large or busy tables.

 Use `REINDEX CONCURRENTLY` when minimizing downtime is important.

3 Supported Index Types

Not all index types in PostgreSQL fully support concurrent reindexing. Understanding the capabilities of each is crucial when planning a reindex operation.

Index Type	Concurrent Reindex Support	Notes
B-tree	 Fully supported	Most common; ideal for concurrent reindexing
GiST	 Limited support	Used for geometric or full-text search; concurrency may be restricted
GIN	 Not fully supported	Reindexing must be done exclusively
SP-GiST	 Not supported	Only supports exclusive reindex
BRIN	 Not supported	Use only during low activity periods

 Always confirm index type compatibility before attempting concurrent reindexing.

4 Performance Trade-Offs

Choosing the right reindexing mode often involves a trade-off between **speed** and **system availability**.

Concurrent Reindexing

-  Pros: Keeps systems online, avoids locks
-  Cons: Slower execution, uses more temporary disk space

- Best for: Production databases with live traffic

Exclusive Reindexing

-  Pros: Faster, uses fewer resources
-  Cons: Requires full table lock — can block all activity
- Best for: Development/staging environments or during scheduled downtime

Key Takeaway

Reindexing isn't just a maintenance task — it's a **performance optimization strategy**. To use it effectively:

- Understand when and why reindexing is needed
- Choose the right mode based on your workload and system demands
- Know the index types you're working with to avoid surprises

In the next section, we'll walk through a hands-on demo using PostgreSQL 17 to put these concepts into action.

Hands-On Demo: Reindexing in PostgreSQL 17

PostgreSQL offers powerful indexing features that help databases scale efficiently and serve queries faster. However, over time, frequent writes (INSERTs, UPDATEs, DELETEs) can lead to **index fragmentation**. This hands-on demo will walk you through how to simulate that fragmentation, detect it, and resolve it using **safe, non-blocking reindexing** with PostgreSQL 17.

We'll use a set of large mock tables with a column called `task_id`, each representing workloads from different projects in a hypothetical system. Our steps include data loading, simulating bloat, generating reindex commands dynamically, and validating improvements.

Step 1: Load Sample Data

To simulate a real production scenario, we'll create 10 large tables — each with 1 to 10 million rows. These tables represent task logs from different teams or time windows. Each record will include a single numeric ID: `task_id`.

Objective:

Create millions of rows to simulate heavy workloads and generate large indexes.

Example Commands:

```
psql -c "CREATE TABLE project_tasks_1 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_2 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_3 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_4 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_5 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_6 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_7 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_8 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_9 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
psql -c "CREATE TABLE project_tasks_10 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
```

```
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_1 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_2 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_3 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_4 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_5 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_6 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_7 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_8 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_9 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE TABLE project_tasks_10 AS SELECT generate_series AS task_id FROM generate_series(1, 1000000);"
SELECT 1000000
SELECT 2000000
SELECT 3000000
SELECT 1000000
```

```
SELECT 2000000
SELECT 3000000
SELECT 1000000
SELECT 2000000
SELECT 3000000
SELECT 10000000
[postgres@ip-172-31-92-215 ~]$
```

After creating the tables, we build indexes on `task_id`:

```
psql -c "CREATE INDEX idx_project_tasks_1_id ON project_tasks_1 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_2_id ON project_tasks_2 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_3_id ON project_tasks_3 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_4_id ON project_tasks_4 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_5_id ON project_tasks_5 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_6_id ON project_tasks_6 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_7_id ON project_tasks_7 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_8_id ON project_tasks_8 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_9_id ON project_tasks_9 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_10_id ON project_tasks_10 (task_id);"
```

```
[postgres@ip-172-31-92-215 ~]$
[postgres@ip-172-31-92-215 ~]$ psql -c "CREATE INDEX idx_project_tasks_1_id ON
psql -c "CREATE INDEX idx_project_tasks_2_id ON project_tasks_2 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_3_id ON project_tasks_3 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_4_id ON project_tasks_4 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_5_id ON project_tasks_5 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_6_id ON project_tasks_6 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_7_id ON project_tasks_7 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_8_id ON project_tasks_8 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_9_id ON project_tasks_9 (task_id);"
psql -c "CREATE INDEX idx_project_tasks_10_id ON project_tasks_10 (task_id);"
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

```
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
[postgres@ip-172-31-92-215 ~]$
```

 Each table now has millions of entries and an indexed column, setting the stage for fragmentation.

Step 2: Simulate Fragmentation

Indexes become bloated over time when rows are frequently deleted or updated. PostgreSQL keeps “dead” entries to maintain MVCC (Multi-Version Concurrency Control) but doesn’t reclaim index space until a **reindex** operation.

To mimic this, we’ll delete **almost all rows** in each table:

```
psql -c "DELETE FROM project_tasks_1 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_2 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_3 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_4 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_5 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_6 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_7 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_8 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_9 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_10 WHERE task_id >= 1;"
```

```
[postgres@ip-172-31-92-215 ~]$  
[postgres@ip-172-31-92-215 ~]$ psql -c "DELETE FROM project_tasks_1 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_2 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_3 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_4 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_5 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_6 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_7 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_8 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_9 WHERE task_id >= 1;"  
psql -c "DELETE FROM project_tasks_10 WHERE task_id >= 1;"  
DELETE 1000000  
DELETE 2000000  
DELETE 3000000  
DELETE 1000000  
DELETE 2000000  
DELETE 3000000  
DELETE 1000000  
DELETE 2000000  
DELETE 3000000  
DELETE 10000000  
[postgres@ip-172-31-92-215 ~]$
```

⚠️ At this point, the tables appear “empty” but their indexes are still bloated with old entries that consume space and reduce performance.

🔧 Step 3: Generate Reindex Commands Dynamically

Rather than hard-coding commands for each table, we can automatically generate reindexing SQL using PostgreSQL’s catalog views. This method scales across all tables in your `public` schema.

💡 Query:

```
SELECT  
  'REINDEX TABLE CONCURRENTLY ' || quote_ident(relname) ||  
  ' /* Size: ' || pg_size.pretty(pg_total_relation_size(C.oid)) || ' */;' AS reindex_command  
FROM pg_class C  
JOIN pg_namespace N ON N.oid = C.relnamespace
```

```

WHERE nspname = 'public'
  AND relkind = 'r'
ORDER BY pg_total_relation_size(C.oid) ASC;

```

What This Does:

- Lists all user tables.
- Shows current size for each table (helpful for tracking improvements).
- Outputs safe, ready-to-run REINDEX TABLE CONCURRENTLY commands.

Example output:

```

postgres=# SELECT
  'REINDEX TABLE CONCURRENTLY ' || quote_ident(relname) ||
  ' /* Size: ' || pg_size.pretty(pg_total_relation_size(C.oid)) || ' */;' AS re
FROM pg_class C
JOIN pg_namespace N ON N.oid = C.relnamespace
WHERE nspname = 'public'
  AND relkind = 'r'
ORDER BY pg_total_relation_size(C.oid) ASC;
          reindex_command
-----+
REINDEX TABLE CONCURRENTLY project_tasks_1 /* Size: 56 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_4 /* Size: 56 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_7 /* Size: 56 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_5 /* Size: 112 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_2 /* Size: 112 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_8 /* Size: 112 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_6 /* Size: 168 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_9 /* Size: 168 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_3 /* Size: 168 MB */;
REINDEX TABLE CONCURRENTLY project_tasks_10 /* Size: 560 MB */;
(10 rows)

postgres=#
postgres=#
...

```

You can then copy and paste these into your terminal.

🔧 Step 4: Execute Reindexing

Now, execute the generated commands for each table one by one. This will rebuild each table's indexes without locking them for reading or writing.

```
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_1;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_2;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_3;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_4;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_5;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_6;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_7;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_8;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_9;"
```

[Open in app ↗](#)

Medium



Search



```
[postgres@ip-172-31-92-215 ~]$  
[postgres@ip-172-31-92-215 ~]$ psql -c "REINDEX TABLE CONCURRENTLY project_task  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_2;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_3;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_4;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_5;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_6;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_7;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_8;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_9;"  
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_10;"  
REINDEX  
REINDEX
```

REINDEX

[postgres@ip-172-31-92-215 ~]\$

- ✓ **REINDEX CONCURRENTLY** is PostgreSQL's safest way to rebuild indexes. It creates a new index, synchronizes it with changes, and then replaces the old one — **without blocking queries or transactions**.

🔧 Step 5: Validate Results

After reindexing, it's important to confirm that the operation worked and that your indexes are now smaller and more efficient.

🔍 Check table and index sizes:

```
psql -c "\dt+"
psql -c "\di+"
```

```
[postgres@ip-172-31-92-215 ~]$ psql -c "\dt+"
                                         List of relations
 Schema |        Name         | Type  | Owner   | Persistence | Access method | S
-----+---------------------+-----+-----+-----+-----+-----+
 public | project_tasks_1  | table | postgres | permanent  | heap          | 1
 public | project_tasks_10 | table | postgres | permanent  | heap          | 1
 public | project_tasks_2  | table | postgres | permanent  | heap          | 1
 public | project_tasks_3  | table | postgres | permanent  | heap          | 1
 public | project_tasks_4  | table | postgres | permanent  | heap          | 1
 public | project_tasks_5  | table | postgres | permanent  | heap          | 1
 public | project_tasks_6  | table | postgres | permanent  | heap          | 1
 public | project_tasks_7  | table | postgres | permanent  | heap          | 1
 public | project_tasks_8  | table | postgres | permanent  | heap          | 1
 public | project_tasks_9  | table | postgres | permanent  | heap          | 1
(10 rows)
```

```
[postgres@ip-172-31-92-215 ~]$ psql -c "\di+"
```

Schema	Name	Type	Owner	Table	Persi
<hr/>					
public	idx_project_tasks_10_id	index	postgres	project_tasks_10	perma
public	idx_project_tasks_1_id	index	postgres	project_tasks_1	perma
public	idx_project_tasks_2_id	index	postgres	project_tasks_2	perma
public	idx_project_tasks_3_id	index	postgres	project_tasks_3	perma
public	idx_project_tasks_4_id	index	postgres	project_tasks_4	perma
public	idx_project_tasks_5_id	index	postgres	project_tasks_5	perma
public	idx_project_tasks_6_id	index	postgres	project_tasks_6	perma
public	idx_project_tasks_7_id	index	postgres	project_tasks_7	perma
public	idx_project_tasks_8_id	index	postgres	project_tasks_8	perma
public	idx_project_tasks_9_id	index	postgres	project_tasks_9	perma
(10 rows)					

[postgres@ip-172-31-92-215 ~]\$

This will list table and index sizes. You should notice:

- Decreased index sizes (e.g., a 70% reduction in some cases)
- Compact and optimized index structures
- Improved query performance on indexed fields

Optional: Run `EXPLAIN ANALYZE` before and after reindexing on typical queries to benchmark improvement.

Final Thoughts

This hands-on exercise demonstrates how even the most well-tuned PostgreSQL database can silently accumulate **index bloat** — especially in write-heavy workloads.

Through this step-by-step walkthrough, we have:

- Created large tables (`project_tasks_*`) with indexes
- Simulated index fragmentation by deleting data
- Generated dynamic `REINDEX TABLE CONCURRENTLY` commands
- Executed them safely on a live database

- Validated storage and performance improvements

Takeaways for Real-World DBAs

- Automate index health checks as part of your regular maintenance schedule.
- Use dynamic SQL to batch-generate reindex commands.
- Always prefer `REINDEX CONCURRENTLY` in production environments to minimize downtime.
- Combine reindexing with monitoring tools (like `pgstattuple`, `pg_stat_user_indexes`, or size differencing) to track and optimize index usage.

When Should You Reindex?

In PostgreSQL, reindexing is not something that needs to be done daily — but knowing when to reindex is critical for maintaining database performance and storage efficiency. Indexes in PostgreSQL can become bloated over time due to the nature of MVCC (Multi-Version Concurrency Control). The key is to recognize the right scenarios where reindexing is beneficial.

Let's explore the most common triggers:

Frequent Updates or Deletes

Whenever rows are updated or deleted in PostgreSQL, the old row versions remain in the index until cleaned up. These “dead tuples” accumulate, increasing the index size and decreasing lookup speed.

If your workload includes:

- Constant updates to customer records,
- Frequent deletions of temporary or expired data,
- Heavy batch jobs that modify large tables,

...then your indexes are likely accumulating fragmentation. **Reindexing is highly recommended in such cases.**

Visible Index Bloat

If you observe that the **index size has grown disproportionately large compared to the data it supports**, that's a classic sign of bloat. Even after `VACUUM` or `ANALYZE`, bloated indexes won't shrink — because the space is only reclaimed by rebuilding the index.

You can identify bloated indexes using tools like:

```
SELECT pg_class.relname AS index_name,
       pg_size.pretty(pg_relation_size(pg_class.oid)) AS index_size
  FROM pg_stat_user_indexes
 JOIN pg_class ON pg_stat_user_indexes.indexrelid = pg_class.oid
 ORDER BY pg_relation_size(pg_class.oid) DESC;
```

```
postgres=# SELECT pg_class.relname AS index_name,
      pg_size.pretty(pg_relation_size(pg_class.oid)) AS index_size
    FROM pg_stat_user_indexes
   JOIN pg_class ON pg_stat_user_indexes.indexrelid = pg_class.oid
  ORDER BY pg_relation_size(pg_class.oid) DESC;
      index_name      | index_size
-----+-----
 idx_project_tasks_1_id | 8192 bytes
 idx_project_tasks_2_id | 8192 bytes
 idx_project_tasks_3_id | 8192 bytes
 idx_project_tasks_4_id | 8192 bytes
 idx_project_tasks_5_id | 8192 bytes
 idx_project_tasks_6_id | 8192 bytes
 idx_project_tasks_7_id | 8192 bytes
 idx_project_tasks_8_id | 8192 bytes
 idx_project_tasks_9_id | 8192 bytes
 idx_project_tasks_10_id | 8192 bytes
```

```
(10 rows)
```

```
postgres=#
```

Query Performance Starts to Drop

If queries that rely on indexed columns become unexpectedly slow, and query plans show **sequential scans instead of index scans**, this could mean:

- The index has become inefficient due to bloat.
- The PostgreSQL planner is avoiding the index due to its poor selectivity.

Reindexing can **rebuild the index tree**, reset planner statistics, and **help PostgreSQL make better decisions**.

Disk Usage Spikes

Bloated indexes take up unnecessary disk space. When index files grow rapidly without a proportional increase in actual data, it's time to reindex.

If your monitoring tools or `pg_total_relation_size()` show high growth in index sizes, reindexing can **free up gigabytes** of space — especially in OLTP workloads.

Rarely Updated or Static Tables

On the other hand, if a table is rarely updated and its data remains stable (e.g., country lists, configuration values, audit logs), then its indexes are not likely to bloat. Reindexing these tables provides **minimal benefit** and can be skipped unless space or performance issues are observed.

PostgreSQL Reindexing: Quick Summary

Let's summarize the **key benefits** of reindexing in PostgreSQL:

Prevents Index Bloat

Indexes can become filled with dead entries that slow down queries and waste disk. Reindexing completely **rebuilds the index**, removing fragmentation.

Restores Query Performance

Queries that previously relied on bloated indexes will benefit from faster lookups and more efficient execution plans after reindexing.

Saves Disk Space

Reindexing shrinks index files by eliminating unused pages and compacting structure, which is especially valuable on cloud platforms where storage costs matter.

Ensures Long-Term PostgreSQL Stability

Databases that run for years without index maintenance will degrade. Proactive reindexing ensures **consistent performance**, fewer surprises, and better user experience.

Conclusion: Reindexing for a Healthier PostgreSQL

PostgreSQL's MVCC model gives it exceptional concurrency and consistency. However, this design can lead to **inevitable index degradation** over time. As your workload scales, so do your **performance risks** if bloat is left unchecked.

Reindexing is the solution.

And PostgreSQL 17 makes it even more powerful with enhanced **REINDEX CONCURRENTLY** support — meaning you can **rebuild indexes without disrupting production workloads**.

With smart reindexing, you can:

-  Avoid costly query slowdowns due to bloated indexes
-  Reduce disk consumption — often by 50% or more for bloated tables
-  Maintain replication health, as large indexes can delay WAL shipping
-  Minimize maintenance windows by using non-blocking concurrent rebuilds
-  Ensure long-term PostgreSQL stability and performance

🎯 Final Takeaway for DBAs

Think of reindexing as a **routine database tune-up**. Just like cars need oil changes, databases need index maintenance. Integrate reindexing into your quarterly or monthly database maintenance cycle — especially for high-write systems.

- ✓ Don't wait for slowness.
- ✓ Don't let indexes silently bloat.
- ✓ Make reindexing a part of your PostgreSQL hygiene checklist.

Your database will thank you.

📢 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) 👉 <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

AWS

Oracle

Database

J

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

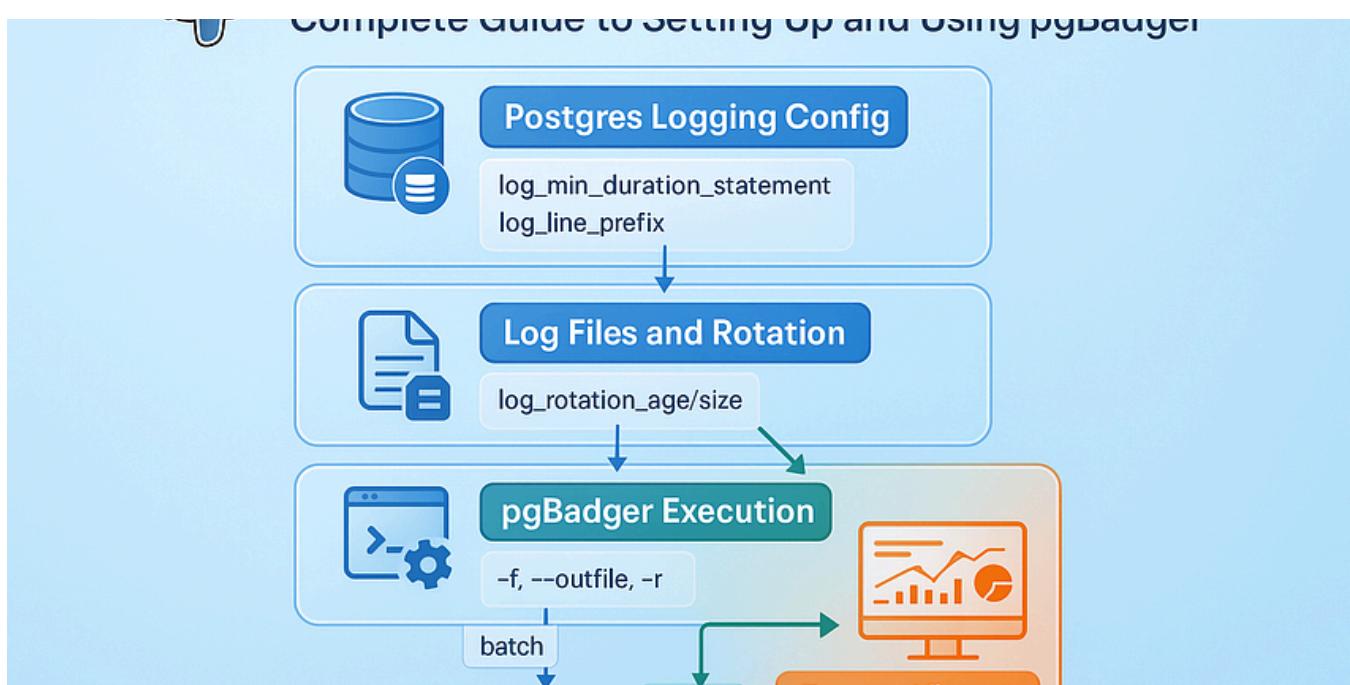
No responses yet



Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy



J Jeyaram Ayyalusamy 

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

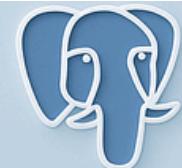
Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

Jun 23  52

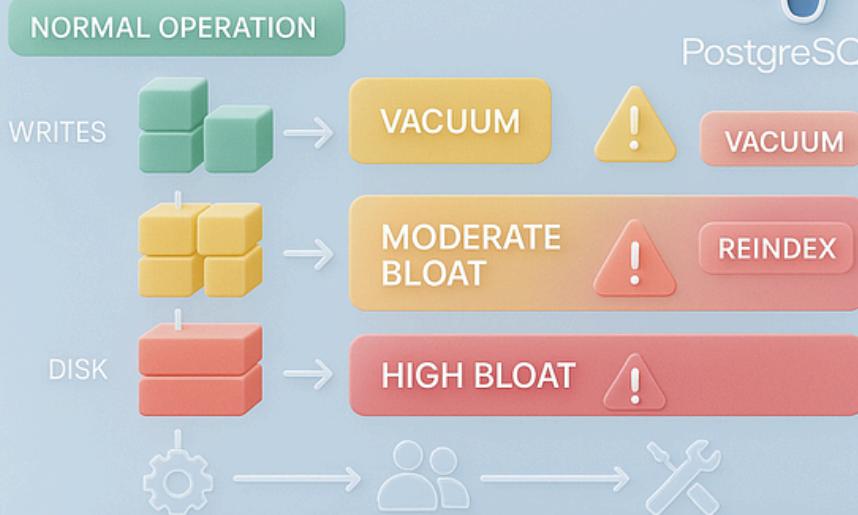


...

Mastering WAL in PostgreSQL 17: Complete Guide for DBAs



PostgreSQL



J Jeyaram Ayyalusamy 

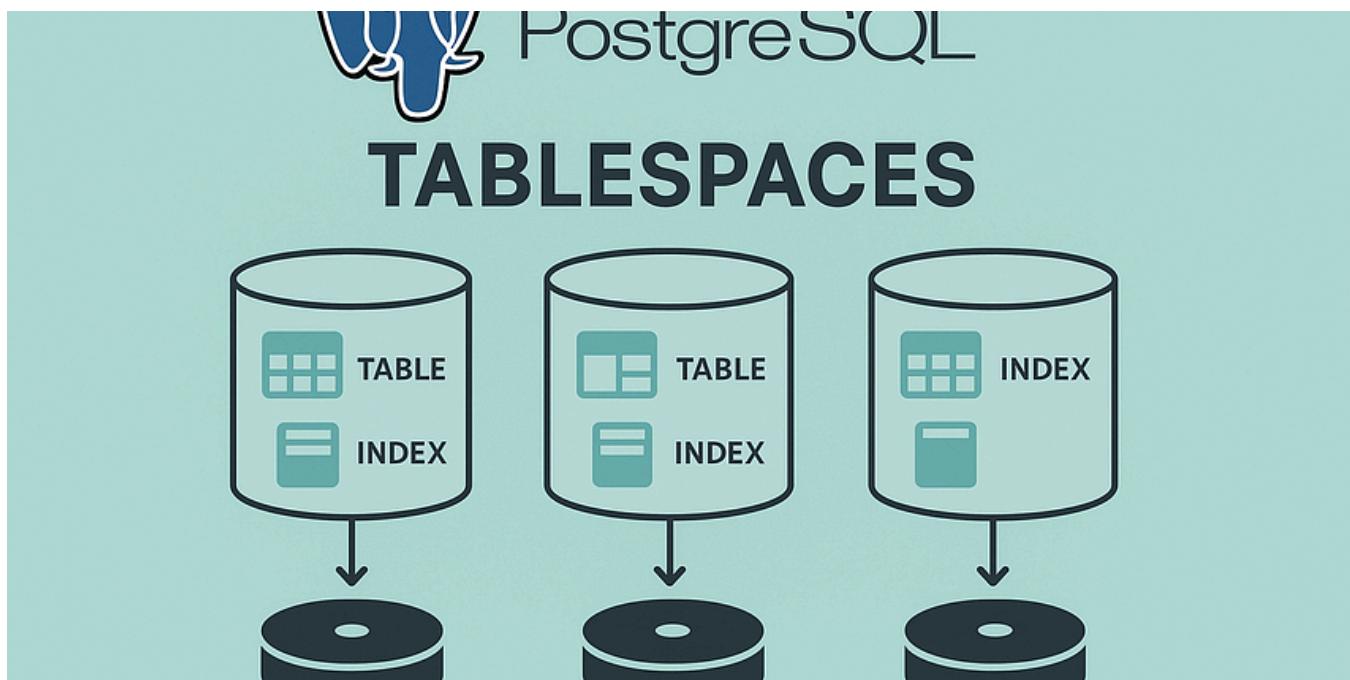
Understanding and Managing Bloating in PostgreSQL: A Complete Guide for DBAs

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

Jun 25  52



...



J Jeyaram Ayyalusamy

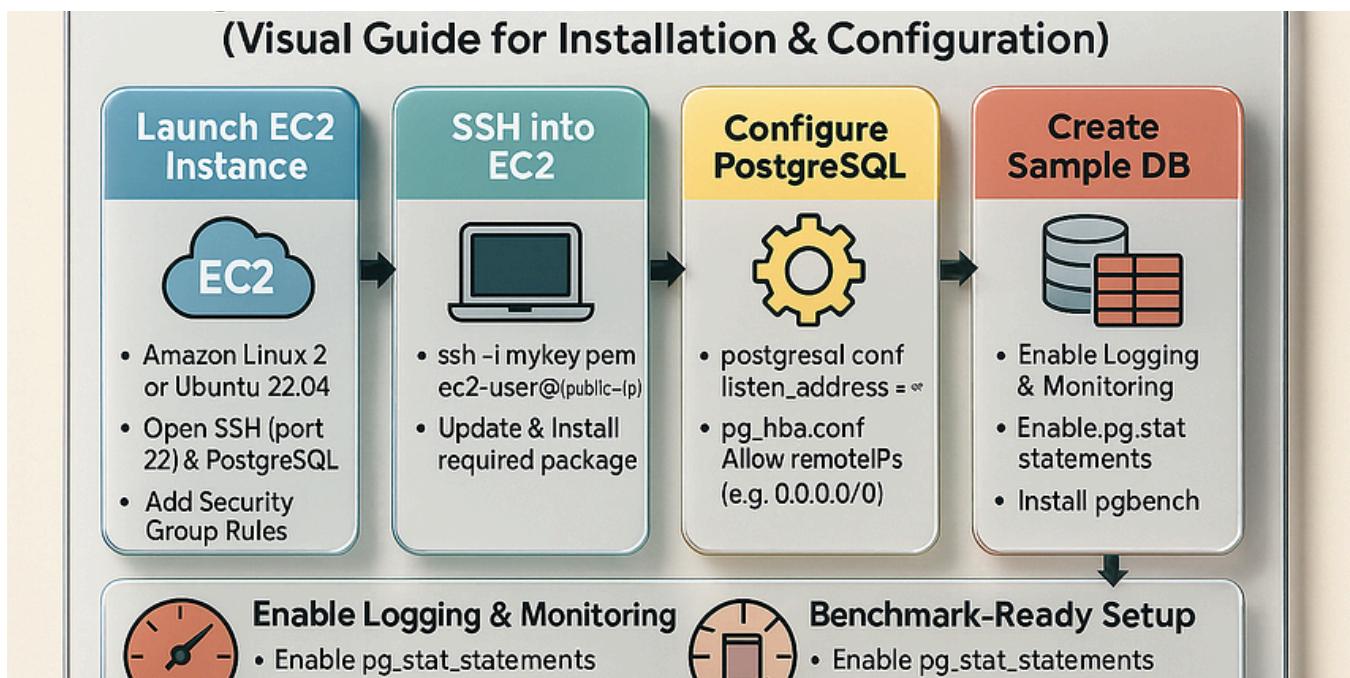
PostgreSQL Tablespaces Explained: Complete Guide for PostgreSQL 17 DBAs

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

Jun 12 8



...



J Jeyaram Ayyalusamy

PostgreSQL 17 on AWS EC2—Full Installation & Configuration Walkthrough

Setting up PostgreSQL 17 on AWS EC2 might seem complex at first—but once you break down each component, it becomes an efficient and...

1d ago 👏 50



...

See all from Jeyaram Ayyalusamy

Recommended from Medium



 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



...

The screenshot shows a PostgreSQL performance tuning interface. On the left, there's a sidebar with sections like 'PostgreSQL Features', 'PostgreSQL Metrics', 'PostgreSQL Tools', and 'PostgreSQL Configuration'. The main area features a large blue hexagonal logo. Below it, there are several cards with metrics: 'PostgreSQL Version' (16.0), 'PostgreSQL Status' (OK), 'PostgreSQL Memory Usage' (1.2GB / 1.5GB), 'PostgreSQL CPU Usage' (0.5%), 'PostgreSQL Disk Usage' (100%), 'PostgreSQL Network Usage' (0.0%), and 'PostgreSQL Locks' (0%). At the bottom, there are links for 'PostgreSQL Performance Tuning' and 'PostgreSQL Configuration'.

Rizqi Mulki

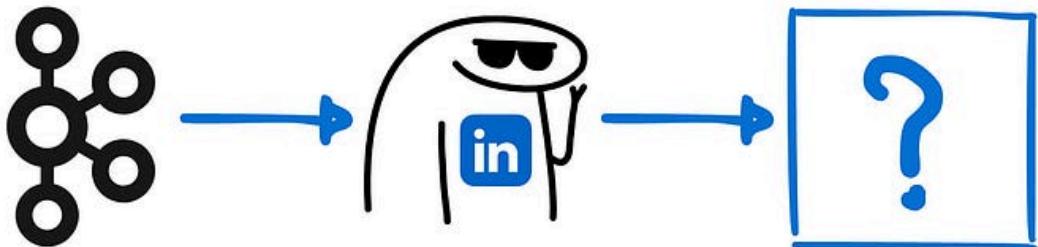
Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55



LinkedIn is moving from Kafka to this



The company that created Kafka is replacing it with a new solution

In Data Engineer Things by Vu Trinh

The company that created Kafka is replacing it with a new solution

How did LinkedIn build Northguard, the new scalable log storage

Jul 17 330 6

```

1 explain
2 select *
3 from payment_lab
4 where customer_id=10 ;

```

Statistics 1 Results 2

explain select * from payment_lab where custon Enter a SQL expression

Grid	QUERY PLAN
1	Seq Scan on payment_lab (cost=0.00..290.45 rows=23 width=26)
2	Filter: (customer_id = 10)

Muhammet Kurtoglu

Postgresql Query Performance Analysis

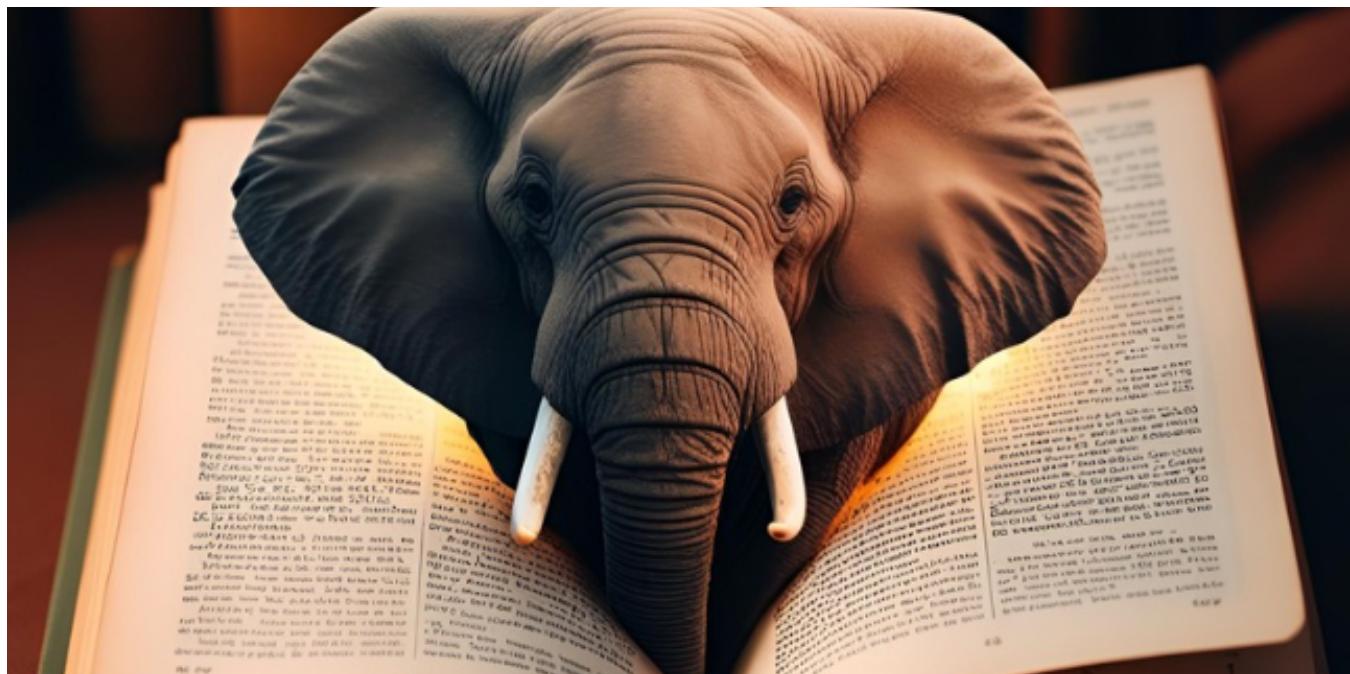
SQL tuning is critically important for ensuring database performance, reliability, and scalability. In most cases, performance issues in a...

6d ago 10

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

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...

star May 14 clapping 58 comment 1



...

See more recommendations