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

J

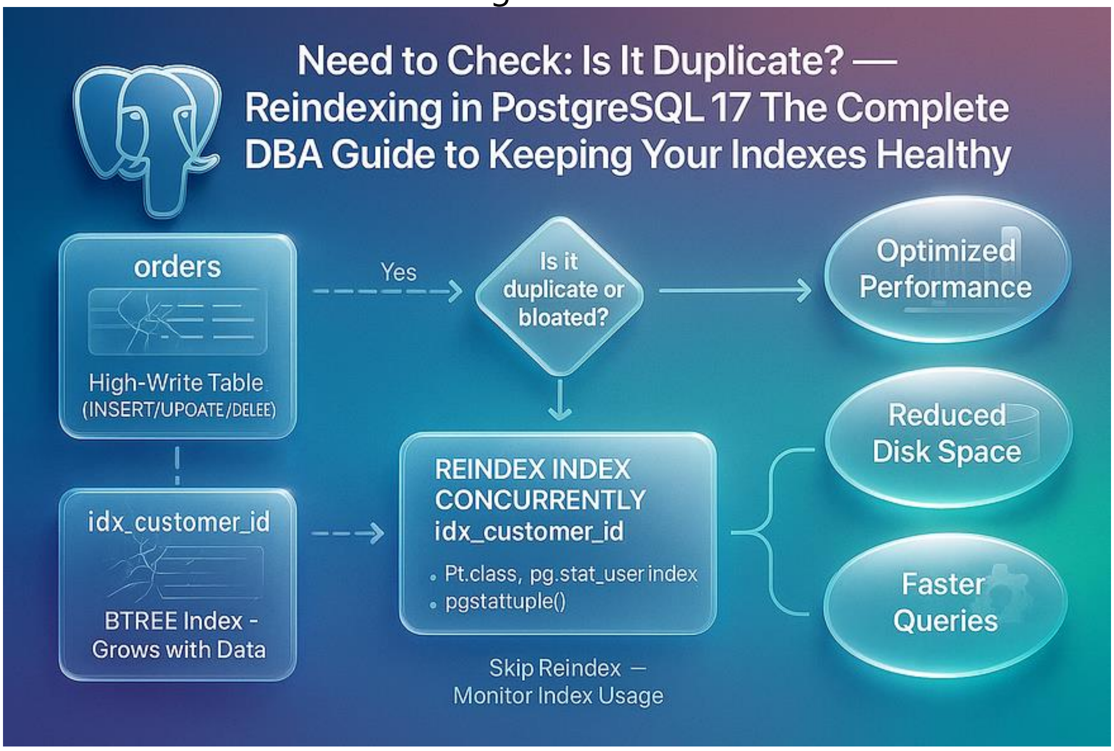
Jeyaram Ayyalusamy

16 min read

·

Jul 2, 2025

Press enter or click to view image in full size



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 **frequent 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.

1️⃣ 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.

Press enter or click to view image in full size

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.

Press enter or click to view image in full size

3️⃣ 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.

4️⃣ 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(1,2000000);"
psql -c "CREATE TABLE project_tasks_3 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
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(1,2000000);"
psql -c "CREATE TABLE project_tasks_6 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
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(1,2000000);"
psql -c "CREATE TABLE project_tasks_9 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
psql -c "CREATE TABLE project_tasks_10 AS SELECT generate_series AS task_id FROM
generate_series(1,10000000);"



[postgres@ip-172-31-92-215 ~]$
[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);"
psql -c "CREATE TABLE project_tasks_2 AS SELECT generate_series AS task_id FROM
generate_series(1,2000000);"
psql -c "CREATE TABLE project_tasks_3 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
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(1,2000000);"
psql -c "CREATE TABLE project_tasks_6 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
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(1,2000000);"
psql -c "CREATE TABLE project_tasks_9 AS SELECT generate_series AS task_id FROM
generate_series(1,3000000);"
psql -c "CREATE TABLE project_tasks_10 AS SELECT generate_series AS task_id FROM
generate_series(1,10000000);"
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
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);"
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
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;
                         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;"
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_10;"



[postgres@ip-172-31-92-215 ~]$
[postgres@ip-172-31-92-215 ~]$ 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;"
psql -c "REINDEX TABLE CONCURRENTLY project_tasks_10;"
REINDEX
REINDEX
REINDEX
REINDEX
REINDEX
REINDEX
REINDEX
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 | Size
| Description
--------+----------------------+-------+----------+-------------+---------------+-------
```

```
+-------------
 public | project_tasks_1  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_10 | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_2  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_3  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_4  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_5  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_6  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_7  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_8  | table | postgres | permanent   | heap           | 16 kB
|
 public | project_tasks_9  | table | postgres | permanent   | heap           | 16 kB
|
(10 rows)

[postgres@ip-172-31-92-215 ~]$ psql -c "\di+"
                                                  List of relations
 Schema |          Name          | Type  |  Owner   |      Table       |
Persistence | Access method |    Size    | Description
--------+------------------------+-------+----------+------------------+-----------
--+---------------+------------+-------------
 public | idx_project_tasks_10_id | index | postgres | project_tasks_10 | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_1_id  | index | postgres | project_tasks_1  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_2_id  | index | postgres | project_tasks_2  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_3_id  | index | postgres | project_tasks_3  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_4_id  | index | postgres | project_tasks_4  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_5_id  | index | postgres | project_tasks_5  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_6_id  | index | postgres | project_tasks_6  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_7_id  | index | postgres | project_tasks_7  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_8_id  | index | postgres | project_tasks_8  | permanent
| btree         | 8192 bytes |
 public | idx_project_tasks_9_id  | index | postgres | project_tasks_9  | permanent
| btree         | 8192 bytes |
(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 diffing) 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=#
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.