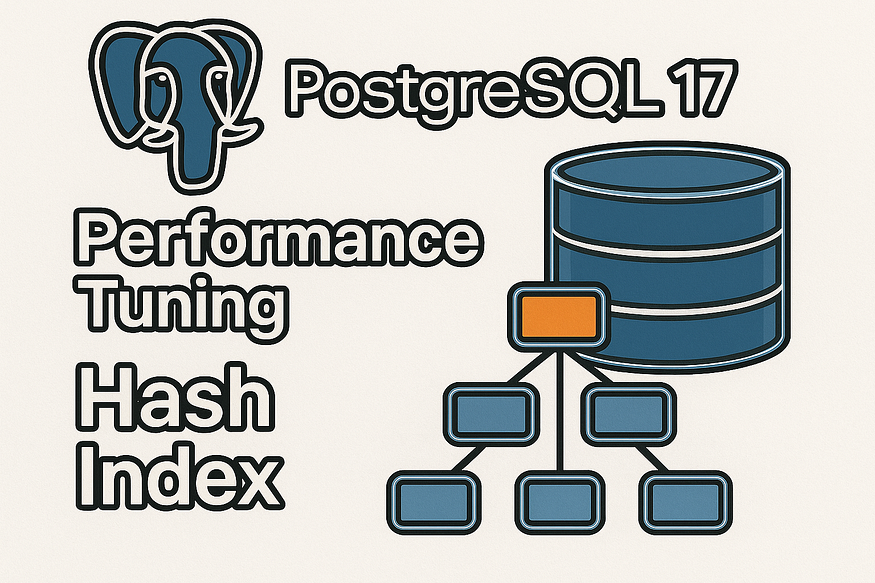
# **18 - PostgreSQL 17 Performance Tuning: Hash Index**



Indexes are the backbone of database performance tuning. In PostgreSQL, the default index type is ****B-Tree****, which handles a wide variety of queries very efficiently. But sometimes, a different type of index is a better fit.

## **What is a Hash Index in PostgreSQL?**

A ****Hash Index**** is a special type of index in PostgreSQL that is optimized for ****equality lookups**** — queries where you check if a column equals a specific value (e.g., WHERE email = 'user123@example.com').

Instead of storing values in a sorted order like a ****B-Tree index****, a hash index uses a ****hashing function****. The value you search for is hashed, and PostgreSQL jumps directly to the matching location in the index.

## **Key Characteristics**

* ****Best for = operator****  
  Works only for equality conditions. You cannot use it for range queries (<, >, BETWEEN) or sorting.
* ****Fast lookups****  
  For exact matches, hash indexes can be extremely fast, especially with large datasets.
* ****Compact structure****  
  They don’t need to maintain order, so storage is usually smaller compared to B-Tree indexes.
* ****Production-ready in PostgreSQL 17****  
  Older PostgreSQL versions had limitations: hash indexes weren’t WAL-logged (not crash safe). Starting from PostgreSQL 10 and now in ****PostgreSQL 17****, they are fully durable and safe to use.

## **When to Use Hash Indexes**

✅ Use when:

* You often query with equality checks (=).
* The column has many unique values (e.g., email, username, ID).

❌ Avoid when:

* You need range lookups (>, <, BETWEEN).
* You want the index to support sorting or ordering.
* You already benefit from a ****B-Tree index****, which supports equality and ranges.

👉 In short:  
A ****Hash Index**** in PostgreSQL is a ****special-purpose index**** that makes equality lookups ****blazing fast****, but it’s not a general replacement for the default ****B-Tree index****.

## **Step 1: Creating the**employees**table**

We’ll start by creating a simple employees table. It contains employee details, including email, which will be the focus of our equality lookups.

CREATE TABLE employees (  
 emp\_id BIGINT,  
 emp\_name TEXT,  
 email TEXT,  
 dept TEXT,  
 salary NUMERIC  
);

postgres=# CREATE TABLE employees (  
 emp\_id BIGINT,  
 emp\_name TEXT,  
 email TEXT,  
 dept TEXT,  
 salary NUMERIC  
);  
CREATE TABLE  
postgres=#

📌 Here:

* emp\_id is a unique identifier.
* email will be queried often, and we will later add a ****hash index**** on it.
* Other columns (name, dept, salary) are for realism and testing.

## **Step 2: Inserting 10 million rows**

To test performance properly, we need a ****large dataset****. We’ll generate 10 million employee records with distinct email addresses.

-- Insert 10 million rows with random emails and departments  
INSERT INTO employees (emp\_name, email, dept, salary)  
SELECT  
 'Employee\_' || g,  
 'user' || g || '@example.com',  
 'Dept\_' || (g % 20), -- 20 departments  
 (random()\*100000)::NUMERIC(10,2) -- salary between 0 and 100k  
FROM generate\_series(1, 10000000) g;

postgres=# INSERT INTO employees (emp\_name, email, dept, salary)  
SELECT  
 'Employee\_' || g,  
 'user' || g || '@example.com',  
 'Dept\_' || (g % 20), -- 20 departments  
 (random()\*100000)::NUMERIC(10,2) -- salary between 0 and 100k  
FROM generate\_series(1, 10000000) g;  
INSERT 0 10000000  
postgres=#

✅ After this step, the table has 10 million rows, each with a unique email (user1@example.com, user2@example.com, …, user10000000@example.com).

## **Step 3: Analyzing the table**

Whenever you bulk-load data, it’s important to run ****ANALYZE****. This updates the PostgreSQL query planner with accurate statistics.

ANALYZE employees;

postgres=# ANALYZE employees;  
ANALYZE  
postgres=#

Without this step, the planner might guess row counts incorrectly and choose suboptimal execution plans.

## **Step 4: Running a query without an index**

Let’s run a simple lookup: find the row where email = 'user5000000@example.com'.

EXPLAIN ANALYZE   
SELECT \* FROM employees WHERE email = 'user5000000@example.com';

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM employees WHERE email = 'user5000000@example.com';  
 QUERY PLAN  
------------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..164395.43 rows=1 width=62) (actual time=4931.459..4936.369 rows=1 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on employees (cost=0.00..163395.33 rows=1 width=62) (actual time=4107.889..4913.315 rows=0 loops=3)  
 Filter: (email = 'user5000000@example.com'::text)  
 Rows Removed by Filter: 3333333  
 Planning Time: 0.469 ms  
 Execution Time: 4936.386 ms  
(8 rows)  
  
postgres=#

Time: 4936.386 ms (00:04.936)  
postgres=#

📊 What happened?

* PostgreSQL performed a ****sequential scan****, reading all 10M rows.
* Only ****one row**** matched, but 3,333,333 rows had to be checked and discarded.
* The execution time was ~4.9 seconds — too slow for real-world workloads.

## **Step 5: Creating a hash index**

Now, let’s create a ****Hash Index**** on the email column.

CREATE INDEX idx\_employee\_email\_hash ON employees USING hash (email);

postgres=# CREATE INDEX idx\_employee\_email\_hash ON employees USING hash (email);  
CREATE INDEX  
postgres=#

📌 Explanation:

* USING hash tells PostgreSQL to build a ****hash-based index**** instead of the default B-Tree.
* Hash indexes are optimized for ****equality checks****, not range scans.

## **Analyzing the table**

Whenever you bulk-load data or creating new indexes, it’s important to run ****ANALYZE****. This updates the PostgreSQL query planner with accurate statistics.

ANALYZE employees;

postgres=# ANALYZE employees;  
ANALYZE  
postgres=#

## **Step 6: Query plan after using the hash index**

Run the same query again:

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM employees WHERE email = 'user5000000@example.com';  
 QUERY PLAN  
------------------------------------------------------------------------------------------------------------------------------------  
 Index Scan using idx\_employee\_email\_hash on employees (cost=0.00..8.02 rows=1 width=62) (actual time=0.012..0.013 rows=1 loops=1)  
 Index Cond: (email = 'user5000000@example.com'::text)  
 Planning Time: 0.130 ms  
 Execution Time: 0.030 ms  
(4 rows)  
  
postgres=#

Time: 0.030 ms  
postgres=#

🔥 Huge improvement:

* The sequential scan is replaced with an ****Index Scan****.
* The database jumps directly to the matching row using the hash index.
* Execution time dropped from ****~5.2 seconds to less than 1 millisecond****.

## **Step 7: Understanding when to use Hash Indexes**

Hash indexes are ****specialized****. They are not a replacement for B-Tree indexes, but they excel in certain cases:

✅ ****When Hash Indexes Shine:****

* Queries with equality conditions (=).
* Lookups on columns with high cardinality (many unique values), like email, username, or UUID.

❌ ****When Not to Use:****

* Range queries (<, >, BETWEEN) → Use B-Tree instead.
* Sorting or ordering → B-Tree indexes are required.

## **Step 8: Improvements in PostgreSQL 17**

Historically, hash indexes had limitations (not WAL-logged, crash unsafe). But now:

* ****WAL-logged:**** Hash indexes survive crashes safely.
* ****VACUUM/ANALYZE aware:**** They integrate smoothly with normal maintenance.
* ****Production-ready:**** Fully safe and recommended for workloads needing equality lookups.

## **Step 9: Verifying index size**

Sometimes indexes can be large. You can check how much space they take compared to the table:

SELECT   
 pg\_size\_pretty(pg\_relation\_size('employees')) AS table\_size,  
 pg\_size\_pretty(pg\_indexes\_size('employees')) AS indexes\_size,  
 pg\_size\_pretty(pg\_total\_relation\_size('employees')) AS total\_size;

postgres=# SELECT  
 pg\_size\_pretty(pg\_relation\_size('employees')) AS table\_size,  
 pg\_size\_pretty(pg\_indexes\_size('employees')) AS indexes\_size,  
 pg\_size\_pretty(pg\_total\_relation\_size('employees')) AS total\_size;  
 table\_size | indexes\_size | total\_size  
------------+--------------+------------  
 870 MB | 256 MB | 1126 MB  
(1 row)  
  
postgres=#

This helps you understand the ****storage overhead**** of adding hash indexes.

## **Final Thoughts**

This experiment shows how ****Hash Indexes in PostgreSQL 17**** can make equality lookups ****thousands of times faster****.

* Without indexes, queries degrade to slow sequential scans.
* With a hash index, lookups become nearly instant.
* The choice of index type matters — ****match the index to the query pattern****.

👉 If your workload involves frequent equality lookups on columns like email, username, or unique IDs, hash indexes are a powerful option.