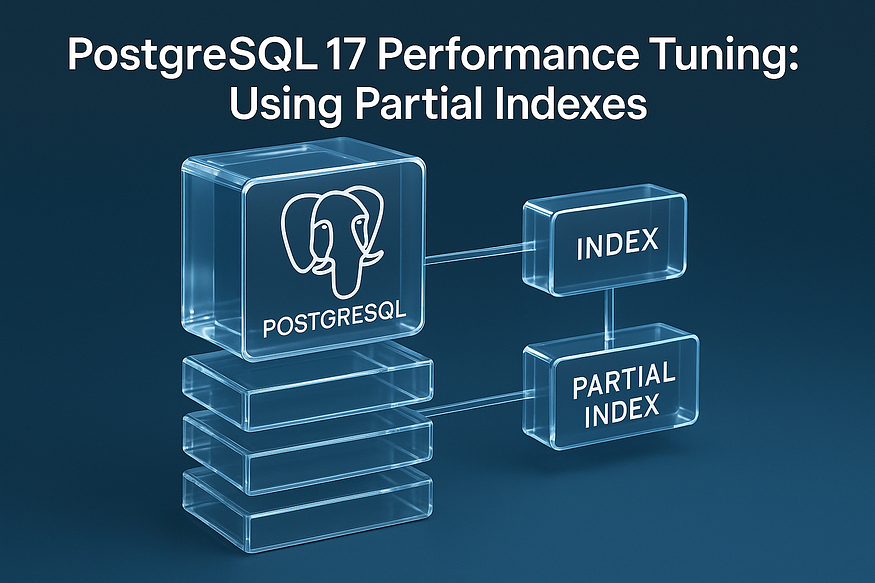
# **17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes**



Even though we usually create indexes on all rows of a table, PostgreSQL also allows you to create an index on ****just a subset of records****. This type of index is called a ****partial index****.

A partial index is smaller than a full index because it only covers rows that match a condition. Since there’s less data in the index, it is faster to scan and cheaper to maintain. This is especially useful when queries consistently filter for a ****small subset**** of rows.

## **Why Use Partial Indexes?**

Imagine a ****task management system**** stored in a products-like table. Rows representing tasks still pending are flagged as ****open****, while completed ones are marked as ****done****.

Over time, most rows will be marked ****done****, and only a small fraction will be ****open****. But in practice, users will frequently query the ****open**** tasks.

👉 If we create a full index on the status column, PostgreSQL will index all 10 million rows (both open and done). This consumes space and slows down writes, even though most queries only care about the small subset (open).

👉 Instead, if we create a ****partial index only for status = 'open'****, PostgreSQL can use a much smaller, faster index, while ignoring rows that don’t matter.

## **Step 1: Create Products Table with 10 Million Rows**

CREATE TABLE products (  
 product\_id BIGSERIAL PRIMARY KEY,  
 product\_name TEXT,  
 category TEXT,  
 status TEXT, -- 'open' or 'done'  
 price NUMERIC,  
 stock\_qty INT  
);

postgres=# CREATE TABLE products (  
 product\_id BIGSERIAL PRIMARY KEY,  
 product\_name TEXT,  
 category TEXT,  
 status TEXT, -- 'open' or 'done'  
 price NUMERIC,  
 stock\_qty INT  
);  
CREATE TABLE  
postgres=#

-- 1M rows with status = 'open' (~10%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT   
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'open'  
FROM generate\_series(1, 1000000) g;  
  
-- 2.5M rows with status = 'done' (~25%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT   
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'done'  
FROM generate\_series(1000001, 3500000) g;  
  
-- 3M rows with status = 'denied' (~30%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT   
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'denied'  
FROM generate\_series(3500001, 6500000) g;  
  
-- 3.5M rows with status = 'new' (~35%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT   
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'new'  
FROM generate\_series(6500001, 10000000) g;

postgres=# -- 1M rows with status = 'open' (~10%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'open'  
FROM generate\_series(1, 1000000) g;  
  
-- 2.5M rows with status = 'done' (~25%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'done'  
FROM generate\_series(1000001, 3500000) g;  
  
-- 3M rows with status = 'denied' (~30%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'denied'  
FROM generate\_series(3500001, 6500000) g;  
  
-- 3.5M rows with status = 'new' (~35%)  
INSERT INTO products (product\_name, category, price, stock\_qty, status)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 50),  
 round((random() \* 500)::numeric, 2),  
 (random() \* 1000)::int,  
 'new'  
FROM generate\_series(6500001, 10000000) g;  
  
INSERT 0 1000000  
INSERT 0 2500000  
INSERT 0 3000000  
INSERT 0 3500000  
postgres=#  
postgres=# ANALYZE products;  
ANALYZE  
postgres=#

postgres=# SELECT status, COUNT(\*)  
FROM products  
GROUP BY status;  
 status | count  
--------+---------  
 denied | 3000000  
 done | 2500000  
 new | 3500000  
 open | 1000000  
(4 rows)

## **Step 2: Full Index vs Partial Index**

## **Full Index on Status**

CREATE INDEX idx\_products\_status ON products(status);

postgres=# CREATE INDEX idx\_products\_status ON products(status);  
CREATE INDEX  
postgres=#

postgres=# analyze products;  
ANALYZE  
postgres=#

Covers ****all 10M rows****.

* Size is large (hundreds of MBs).
* Every insert/update must maintain this index, even for rows where status = 'done'.

## **Query Example**

Now run a query for open tasks:

EXPLAIN ANALYZE  
SELECT \* FROM products WHERE status = 'open';

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM products WHERE status = 'open';  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on products (cost=10979.37..203257.63 rows=985669 width=49) (actual time=24.562..5823.624 rows=1000000 loops=1)  
 Recheck Cond: (status = 'open'::text)  
 Heap Blocks: exact=10310  
 -> Bitmap Index Scan on idx\_products\_status (cost=0.00..10732.95 rows=985669 width=0) (actual time=22.815..22.816 rows=1000000 loops=1)  
 Index Cond: (status = 'open'::text)  
 Planning Time: 1.067 ms  
 Execution Time: 6343.627 ms  
(7 rows)  
  
postgres=#

## **Partial Index on Status = ‘open’**

DROP INDEX idx\_products\_status;

postgres=# DROP INDEX idx\_products\_status;  
DROP INDEX  
postgres=#

CREATE INDEX idx\_products\_open ON products(status) WHERE status = 'open';

postgres=# CREATE INDEX idx\_products\_open ON products(status) WHERE status = 'open';  
CREATE INDEX  
postgres=#

analyze products;

postgres=# analyze products;  
ANALYZE  
postgres=#

* Covers only ****1M rows**** (10% of total).
* Much smaller, cheaper to maintain.
* Perfect for queries that only look at open rows.

## **Step 3: Query Example**

Now run a query for open tasks:

EXPLAIN ANALYZE  
SELECT \* FROM products WHERE status = 'open';

postgres=# EXPLAIN ANALYZE  
SELECT \* FROM products WHERE status = 'open';  
 QUERY PLAN  
------------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on products (cost=8417.18..200578.87 rows=956335 width=49) (actual time=24.340..706.699 rows=1000000 loops=1)  
 Recheck Cond: (status = 'open'::text)  
 Heap Blocks: exact=10310  
 -> Bitmap Index Scan on idx\_products\_open (cost=0.00..8178.10 rows=956335 width=0) (actual time=23.030..23.031 rows=1000000 loops=1)  
 Planning Time: 1.010 ms  
 Execution Time: 1294.439 ms  
(6 rows)  
  
postgres=#

👉 PostgreSQL will use idx\_products\_open, skipping all the done rows. The query is much faster because the index is tiny.

If you query for status = 'done', PostgreSQL won’t use this index — it will either do a sequential scan or use another index, depending on the query.

## **Step 4: Measuring Index Space**

Check how much space indexes use compared to the table:

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

postgres=# SELECT  
 pg\_size\_pretty(pg\_relation\_size('products')) AS table\_only,  
 pg\_size\_pretty(pg\_indexes\_size('products')) AS indexes\_only,  
 pg\_size\_pretty(pg\_total\_relation\_size('products')) AS total\_with\_indexes;  
 table\_only | indexes\_only | total\_with\_indexes  
------------+--------------+--------------------  
 805 MB | 221 MB | 1027 MB  
(1 row)  
  
postgres=#

It’s common to see indexes taking ****more space than the table itself****. That’s why it’s so important to measure index usage vs cost. Partial indexes help reduce wasted space.

## **When to Use Partial Indexes**

Partial indexes make sense when:

* Your table has ****a few very common values**** (e.g., status = 'done') and queries mostly target the ****less frequent values**** (e.g., status = 'open').
* The excluded values cover at least ****25% or more**** of the table.
* You know your data patterns well (e.g., gender, nationality, active/inactive users).

Partial indexes trade ****bigger space savings**** and ****faster lookups**** for less flexibility, but when applied wisely, they can make queries significantly faster while reducing index maintenance costs.

## **Key Takeaways**

* A full index on 10M rows can be huge and expensive to maintain.
* A partial index only on frequently queried rows (e.g., status = 'open') is ****smaller, cheaper, and faster****.
* Always measure ****space usage vs query benefit**** before adding indexes.
* PostgreSQL 17 makes partial indexes just as powerful as full ones — but far more efficient in the right scenarios.

✅ Partial indexes are a powerful tuning tool when your workload is focused on a ****small subset of values**** inside a very large table.