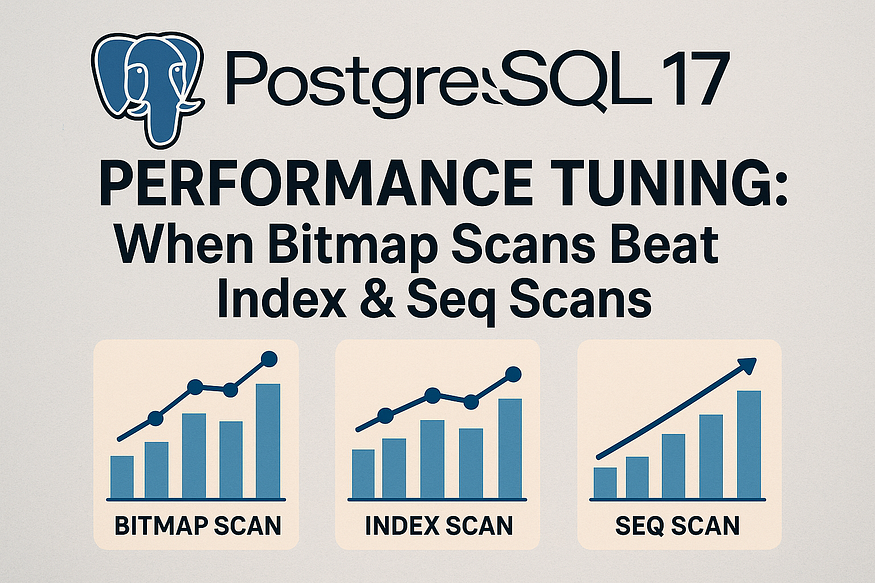
# **14 - PostgreSQL 17 Performance Tuning: When Bitmap Scans Beat Index & Seq Scans**



PostgreSQL chooses among three broad strategies for filtering rows:

* ****Index Scan**** when the result set is tiny (needle in a haystack).
* ****Sequential Scan**** when most rows match (just read the whole table).
* ****Bitmap (Index + Heap) Scan**** for the *in-between*: too many rows for a fast Index Scan, too few to justify a full Seq Scan, or when ****combining multiple indexes****.

Bitmap scans build an in-memory ****bitmap of row locations (TIDs)**** from one or more indexes, then visit each heap page ****once****, pulling all needed rows per page. This dramatically reduces random I/O compared to a plain Index Scan over many matching rows.

Below is a reproducible demo you can paste into psql.

## **PostgreSQL 17 Performance Tuning: Why Bitmap Heap Scan Can Be Slower Than Seq Scan — and How**CLUSTER**Fixes It**

When tuning queries in PostgreSQL, indexes don’t always guarantee faster execution. In fact, sometimes a ****Bitmap Heap Scan with an index**** can be significantly slower than a simple ****Sequential Scan****. In this section, we’ll walk through a hands-on test with ****10 million rows**** and explain the query plans in detail.

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

CREATE TABLE products (  
 product\_id BIGINT,  
 product\_name TEXT,  
 category TEXT,  
 price NUMERIC,  
 stock\_qty INT  
);

This table will hold product information such as name, category, price, and stock quantity.

## **Step 2: Insert 10M rows**

INSERT INTO products (product\_name, category, price, stock\_qty)  
SELECT  
 'Product\_' || g,  
 'Category\_' || (g % 10),   
 (random()\*500)::NUMERIC(10,2),   
 (random()\*100)::INT   
FROM generate\_series(1, 10000000) AS g;

* 10 evenly distributed categories (Category\_0 … Category\_9).
* Random prices between 0.00 and 500.00.
* Stock quantities between 0 and 100.

## **Step 3: Analyze the table**

ANALYZE products;

This updates statistics for the planner.

## **Step 4: Query plan without index (Seq Scan)**

EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;

## **PostgreSQL 17 Performance Tuning: Understanding the Parallel Seq Scan**

When analyzing query performance in PostgreSQL, looking at the execution plan is crucial. Let’s break down the following plan in detail:

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;  
 QUERY PLAN  
----------------------------------------------------------------------------------------------------------------------------------  
 Gather (cost=1000.00..146183.08 rows=97665 width=44) (actual time=3.077..3272.108 rows=100179 loops=1)  
 Workers Planned: 2  
 Workers Launched: 2  
 -> Parallel Seq Scan on products (cost=0.00..135416.58 rows=40694 width=44) (actual time=0.983..3228.192 rows=33393 loops=3)  
 Filter: (stock\_qty = 76)  
 Rows Removed by Filter: 3299940  
 Planning Time: 0.046 ms  
 Execution Time: 3363.344 ms  
(8 rows)

* ****What it means:****  
  The Gather node collects results from multiple parallel workers. PostgreSQL planned for 2 workers, and both were successfully launched.
* ****Why it matters:****  
  This is the coordination point where PostgreSQL merges all rows found by parallel workers and returns them as a single result set.

## **2. Parallel Seq Scan on**products

* ****What it means:****  
  Each worker performed a ****Sequential Scan**** across a different portion of the products table. A sequential scan means PostgreSQL reads rows in physical order directly from the heap.
* ****Rows processed:****
* Each worker scanned about ****3.3 million rows**** (rows=33393 actually returned per worker).
* The filter stock\_qty = 76 discarded the vast majority of rows, as shown in Rows Removed by Filter: 3,299,940.
* ****Why it matters:****  
  Even though every row in the table was touched, parallel workers allowed the job to be divided, making it faster than if a single process had done all the scanning.

## **3. Costs vs. Actual Times**

* ****Cost estimates:****
* cost=1000.00..146183.08 means the planner estimated this query would have a base startup cost of 1000 and a total cost of ~146k.
* ****Actual execution:****
* The query started returning rows after ~3 ms.
* The entire scan finished in about ****3.27 seconds**** (actual time=3.077..3272.108).

👉 The actual time aligns reasonably with planner expectations, which means PostgreSQL chose an appropriate strategy.

## **4. Planning and Execution Time**

* ****Planning Time:**** 0.046 ms — PostgreSQL needed almost no time to decide on this plan.
* ****Execution Time:**** 3363.344 ms — The real cost of scanning 10 million rows and filtering them.

## **Why the Sequential Scan Was Efficient**

Even though scanning the whole table sounds expensive, it performed well here because:

1. ****Parallelism:**** Multiple workers shared the workload.
2. ****Sequential I/O:**** Reading pages in order is much cheaper than jumping around randomly.
3. ****Data selectivity:**** The condition matched about 1% of rows, but scanning was still competitive because sequential reads are highly optimized.

✅ ****Key Takeaway:****  
In PostgreSQL, a sequential scan (especially parallelized) can be faster than an index-based plan when the query touches a large percentage of the table or when rows are spread across many pages.

## **Step 5: Create index on**stock\_qty

CREATE INDEX idx\_products\_stock ON products(stock\_qty);

## **Step 6: Query plan with index (Bitmap Heap Scan)**

EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;

****Query Plan Output****

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on products (cost=1089.35..89823.31 rows=97667 width=44) (actual time=21.747..42485.726 rows=100179 loops=1)  
 Recheck Cond: (stock\_qty = 76)  
 Heap Blocks: exact=58440  
 -> Bitmap Index Scan on idx\_products\_stock (cost=0.00..1064.94 rows=97667 width=0) (actual time=9.309..9.310 rows=100179 loops=1)  
 Index Cond: (stock\_qty = 76)  
 Planning Time: 4.174 ms  
 Execution Time: 42559.194 ms  
(7 rows)  
  
postgres=#

****Explanation:****

* ****Bitmap Index Scan****: Quickly finds ~100k row pointers (TIDs) matching stock\_qty = 76.
* ****Bitmap Heap Scan****: Fetches those rows from the heap but must touch ****58,440 pages**** scattered across the table.
* ****Execution Time (~42.6s)****: Much slower than the Seq Scan because of ****random I/O****.

👉 Lesson: An index doesn’t help if matching rows are scattered everywhere.

## **Step 7: Cluster the Table**

CLUSTER products USING idx\_products\_stock;

When you run the ****CLUSTER**** command in PostgreSQL, the database physically ****rewrites the table**** so that its rows are stored on disk in the same order as the specified index (in this case, idx\_products\_stock).

In our example, rows are reordered so that all entries with the same ****stock\_qty**** value are stored ****contiguously**** on disk.

## **Why Does This Improve Performance?**

1. ****Data Locality****

* Normally, even if you have an index, the corresponding rows may be scattered across different disk pages.
* After clustering, rows with the same stock\_qty are stored together.
* This means fewer page reads are needed when scanning rows that share the same value, reducing I/O.

2. ****Faster Sequential Access****

* PostgreSQL can fetch grouped rows in ****sequential blocks**** instead of random disk access.
* Sequential reads are much faster than random reads, especially on large tables.

3. ****Improved Index Efficiency****

* The index idx\_products\_stock now directly maps to data that is already in physical order.
* This reduces the number of heap fetches, since PostgreSQL doesn’t have to jump around to find matching tuples.

4. ****Better Cache Utilization****

* When rows are stored together, PostgreSQL can load an entire block of related rows into memory.
* This improves buffer cache hit ratio, making repeated queries faster.

5. ****Query Patterns Benefit Most****

* Queries like:

SELECT \* FROM products WHERE stock\_qty = 76;

* or

SELECT \* FROM products WHERE stock\_qty BETWEEN 70 AND 80;

* run significantly faster because the database engine reads ****fewer scattered pages****.

## **Key Considerations**

* ****One-time Reorganization****:  
  CLUSTER is not automatic. If new rows are inserted or updated, the physical order can get out of sync again.
* ****Use Periodically****:  
  For frequently queried tables where data distribution changes over time, consider re-clustering during maintenance windows.
* ****Table Locking****:  
  CLUSTER requires an ****exclusive lock****, meaning the table is not available for reads/writes during the operation.

✅ In summary, clustering improves performance by ****reducing I/O, improving sequential access, and aligning physical storage with logical query patterns****. This leads to ****faster lookups and more efficient scans**** for queries that filter or group by the clustered column.

## **Step 8: Query plan after clustering**

EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;

****Query Plan Output****

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;  
 QUERY PLAN  
---------------------------------------------------------------------------------------------------------------------------------------  
 Bitmap Heap Scan on products (cost=1089.35..89823.31 rows=97667 width=44) (actual time=6.071..378.748 rows=100179 loops=1)  
 Recheck Cond: (stock\_qty = 76)  
 Heap Blocks: exact=835  
 -> Bitmap Index Scan on idx\_products\_stock (cost=0.00..1064.94 rows=97667 width=0) (actual time=5.550..5.551 rows=100179 loops=1)  
 Index Cond: (stock\_qty = 76)  
 Planning Time: 2.360 ms  
 Execution Time: 430.219 ms  
(7 rows)  
  
postgres=#

****Explanation:****

* The Bitmap Index Scan still identifies 100k rows.
* But this time, only ****835 heap pages**** need to be fetched (instead of 58k).
* ****Execution Time (~0.43s)****: Dramatic improvement because rows are now ****physically contiguous****.

## **Step 9: Analyze after clustering**

ANALYZE products;

## **Step 10: Query plan after Analyze (Index Scan)**

EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;

****Query Plan Output****

postgres=# EXPLAIN ANALYZE SELECT \* FROM products WHERE stock\_qty = 76;  
 QUERY PLAN  
--------------------------------------------------------------------------------------------------------------------------------------------  
 Index Scan using idx\_products\_stock on products (cost=0.43..3170.28 rows=108334 width=44) (actual time=0.010..69.161 rows=100179 loops=1)  
 Index Cond: (stock\_qty = 76)  
 Planning Time: 0.148 ms  
 Execution Time: 128.913 ms  
(4 rows)  
  
postgres=#

****Explanation:****

* PostgreSQL switches to a ****direct Index Scan****.
* Since rows are tightly packed and statistics are updated, the planner knows the index scan is optimal.
* ****Execution Time (~0.13s)****: Fastest plan so far.

## **Why Bitmap Heap Scan Was Slower Than Seq Scan**

* ****Without clustering****: The Bitmap Heap Scan had to jump to ****58,440 scattered pages**** → costly random I/O.
* ****Seq Scan**** read the table linearly with parallel workers → faster despite reading more data.

## **Why**CLUSTER**Improved Execution Time**

* ****Physically orders data**** according to the index.
* Rows with stock\_qty = 76 became adjacent → only ****835 pages**** to fetch.
* Execution time dropped from ****42.6s → 0.43s → 0.13s**** after Analyze.

## **Final Takeaways**

* An index doesn’t always guarantee better performance.
* Bitmap Heap Scans can be very slow if data is ****scattered across the heap****.
* Sequential scans may outperform them because of efficient linear I/O and parallelism.
* CLUSTER dramatically improves performance by grouping related rows together.
* After clustering and analyzing, PostgreSQL can switch to the fastest ****Index Scan****.

👉 ****Lesson for tuning:**** Always look at the query plan, not just the presence of indexes. Data organization matters just as much as indexes.