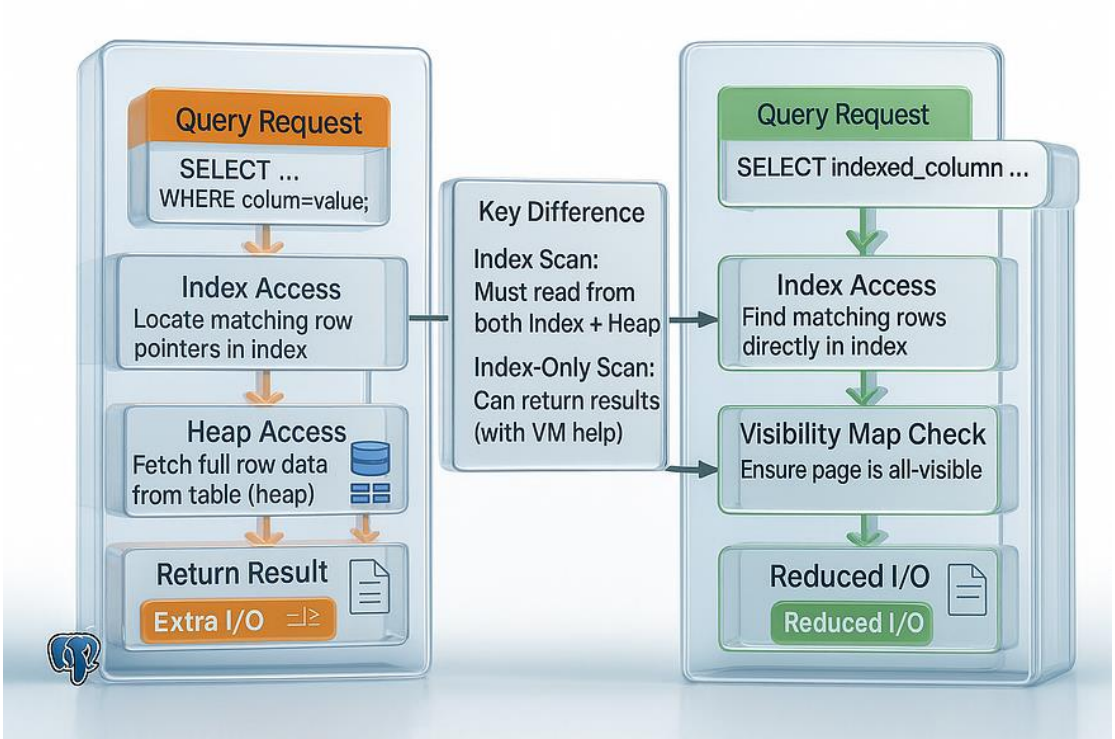
15 - PostgreSQL 17 Performance Tuning: Index Scans vs. Index-Only Scans



So far, we have looked at how indexes work and when PostgreSQL chooses Index Scans or Bitmap Scans. But there's more to indexing: sometimes PostgreSQL can avoid reading the table at all.

This is where **Index-Only Scans** come in. Let's explore the difference with a practical demo.

Index-Only Scans in PostgreSQL — A Deep Dive

What are Index-Only Scans?

An **Index-Only Scan (IOS)** is a powerful optimization in PostgreSQL where the database engine can answer a query **entirely from the index**, without having to fetch the original row data from the table (the heap). This works when the index contains all the columns needed to satisfy the query.

The benefit is simple but profound: **fewer disk reads, fewer buffer accesses, and faster queries.** Since table pages are often much larger and scattered, avoiding them can yield massive performance gains.

Why Index-Only Scans Are Faster

- **Skip the heap reads:** Traditional index scans still need to check the table (heap) to fetch requested columns and confirm row visibility. IOS can skip this if the index already has the data.
- **Compact and cache-friendly:** Indexes are smaller than the table and more likely to fit in memory. Scanning only the index reduces random I/O.
- **Faster queries with limits:** When combined with `ORDER BY` and `LIMIT`, IOS can return results quickly without scanning large portions of the table.

In short: **the less PostgreSQL touches the heap, the faster your query runs.**

The Two Key Conditions for IOS

1. **All required data is in the index**

- Every column in the `SELECT`, `WHERE`, `ORDER BY`, or `GROUP BY` clause must come directly from the index.
- PostgreSQL allows you to add extra non-key columns using `INCLUDE` in an index definition (known as a *covering index*).

Example:

```
CREATE INDEX idx products category cover ON products (category) INCLUDE (product id,
price);
```

2. Here, the index stores `category` for filtering and includes `product_id` and `price` for retrieval.

1. **Visibility map marks the rows as all-visible**

- PostgreSQL uses MVCC (multi-version concurrency control). It must ensure a row is visible to your query snapshot.
- Normally, this requires a heap fetch.
- With the **Visibility Map (VM)**, PostgreSQL knows if a whole heap page contains only visible rows. If so, no heap fetch is needed.
- If the VM doesn't confirm this, PostgreSQL falls back to heap lookups even though the plan still says "Index Only Scan."
- Hint: Run `VACUUM` regularly (or rely on autovacuum) so the visibility map stays updated.

How Index-Only Scans Work (Step-by-Step)

1. PostgreSQL traverses the index to locate entries matching the filter condition.
2. For each entry, it consults the visibility map:

- If marked **all-visible**, PostgreSQL skips the heap entirely.
- If not, PostgreSQL fetches the row from the heap for verification.

1. If the needed columns are included in the index, PostgreSQL returns them directly.

2. If there's an `ORDER BY` that matches the index order, results come out sorted without extra work.
3. If the query uses `LIMIT`, PostgreSQL can stop once enough rows are found.

Example Walkthrough

Suppose we want to quickly fetch product IDs and prices for products in a given category.

```
CREATE INDEX idx_products_category_cover
ON products (category)
INCLUDE (product id, price);



postgres=# CREATE INDEX idx_products_category_cover
ON products (category)
INCLUDE (product id, price);
CREATE INDEX
postgres=#



EXPLAIN (ANALYZE, BUFFERS)
SELECT product_id, price
FROM products
WHERE category = 'Category_2';
```

A good plan might look like:

```
postgres=# EXPLAIN (ANALYZE, BUFFERS)
SELECT product id, price
FROM products
WHERE category = 'Category_2';
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------
 Index Only Scan using idx_products_category_cover on products  (cost=0.56..44995.00
rows=972711 width=19) (actual time=0.023..721.391 rows=1000000 loops=1)
   Index Cond: (category = 'Category_2'::text)
   Heap Fetches: 0
   Buffers: shared hit=7151
 Planning Time: 0.065 ms
 Execution Time: 1333.884 ms
(6 rows)

postgres=#
```

Key observations:

- **Index Only Scan** → PostgreSQL used the index without touching the heap.
- **Heap Fetches: 0** → All rows came from the index, thanks to the visibility map.
- Execution time is extremely low compared to a table scan.

When Index-Only Scans Shine

- **Large tables with selective queries**: Avoiding heap fetches saves thousands or millions of I/O operations.
- **Reporting dashboards and APIs**: Frequent lookups that return a small number of columns.
- **Queries with `ORDER BY` and `LIMIT`**: Results can be returned very quickly from the index.
- **Covering indexes**: When you design indexes to include frequently accessed columns.

When They May Not Help

- **Small tables**: Sequential scans can be just as fast.
- **Frequent updates**: Updated/deleted rows delay pages becoming all-visible, forcing heap fetches.
- **Queries needing non-indexed columns**: PostgreSQL still must visit the heap.
- **Indexes with too many included columns**: Bloated indexes slow down writes and increase storage use.

Best Practices for Effective IOS

- Use **covering indexes** with `INCLUDE` for common queries.

- Keep autovacuum active to maintain the visibility map.
- Avoid excessive updates to indexed columns (prefer HOT updates).
- Don’t over-cover: balance read performance with write costs.
- Regularly verify with:

```
EXPLAIN (ANALYZE, BUFFERS) ...
```

- Look for `Heap Fetches: 0`.

Index-Only Scans in PostgreSQL are a performance booster that let the database serve results **directly from the index** without consulting the heap — provided the index has all the required columns and the visibility map confirms row visibility. They’re especially powerful for large datasets, reporting queries, and queries with limits. With careful index design and good autovacuum practices, they can cut query times dramatically.

1. The Difference Between Index Scan and Index-Only Scan

Suppose we query like this:

```
SELECT product_id, product_name
FROM products
WHERE product_id = 10;

postgres=# SELECT product_id, product_name
FROM products
WHERE product_id = 10;
 product_id | product_name
------------+--------------
         10 | Product 10
(1 row)

postgres=#
```

If there is an index on `product_id`, PostgreSQL does two things:

1. Looks up the row location in the index.
2. Fetches the full row from the table (heap) to get the additional columns.

☐ This is a normal **Index Scan**. It always needs to touch both the index and the table.

But what if we only select columns that already exist inside the index? Then PostgreSQL doesn't need to touch the table at all — it can return results directly from the index. This is called an **Index-Only Scan**.

2. Create the Products Table with 10 Million Rows

For the demo, let's create a large `products` table:

```
CREATE TABLE products (
    product id   BIGSERIAL PRIMARY KEY,
    product_name TEXT,
    category     TEXT,
    price        NUMERIC,
    stock_qty    INT
);
```

```
postgres=# CREATE TABLE products (
    product_id   BIGSERIAL PRIMARY KEY,
    product name TEXT,
    category     TEXT,
    price        NUMERIC,
    stock_qty    INT
);
CREATE TABLE
postgres=#
```

```
-- Insert 10 million rows
INSERT INTO products (product name, category, price, stock qty)
SELECT
    'Product ' || g,
    'Category ' || (g % 10),
    (random()*500)::NUMERIC,
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
```

```
postgres=# -- Insert 10 million rows
INSERT INTO products (product_name, category, price, stock_qty)
```

```
SELECT
    'Product_' || g,
    'Category ' || (g % 10),
    (random()*500)::NUMERIC,
    (random()*100)::INT
FROM generate_series(1, 10000000) g;
ANALYZE products;
INSERT 0 10000000

ANALYZE
postgres=#
```

3. Normal Index Scan

First, create a simple index:

```
CREATE INDEX idx_products_id ON products(product_id);
```

```
postgres=# CREATE INDEX idx_products_id ON products(product_id);
CREATE INDEX
postgres=#
```

Now run this query:

```
SELECT * FROM products WHERE product_id = 42;
```

```
postgres=# SELECT * FROM products WHERE product_id = 42;
 product_id | product_name |  category  |      price       | stock_qty
------------+--------------+------------+------------------+-----------
         42 | Product_42   | Category 2 | 339.575950907242 |        13
(1 row)

postgres=#
```

```
EXPLAIN ANALYZE
SELECT * FROM products WHERE product_id = 42;
```

```
postgres=# EXPLAIN ANALYZE
SELECT * FROM products WHERE product_id = 42;
                                                          QUERY PLAN
------------------------------------------------------------------------------------------------
---------------------------------------
 Index Scan using idx_products_id on products  (cost=0.43..8.45 rows=1 width=49)
```

```
(actual time=1.665..1.667 rows=1 loops=1)
   Index Cond: (product_id = 42)
 Planning Time: 1.170 ms
 Execution Time: 1.687 ms
(4 rows)

postgres=#
```

What happens:

- PostgreSQL uses an **Index Scan**.
- It finds the row quickly via the index, but then still fetches the full row from the table.
- Two lookups per result row (index + heap).

4. Index-Only Scan

Now try a query that selects only the indexed column:

```
EXPLAIN ANALYZE
SELECT product_id FROM products WHERE product_id = 42;
```

```
postgres=# EXPLAIN ANALYZE
SELECT product_id FROM products WHERE product_id = 42;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------
-----------------------------------------
 Index Only Scan using idx_products_id on products  (cost=0.43..4.45 rows=1 width=8)
(actual time=0.060..0.062 rows=1 loops=1)
   Index Cond: (product_id = 42)
   Heap Fetches: 0
 Planning Time: 0.090 ms
 Execution Time: 0.076 ms
(5 rows)

postgres=#
```

What happens:

- PostgreSQL uses an **Index-Only Scan**.

- Since all the needed data (`product_id`) is already inside the index, PostgreSQL skips the table.
- Much faster, because it avoids heap access.

5. Covering Index with `INCLUDE`

What if queries often fetch **one filter column and one extra column**?

Example:

```
EXPLAIN ANALYZE
SELECT product_name, price FROM products WHERE product_id = 42;
```

```
postgres=# EXPLAIN ANALYZE
SELECT product_name, price  FROM products WHERE product_id = 42;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------
-------------------------------------
 Index Scan using products pkey on products  (cost=0.43..8.45 rows=1 width=26)
(actual time=0.033..0.034 rows=1 loops=1)
   Index Cond: (product_id = 42)
 Planning Time: 4.248 ms
 Execution Time: 0.091 ms
(4 rows)

postgres=#
```

Normally, PostgreSQL would still fetch from the heap, because `product_name, price`isn't in the index.

✓ Solution: Create a **covering index** with `INCLUDE`.

```
CREATE INDEX idx products id ON products(product id) INCLUDE (product name, price);
```

```
postgres=# CREATE INDEX idx products id ON products(product id) INCLUDE
(product_name, price);
CREATE INDEX
postgres=#
```

Now the index contains both:

- Search key → `product_id`
- Payload → `product_name, price`

```
ANALYZE products;
```

Run again:

```
EXPLAIN ANALYZE
SELECT product_name, price FROM products WHERE product_id = 42;
```

```
postgres=# EXPLAIN ANALYZE
SELECT product_name, price FROM products WHERE product_id = 42;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------
--------------------------------------------
 Index Only Scan using idx_products_id on products  (cost=0.43..4.45 rows=1
width=26) (actual time=0.018..0.020 rows=1 loops=1)
   Index Cond: (product_id = 42)
   Heap Fetches: 0
 Planning Time: 0.059 ms
 Execution Time: 0.036 ms
(5 rows)

postgres=#
```

☐ PostgreSQL can now return results with an **Index-Only Scan** because both columns are available in the index itself.

Key Takeaways

- **Index Scan** → always fetches the row from the table after finding it in the index.
- **Index-Only Scan** → returns data directly from the index if all requested columns are present there.

- **Covering Indexes (`INCLUDE`)** let you add extra payload columns so more queries can benefit from Index-Only Scans.
- Since PostgreSQL 11, `INCLUDE` makes Index-Only Scans even more powerful for real-world queries.

✅ Index-Only Scans can significantly reduce I/O and speed up queries, especially on large tables like our `products` table with 10 million rows.