Prelude

Before we start with Indexes I just wanted to go back on a few things.

- Bitmap Heap and Bitmap Index Scans
- Show the difference between EXPLAIN and EXPLAIN ANALYZE on a query
- how to update the stats on a table.
    - EXPLAIN ANALYZE does not force the real stats on the planner. Instead, it shows you both the planner's estimates and what actually happened during execution.
- ANALYZE a table to update the stats

Bitmap Heap and Bitmap Index Scans

The query planner may or may not decided on using a Bitmap Scan. It's not because you have an index on a given column and a query with a condition on that column that the planner will always resort to a Bitmap Scan. It may also use a Sequential scan or an Index Scan.

A Bitmap Heap Scan is a two-step process used by PostgreSQL to retrieve data from a table **when there's an index that can be used**. The filtering must involve a column with an index.

The process works as follows: a) First, it creates a **bitmap** in memory, where each bit represents a **table block**. b) It then sets bits (1) for blocks that contain rows matching the query conditions. c) Finally, it scans only the blocks with set bits to retrieve the matching rows.

- **Bitmap Index Scan**: This is the first step of the process. It scans the index to create the in-memory bitmap of which blocks potentially contain matching rows.
- **Bitmap Heap Scan**: This is the second step. It uses the bitmap created by the Bitmap Index Scan to fetch the actual data from the table.

In an EXPLAIN output, these two steps together are coupled together

```
Bitmap Heap Scan on table_name
  Recheck Cond: (column = value)
  ->  Bitmap Index Scan on index_name
        Index Cond: (column = value)
```

The Bitmap Index Scan creates the bitmap, and the Bitmap Heap Scan uses that bitmap to fetch the rows from the table.

Bitmap data structure

The bitmap data structure is created during the execution phase, as part of the Bitmap Index Scan process, and then immediately used by the Bitmap Heap Scan.

The bitmap is created and used on-the-fly during query execution. It exists only in memory and for the duration of the query.

Bitmap Scans are particularly efficient when:

1. The query is expected to return a moderate to large number of rows
2. The rows are scattered across many pages in the table

It's more efficient than a regular Index Scan for larger result sets because it can read the table pages in physical order, reducing random I/O.

Illustration

Let's go back to the trees table with id as the index

SQL
```
EXPLAIN select * from trees where id < 10000
```

The planner follows the Bitmap Scan sequence : Bitmap Index Scan then Bitmap Heap Scan

SQL
```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on trees  (cost=189.37..2777.48 rows=9929 width=50)
   Recheck Cond: (id < 10000)
   ->  Bitmap Index Scan on trees_pkey  (cost=0.00..186.89 rows=9929 wi
         Index Cond: (id < 10000)
(4 rows)
```

But with a weaker condition on the `id`, the planner reverts back to a Sequential Scan.

```SQL
treesdb_v04=# EXPLAIN select * from trees where id < 150000;
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on trees  (cost=0.00..5105.74 rows=149967 width=50)
   Filter: (id < 150000)
(2 rows)
```

Index Scan

If the column is part of an index, then the planner can also decide to directly leverage the indexed data with an **Index Scan**

Index Scans are used for:

- **High Selectivity Queries**: when the query is expected to return a small portion of the table's rows, typically less than 5-10%. The index allows quick access to these few rows without scanning the entire table.
- **Ordered Results**: when the query requires results in the order of the index (e.g., ORDER BY clause matches the index order). The index already maintains data in sorted order, avoiding a separate sorting step.
- **Unique Constraints**: for queries looking up rows based on a unique or primary key. These are typically very fast as they return at most one row.

Index Scan Example

```SQL
treesdb_v04=# explain select * from trees where id = 808;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using trees_pkey on trees  (cost=0.42..8.44 rows=1 width=50
   Index Cond: (id = 808)
```

Recap on Scans

- Sequential Scans: lots of rows regardless of the presence of an index
- Bitmap Scans: index + 2 steps process + block bitmap + less rows
- Index Scans: index + even less rows

Explain and Explain Analyze and Analyze

I connected to the remote server and the `airdb` database.

In that database, there is a `booking` table with 5.6 m rows

SQL

```
                    Table "postgres_air.booking"
    Column     |           Type           | Collation | Nullable | Defau
---------------+--------------------------+-----------+----------+------
 booking_id    | bigint                   |           | not null |
 booking_ref   | text                     |           | not null |
 booking_name  | text                     |           |          |
 account_id    | integer                  |           |          |
 email         | text                     |           | not null |
 phone         | text                     |           | not null |
 update_ts     | timestamp with time zone |           |          |
 price         | numeric(7,2)             |           |          |
Indexes:
    "booking_pkey" PRIMARY KEY, btree (booking_id)
    "booking_booking_ref_key" UNIQUE CONSTRAINT, btree (booking_ref)
    "idx_booking_phone" btree (phone)
```

Note the index on `phone`. Let's pick a random phone number ('918728050') and look at the simple query

SQL

```
select * from booking where phone = '918728050';
```

This returns 1955 rows. Meaning that 1955 booking were made with the same phone number. Bit much but why not.

Explaining the query shows the use of a Bitmap Heap Scan. Which makes sense. The phone column is indedxed and the condition returns a significant number of rows but not too many

SQL

```
EXPLAIN select * from booking where phone = '918728050';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on booking  (cost=11.72..3478.83 rows=940 width=92)
   Recheck Cond: (phone = '918728050'::text)
   ->  Bitmap Index Scan on idx_booking_phone  (cost=0.00..11.48 rows=94
         Index Cond: (phone = '918728050'::text)
```

Notice that the number of rows is 940! far off from the real number of rows 1955!

EXPLAIN ANALYZE returns more info

```SQL
explain ANALYZE select * from booking where phone = '918728050';
                                                          QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on booking  (cost=11.72..3478.83 rows=940 width=92) (
   Recheck Cond: (phone = '918728050'::text)
   Heap Blocks: exact=1943
   ->  Bitmap Index Scan on idx_booking_phone  (cost=0.00..11.48 rows=9
         Index Cond: (phone = '918728050'::text)
 Planning Time: 0.122 ms
 Execution Time: 5.012 ms
```

the interesting bit here is that EXPLAIN ANALYZE returns the **estimated** number of rows (still 940) and the real number of rows (see *actual*).

EXPLAIN ANALYZE also returns

- the real planning and execution times of the query
- the number of blocks scanned : Heap Blocks: exact=1943)
- the number of workers, loops etc

But why is the estimate number of rows still far off ? Shouldn't it be closer to the real number if we use EXPLAIN ANALYZE ?

Good question, and I think you for asking it ;)

We can dig deeper and update the table stats (and hopefully the estimations) with the `ANALYZE table` query;

```SQL
ANALYZE booking;
```

Then rerunning the EXPLAIN query should improve the row number estimation.

```SQL
EXPLAIN select * from booking where phone = '918728050';
                                    QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on booking  (cost=11.83..3528.63 rows=955 width=92)
 ...
```

Now we have 955 estimated rows instead of 940. Bit better but still very off.

Running analyze table several times tends to slowly update the estimation.

At this point, we should look into the particularities of the data distribution in the booking table. And I did not have the time to further investigate.

However, all the stats that are related to a given table are fully accessible in the `pg_stats` table;

```
        Column          |   Type    | Collation | Nullable | Default
------------------------+-----------+-----------+----------+---------
 schemaname             | name      |           |          |
 tablename              | name      |           |          |
 attname                | name      |           |          |
 inherited              | boolean   |           |          |
 null_frac              | real      |           |          |
 avg_width              | integer   |           |          |
 n_distinct             | real      |           |          |
 most_common_vals       | anyarray  |           |          |
 most_common_freqs      | real[]    |           |          |
 histogram_bounds       | anyarray  |           |          |
 correlation            | real      |           |          |
 most_common_elems      | anyarray  |           |          |
 most_common_elem_freqs | real[]    |           |          |
 elem_count_histogram   | real[]    |           |          |
```

Some really interesting info in there.

For the sake of it Let's check what stats are in store for the booking table and the phone attribute:

```
select * from pg_stats
where tablename = 'booking'
and attname = 'phone';
```

we get

```SQL
schemaname             | postgres_air
tablename              | booking
attname                | phone
inherited              | f
null_frac              | 0
avg_width              | 10
n_distinct             | 5453
most_common_vals       | {966483262,727391113,716277086,246887278,46877
most_common_freqs      | {0.0012666667,0.0011666666,0.0011333333,0.0011
histogram_bounds       | {1000052124,1009391378,102170194,1031634384,10
correlation            | 0.021118335
most_common_elems      | [null]
most_common_elem_freqs | [null]
elem_count_histogram   | [null]
```

So much to explore !!!

Prelude Conclusion

In conclusion

3 types of scans

- Sequential: the whole table, large number of rows
- Bitmap: index + 2 steps + bitmap (0,1), significant volume of rows
- Index: index, few rows

and

- `EXPLAIN ANALYZE` gives both estimated and real numbers (time, rows, number of blocks...)
- `ANALYZE table` updates the stats for a given table
- all stats are in the `pg_stats` table

Let's now dive into INDEXes