# 1. reading the plan

## 1.1 Task 1 – TABLE WITHOUT INDEX

### 1. Create table test\_index:

CREATE TABLE labs.test\_index\_plan (

num float NOT NULL, load\_date timestamptz NOT NULL );

### 2. Fill the table with a lot of test data:

INSERT INTO labs.test\_index(num, load\_date)

SELECT random(), x

FROM generate\_series('2017-01-01 0:00'::timestamptz, '2021-12-31 23:59:59'::timestamptz, '10 seconds'::interval) x;

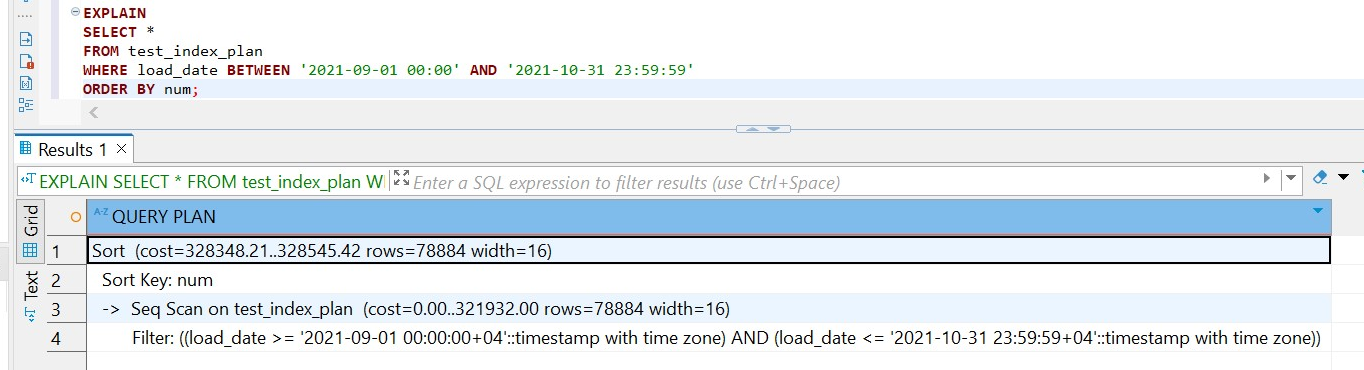
### 3. Check the plan of the select (twice at least, is any difference in plans?). Disable the parallel query planning If it needed:

SET max\_parallel\_workers\_per\_gather = 0;

SELECT \* FROM labs.test\_index\_plan

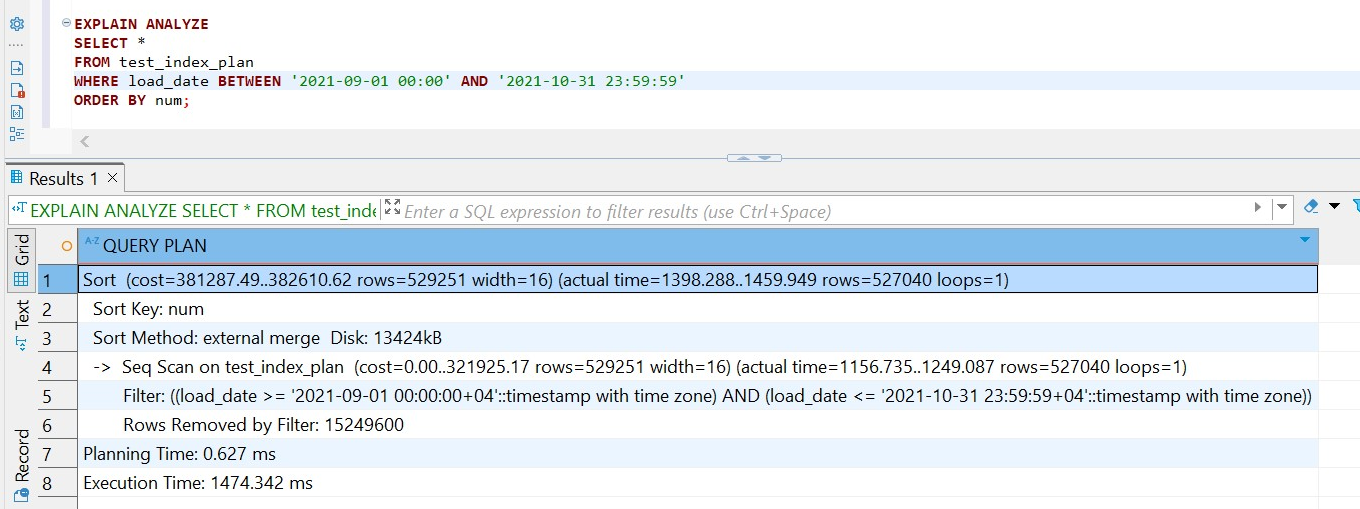
WHERE load\_date BETWEEN '2021-09-01 0:00' AND '2021-10-31 11:59:59'

ORDER BY 1;

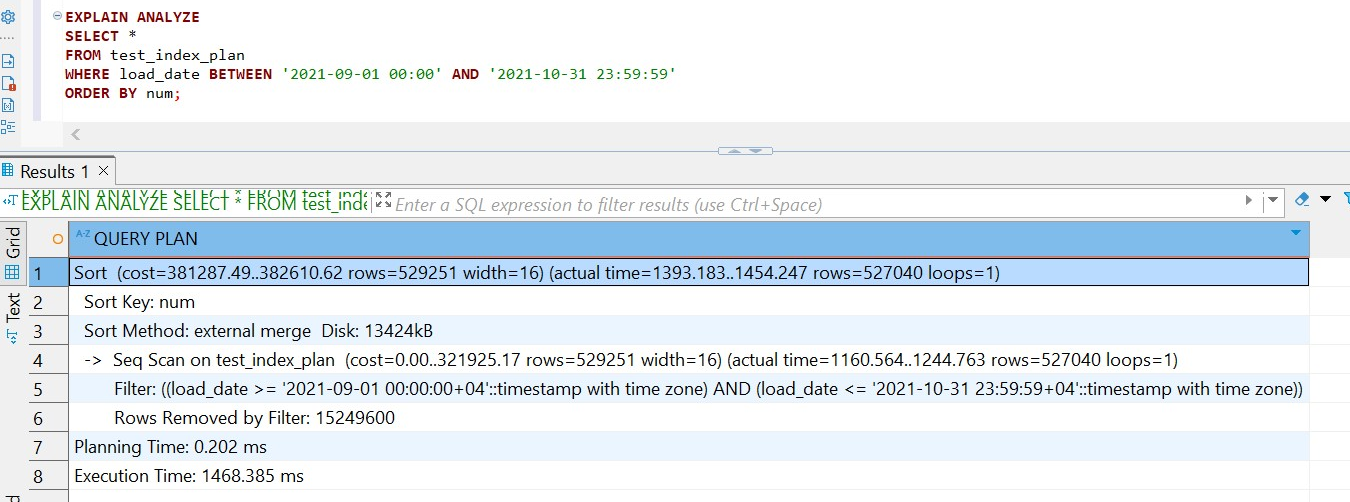
For **EXPLAIN**

For **EXPLAIN ANALYZE**

1st time

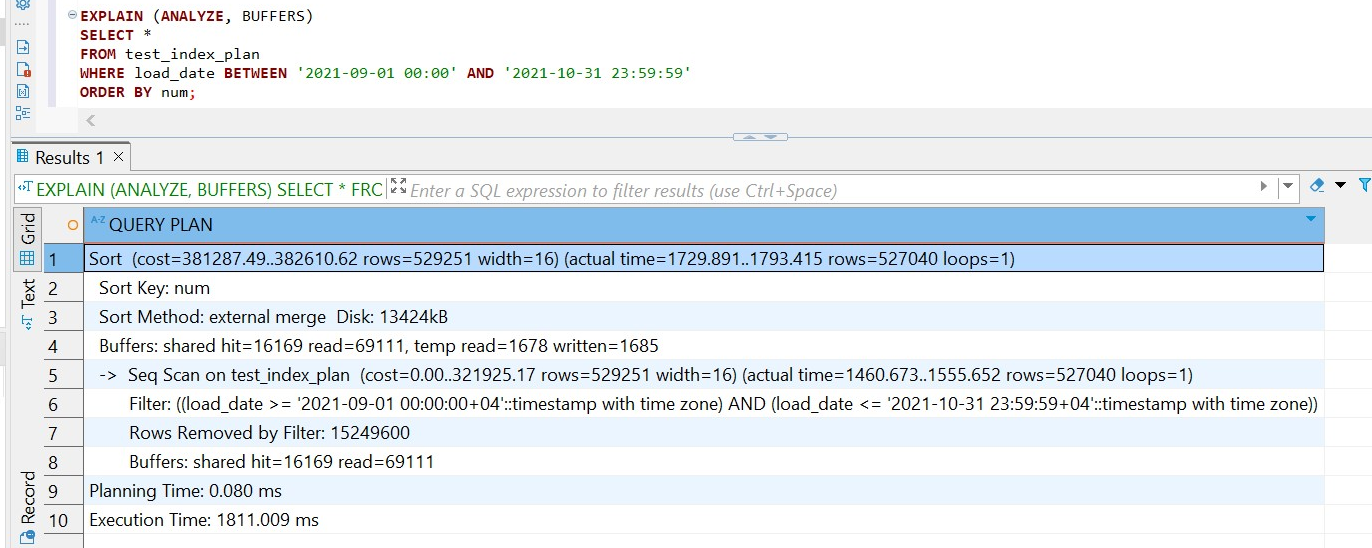


2nd time

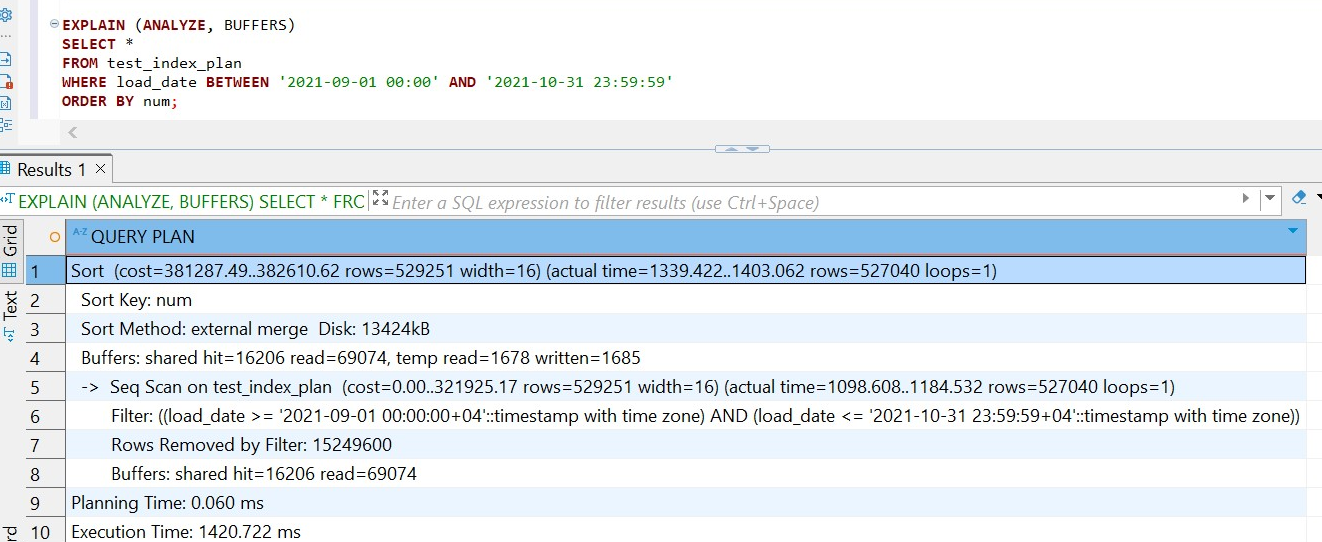


For **EXPLAIN ANALYZE(ANALYZE BUFFERS)**

1st time



2nd time



The first query was run with just EXPLAIN, which displays the **estimated execution plan** without actually running the query. The query filtered rows where load\_date was between '2021-09-01 00:00:00' and '2021-10-31 23:59:59', and sorted the results by the num column.

The second query was run with EXPLAIN ANALYZE, which not only planned the query but also executed it and reported real performance metrics. The scan type remained a **Sequential Scan**, but this time PostgreSQL reported that it actually returned **527,040 rows**, and filtered out **what was left, basically** almost the entire table.

The third query added the BUFFERS option, which provided detailed information about I/O behavior. PostgreSQL again used a **Sequential Scan**, returned the same **527,040 rows**, and used **external merge sort**. The execution time was longer — **1,811 ms** — reflecting that this was the **first run**, when most of the required pages were not yet cached in memory. This is consistent with the cold cache behavior, where PostgreSQL had to fetch most data blocks directly from disk.

This clearly demonstrated that when a table lacks indexes, PostgreSQL must use a **sequential scan**, reading **all rows** in the table regardless of how small the filter range is. This leads to high I/O costs and unnecessary processing, especially for large tables. The key takeaway is that without indexes, PostgreSQL cannot optimize access paths and must scan the full table, making query performance highly dependent on table size and caching state.

## 1.2 TASK 2 – ADDING INDEX

1. Create B-Tree Index on test\_index\_plan table for load\_date column.

CREATE INDEX IF NOT EXISTS idx\_test\_index\_plan\_loaddate

ON test\_index\_plan (load\_date);

### 2. Check the plan of the select (twice at least, is any difference in plans?). Disable the parallel query planning If it needed:

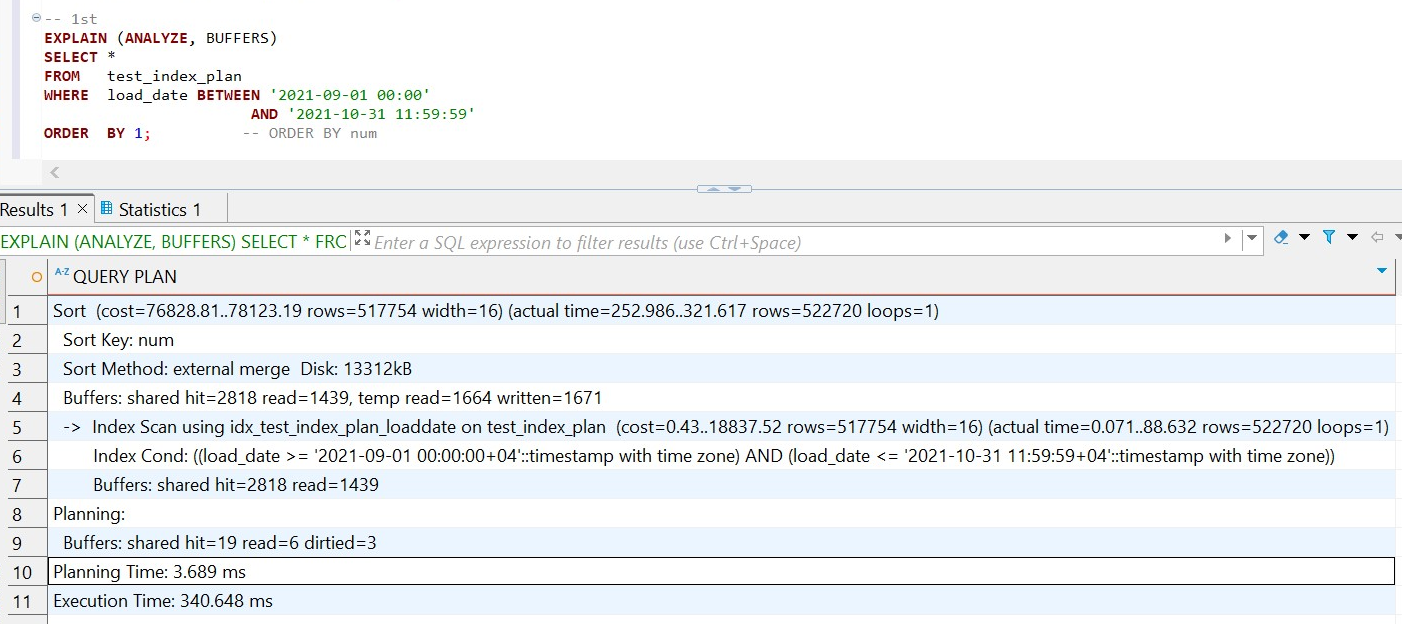
SET max\_parallel\_workers\_per\_gather = 0;

SELECT \* FROM labs.test\_index\_plan

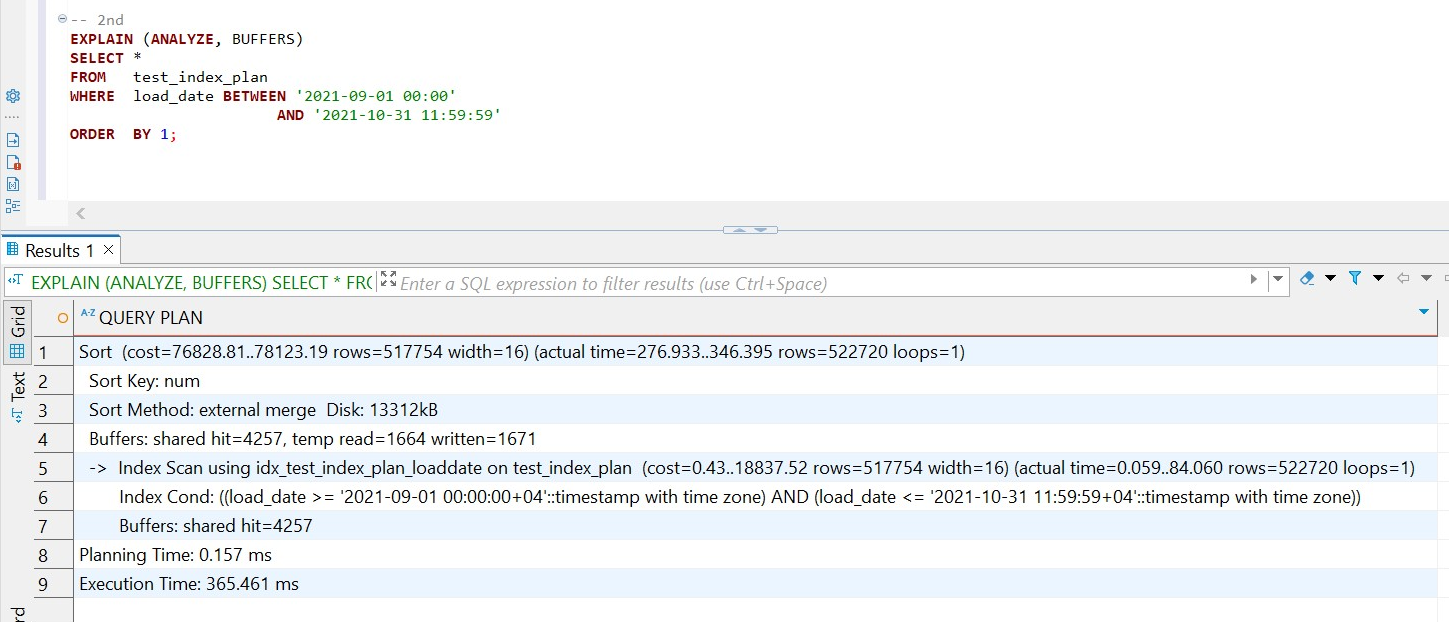
WHERE load\_date BETWEEN '2021-09-01 0:00' AND '2021-10-31 11:59:59'

ORDER BY 1;

From 1st run



From 2nd run



From those results we see that after a B-tree index is added on a column load\_date, PostgreSQL automatically switches from a costly sequential scan to an **Index Scan**, which improves performance. The query returned the same number of rows, but the index enabled the database to avoid reading the entire table. As a result, the execution time dropped from 1.5–1.8 seconds down to around **340 milliseconds**. Additionally, the performance further improved during the second execution due to **caching**, where many required pages were already in memory. The planner correctly reused the optimal plan, and disk I/O was minimized. Although the filtering part of the query became highly efficient, the final sorting step remained a performance cost. Because the index was only on load\_date and not on num, PostgreSQL had to sort the filtered result set using an external merge sort, which required temporary disk space. All of this shows the significant performance benefits of using an index for filtering large datasets and highlights how PostgreSQL adapts its query execution plan when indexes are available.

### 3. What can be done to query to use INDEX ONLY SCAN method?

An Index-Only Scan is possible when both of these conditions are true:

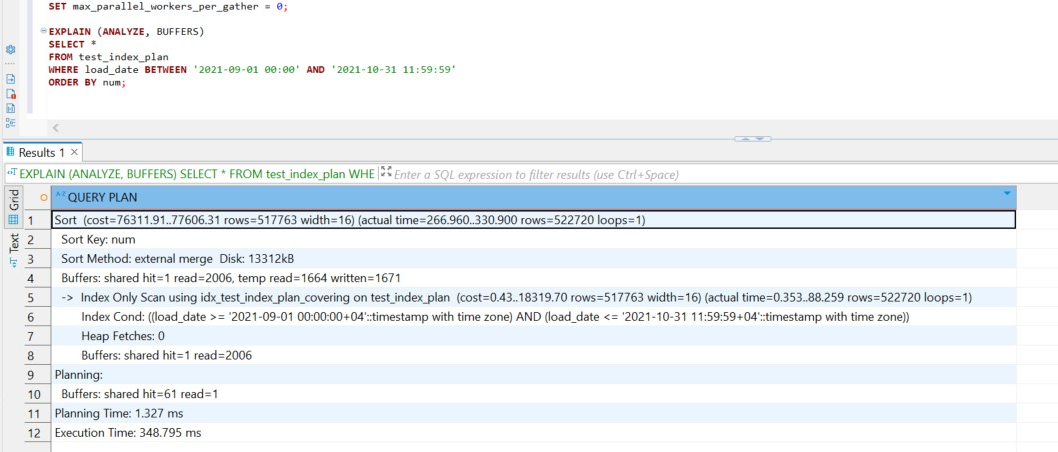
* Every column requested by the query is already in the index (so we have to create a new index)

CREATE INDEX idx\_test\_index\_plan\_loaddate\_num

ON test\_index\_plan (load\_date, num);

* Visibility map is “all visible”

We can run VACUUM so those pages are marked as all visible



By creating a covering index on all required columns (load\_date, num) and running VACUUM, PostgreSQL was able to avoid heap access entirely.

This resulted in a more efficient execution plan with:

* Reduced I/O load
* Slightly improved execution time
* Maintained accuracy in filtering and sorting

This proves that Index Only Scans can significantly enhance performance, especially for read-heavy workloads, provided the visibility conditions and indexing requirements are met.

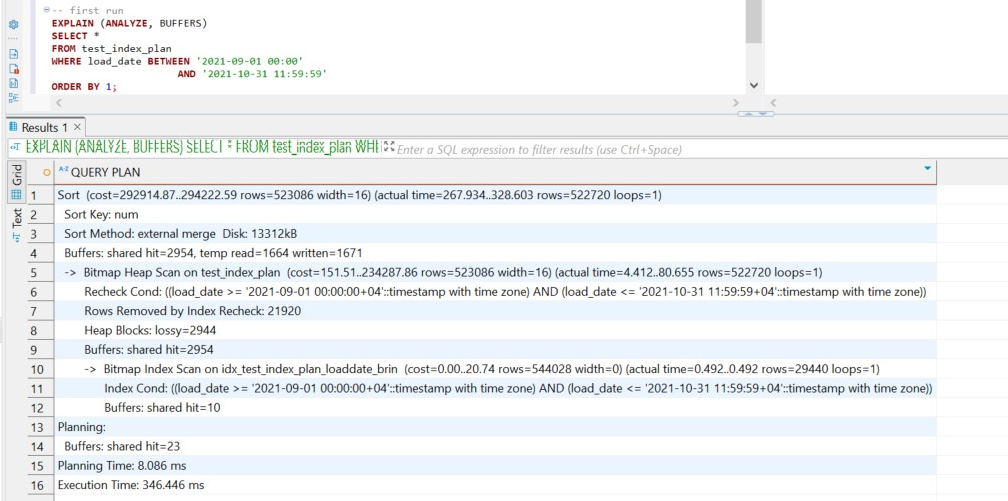
### 4. DROP B-tree Index from test\_index\_plan table and create BRIN index on test\_index\_plan table for load\_date column. Check the plan of the select (twice at least, is any difference in plans?).

SELECT \* FROM labs.test\_index\_plan

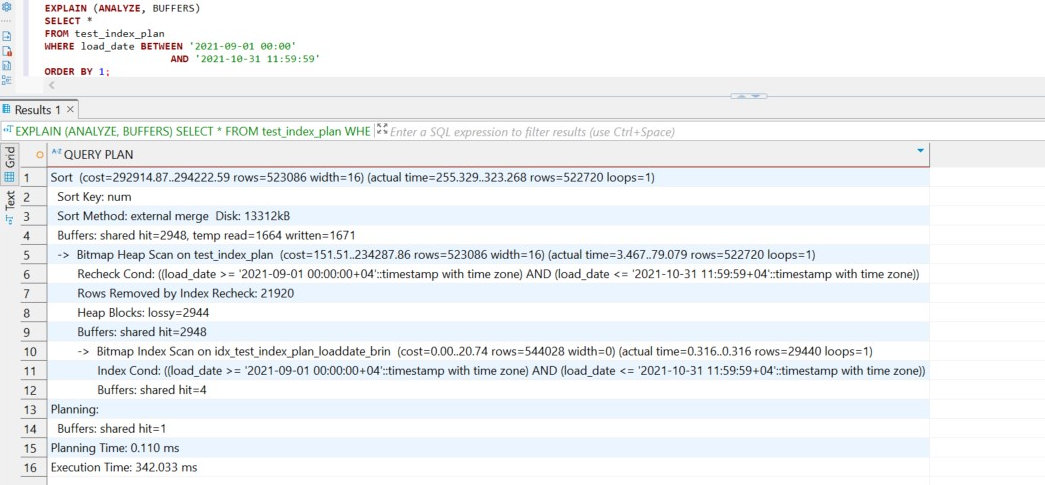
WHERE load\_date BETWEEN '2021-09-01 0:00' AND '2021-10-31 11:59:59'

ORDER BY 1;

1st run



2nd run



In both runs of the query, PostgreSQL selected a **Bitmap Index Scan** followed by a **Bitmap Heap Scan**. This behavior is expected with BRIN indexes, which store summaries of block ranges rather than specific row pointers.

BRIN indexes are less precise than B-tree indexes, but they are extremely lightweight and efficient for tables where the column values are mostly in order. The execution time was slightly higher than with B-tree or index-only scans, but still **10 times faster** than the original full-table Sequential Scan. This proves that BRIN indexes are an efficient option for large, append-only datasets, especially where disk space and index creation time are considerations. They provide considerable performance improvements over sequential scans while using minimal storage and maintenance overhead.

# 2. ADDING DATA WITH INSERT AND COPY

## 2.1 TASK 3 BULK INSERT

1. Create new table:

CREATE TABLE labs.test\_inserts (

num float NOT NULL, load\_date

timestamptz NOT NULL );

### 2. Add B-Tree index on the table test\_inserts on load\_date column.

CREATE INDEX idx\_test\_inserts\_loaddate

ON test\_inserts (load\_date);

### 3. INSERT into test\_inserts by using:

SELECT num, load\_date

FROM labs.test\_index\_plan;

### 4. Create new table\*:

CREATE TABLE emp (

empno NUMERIC(4) NOT NULL CONSTRAINT emp\_pk PRIMARY KEY,

ename VARCHAR(10) UNIQUE,

job VARCHAR(9), mgr

NUMERIC(4),

hiredate DATE );

### 5. Rewrite INSERT statements to more efficient way, run it:

INSERT INTO emp VALUES (1,'SMITH','CLERK',13,'17-DEC-80');

INSERT INTO emp VALUES (2,'ALLEN','SALESMAN',6,'20-FEB-81');   
 INSERT INTO emp VALUES (3,'WARD','SALESMAN',6,'22-FEB-81');   
 INSERT INTO emp VALUES (4,'JONES','MANAGER',9,'02-APR-81');   
 INSERT INTO emp VALUES (5,'MARTIN','SALESMAN',6,'28-SEP-81');   
 INSERT INTO emp VALUES (6,'BLAKE','MANAGER',9,'01-MAY-81');

INSERT INTO emp VALUES (7,'CLARK','MANAGER',9,'09-JUN-81');

INSERT INTO emp VALUES (8,'SCOTT','ANALYST',4,'19-APR-87');

INSERT INTO emp VALUES (9,'KING','PRESIDENT',NULL,'17-NOV-81');

INSERT INTO emp VALUES (10,'TURNER','SALESMAN',6,'08-SEP-81');

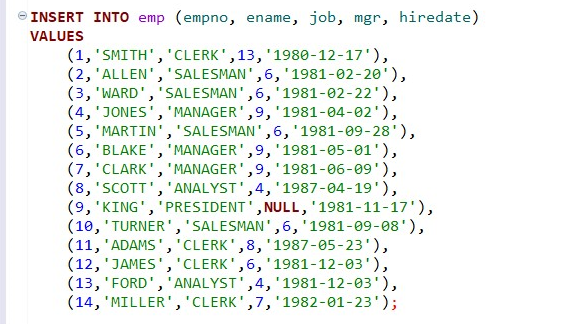
INSERT INTO emp VALUES (11,'ADAMS','CLERK',8,'23-MAY-87');

INSERT INTO emp VALUES (12,'JAMES','CLERK',6,'03-DEC-81');

INSERT INTO emp VALUES (13,'FORD','ANALYST',4,'03-DEC-81');

INSERT INTO emp VALUES (14,'MILLER','CLERK',7,'23-JAN-82')

Instead of this code that I have highlited and was provided in task I used



## 2.2 TASK 4 COPY COMMAND

1. Use COPY Command to export your test\_index\_plan table into csv file:

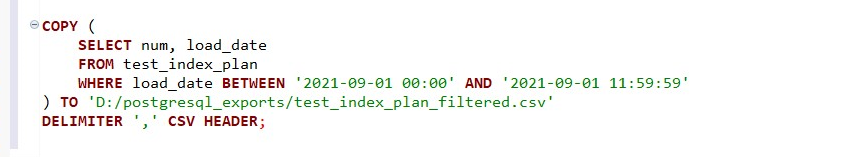
COPY labs.test\_index\_plan TO

‘\path\_to\_file\test\_index\_plan.csv’ DELIMITER ',' CSV HEADER;

### Change command to export column load\_date from test\_index\_plan with quotes and num column without quotes.



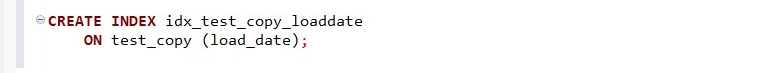
### 2. Use COPY to export data from test\_index\_plan table into csv file ‘test\_index\_plan\_short.csv’ where load\_date between '2021-09-01 0:00' AND '2021-09-01 11:59:59'



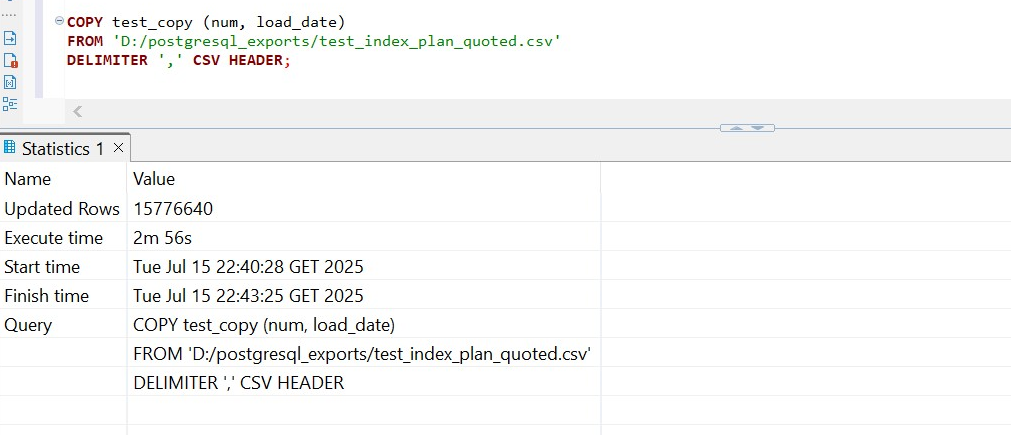
### 3. Create new table: CREATE TABLE labs.test\_copy ( num float NOT NULL, load\_date timestamptz NOT NULL );



### 4. Add B-Tree index on the table test\_inserts on load\_date column.



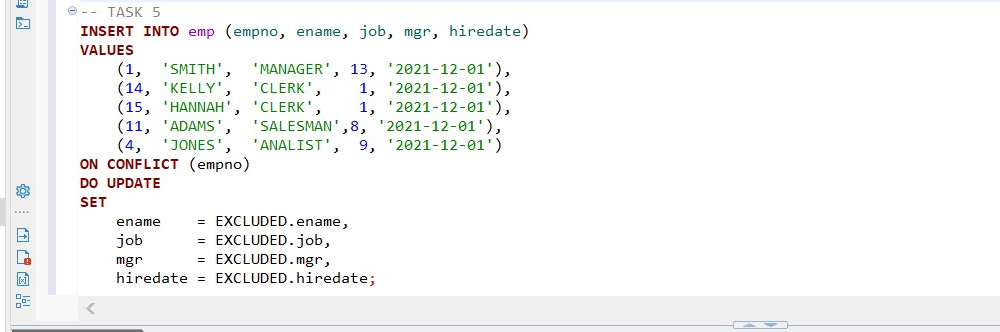
### 5. COPY into test\_copy by using test\_index\_plan.csv file.



In my case copying took almost 3 minutes, while INSERT INTO took less than a minute. Generally COPY is faster for loading data from external sources, in this case the INSERT INTO ... SELECT method outperformed it because it bypassed disk access, avoided parsing overhead, and operated entirely in memory. This highlights that while COPY is ideal for loading external data, INSERT ... SELECT can be faster when moving data within the database.

## 2.3 TASK 5 UPSERT STATEMENT

### Add into emp table following information in one UPSERT statement:



In this task, we used the UPSERTto insert or update multiple employee records in the `emp` table. The goal was to insert five employee entries, while ensuring that if any of the rows already, their data would be updated instead of causing a conflict.

1. One existing employee (`empno = 1`, SMITH) had their job title updated from `CLERK` to `MANAGER`.
2. Employees with `empno = 14` and `15` were added as new rows.
3. Employee `ADAMS` (`empno = 11`) had their job updated to `SALESMAN`.
4. Employee `JONES` (`empno = 4`) was updated to the job title `ANALIST` (as spelled in the original input).

The UPSERT statement successfully inserted or updated all provided employee records in a single, atomic operation. This approach ensures data consistency and is highly efficient compared to executing separate `INSERT` and `UPDATE` statements.