

| Business Template  **Data Access and Query Optimizer** |
| --- |
|  |

### 1. Reading the Plan

### 1.1 Task 1 – Table Without Index

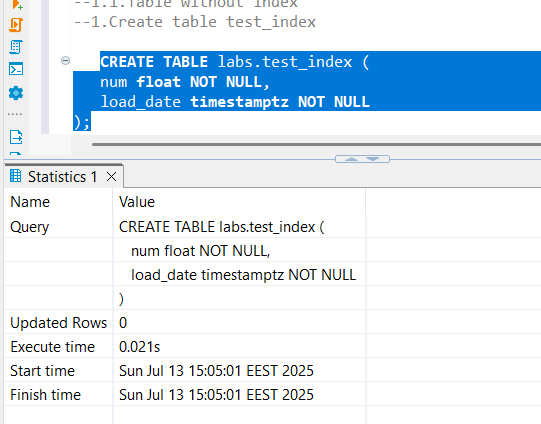
**🎯 Task Objective**:  
 Read the execution plan and describe **what happened** and **why**, including **screenshots where needed**.

### Compare Execution Plans

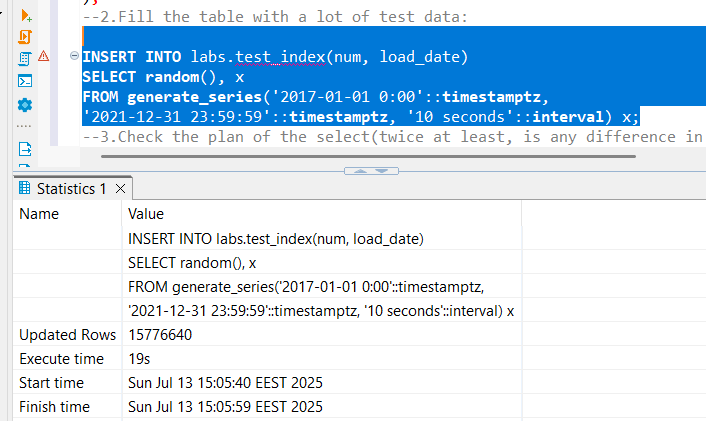
Check and understand the differences between the following PostgreSQL commands:

* EXPLAIN
* EXPLAIN ANALYZE
* EXPLAIN (ANALYZE, BUFFERS)

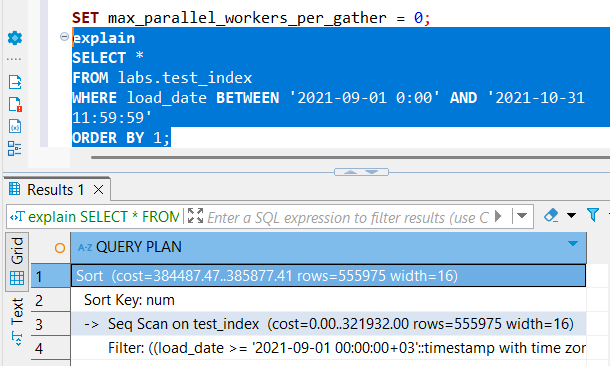
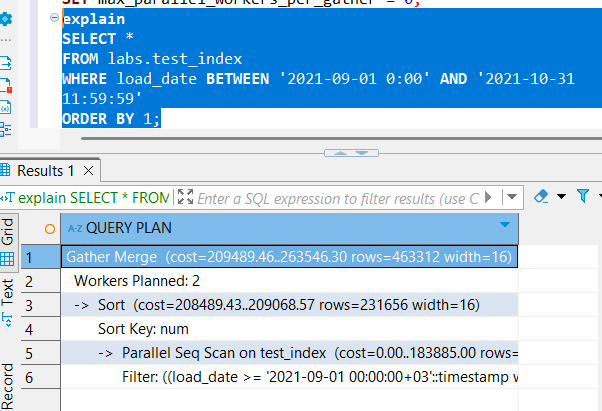
#### **Step 1: Create the Table**



#### **Step 2: Insert Test Data**

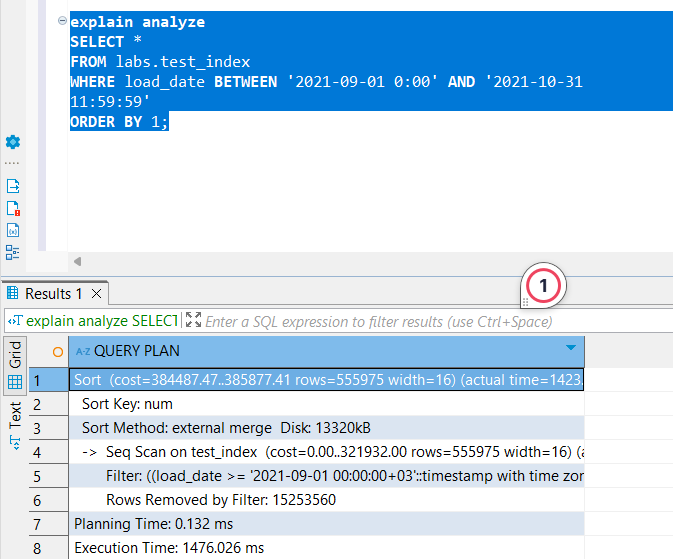
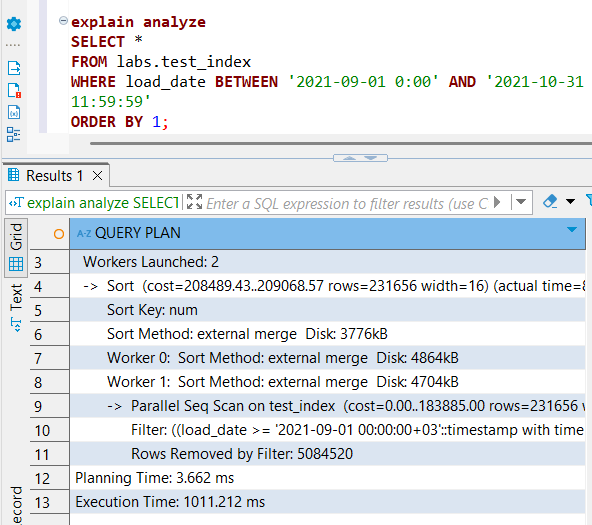


#### **Step 3: Analyze the Execution Plan**



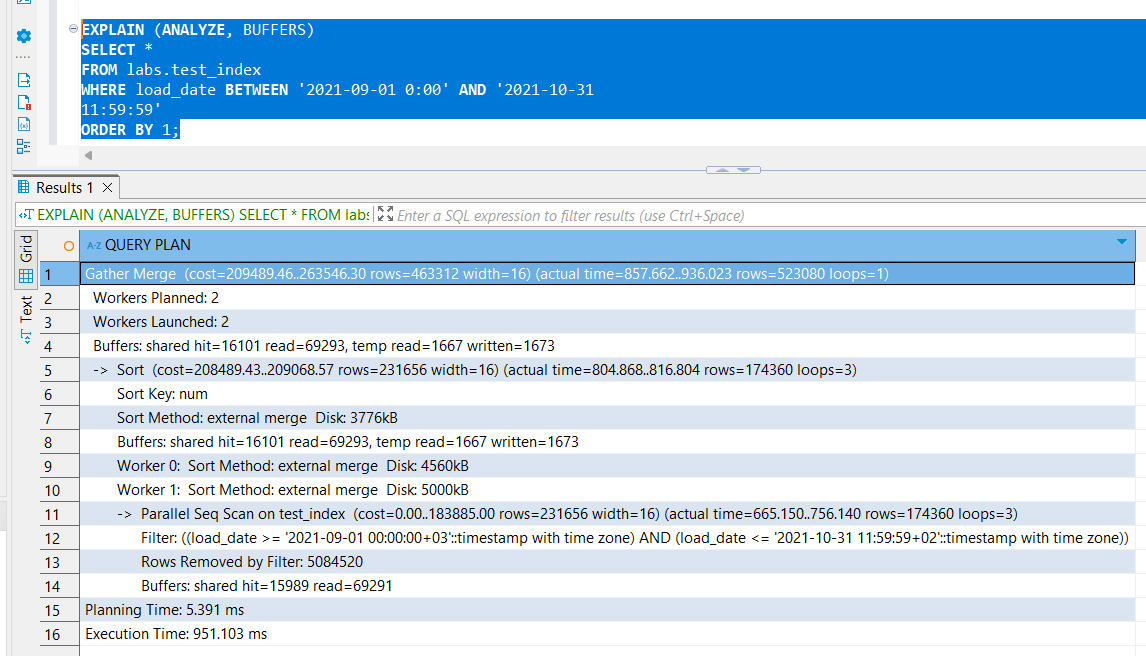
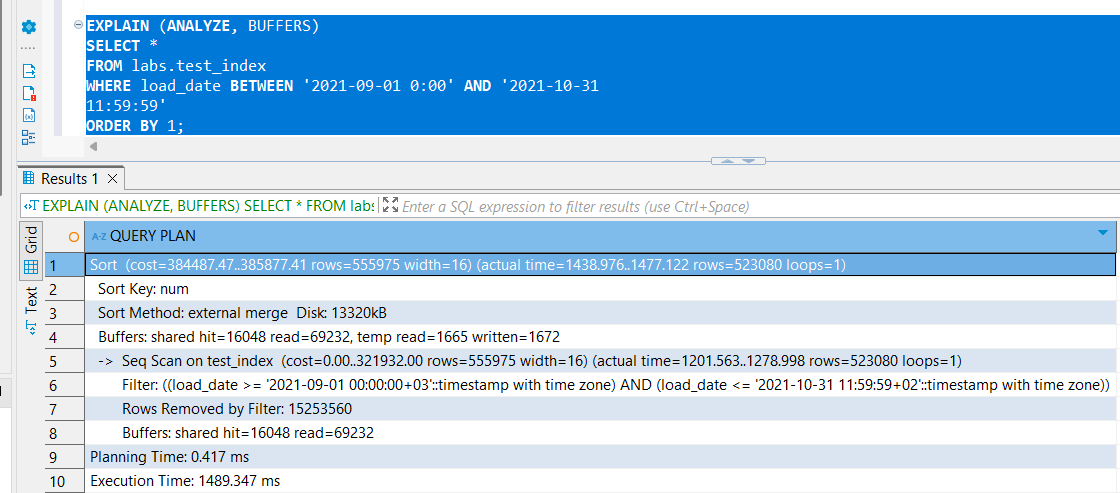
When we **disable parallel query planning**, the **cost of the SORT (ORDER BY 1) operation increases significantly—almost doubling**. This makes sense because **parallel execution helps optimize expensive operations** like sorting large datasets.

When **parallelism is enabled**, the query planner chooses a more efficient execution plan, such as **Parallel Seq Scan**, which distributes the workload across multiple CPU cores. This is beneficial for performance.



Now, when we run EXPLAIN ANALYZE, we can see a detailed breakdown of the **time taken by each step** in the execution. With parallel planning **enabled**, the **execution time is lower**, although the **planning time is slightly higher** due to the added complexity of evaluating parallel strategies.

However, when we **set the parallel workers to 0** (effectively disabling parallel processing), the planner chooses a different plan, such as a **simple sequential scan** on the test table. While the **planning time is slightly reduced**, the **overall execution time increases by nearly 50%**, showing that the lack of parallelism significantly impacts performance.



| **Metric** | **Parallel Plan** | **Non-Parallel Plan** | **Difference** |
| --- | --- | --- | --- |
| shared hit | 16,101 | 16,048 | +53 (parallel) |
| shared read | 69,293 | 69,232 | +61 (parallel) |
| temp read | 1,667 | 1,665 | +2 (parallel) |
| temp written | 1,673 | 1,672 | +1 (parallel) |

The **buffer usage is almost identical**, suggesting that **both plans read roughly the same amount of data**; no major difference in how much data was cached or spilled to disk. The main advantage of parallel processing isn t in I/O reduction, but in CPU-bound processing. Parallel workers(Worker 0 and Worker 1) can split the workload(scan, filter, sort) leading to faster execution time even if I/O is similar. While we see lower execution time in the parallel processing case, the buffer stats are similar because we read data from the same disk, the same data is processed, the amount of disk and memory usage doesn’t drastically change.

**Note**:  
 Capture screenshots of each plan and include observations about:

* Execution time
* Planning method
* Any buffer usage (in EXPLAIN (ANALYZE, BUFFERS))

### 

### 1.2 Task 2 – Adding index

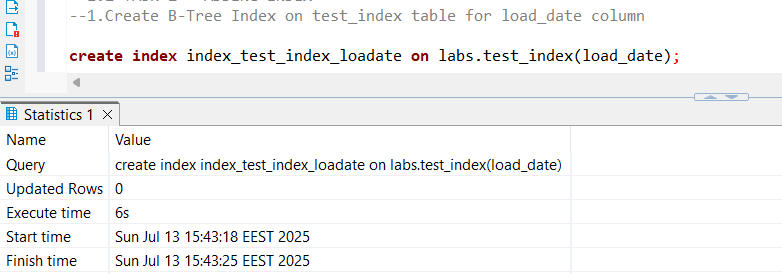
**🔍 Task Result**:  
 Read the plan and describe **what happened and why**, including **screenshots where needed**.

Check the difference between execution plans generated by the following commands:

* EXPLAIN
* EXPLAIN ANALYZE
* EXPLAIN (ANALYZE, BUFFERS)

### Step 1: Create a B-Tree Index

Create a B-Tree index on the load\_date column of the test\_index\_plan table:



### 

### 

### 

### 

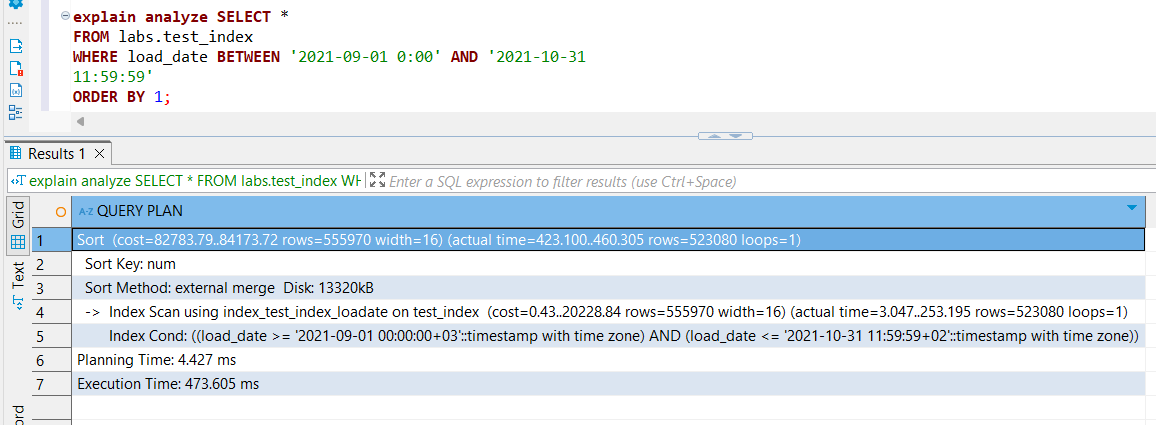
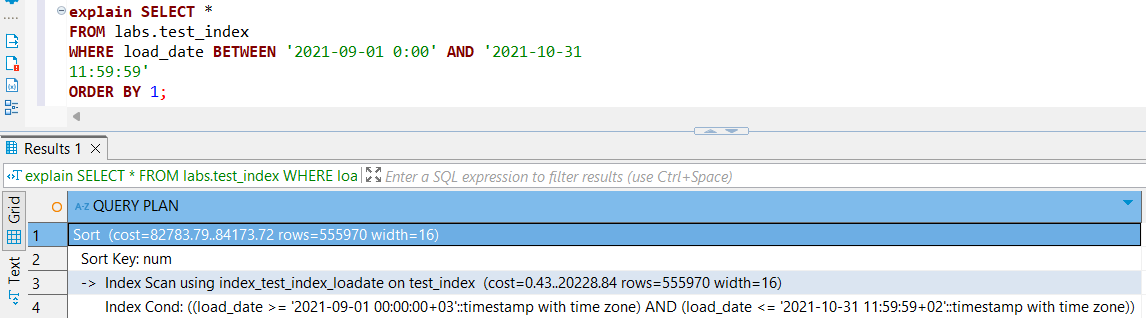
### 

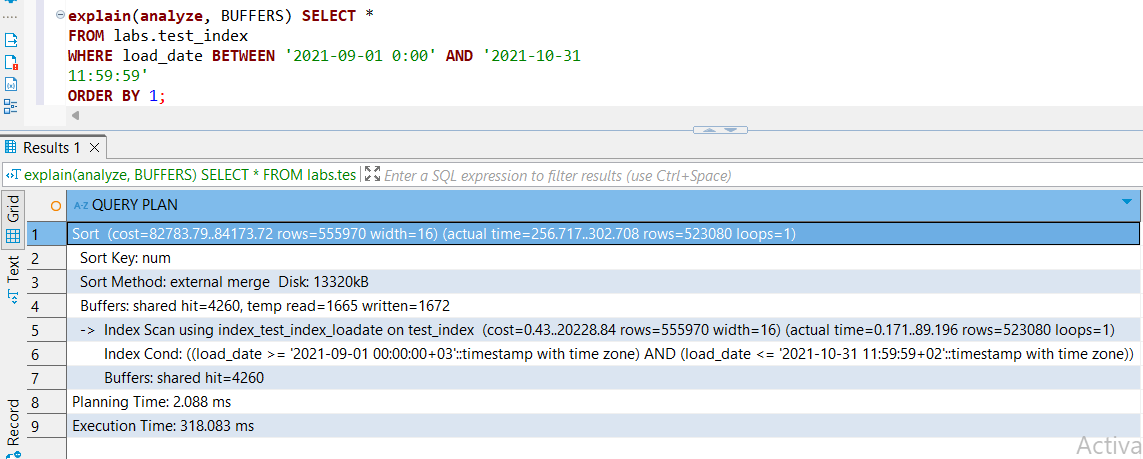
### 

### 

### Step 2: Check the Execution Plan (At Least Twice)

If needed, disable parallel query planning SET max\_parallel\_workers\_per\_gather = 0; Compare the plans with and without the index, using the three forms of EXPLAIN.





| **Metric** | **Value** | **Interpretation** |
| --- | --- | --- |
| **Execution time** | 318 ms | ✅ Much faster than before |
| **Planning time** | 2.088 ms | ⚡ Still fast, negligible increase |
| **shared hit** | 4,260 | ✅ Pages read from memory (cached) |
| **shared read** | *(none)* | ✅ No disk reads — everything was served from cache |
| **temp read/write** | 1665 / 1672 | ❗ Temp files still used — likely due to sorting, joining, or aggregation |

In addition to the improvements brought by using a B-tree index—compared to previous executions that used sequential or parallel sequential scans—we also observed that during the second run (when using EXPLAIN (ANALYZE, BUFFERS)), both planning and execution times were noticeably shorter. This is likely due to PostgreSQL's internal caching mechanisms. After the first run, much of the necessary data (e.g., table pages, index pages, and execution plan metadata) is cached in memory. Therefore, the second execution benefits from:

* **Buffer cache**: Data already loaded from disk is served from memory, reducing I/O.
* **Plan cache effects** (in some cases): PostgreSQL may avoid recomputing parts of the plan if the same query is run repeatedly in a prepared statement or similar context.

### Step 3: Optimize for INDEX ONLY SCAN

**❓ What can be done to enable an INDEX ONLY SCAN?**

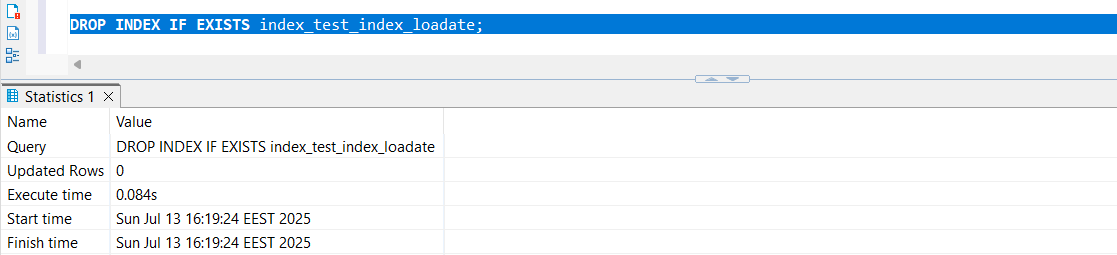
To enable an Index Only Scan, the query must **reference only the columns that are included in the index**. If the query needs columns that are not in the index, PostgreSQL will have to access the actual table (heap), and it will fall back to a regular Index Scan or Sequential Scan.

Additionally, PostgreSQL must be able to confirm the visibility of rows using the **visibility map**. If the table hasn’t been vacuumed recently, PostgreSQL may still need to check the heap even if all columns are in the index, which would prevent a true Index Only Scan.

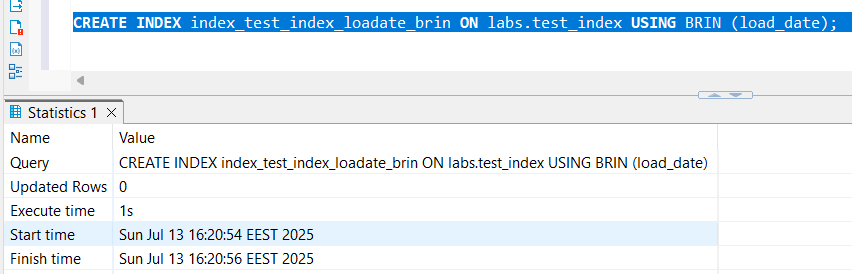
### 

### Step 4: Replace B-Tree with BRIN Index

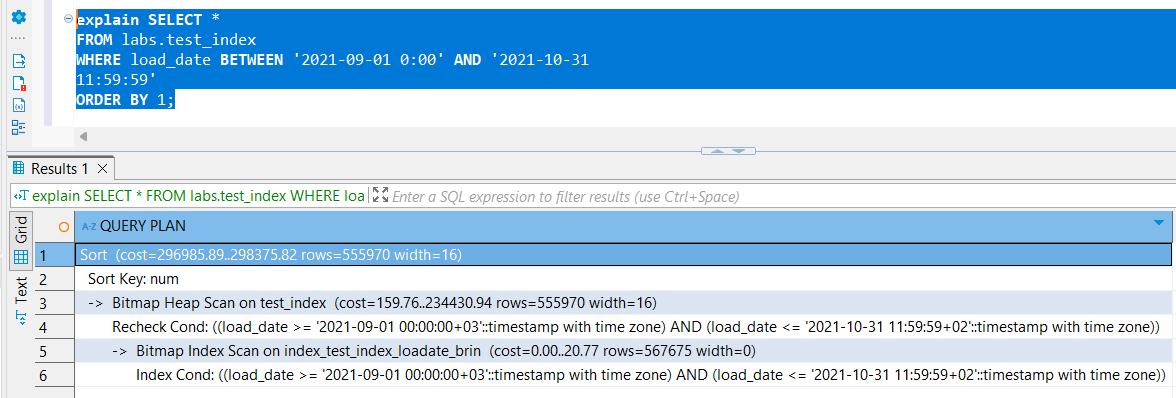
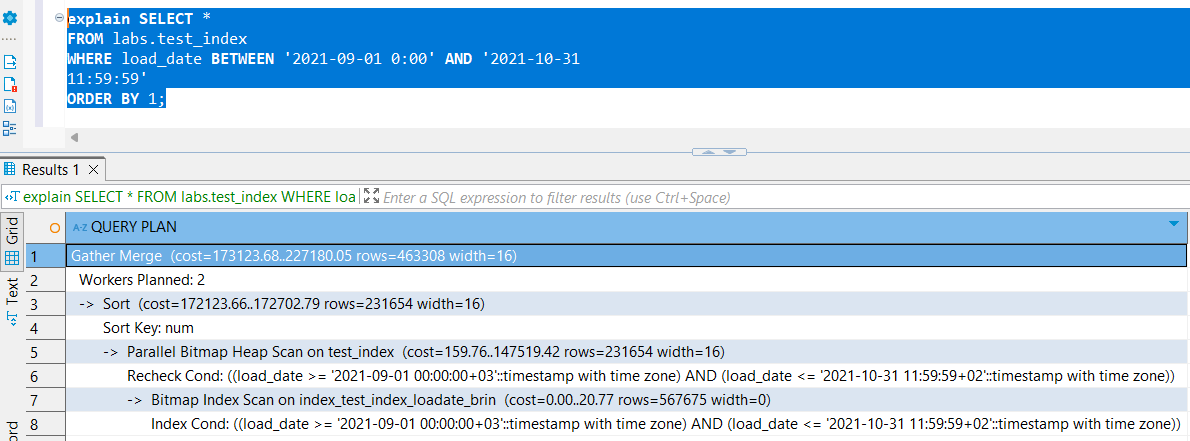
1)Drop the existing B-Tree index:

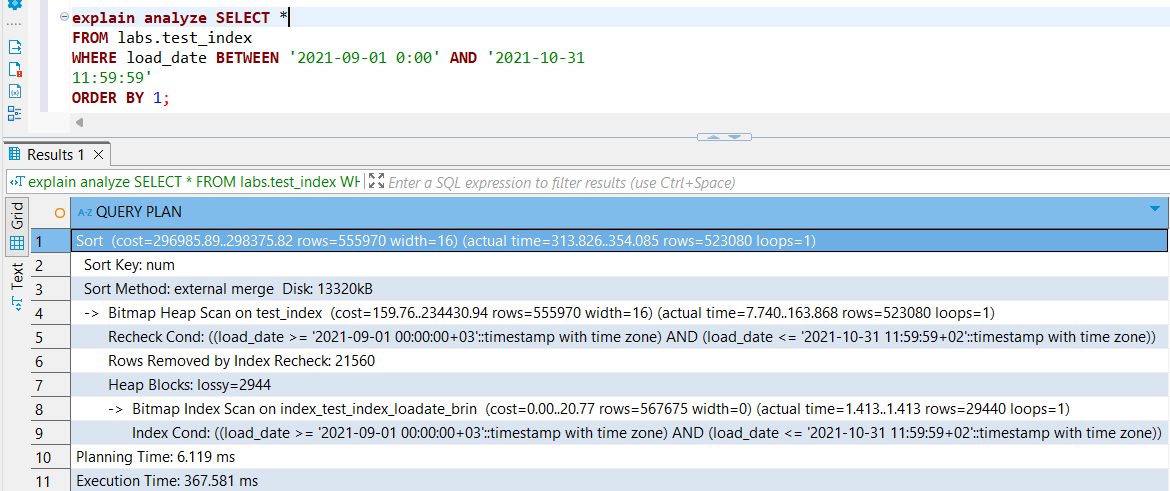
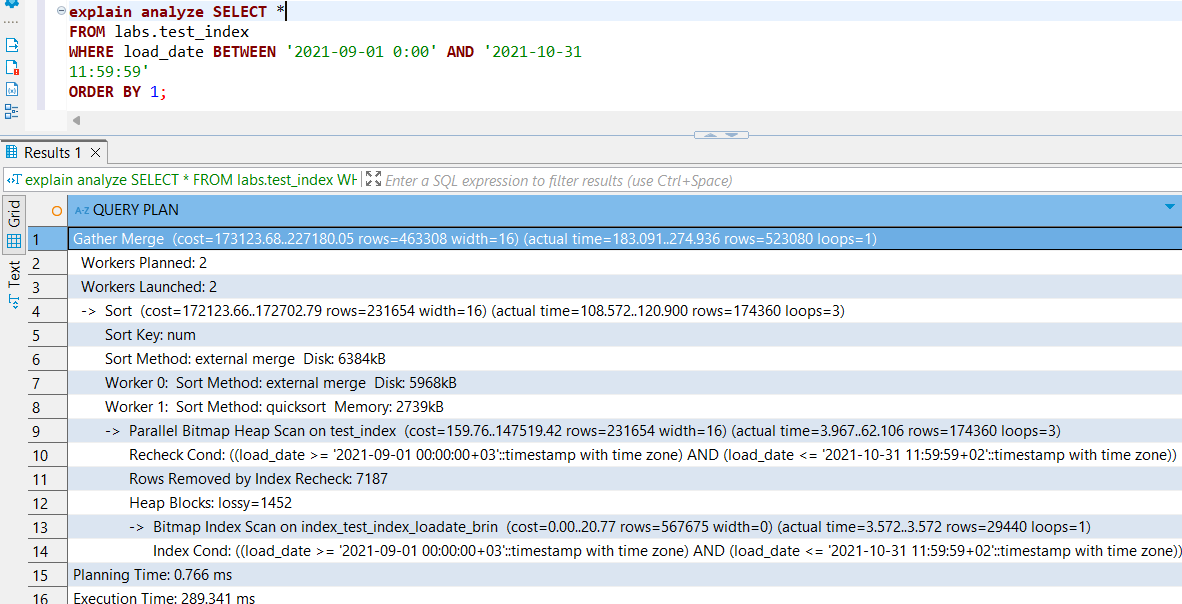


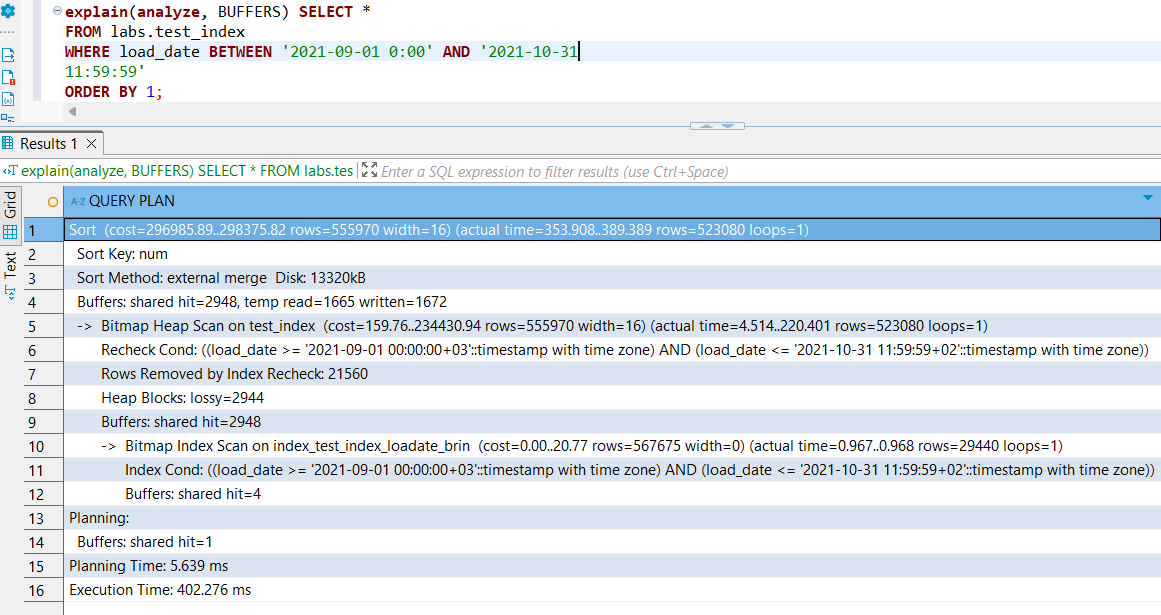
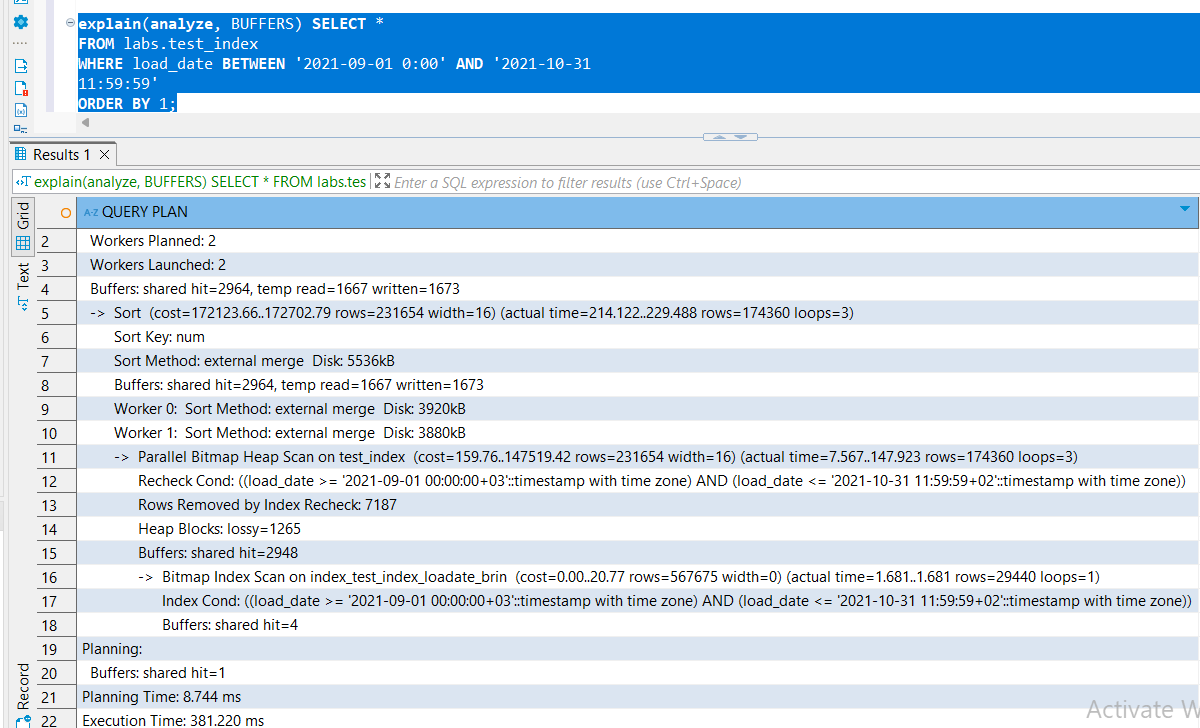
2)Create a **BRIN** index on the load\_date column:



3)Check the plan again (twice, at least) using:







The first thing I noticed is that with the **B-tree index**, the query planner did **not opt for parallel execution**, likely because the index itself was efficient enough to make parallelism unnecessary or even more expensive. In contrast, with the **BRIN index**, the planner initially **chose to use parallel execution** until I manually disabled it (SET max\_parallel\_workers\_per\_gather = 0). This suggests that BRIN's coarser granularity may benefit more from parallelism to compensate for less precise filtering.

Another observation: with B-tree indexing, the **execution time decreased on the second run**, likely due to caching (i.e. data was already loaded into memory). However, for the BRIN index, the **execution time actually increased** on the second run, which could be due to BRIN's reliance on scanning more blocks, especially if the filtered data isn’t as physically localized.

When comparing EXPLAIN ANALYZE runs:

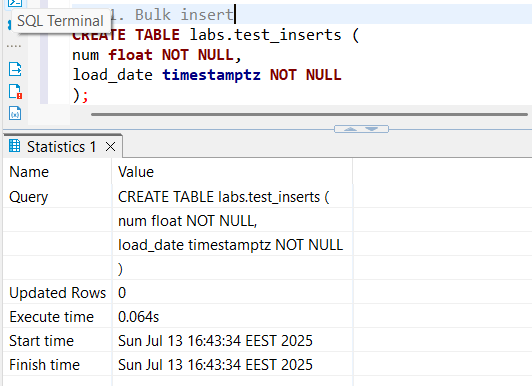
* On the **first run**, BRIN showed **better performance than B-tree**, likely due to less index overhead and use of parallelism.
* However, in the **second run**, using EXPLAIN (ANALYZE, BUFFERS) for both, BRIN showed **higher execution times** than B-tree — both with and without parallelism. This may indicate that BRIN, while space-efficient, can become less performant than B-tree under repeated access patterns where precision matters more than raw speed.

### 2. Adding data with insert and copy

### 2.1 Task 3 – Bulk Insert

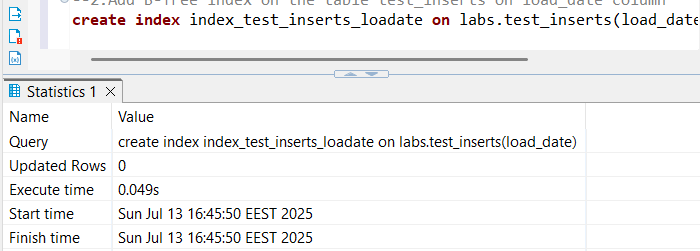
**Task Result**:  
 Provide SQL queries where needed. Describe **what happened and why**, including **screenshots where applicable**.

### Step 1: Create New Table test\_inserts



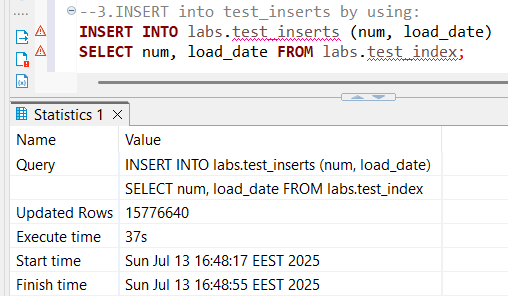
### Step 2: Add a B-Tree Index

Add a B-Tree index on the load\_date column of the test\_inserts table:



### Step 3: Perform Bulk Insert

Insert data from the test\_index\_plan table into test\_inserts:

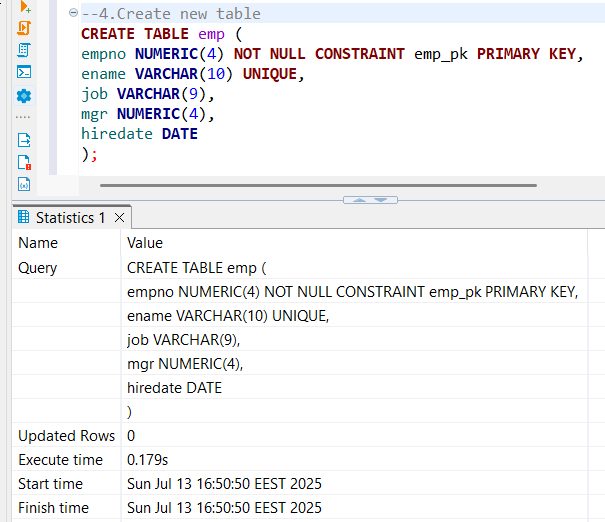


It took longer to bulk insert data into my table from another table than it took to generate that data initially. This may be because, beyond just generating the data, our program has to update and check indexes we imposed on the target table, which can slow down the process. Additionally, performing the entire bulk insert within a single transaction can be more expensive than breaking it into smaller chunks across multiple transactions.

Inserting data into a table—especially in bulk—involves several overheads, such as:

* Writing to disk, which is slower than working in memory
* Updating indexes on the target table
* Enforcing constraints like primary keys, foreign keys, and unique constraints
* Logging changes for durability (transaction logs or write-ahead logs)
* Potentially locking the table or rows during the insert

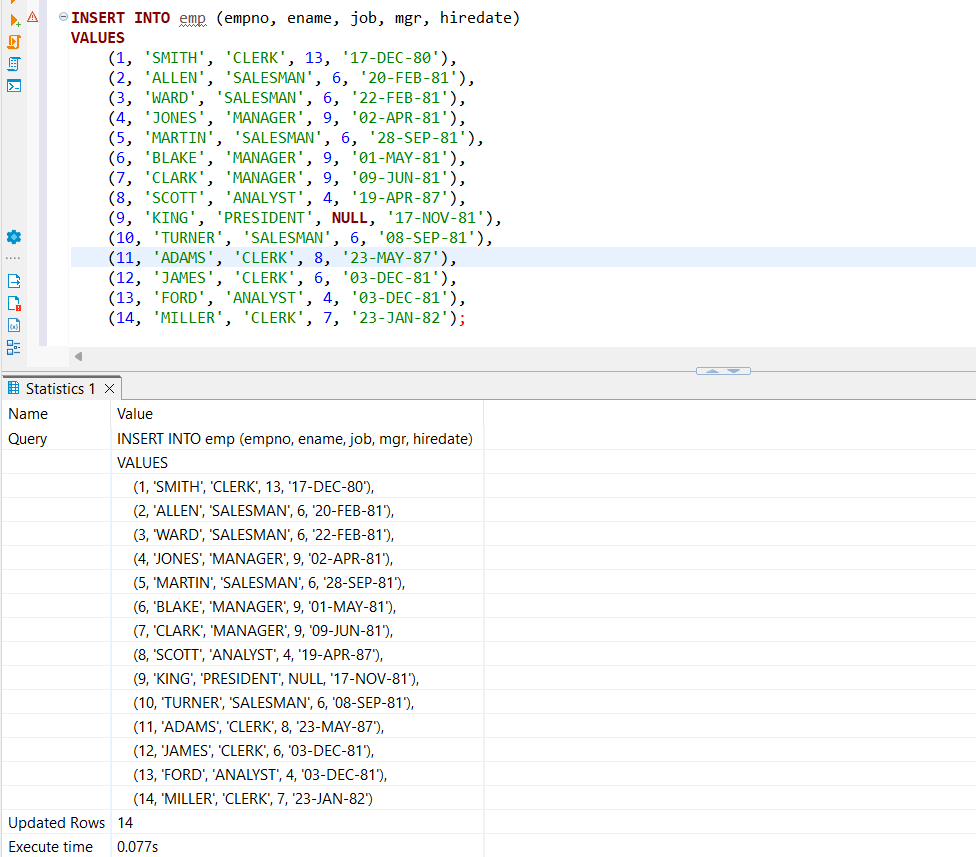
### Step 4: Create Table emp



ℹ️ **Note**: *Do not delete the emp table after this module.*

### Step 5: Rewrite INSERT Statements for Efficiency

Rather than multiple single-row inserts, group inserts in a more efficient form:

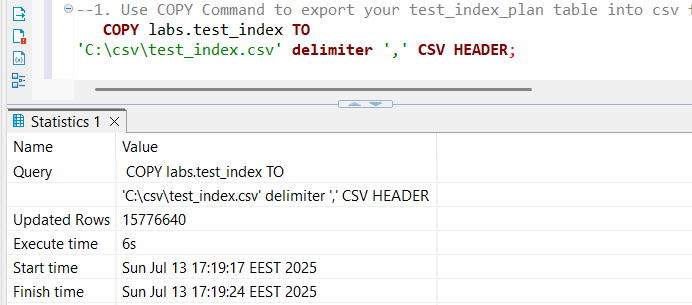


### 2.2 Task 4 – Copy Command

**Task Result**:  
 Provide SQL queries where needed. Describe **what happened and why**, including **screenshots where applicable**.

### Step 1: Export Full Table to CSV

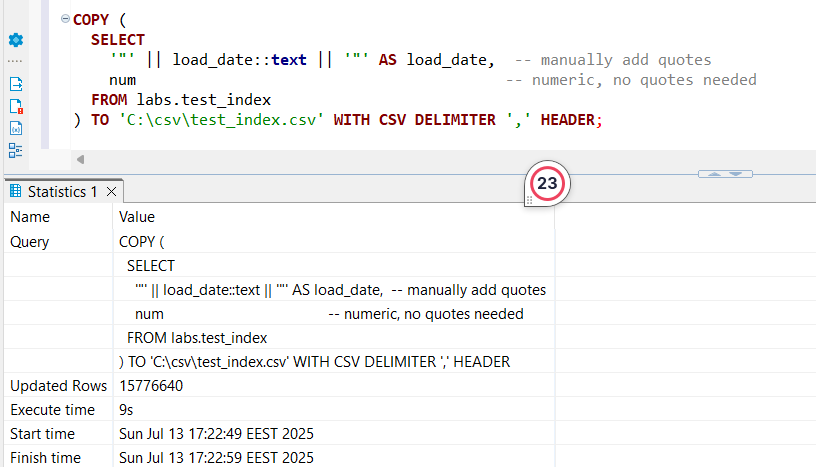
Use the COPY command to export the entire test\_index table to a CSV file:



### Modify Export Format

Update the COPY command to export:

* load\_date **with quotes**
* num **without quotes**

****

**It took 3 seconds longer, as this time we had an extra operation to be done inside the copy.**

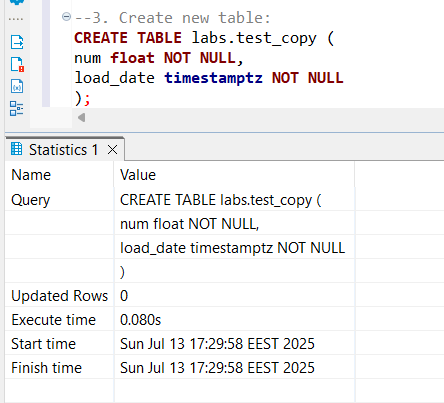
### Step 2: Export Filtered Rows to CSV

Export only rows where load\_date is between '2021-09-01 0:00' and '2021-09-01 11:59:59':

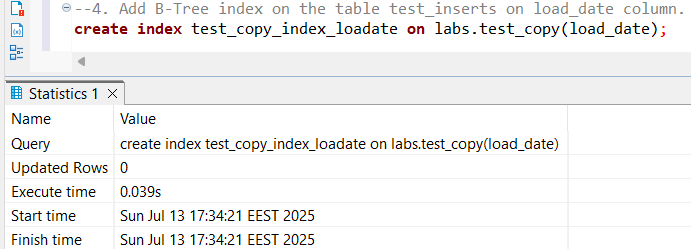
### 

The process was faster than before, suggesting that filtering is not a particularly expensive operation for the COPY command; however, the amount of data (space it occupies) still directly impacts the time it takes.

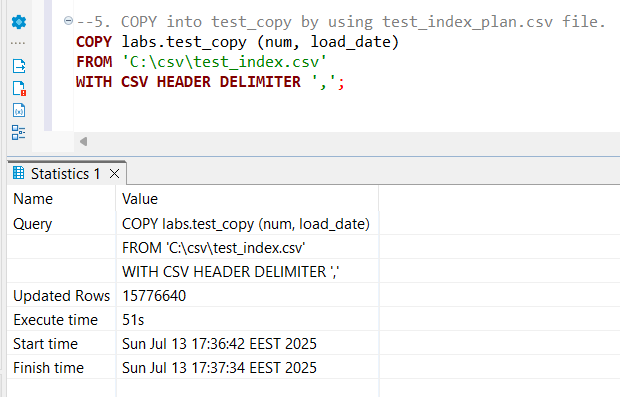
### Step 3: Create New Table test\_copy



### Step 4: Add B-Tree Index on test\_copy



### Step 5: Import Data Using COPY



So, apparently copying (using the COPY command) from a CSV file that we populated from a table and then copying from the CSV to populate another table takes longer (51 seconds) compared to directly inserting the block of rows from the first table to the second (37 seconds), which makes sense because the CSV method involves additional overhead such as writing data to disk, reading it back, and parsing text, whereas direct inserts operate entirely within the database’s internal memory structures, avoiding the costly file I/O and text conversion steps.

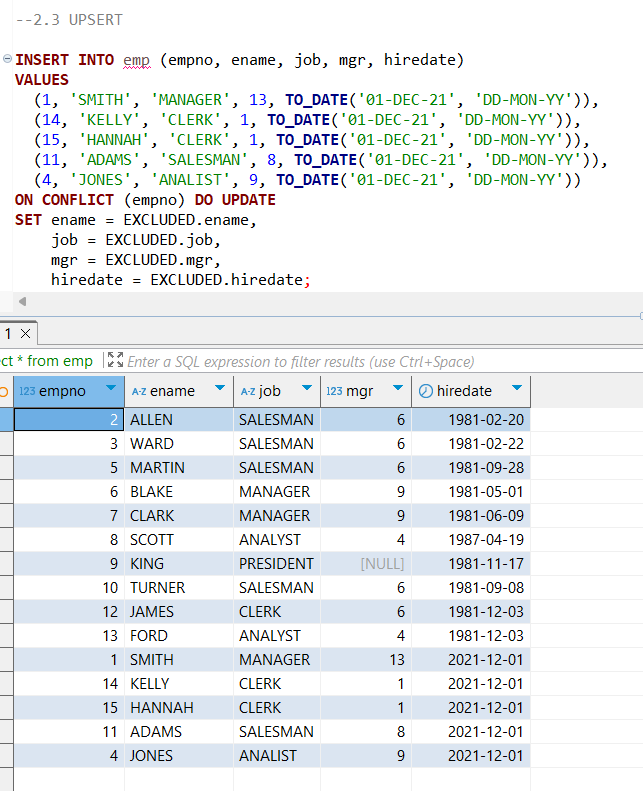
### 2.3 Task 5 – Upsert

**Task Result**:  
 Provide SQL queries where needed. Describe **what happened and why**, including **screenshots where applicable**.

### Step 1: Perform UPSERT on emp Table

Insert or update the following records in **one UPSERT statement**:

| **empno** | **ename** | **job** | **mgr** | **hiredate** |
| --- | --- | --- | --- | --- |
| 1 | SMITH | MANAGER | 13 | 01-DEC-21 |
| 14 | KELLY | CLERK | 1 | 01-DEC-21 |
| 15 | HANNAH | CLERK | 1 | 01-DEC-21 |
| 11 | ADAMS | SALESMAN | 8 | 01-DEC-21 |
| 4 | JONES | ANALIST | 9 | 01-DEC-21 |



When performing the UPSERT, we effectively implemented a **Slowly Changing Dimension (SCD) Type 1**. This means that on conflict—such as encountering the same primary key—we simply update the existing record with the new values provided, overwriting the old data.

In contrast, implementing an **SCD Type 2** requires preserving the history of changes by inserting new records with additional columns to track versions or effective dates. This cannot be achieved with a single UPSERT statement alone; it requires multiple queries or more complex logic.

Thus, in our UPSERT, we tested that when a conflict occurs (i.e., the primary key already exists), the existing row is updated with the new information instead of ignoring the change or inserting a new row.