

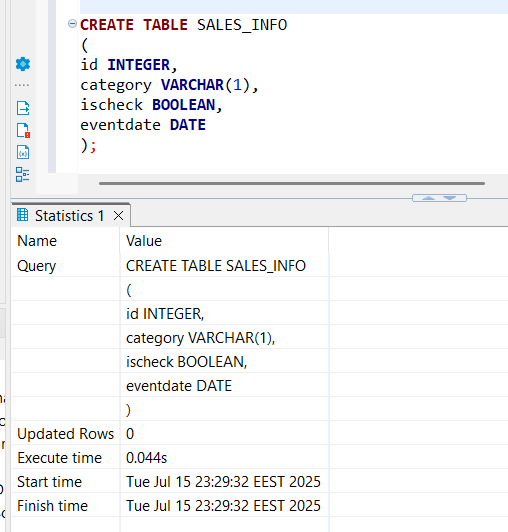
| Business Template  **Partitions** |
| --- |
|  |

### 1. Partitioning

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

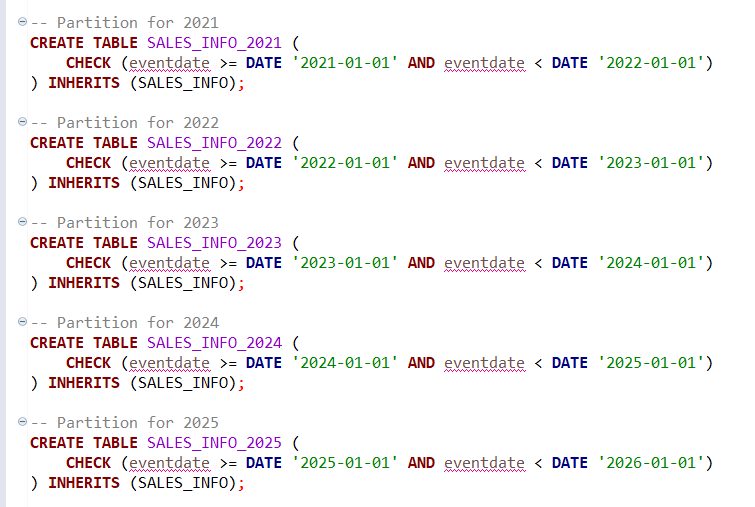
### 1.1 Task 1 – Use inheritance

Create table:



Apply partitioning by using inheritance:

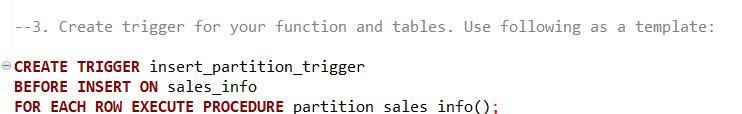
#### **Step 1: Create 4-5 child tables with partitioning by eventdate column. One partition is one year.**



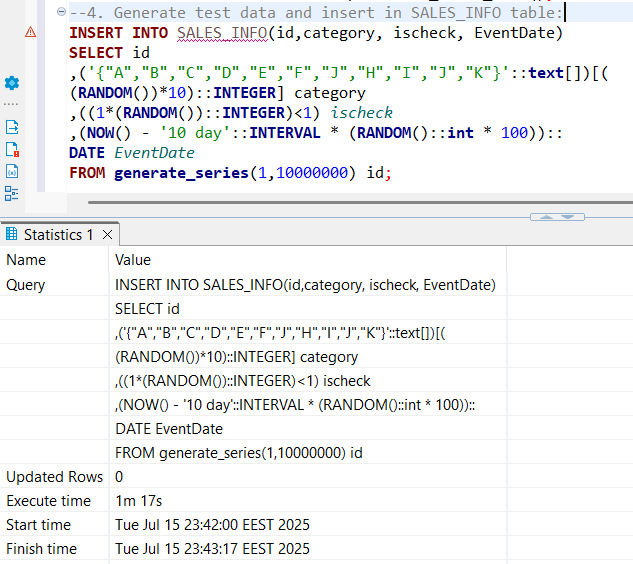
#### **Step 2:Create partition function for your tables.**



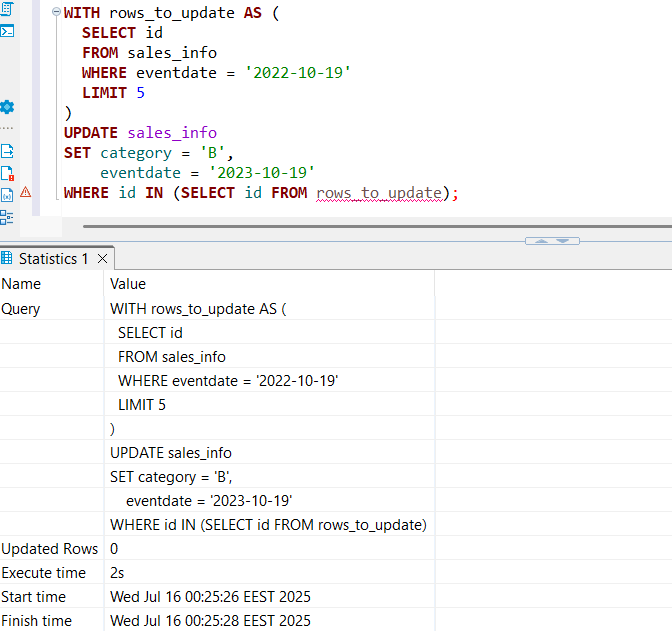
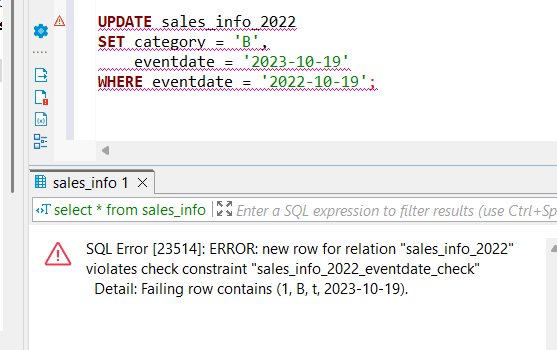
#### **Step 3: Create trigger for your function and tables.**



#### **Step 4:Generate test data and insert in SALES\_INFO table**



#### **Step 5:Update some rows in SALES\_INFO and set another eventdate.**

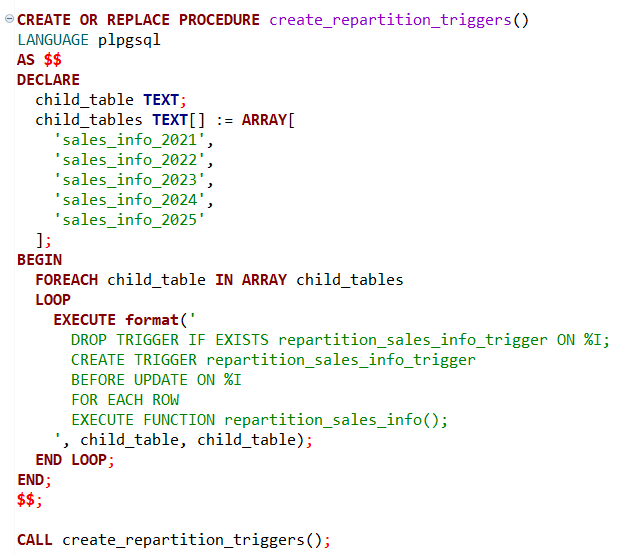
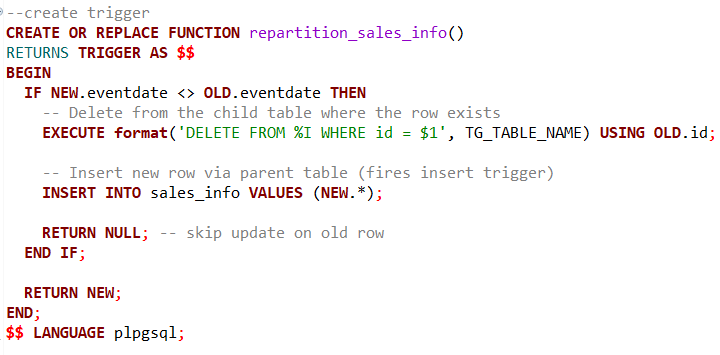


When I try to run a normal UPDATE on the parent table, I get an error because the update violates the partition check constraints in the child tables. This happens because rows physically reside in child tables, and updating a partition key column (like eventdate) in place can break the partition rules.

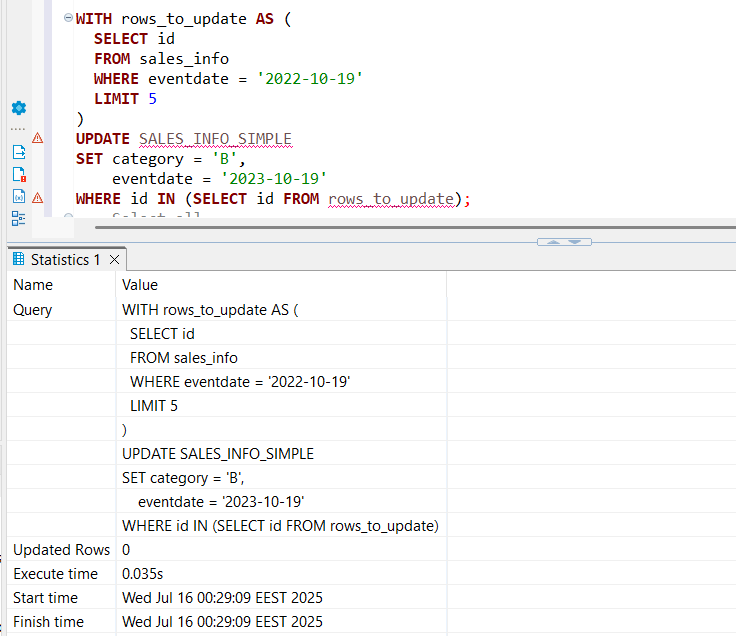
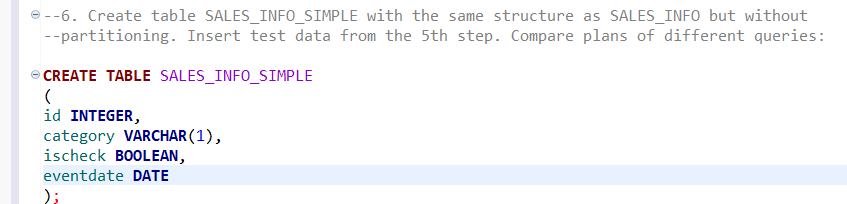
To handle this, I created a trigger function that fires **before UPDATE** on each child partition. When an UPDATE changes the partition key (eventdate), the trigger:

* Deletes the old row from the child table where it currently lives.
* Inserts the updated row into the **parent table**.
* The parent’s INSERT trigger then automatically routes this row to the correct child partition based on the new eventdate.

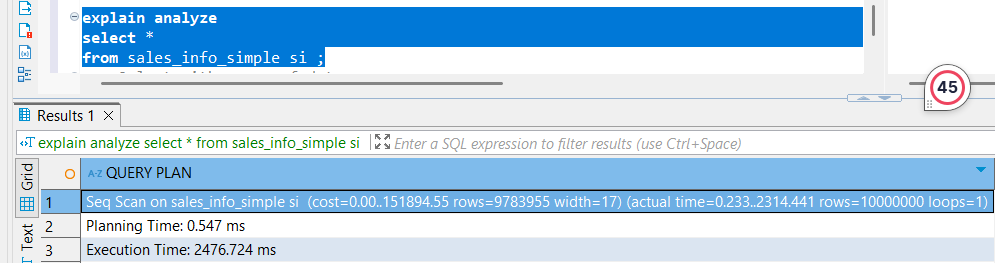
This way, the partitioning is maintained transparently during updates that change the partition key.



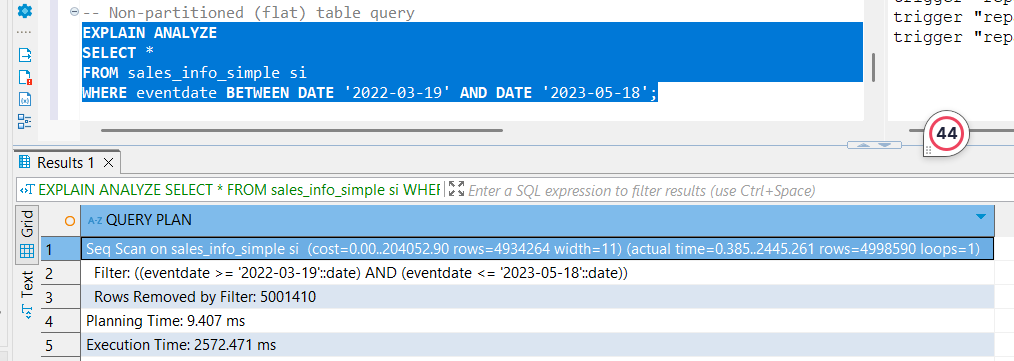
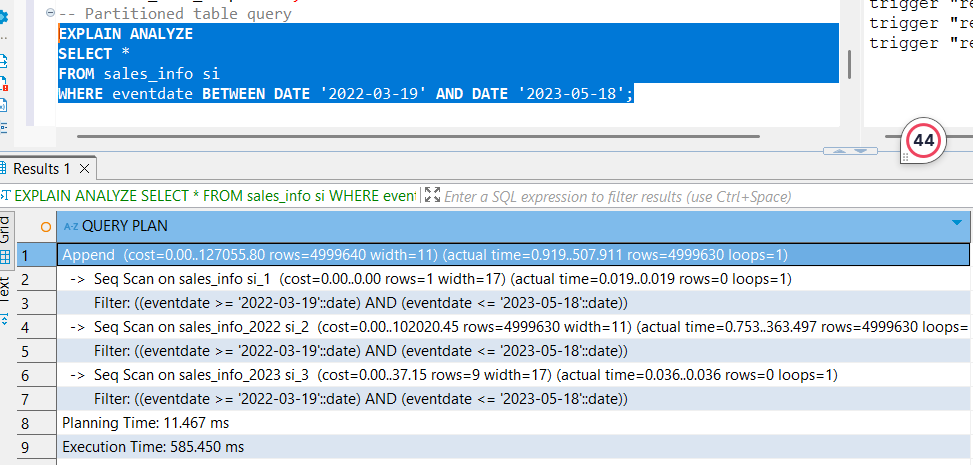
#### **Step 6:Create table SALES\_INFO\_SIMPLE with the same structure as SALES\_INFO but without partitioning. Insert test data from the 5th step. Compare plans of different queries:**



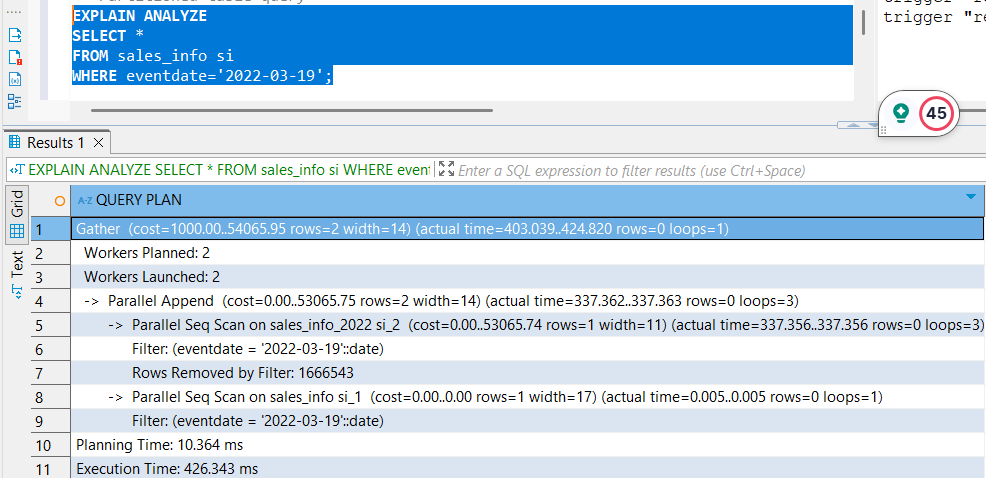
#### **Select all**



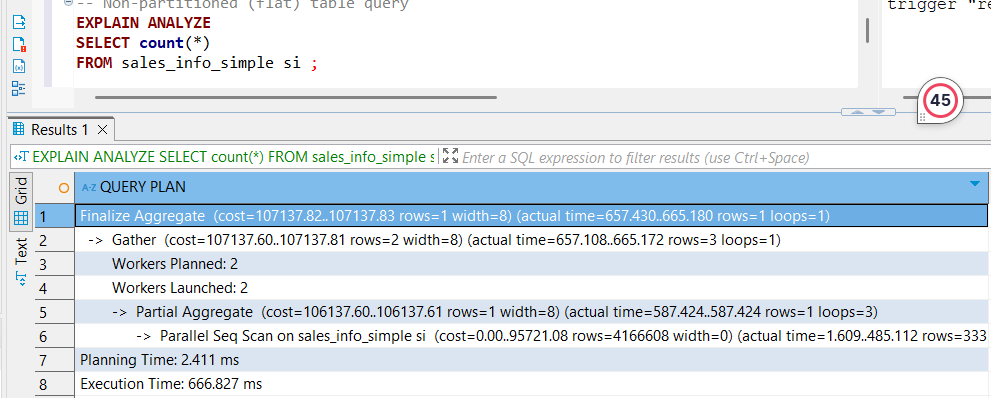
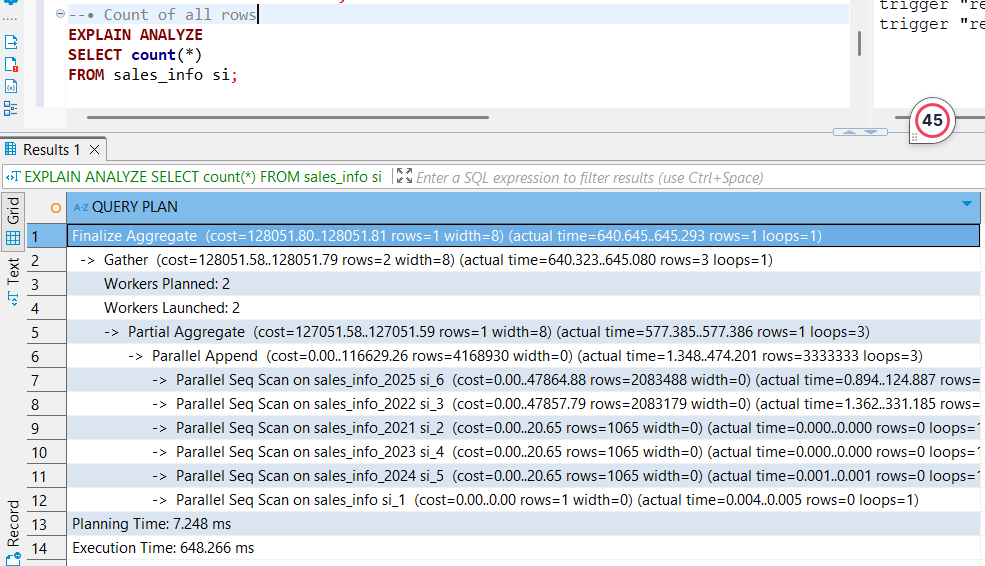
#### **Select with range of dates**



#### **Select exact date**



#### **Count of all rows**



#### **Count of rows with range of dates**

#### 

When using inheritance-based partitioning in PostgreSQL, the parent table (sales\_info) acts as a **logical container**, while the actual data resides in the child tables (sales\_info\_2021, sales\_info\_2022, etc.). These partitions are defined by eventdate, allowing PostgreSQL to route inserts (and with custom triggers, even updates) to the correct child table.

In performance testing, we observed that **partitioned tables consistently outperformed** their non-partitioned counterpart (sales\_info\_simple) across most query types — including:

* Selecting rows by a specific eventdate
* Counting all rows
* Counting rows within a date range

In these cases, **PostgreSQL used parallel execution** (e.g., 2 workers) across partitions and leveraged **partition pruning** or **constraint exclusion** to avoid scanning irrelevant child tables. This optimization significantly reduced execution time, even when planning time was slightly higher.

However, for queries like SELECT \* or broad range selects, both the partitioned and non-partitioned tables used **sequential scans** — though for the partitioned version, this meant **multiple independent scans**, one per child table. In such cases, the non-partitioned version may appear simpler, but the partitioned one still benefits from smaller, parallelizable scans, especially when partitions are large.

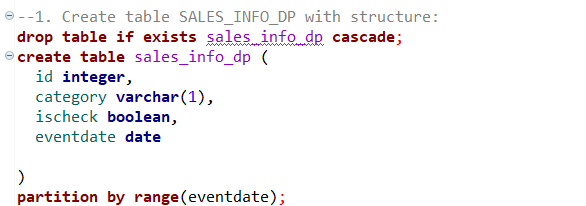
While inheritance-based partitioning introduces **additional complexity for updates** (requiring triggers when moving rows between partitions), it pays off in **read-heavy workloads** where **temporal or range-based queries dominate**. The combination of partition pruning and parallelism leads to a **noticeable execution time advantage**, especially at scale.

#### **Step 7:Delete one of partition (the oldest one). Create some general table like sales\_info\_3000 with the same structure as sales\_info and add it as new partition.**

#### 

### 1.2 Task 2 – Use Declarative Partitioning

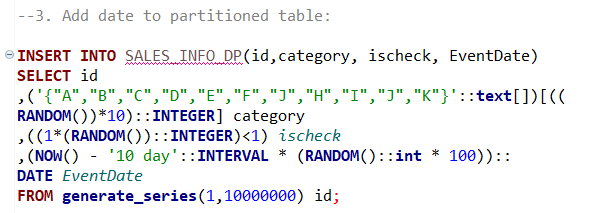
#### **Step 1: Create table SALES\_INFO\_DP.And make it partitioned by eventdate.**



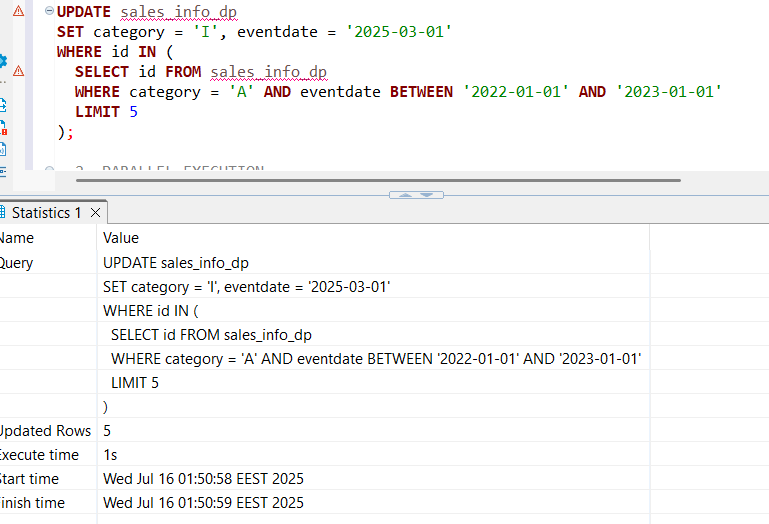
#### **Step 2:Create 4-5 child tables with partitioning by eventdate column. One partition is one year. Each child table should be partitioned by list on category column. Use 2 lists of values and one default partition here. As a result you should have SALES\_INFO\_DP table with composite partitioning by range and list.**

#### 

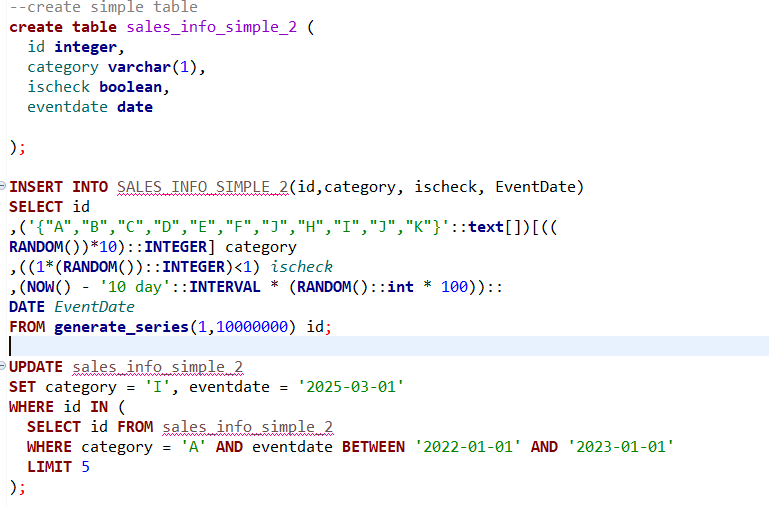
#### **Step 3:Add date to partitioned table.**



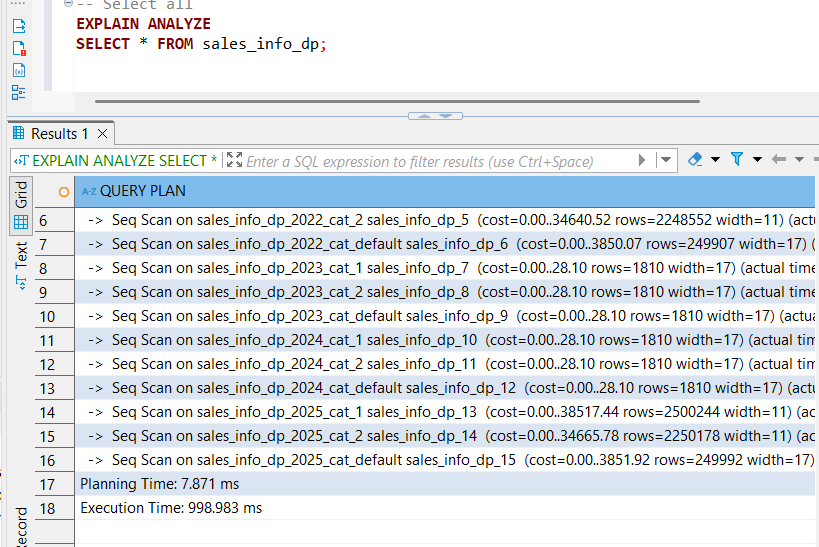
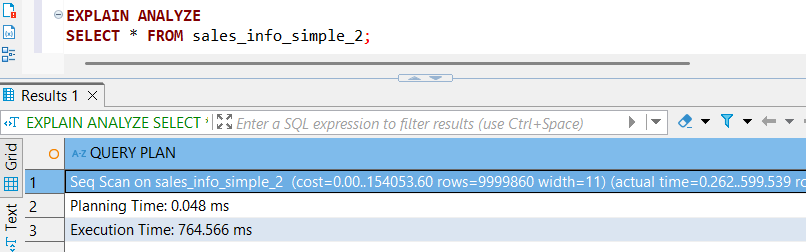
#### **Step 4:Update some rows in SALES\_INFO\_DP and set another category.**



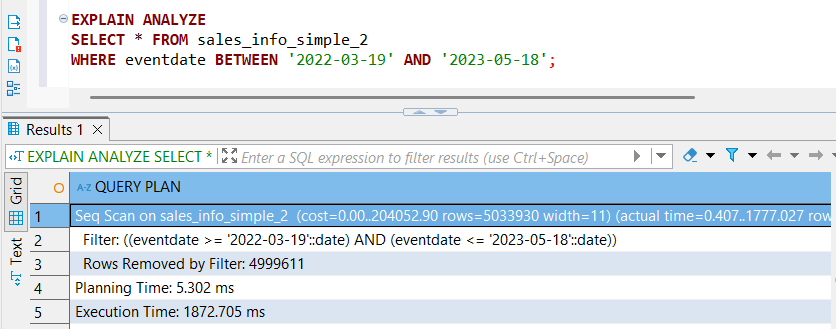
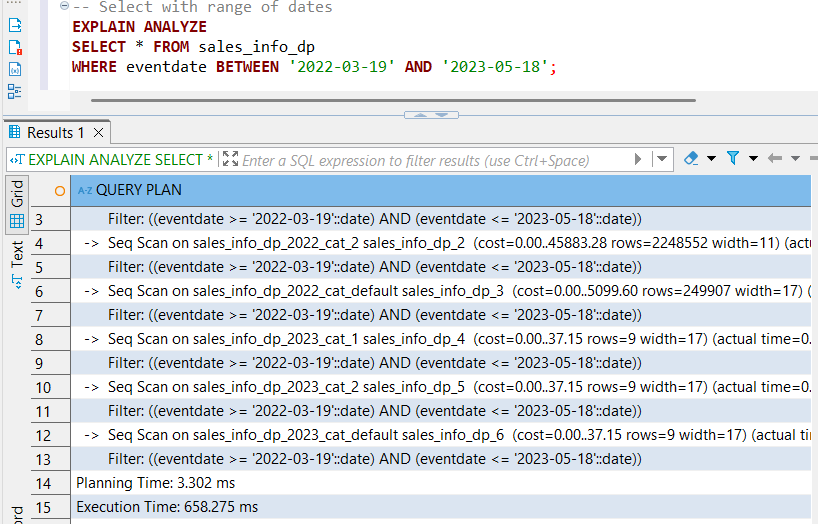
#### **Step 5:Compare plans of different queries for tables SALES\_INFO\_DP and SALES\_INFO\_SIMPLE:**



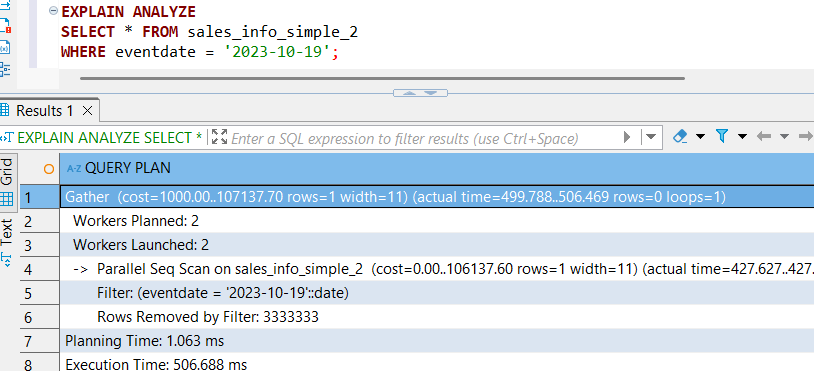
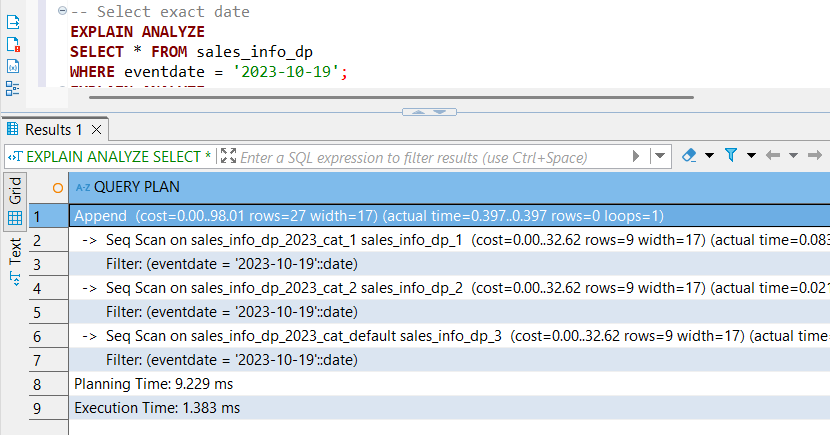
#### **Select all(>)**



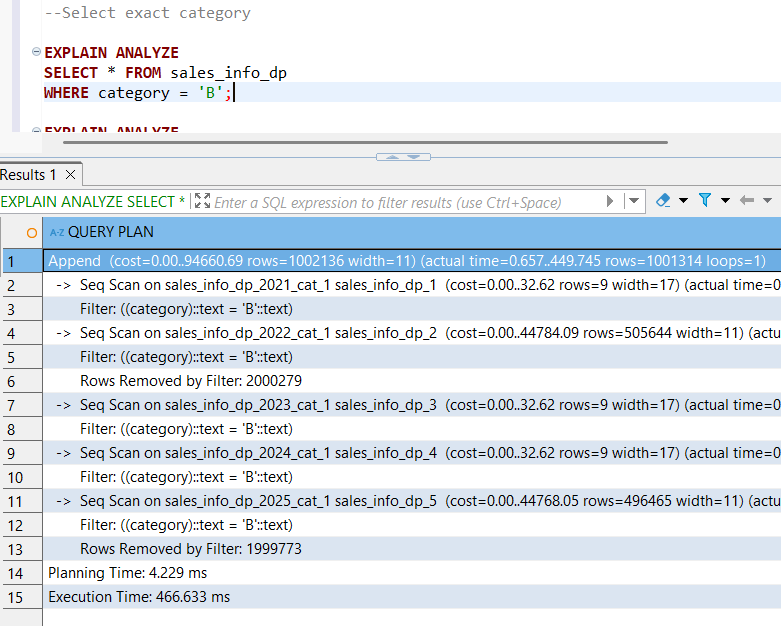
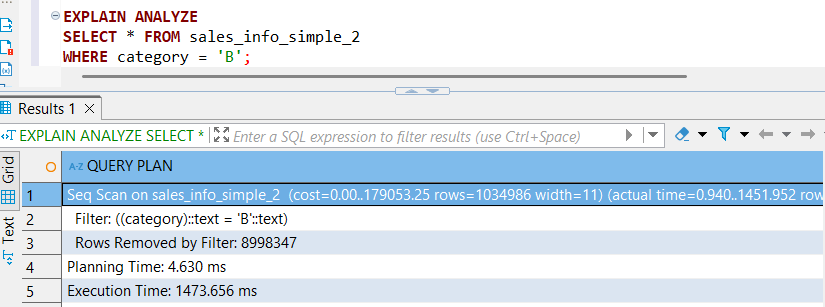
#### **Select with range of dates(<)**

****

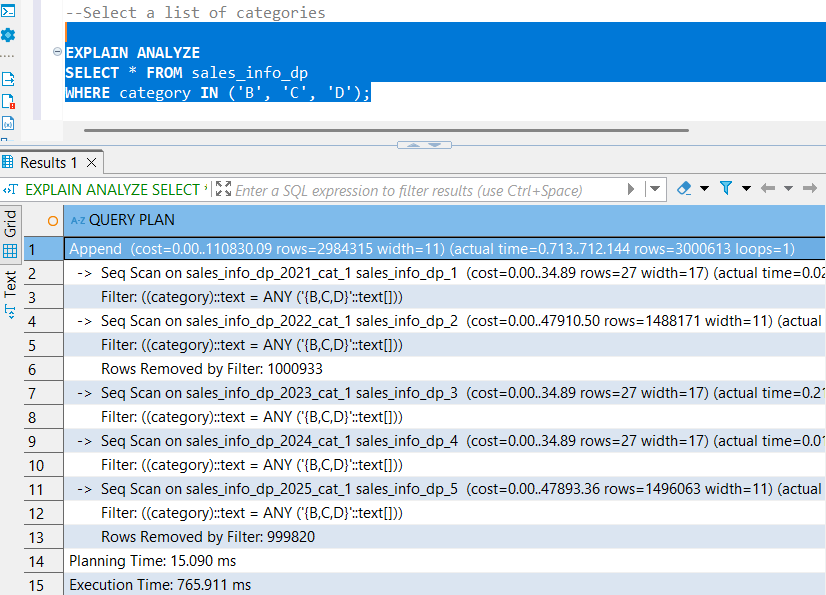
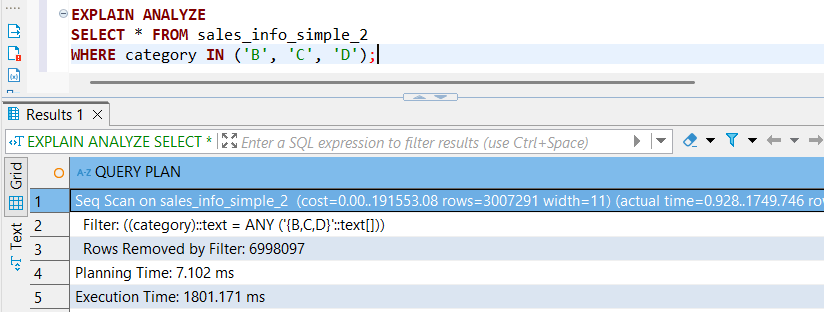
#### **Select exact date(<)(parallel processing with 2 workers for the simple one)**

****

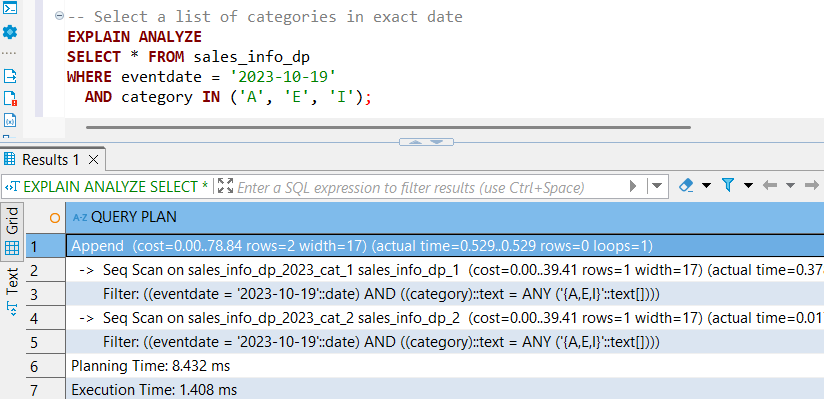
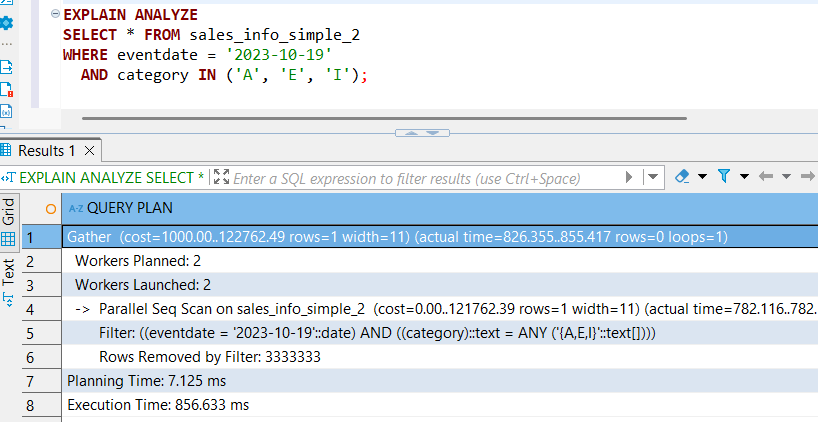
#### **Select exact category(<)**



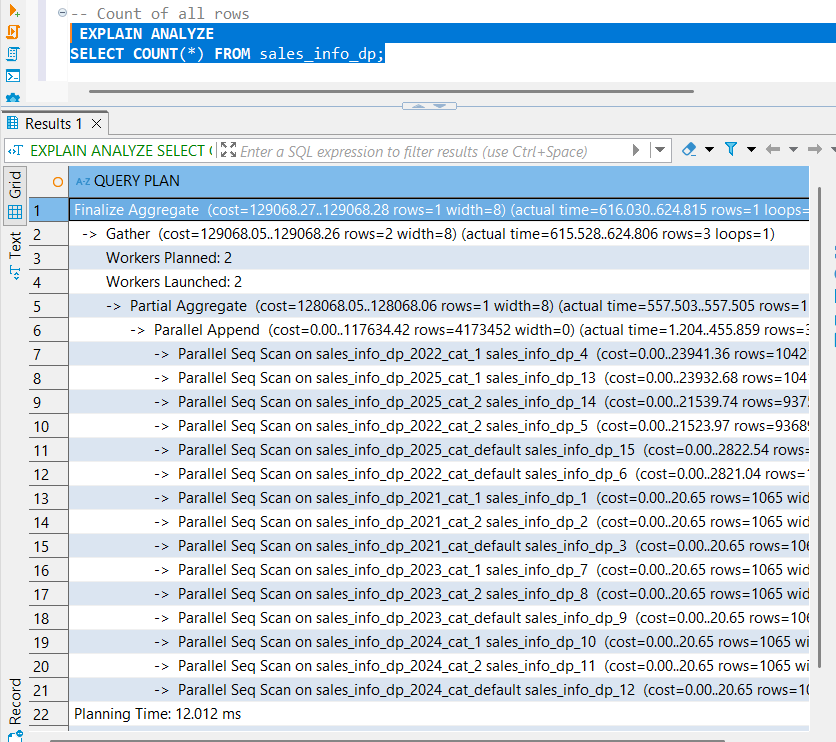
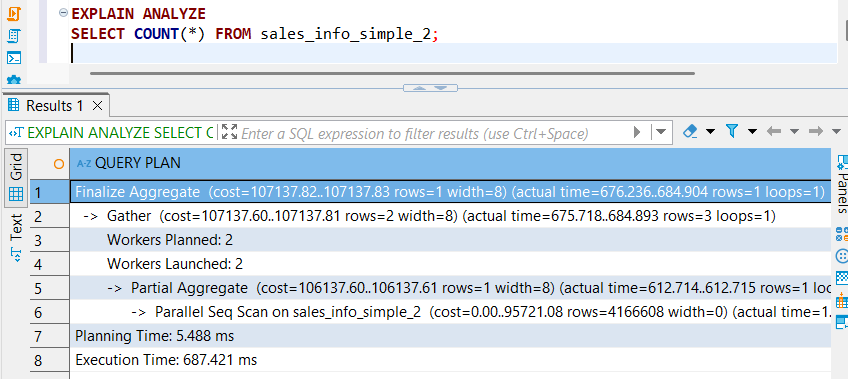
#### **Select a list of categories(<)**



#### **Select a list of categories in exact date(<)**



#### **Count of all rows(<) (both parallel seq scan)**



#### **Count of rows with range of dates(<)(both parallel seq scan)**

#### 

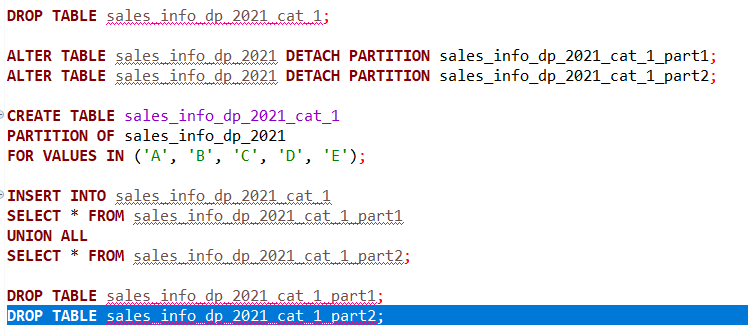
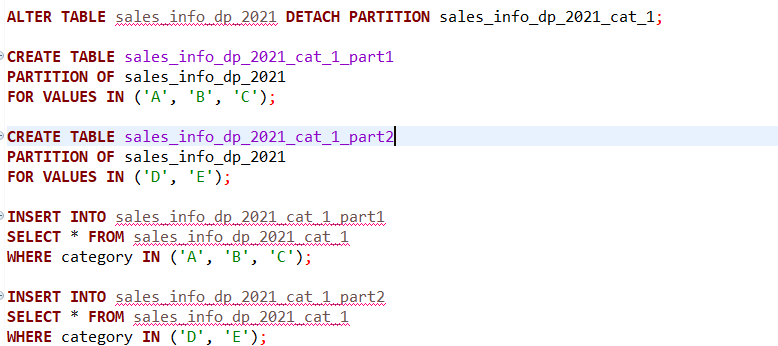
From our comparative analysis, the **composite partitioned table (SALES\_INFO\_DP) consistently outperformed** the unpartitioned table (SALES\_INFO\_SIMPLE) across almost all query types — except for full-table scans (SELECT \*), where the unpartitioned table was faster due to the absence of partition-planning overhead.

Key observations:

* **Partition Pruning Effectiveness**: For queries with eventdate, category, or both, the planner effectively **pruned irrelevant partitions**, significantly reducing scan time in the partitioned setup.
* **Parallel Execution**:  
  + **SALES\_INFO\_SIMPLE** more frequently utilized **parallel sequential scans** — even for selective queries like filtering by a single date.
  + **SALES\_INFO\_DP** also used parallelism, but it was more selective — showing up mainly in aggregation queries (e.g., COUNT(\*)) or wide scans with minimal pruning.
* **Performance Summary**:  
  + SELECT \* → **Faster in SIMPLE**
  + All filtered queries (by date, category, list, or combination) → **Faster in DP**
  + COUNT(\*) and range counts → **Faster in DP**, despite both using parallel scans.

Overall, while the **unpartitioned table benefits from simpler planning and frequent parallel scan use**, the **partitioned table gains superior execution efficiency** by leveraging **partition pruning**, making it ideal for large, filter-heavy analytical workloads.

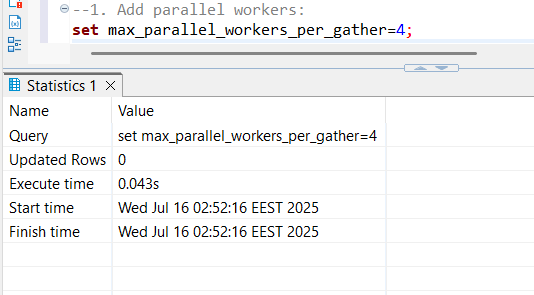
#### **Step 6:For one of the child tables with range partition by eventdate split one list partition for two. Return partition ('A','B','C','D','E'). Drop newly created (for ('A','B','C') and ('D','E')).**



### 2. Parallel execution

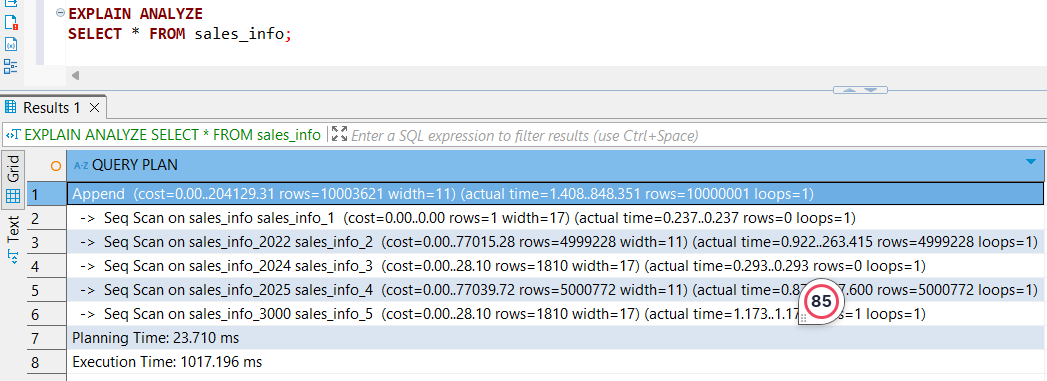
### 2.1 Task 3 – Use Parallel Quering

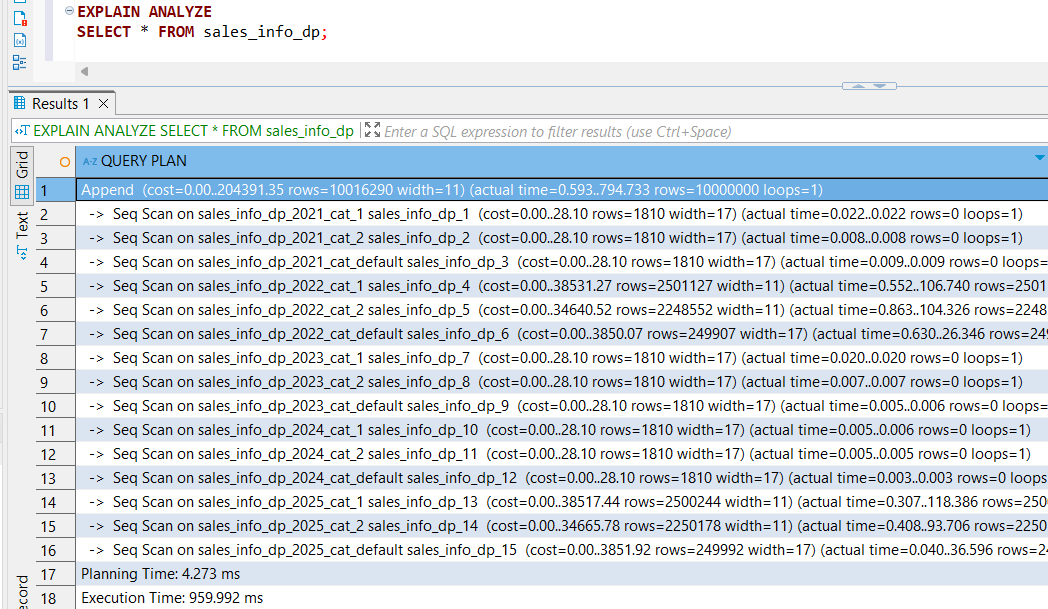
#### **Step 1: Add parallel workers.**

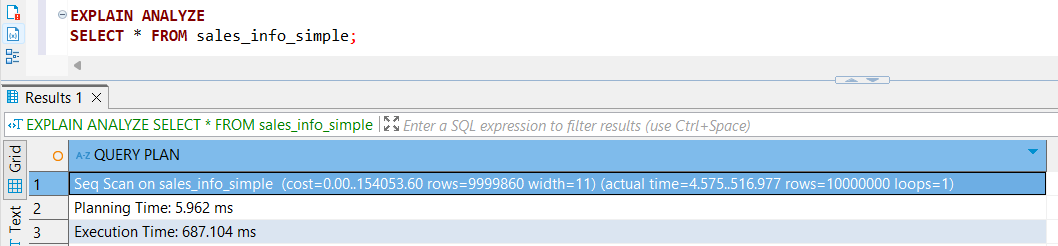


#### **Step 2:Analyze plans for tables SALES\_INFO, SALES\_INFO\_DP and SALES\_INFO\_SIMPLE by querying:**

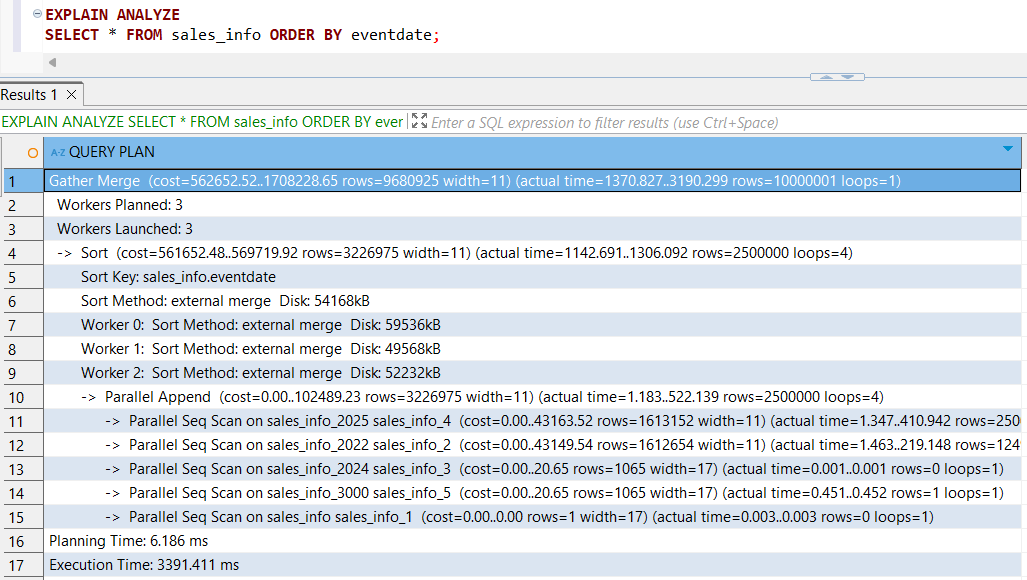
#### **Select all from tables(sales\_info\_simple>sales\_info\_dp>sales\_info)**

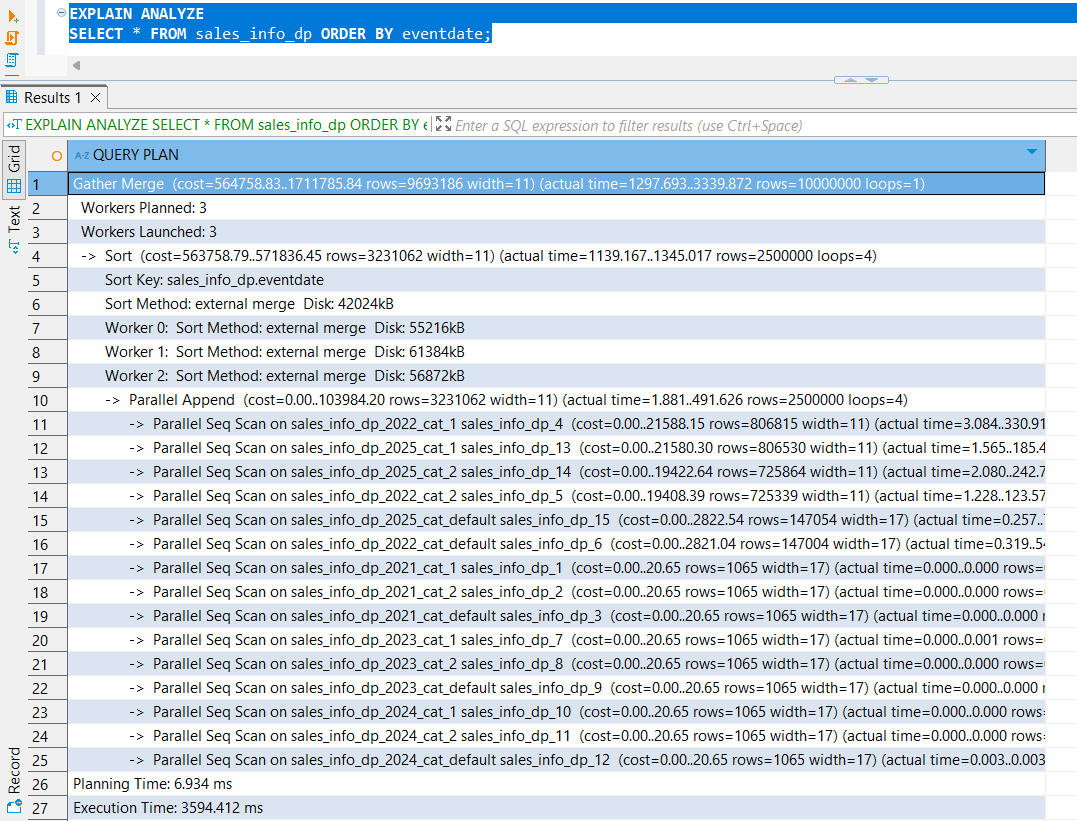


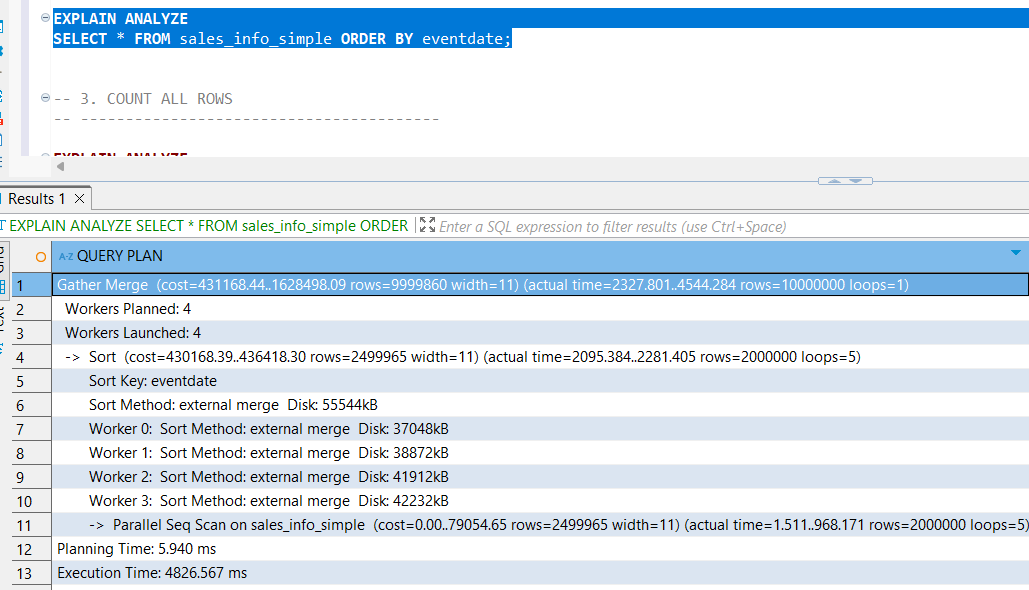




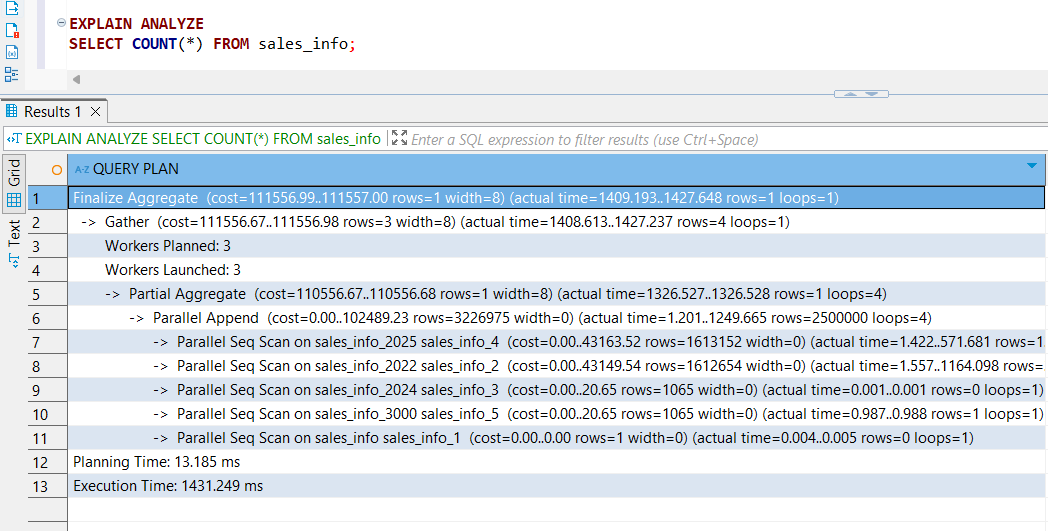
#### **Add order by eventadate(sales\_info\_simple<sales\_info\_dp<sales\_info)(all parallel processing with 3 workers except or the nonpartitioned with 4)**

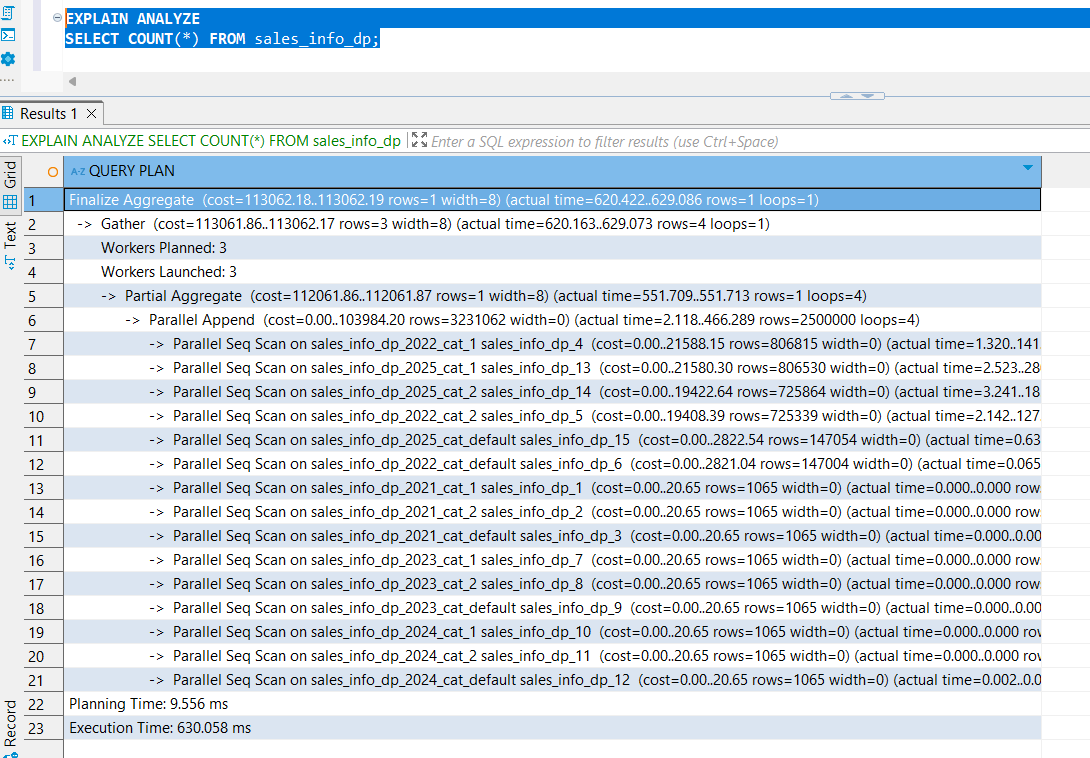






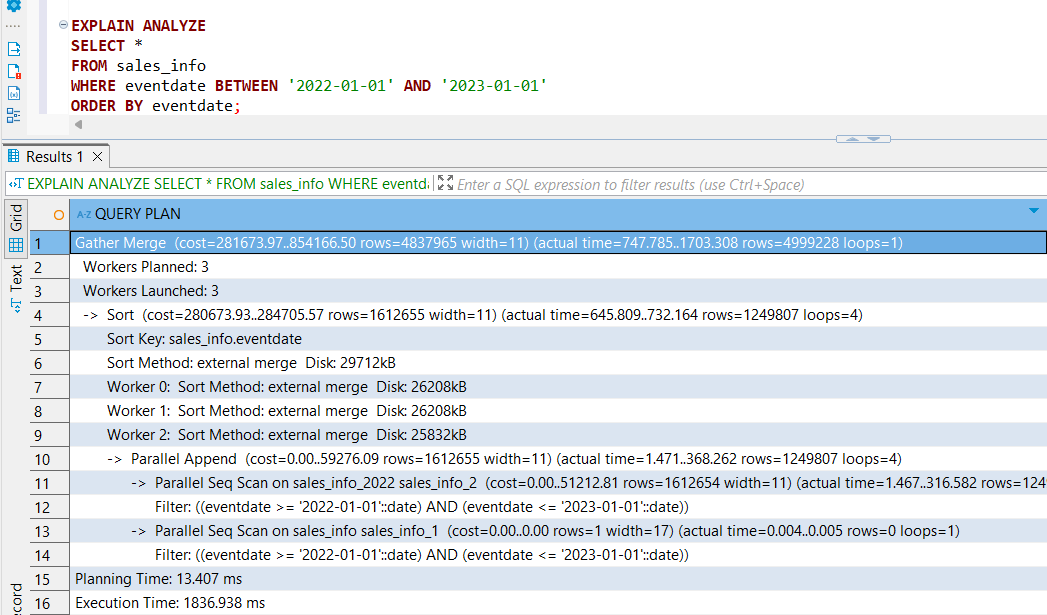
#### **Select count of all rows(sales\_info\_simple>sales\_info\_dp>sales\_info)(all parallel processing with 3 workers except or the nonpartitioned with 4)**

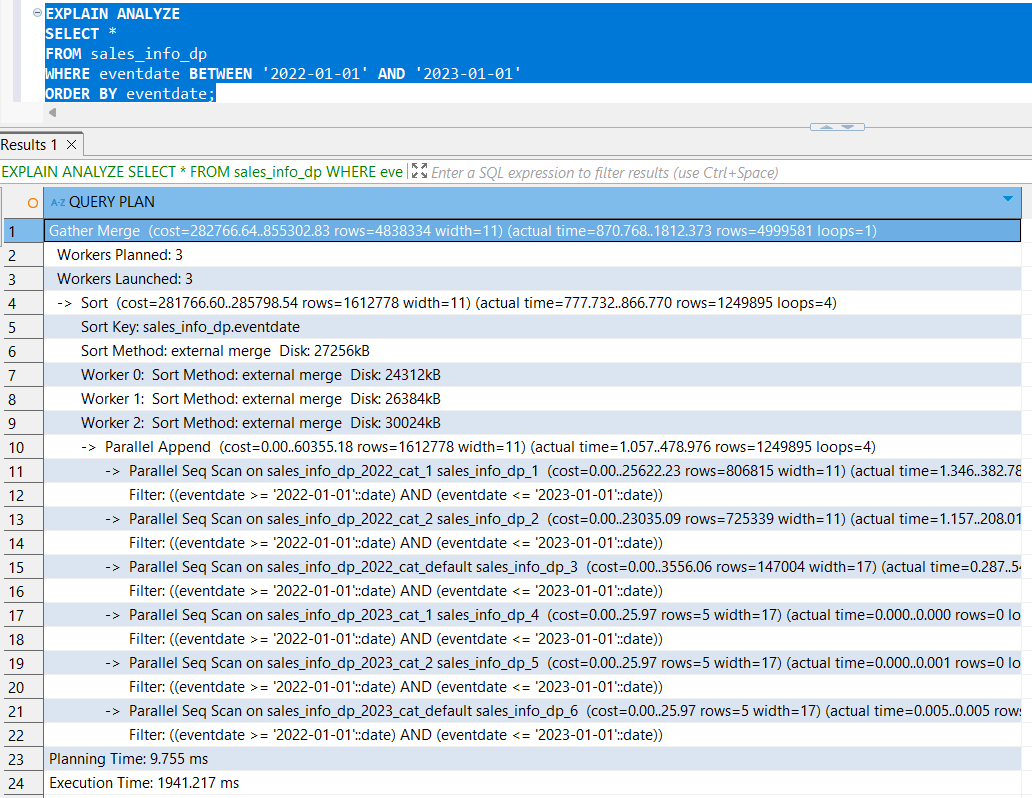


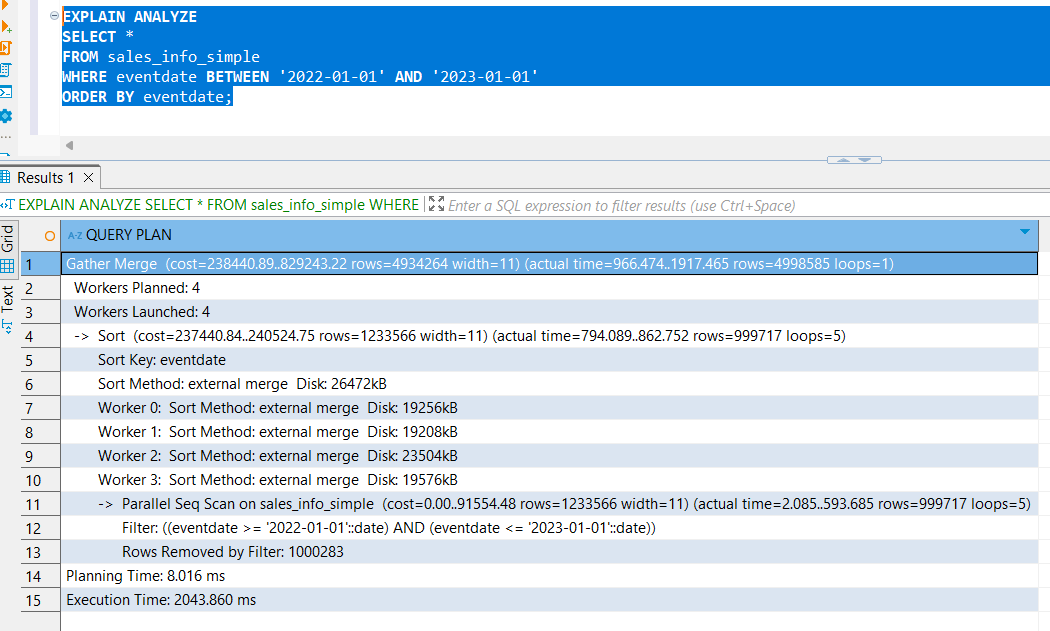




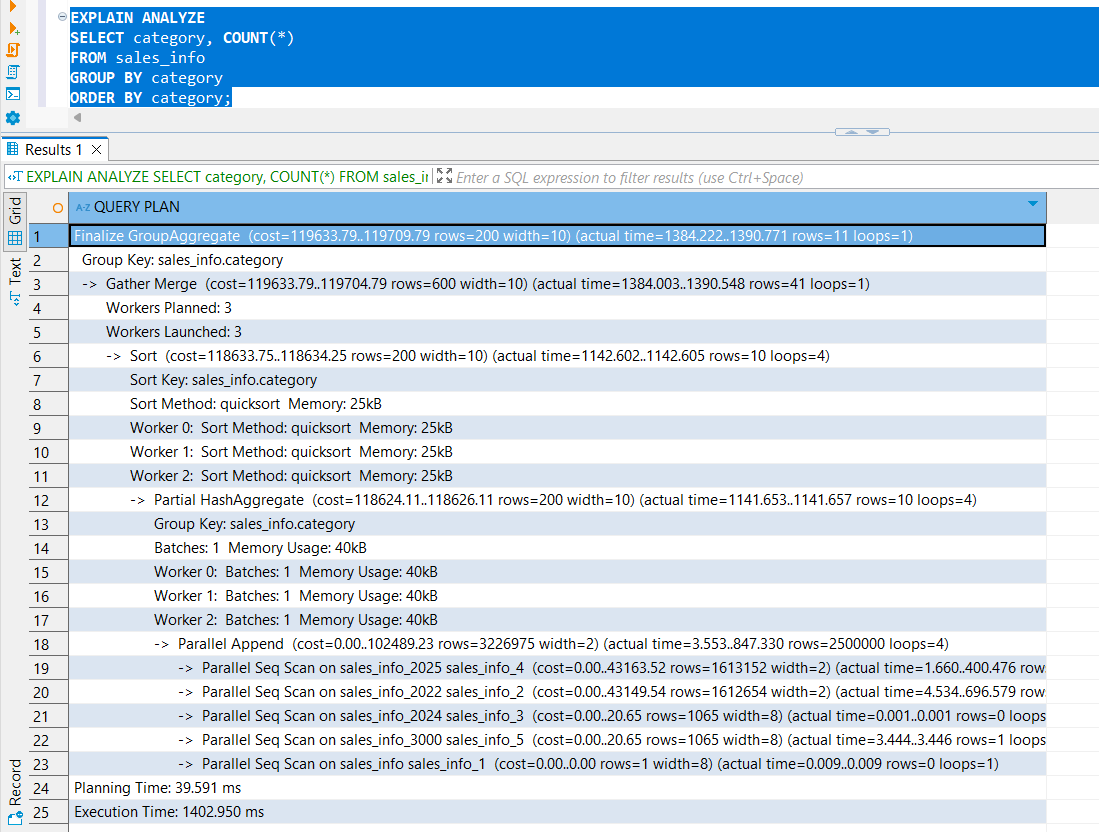
#### **Add range of dates(sales\_info\_simple<sales\_info\_dp<sales\_info)(all parallel processing with 3 workers except or the nonpartitioned with 4)**

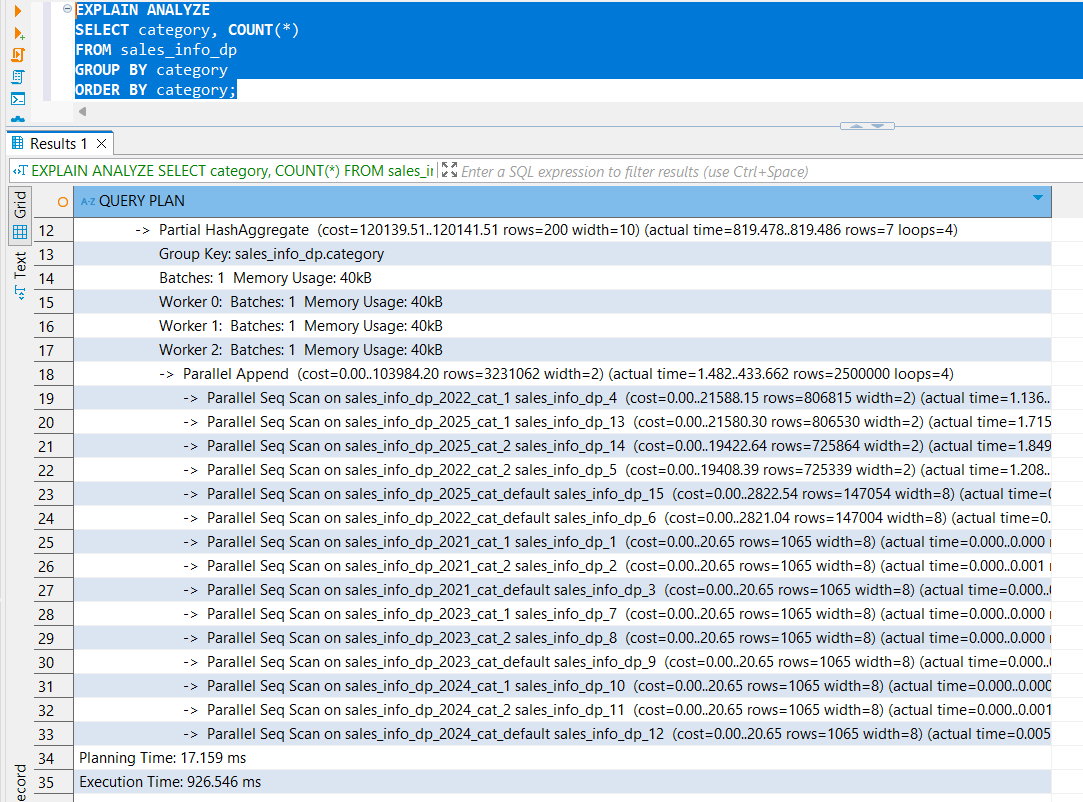


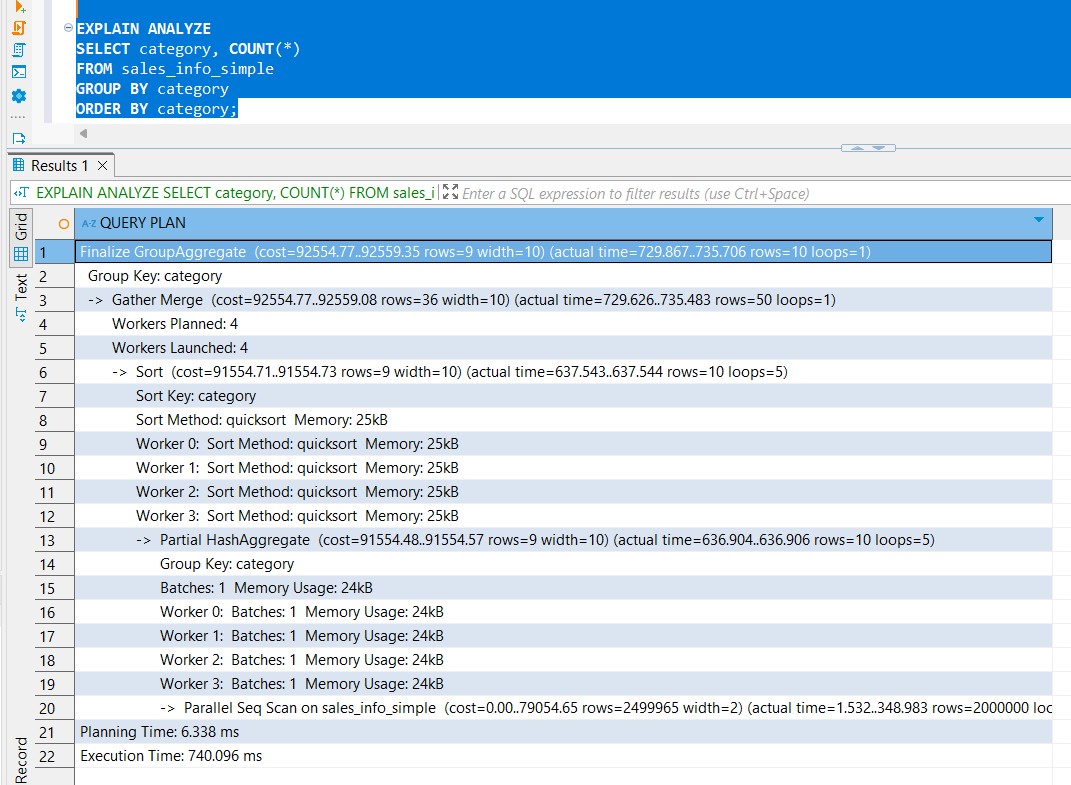




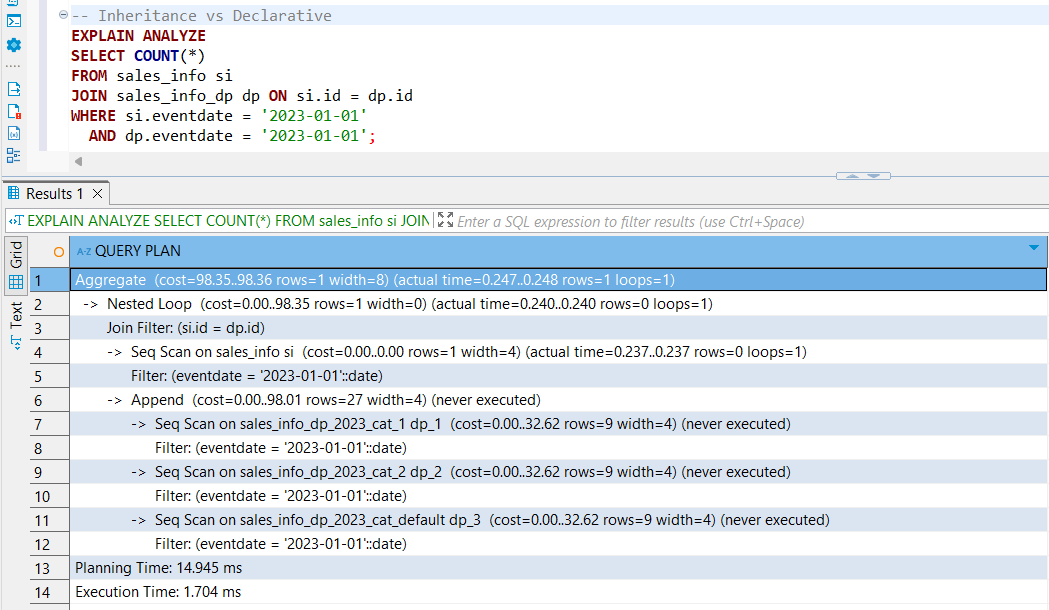
#### **Add grouping by category(sales\_info\_simple>sales\_info\_dp>sales\_info)(all parallel processing with 3 workers except or the nonpartitioned with 4)**

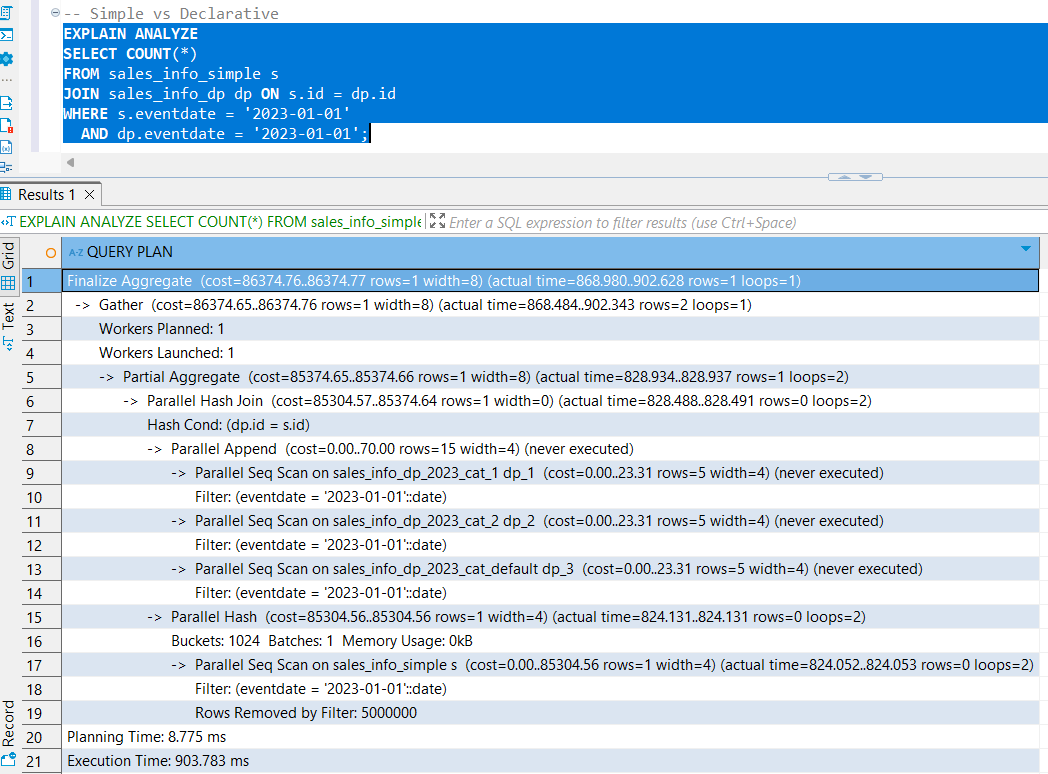






#### **Join SALES\_INFO and SALES\_INFO\_DP on id and count rows on exact date.(join sales\_info\_simple is slower than sales\_info\_dp and sales\_info)( parallel processing with 1 workers in the second one and nested loop on the shorter one)**





#### 1. SELECT all rows

#### Result: sales\_info\_simple > sales\_info\_dp > sales\_info

#### Explanation: The non-partitioned table delivered the fastest results due to a straightforward sequential scan without partitioning overhead. In contrast, both partitioned tables had to traverse partition metadata and structures, slowing down performance.

#### 2. SELECT with ORDER BY eventdate

#### Result: sales\_info\_simple < sales\_info\_dp < sales\_info

#### Explanation: Despite having more workers (4), sales\_info\_simple performed worse. Sorting a large, flat dataset is expensive. The partitioned tables benefited from working with smaller partitions, making sorting more efficient overall, even with fewer workers (3).

#### 3. SELECT COUNT(\*)

#### Result: sales\_info\_simple > sales\_info\_dp > sales\_info

#### Explanation: The simple table again excelled due to full parallel sequential scan and minimal structure. Partitioned tables incurred overhead from scanning each partition individually, despite parallel processing (3 workers).

#### 4. SELECT with a range of eventdate

#### Result: sales\_info\_simple < sales\_info\_dp < sales\_info

#### Explanation: Declarative partitioning significantly improved performance by pruning irrelevant partitions based on the date range. However, the flat table was still faster for broad filters due to full-table scan optimization and more available parallel workers.

#### 5. GROUP BY category

#### Result: sales\_info\_simple > sales\_info\_dp > sales\_info

#### Explanation: Flat tables allow the planner to efficiently distribute the grouping operation across multiple threads without partitioning logic. The older sales\_info structure lagged due to inefficient planner pruning.

#### 6. JOIN sales\_info\_dp and sales\_info on id with date filter

#### Result: sales\_info\_dp & sales\_info > sales\_info\_simple & sales\_info\_dp

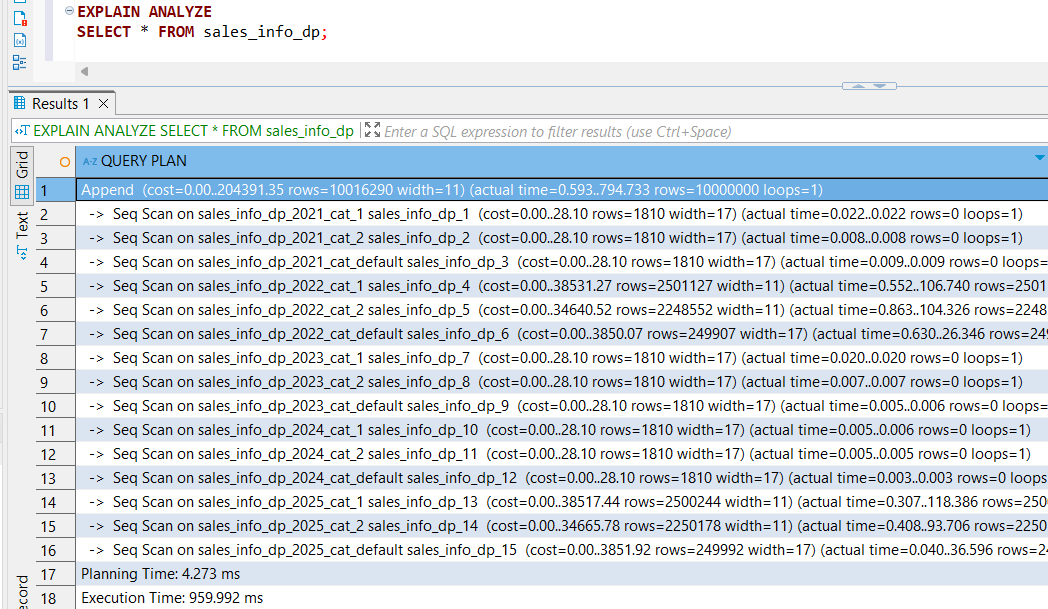
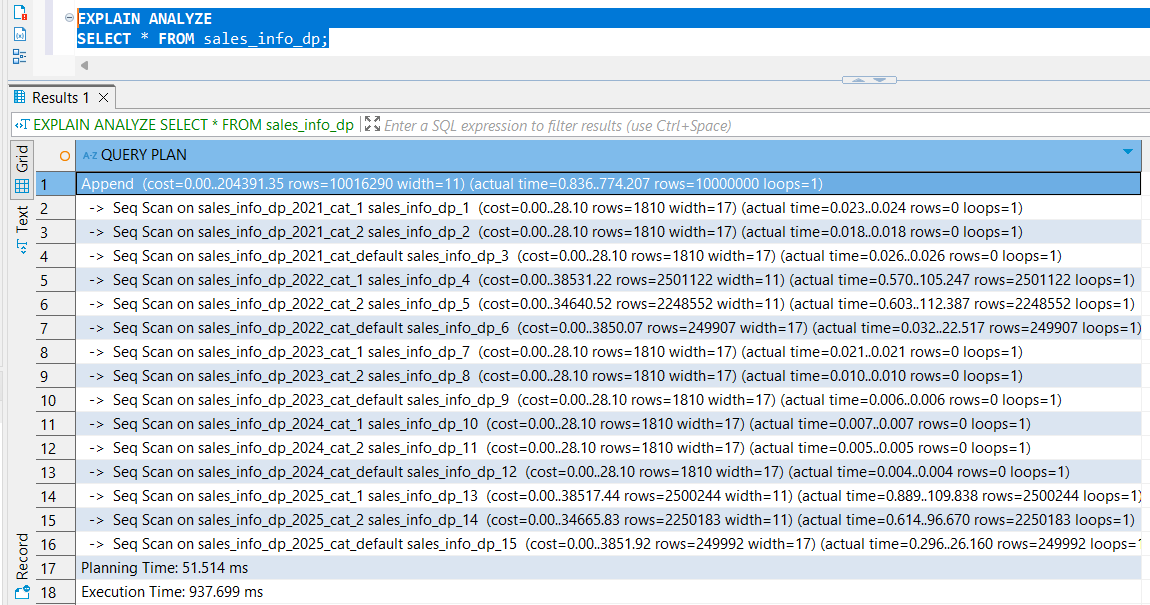
#### Explanation: Declarative partitioning join with sales info provided the best performance due to targeted pruning and parallel nested loop/hash join across smaller chunks. Surprisingly, the simple table join with the declarative partitioning table was slowest, as it required scanning the entire dataset and applying filters post-join.

#### 

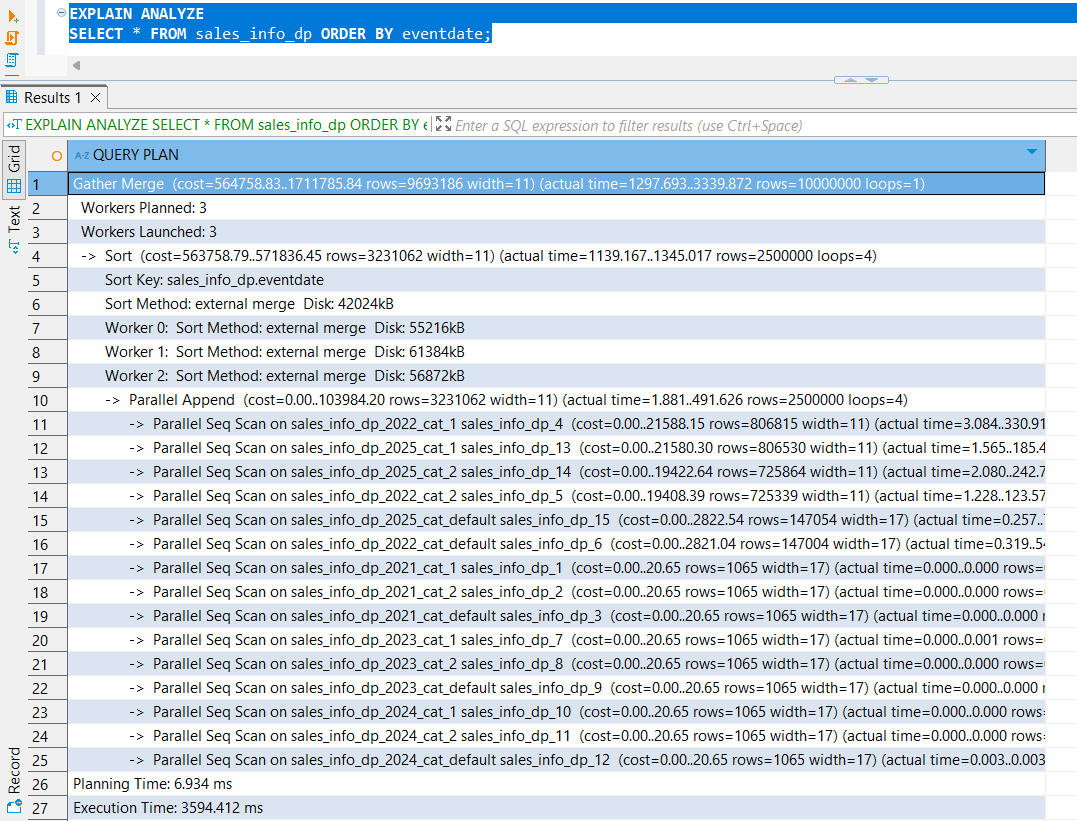
| **Query Type** | **Simple (Non-Partitioned)** | **Declarative (DP)** | **Inheritance-Based** |
| --- | --- | --- | --- |
| SELECT \* | No workers | No workers | No workers |
| ORDER BY | 4 workers (slowest) | 3 workers | 3 workers |
| COUNT(\*) | 4 workers (fastest) | 3 workers | 3 workers |
| WHERE date BETWEEN | 4 workers | 3 workers | 3 workers |
| GROUP BY | 4 workers (fastest) | 3 workers | 3 workers |
| JOIN | 4 workers (slowest) | 1–2 workers | 1–2 workers |

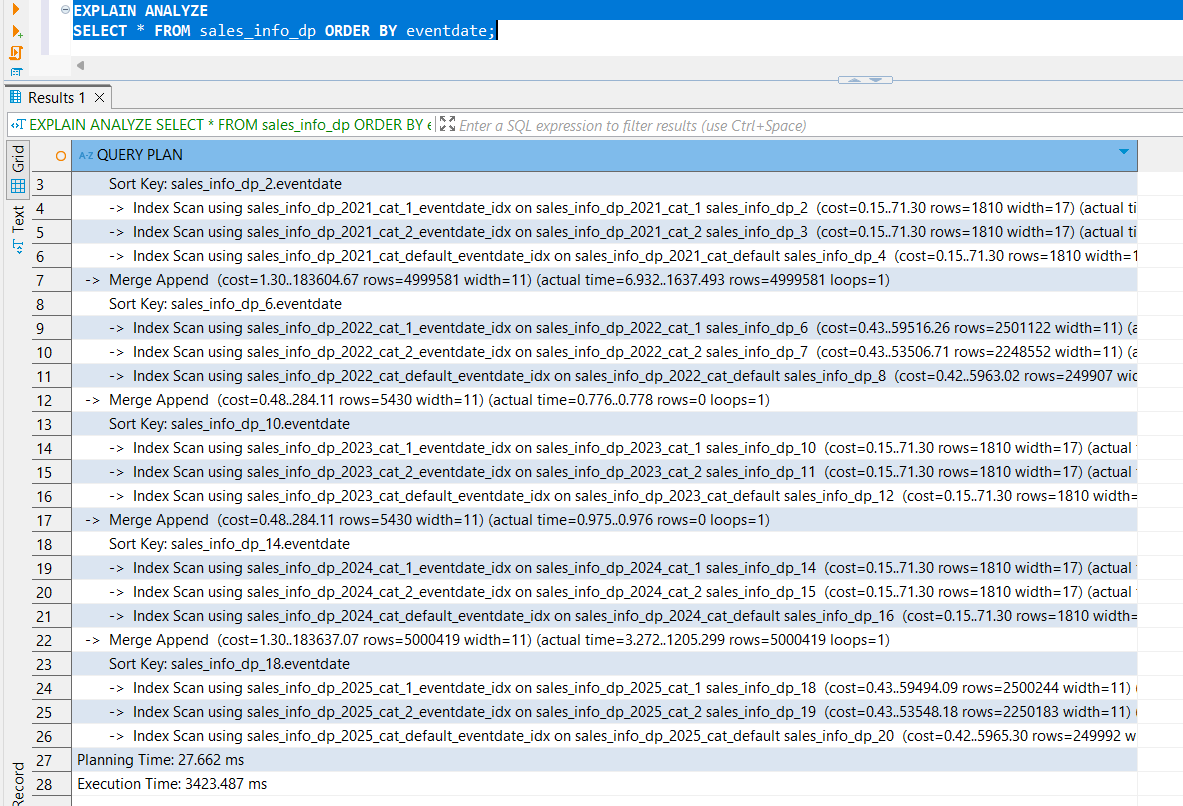
#### **Step 3: Add indexes on any of table with partitions. Check how plans are change.**

1. **Select all from tables(nonindex<index)**

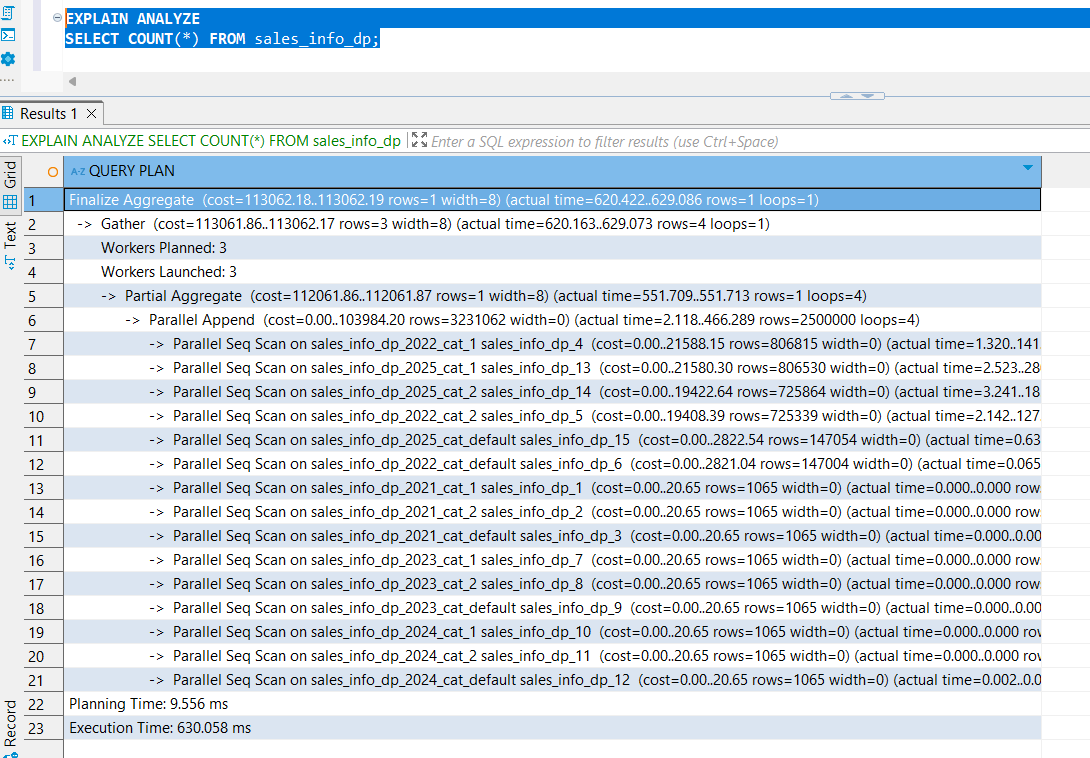
****

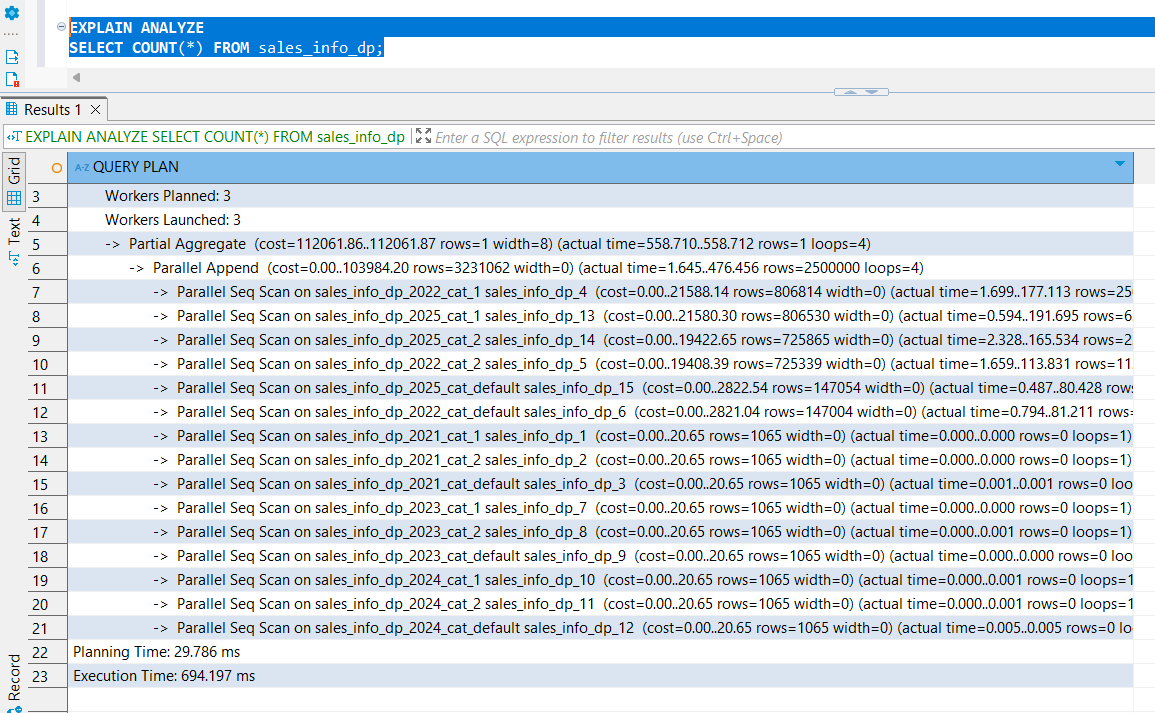
1. **Add order by eventadate(nonindex<index)(nonindex uses parall workers 3 while index uses merge)**



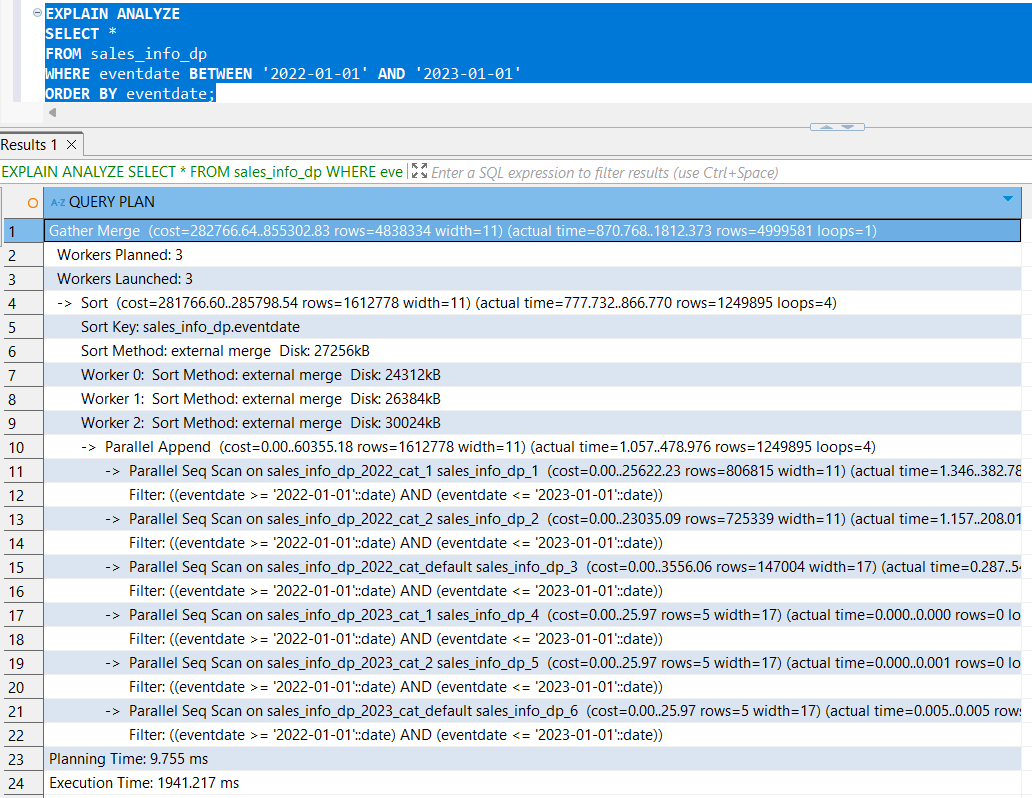


1. **Select count of all rows(nonindex>index)(both uses parall workers 3)**



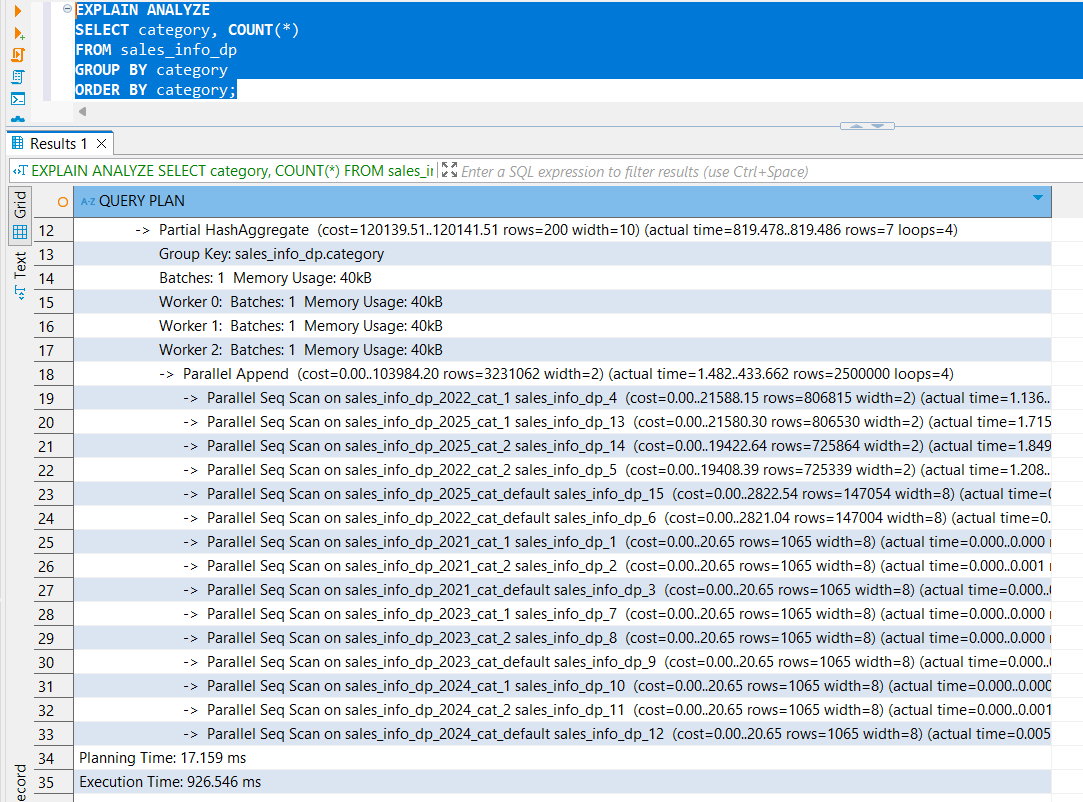


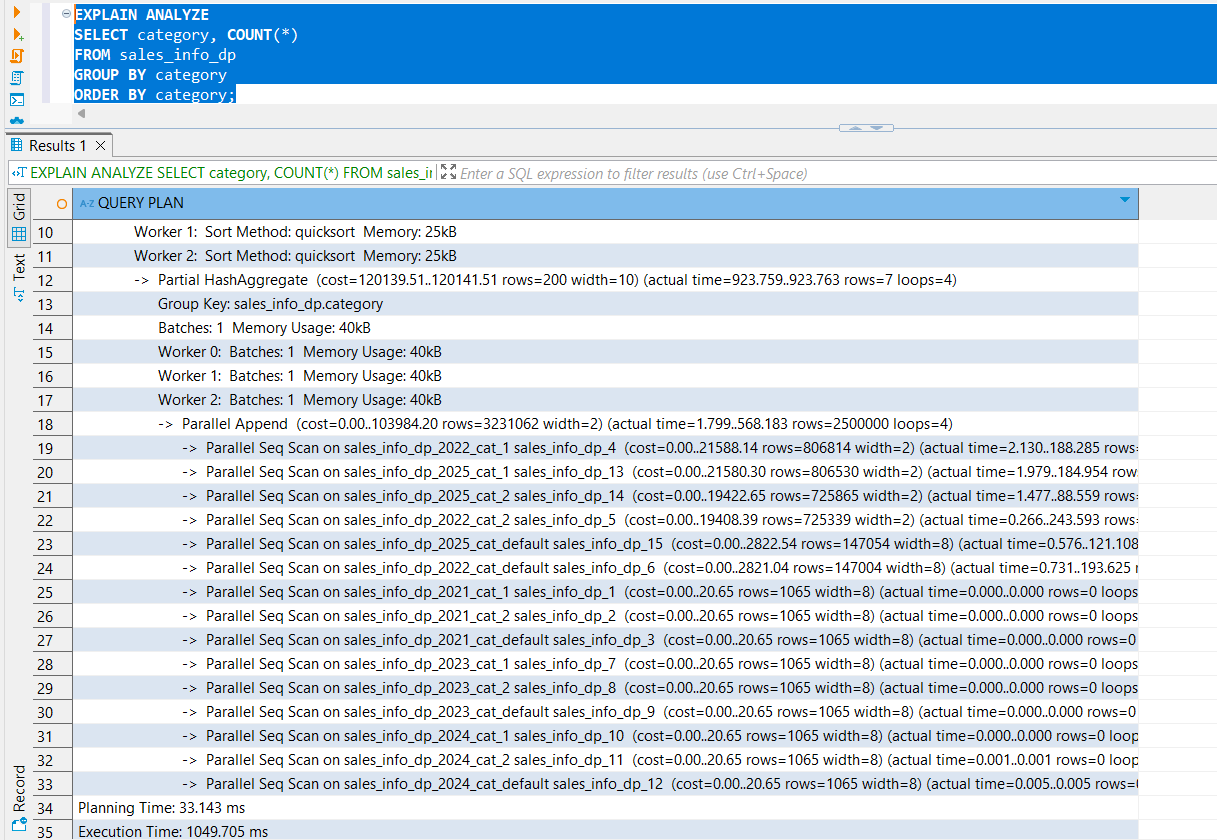
1. **Add range of dates(nonindex>index)(nonindex uses parall workers 3 while index uses merge)**





1. **Add grouping by category(nonindex>index)(both are using 3 parale workers)**





### 

After adding indexes on partitioned tables (e.g., SALES\_INFO\_DP), the PostgreSQL planner adjusted its execution strategies. While some operations benefited from indexing, others performed better without it. Here's a breakdown of the results:

### ✅ Queries that performed better *with indexes*

| **Query** | **Observation** | **Reason** |
| --- | --- | --- |
| **Select All** | nonindex < index | Full table scans are faster without index overhead. Index adds planning complexity without much gain. |
| **Order by eventdate** | nonindex < index (non-index uses **parallel workers**, index uses **merge sort**) | PostgreSQL’s merge sort via index is more precise but may be slower than a full parallel sort on sequential data. Index plan avoids extra workers, but that may cost time. |

### ❌ Queries that performed better *without indexes*

| **Query** | **Observation** | **Reason** |
| --- | --- | --- |
| **Count of All Rows** | nonindex > index (both use parallel workers) | A full sequential scan (nonindex) is more efficient for simple row counts than navigating index pages. |
| **Range of Dates** | nonindex > index (nonindex uses **parallel scan**, index uses **merge**) | Index introduces sorting overhead and limits use of parallelism; sequential range scan can be faster. |
| **Group by Category** | nonindex > index (both use parallel workers) | Aggregation benefits more from direct access to partitioned rows via parallel scan than from index lookups. |

### 

Adding indexes to partitioned tables **does not guarantee better performance across all workloads**. Specifically:

* **Indexes can reduce performance** for broad scans (COUNT, GROUP BY, large ranges), because PostgreSQL can better optimize parallel sequential access.
* **Indexes are more beneficial** for targeted lookups or queries that benefit from ordering (ORDER BY, selective filters)—**but only if they avoid disabling parallelism**.
* **Parallelism matters**: In several cases, **non-indexed scans outperformed indexed scans** simply because they were able to leverage multiple workers more effectively.