

| Business Template  **Join Methods** |
| --- |
|  |

### 1. Join Methods

Read about PostgreSQL parameters enable\_nestloop, enable\_hashjoin, enable\_mergejoin and how they can be used to instruct the planner to choose a join method.

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

### 1.1 Task 1 –Nested Loop Join

#### **Step 1: Create test tables and populate them with test data**

Table Creation

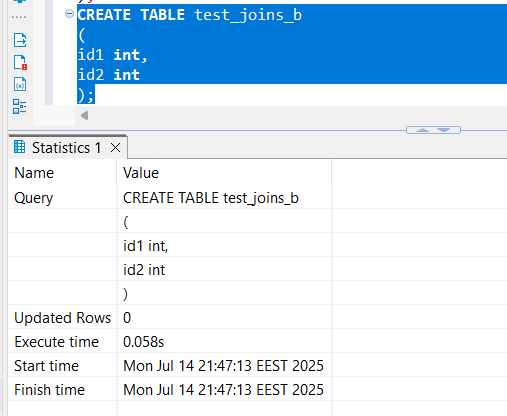
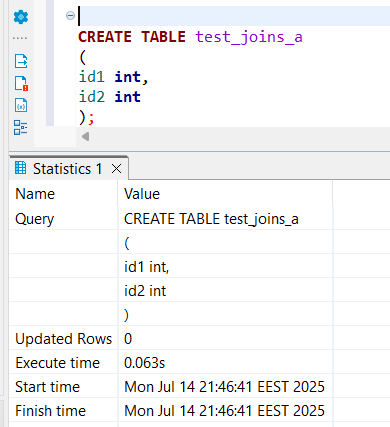


Table population

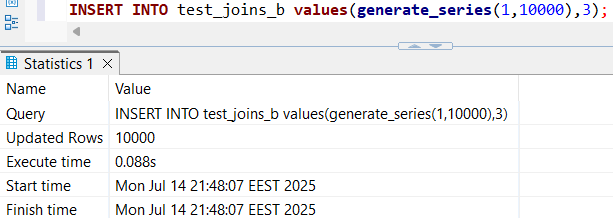
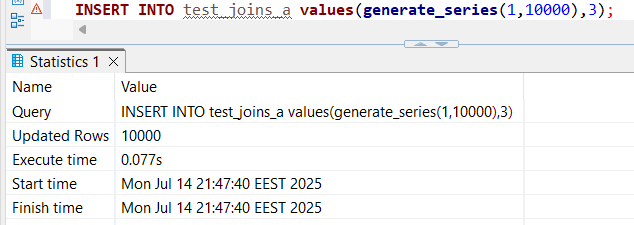
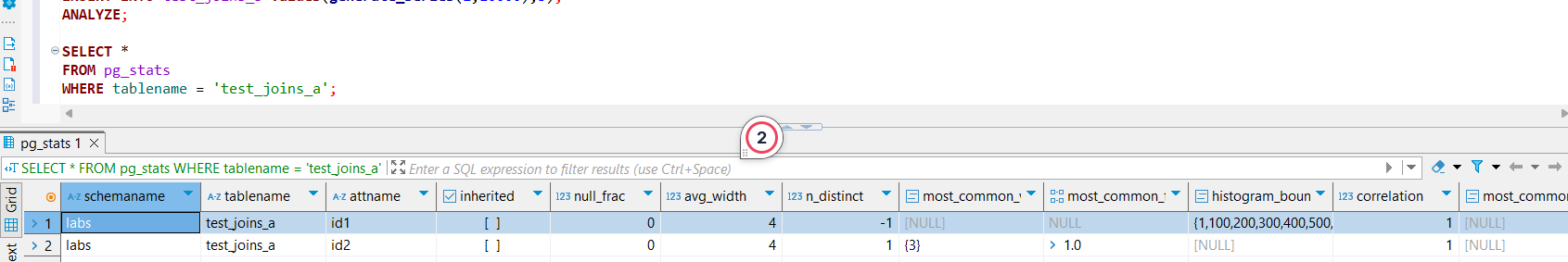
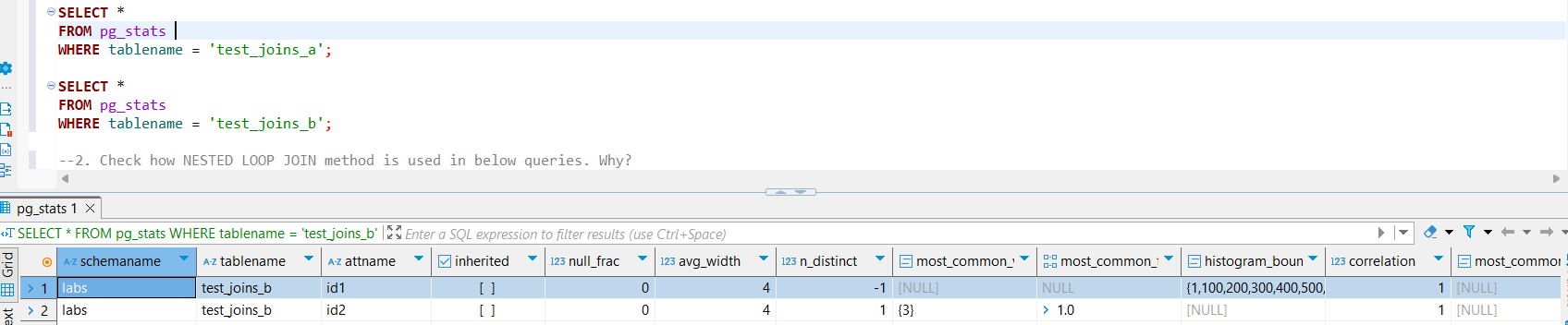
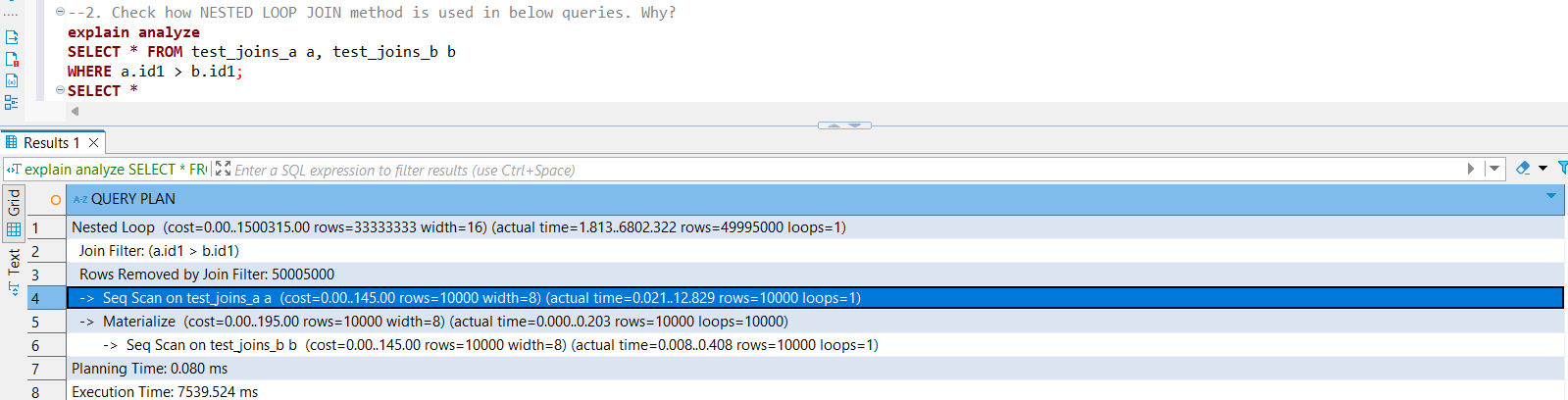


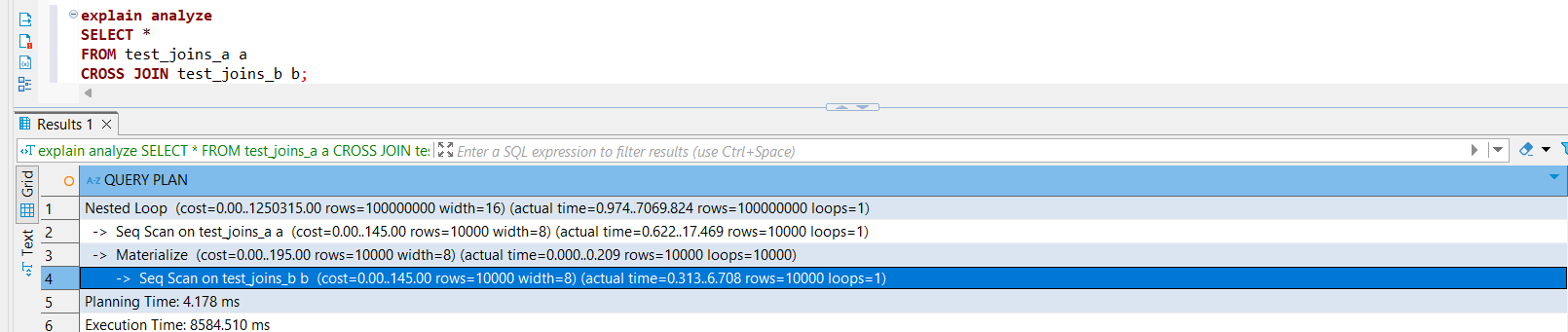
Table statistics





#### **Step 2: Check how NESTED LOOP JOIN method is used in below queries. Why?**





The first query performs a join between test\_joins\_a and test\_joins\_b with a condition a.id1 > b.id1. Internally, this behaves like a nested loop join, where for each row in the outer table (a), the database scans rows in the inner table (b) and **applies the condition immediately** to decide whether to output the row. Although the join condition filters rows during this process, the engine still **examines every candidate pair** generated by iterating over the inner table. This is why execution plans often show a count of **“rows removed by filter”**, indicating pairs checked but discarded because they don’t meet the condition. This means the database **does not produce the full Cartesian product** in memory but evaluates and filters pairs on the fly, which helps reduce output size and improve performance.

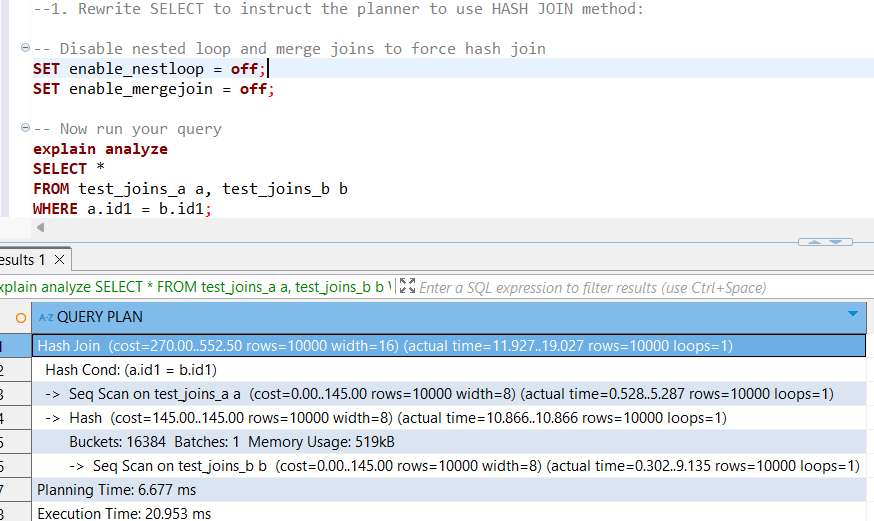
The second query is a straightforward cross join without any filtering condition, so it must produce **all possible combinations** of rows from both tables (the full Cartesian product). Because there is no condition to filter out any pairs, the database outputs every possible row pair, which can be very large and consequently slower to execute.

### 1.2 Task 2 –Hash Join

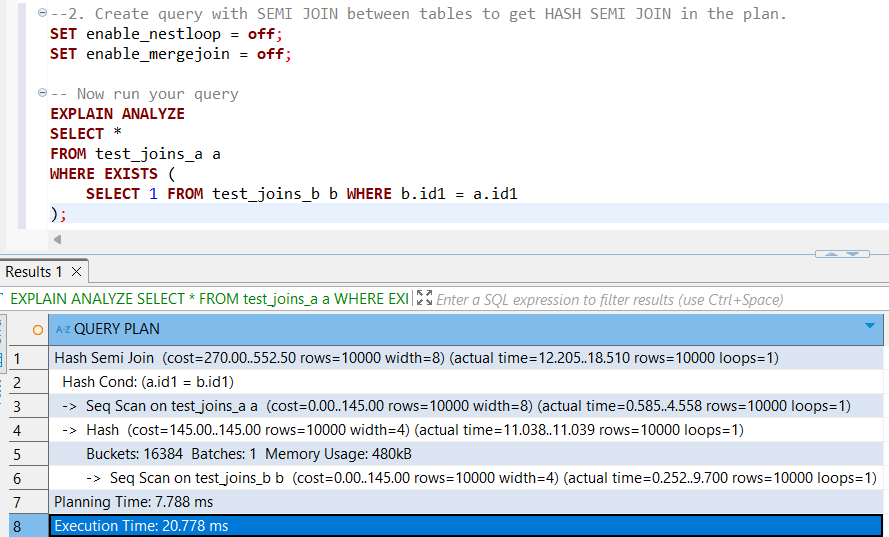
#### 

#### 

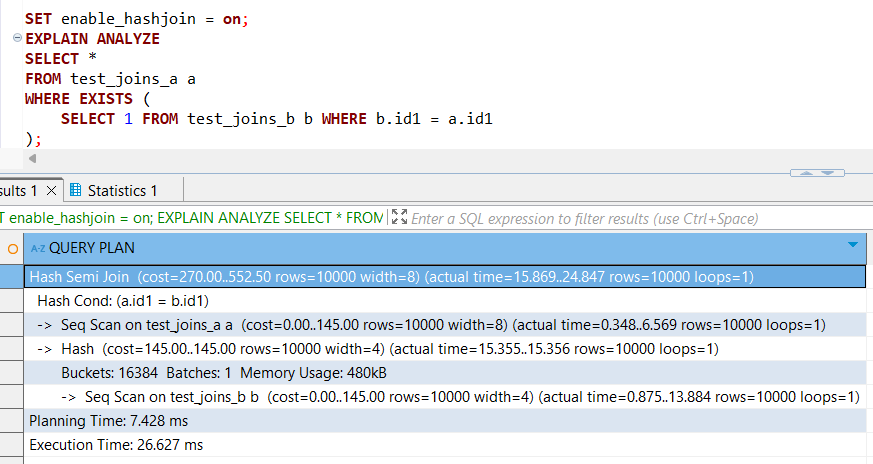
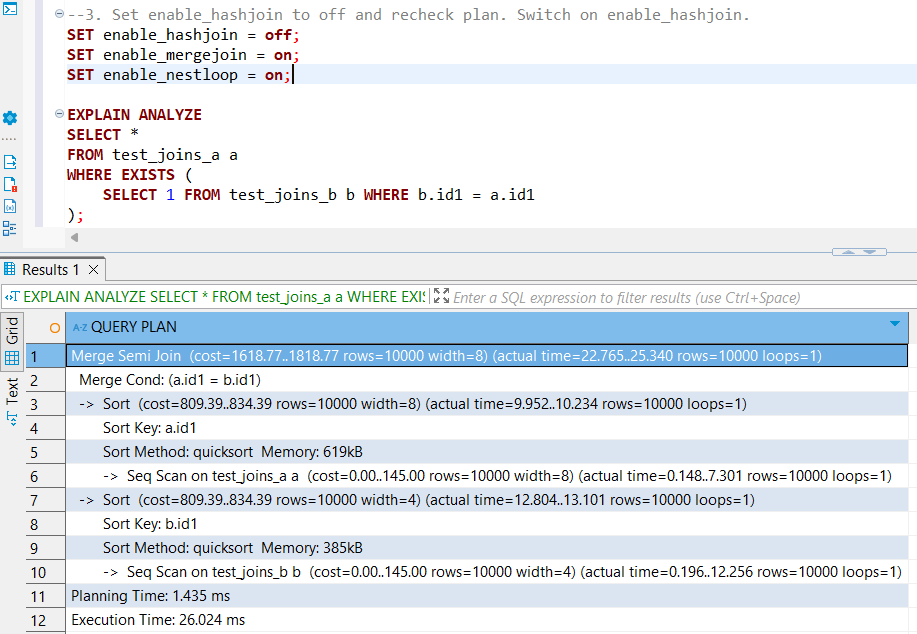
#### **Step 1: Rewrite SELECT to instruct the planner to use HASH JOIN method**



#### **Step 2:Create query with SEMI JOIN between tables to get HASH SEMI JOIN in the plan.**



#### **Step 3: Set enable\_hashjoin to off and recheck plan. Switch on enable\_hashjoin.**



As we can see from the planner’s execution, when I **disabled hash joins**, the optimizer chose a **merge join** instead of a nested loop join. This happens because:

* Initially, without the hash join option, the optimizer’s only ways to join the tables were nested loop or merge joins.
* For queries involving a predicate like a.id = b.id or a semi-join (e.g., WHERE EXISTS (SELECT 1 FROM b WHERE a.id = b.id)), the optimizer prefers merge joins over nested loops if hash joins are not available.
* This preference is because merge joins can efficiently handle equi-join conditions by **sorting both datasets on the join key** (id) and then scanning them in parallel to check for matching rows.

Interestingly, while time efficiency was the main driver before, here we observe that merge joins deliver results slightly faster than nested loops when hash joins are disabled. However, when no restrictions are imposed, the optimizer still prefers hash joins.

This preference likely stems from the fact that hash joins tend to be less costly to implement compared to merge joins, which require sorting both inputs — a potentially expensive operation. Hash joins can build a hash table on one dataset and probe it with the other, avoiding the overhead of sorting.

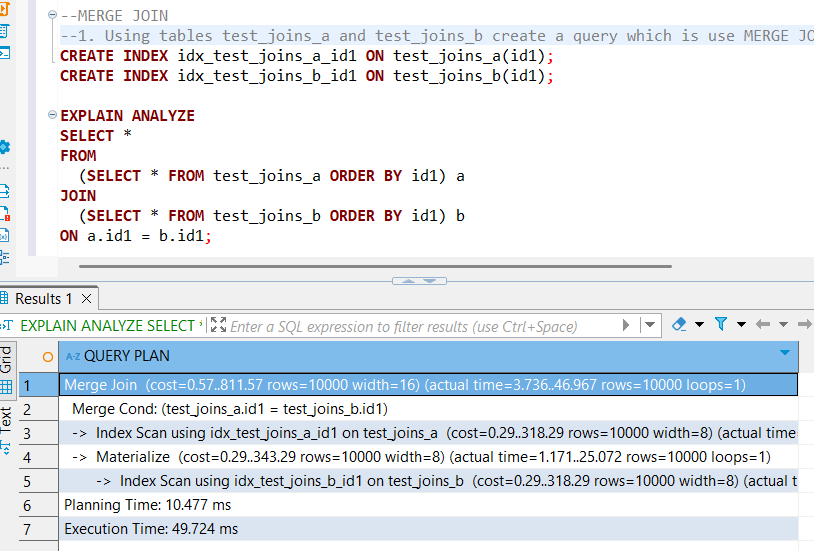
### 1.3 Task 3 –Merge Join

#### 

#### 

#### **Step 1:Using tables test\_joins\_a and test\_joins\_b create a query which is use MERGE JOIN as a join**

#### **Method.**



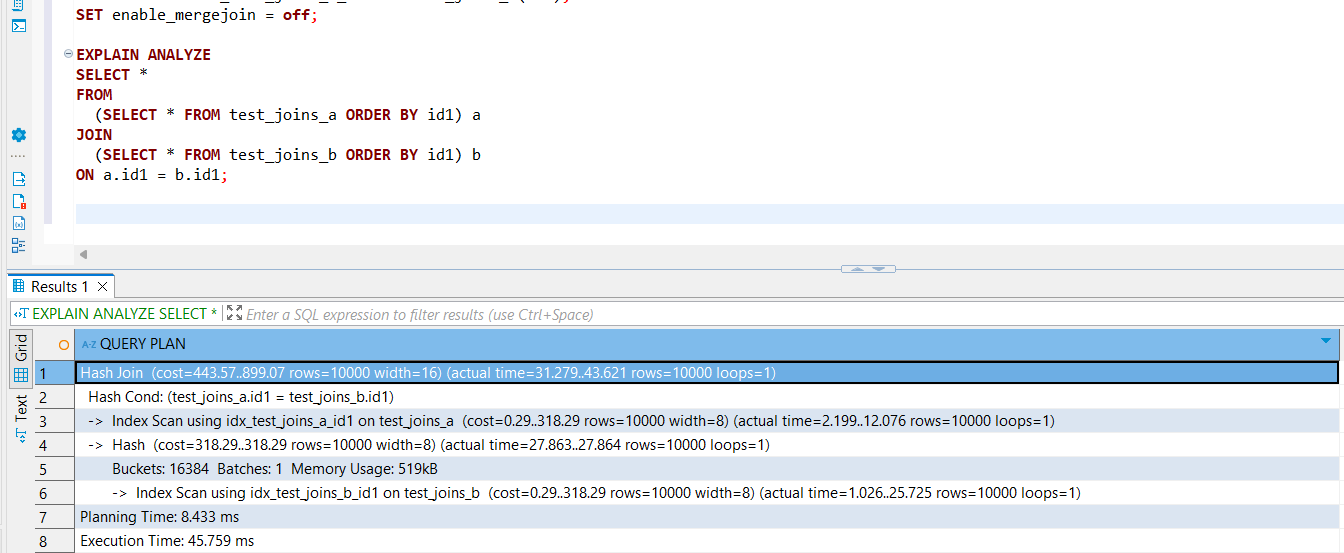
To enable the optimizer to choose a **merge join** naturally, two key factors come into play: **sorted input data** and **efficient access paths**.

1. **Indexes on the join columns** ensure that the data can be accessed in sorted order without extra sorting overhead. When the optimizer knows that both tables can be scanned in order of the join keys (thanks to the indexes), it considers merge join as a viable and efficient method.
2. By including an **ORDER BY on the join keys** in the query, you explicitly tell the optimizer to produce sorted results. This sorting aligns perfectly with the merge join’s requirement that both inputs be sorted on the join columns.
3. Because the data is already sorted or efficiently sortable, the optimizer calculates that the **merge join will be cheaper** compared to other methods like hash join (which builds and probes hash tables and requires memory) or nested loop (which may be more costly with large datasets).

Thus, the combination of indexes and sorting signals to the optimizer that merge join is the optimal choice, balancing CPU and I/O costs effectively.

#### 

#### **Step 2:Set enable\_mergejoin to off and recheck plan. Switch on enable\_mergejoin.**

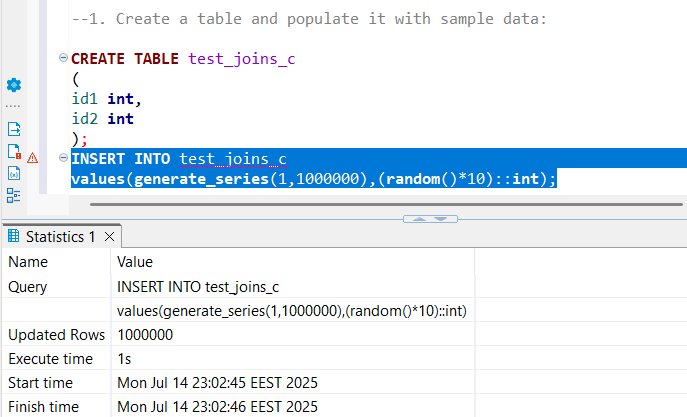


When I set enable\_mergejoin to off, the optimizer chose a hash join instead. Although hash joins can be more costly in some scenarios, they are generally more efficient than nested loops for joins with predicates like a.id = b.id or semi-joins (WHERE EXISTS). This makes sense because nested loop joins are typically optimal only in cases like cross joins or filtering with small datasets. In contrast, hash joins are preferred for equality joins since they use memory more efficiently to quickly match rows without the repeated scanning that nested loops require.

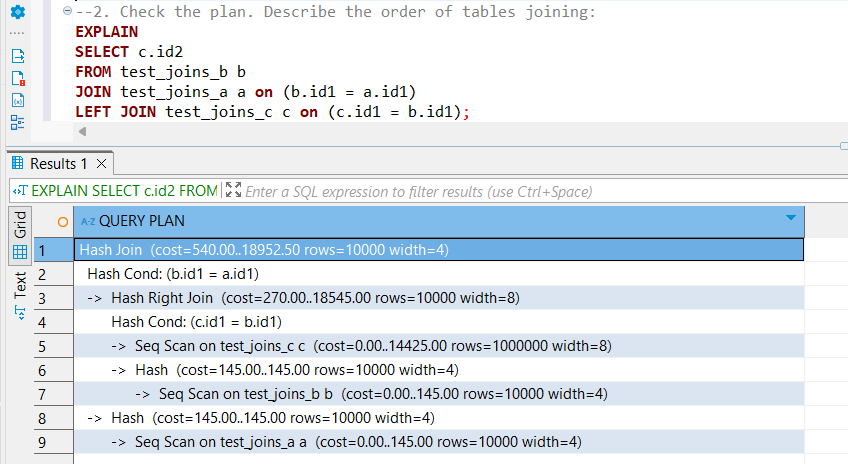
### 2. Join Order and Lateral Join

### 2.1 Task 4 –Changing Join Order

#### **Step 1:Create a table and populate it with sample data:**



#### **Step 2:Check the plan. Describe the order of tables joining:**



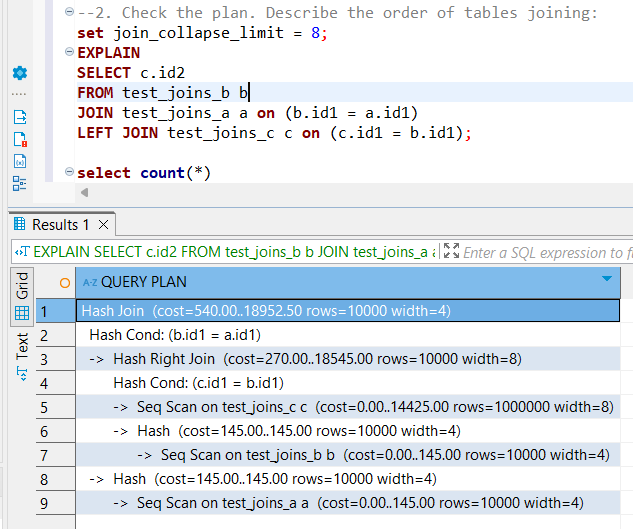
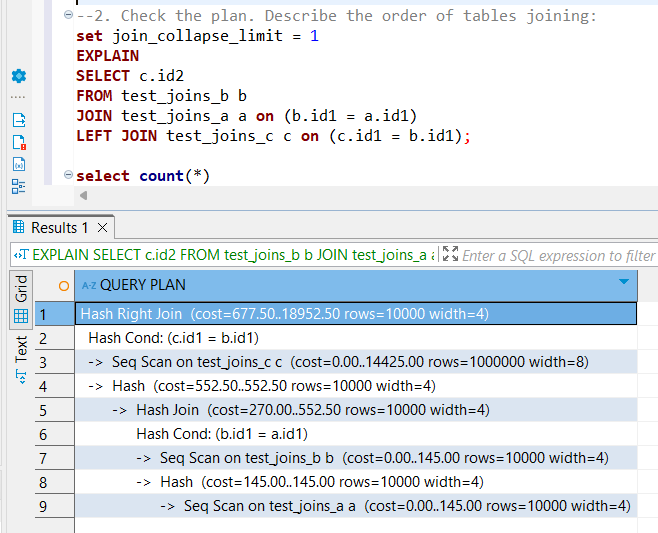
In this query, the optimizer first joins tables **a** and **b**, followed by a join with table **c**. This makes sense because **a** and **b** are smaller tables, so joining them first reduces the number of rows to process early on, using less memory.

Since each of **a** and **b** has only about 10,000 rows, joining them first means fewer rows need to be hashed. Afterwards, the left join with **c** involves hashing against a much larger table (around 1,000,000 rows).

If the optimizer were to join a small table with a very large one first, it would have to hash through the large table multiple times, which is expensive. By joining the smaller tables first, the hash join has a smaller intermediate result to work with before processing the large table, which is more efficient.

#### **Step 3: Set join\_collapse\_limit = 1 and recreate plan for query above. Describe changes if any. Return join\_collapse\_limit = 8.**

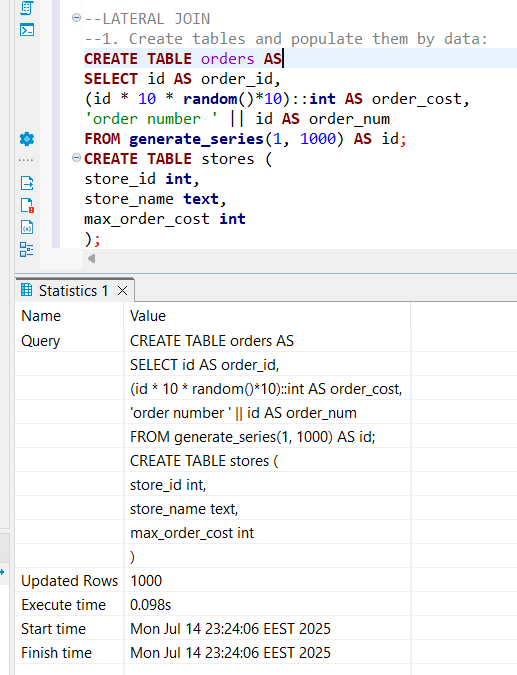
#### 



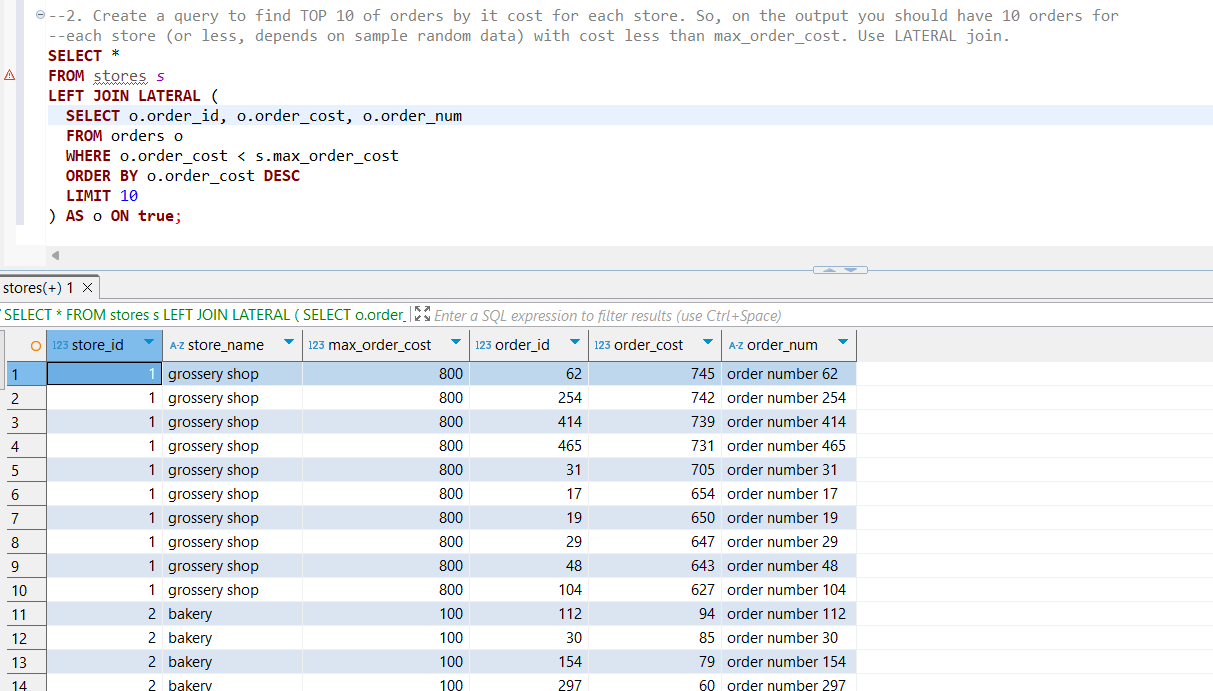
When I set join\_collapse\_limit to 1, the optimizer simply follows the join order as written in my query without trying alternative orders. This happens because join\_collapse\_limit controls how many join combinations the optimizer is allowed to consider before deciding on the join order. Setting it to 1 means the optimizer only considers the first (i.e., the order you impose) and does not explore other join sequences.

### 2.2 Task 5 –Lateral Join

#### **Step 1:Create tables and populate them by data.**



#### **Step 2:Create a query to find TOP 10 of orders by it cost for each store. So, on the output you should have 10 orders for each store (or less, depends on sample random data) with cost less than max\_order\_cost. Use LATERAL join.**



### 2. CTEs

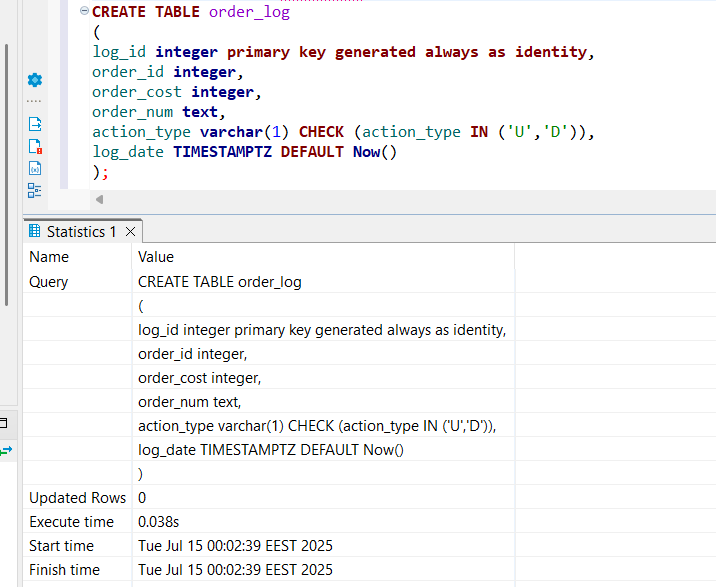
### 3.1 Task 6 –Recursive CTE

#### **Step 1:Use emp table you created before. Select all employee and his manager name and level of management start from president of the company**

#### 

### 3.2 Task 7 –Changing Data CTE

#### **Step 1:Create log table for emp table,**



#### **Step 2:Update all rows for ORDER table:**

#### **a. set new ORDER\_COST = (old ORDER\_COST / 2) where old ORDER\_COST between 100 and 1000**

#### **b. delete all rows where ORDER\_COST < 50**

#### **c. save all updated and deleted rows into log table with action type ‘U’ and ‘D’ relatively. Perform all in one SQL CTE query.**

#### 

In this