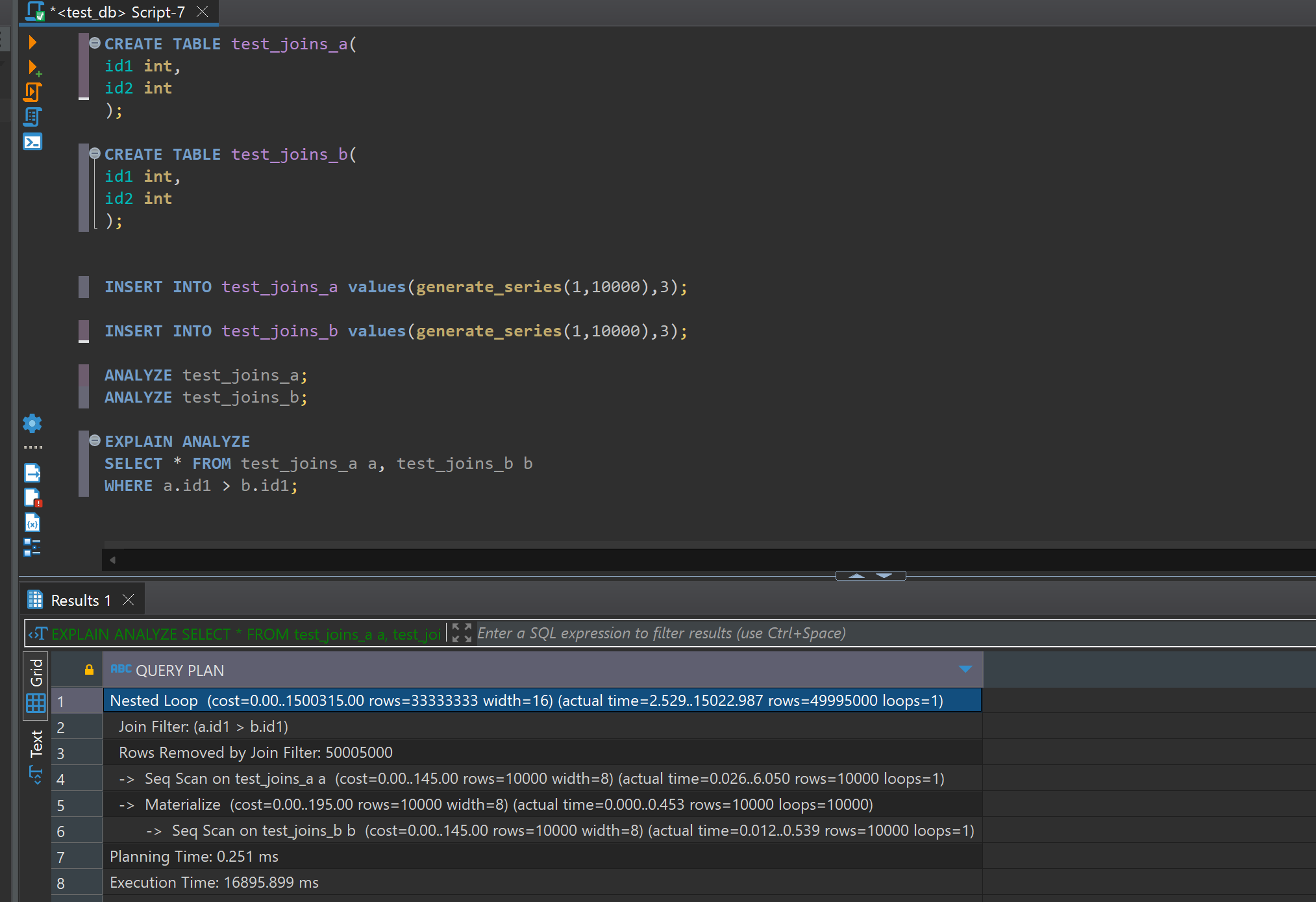
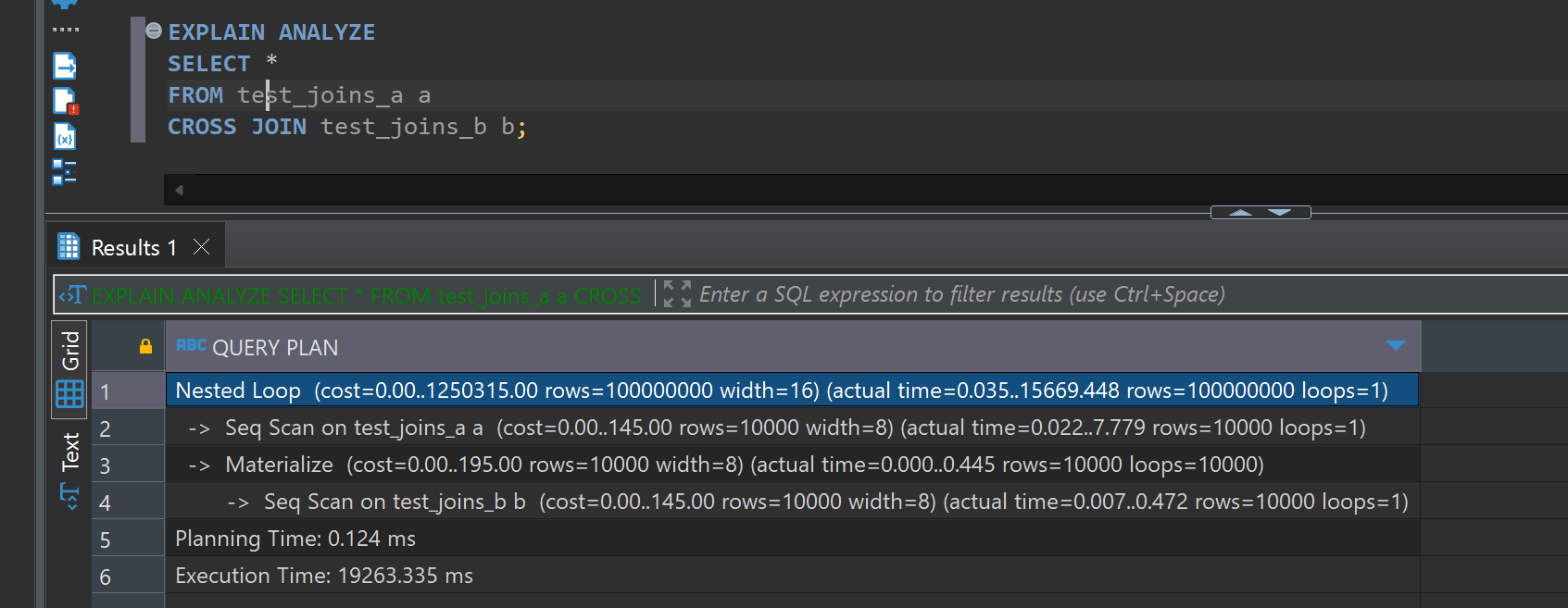
* 1. Join Methods



the planner chose a nested loop join because it needs to evaluate the condition a.id1 > b.id1 for each pair of rows. Since this condition is not an equality condition and there's no index on the tables, a nested loop join is a straightforward choice. Both tables are scanned sequentially. The first table test\_joins\_a is scanned once, and for each row in test\_joins\_a, the second table test\_joins\_b is scanned to find matching rows.

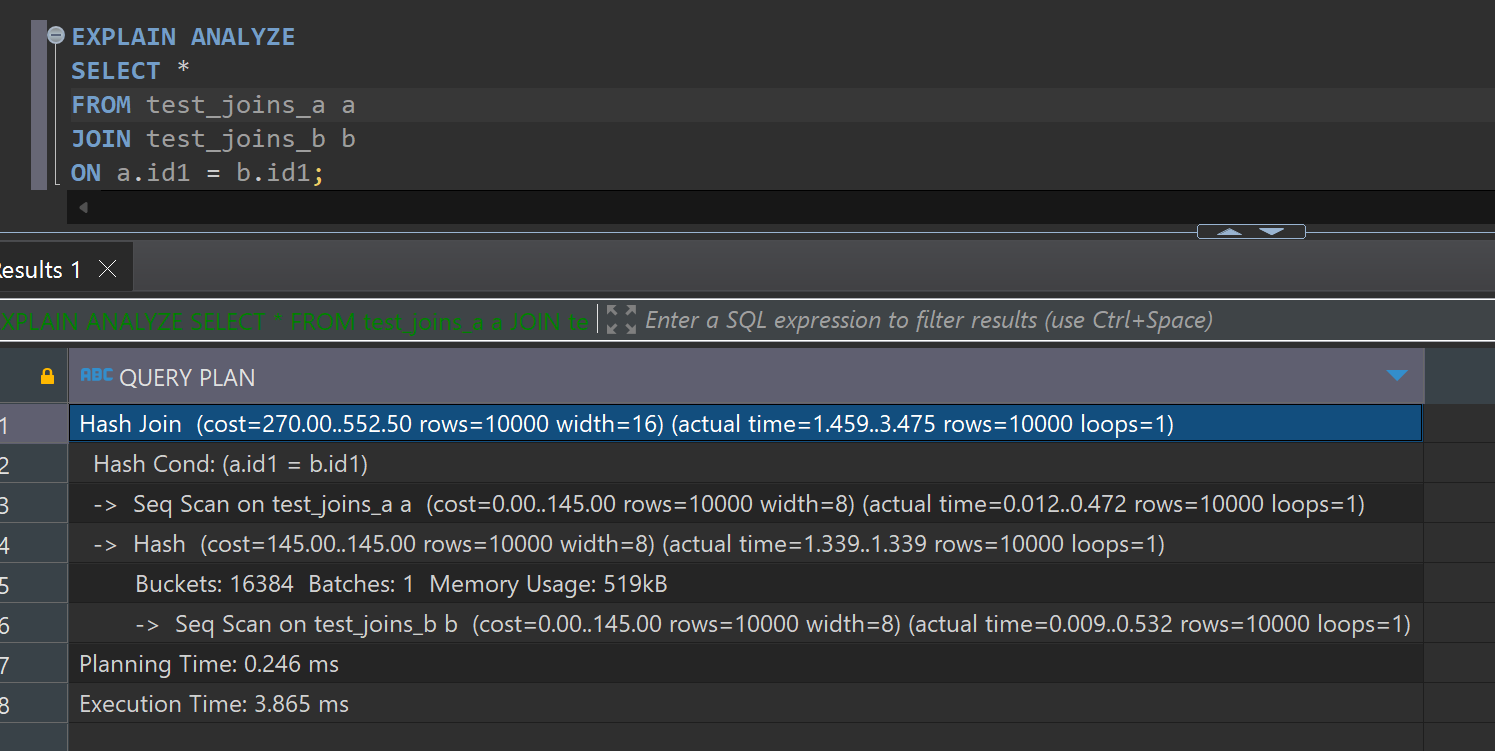


In this case, planner chose a nested loop join again because a cross join requires evaluating every possible pair of rows from test\_joins\_a and test\_joins\_b. This is effectively the definition of a nested loop join. Both tables are scanned sequentially and for each row in test\_joins\_a, all rows in test\_joins\_b are scanned to produce the Cartesian product.

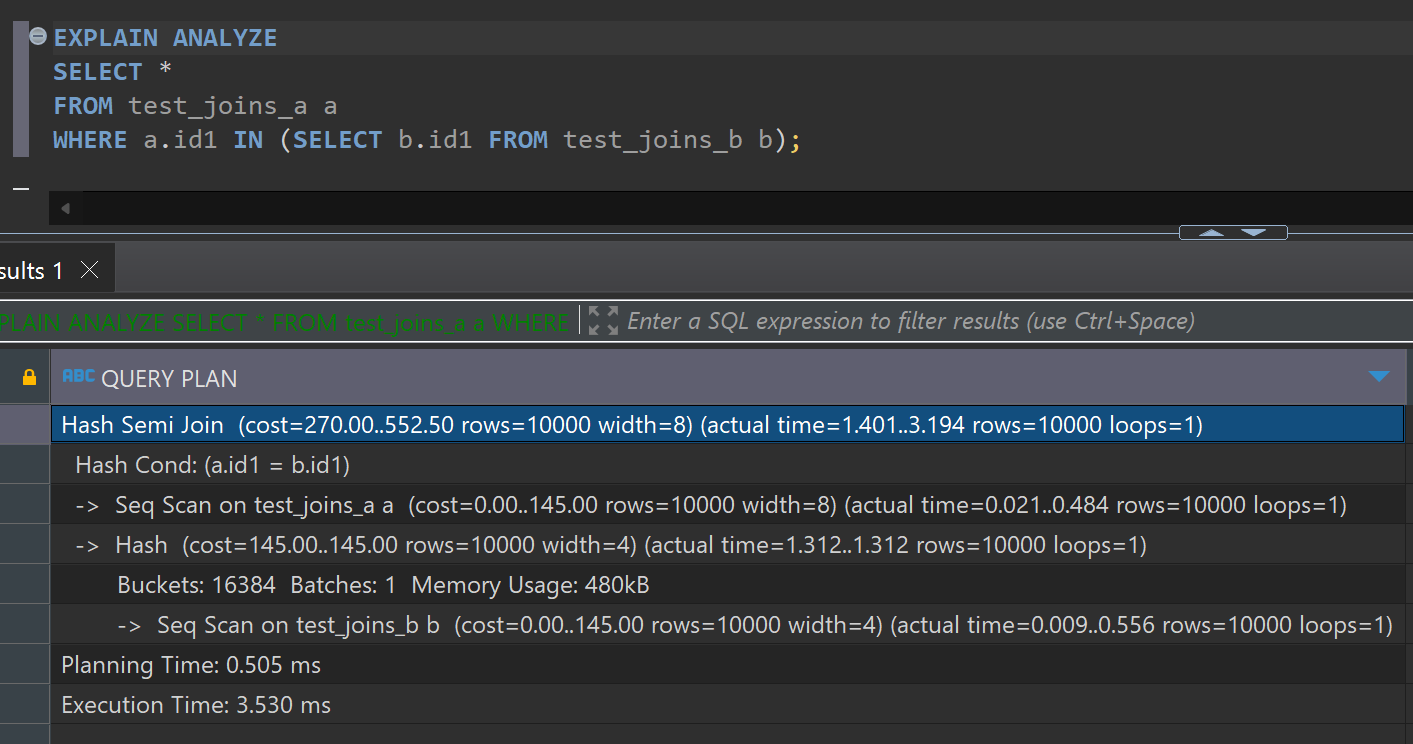
* 1. task 2

1: Rewrite the Query to Instruct the Planner to Use Hash Join

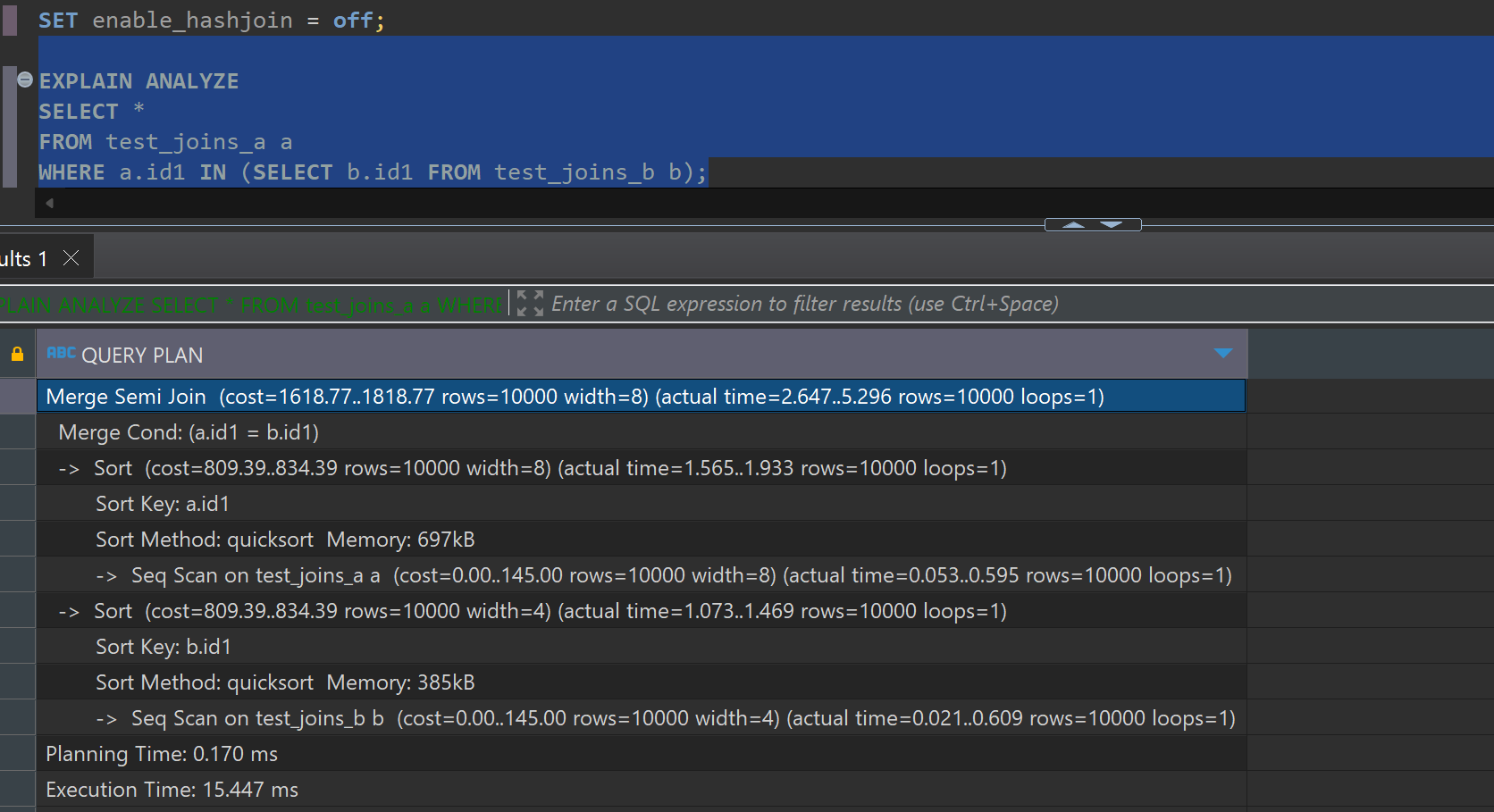
First of all, in order to ensure that query is suitable for a hash join equality condition should be used.



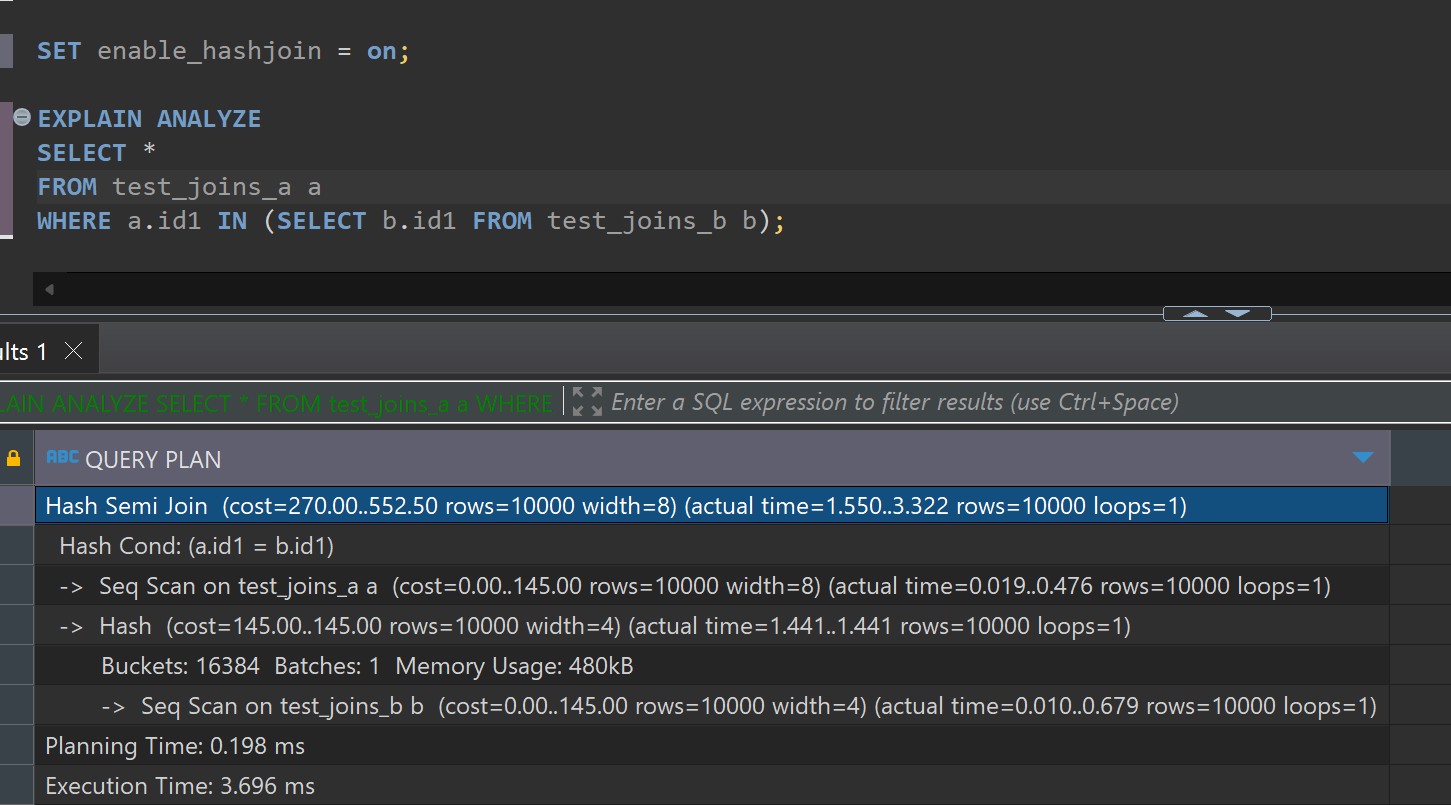
2: Create Query with SEMI JOIN for Hash Semi Join in the Plan

A semi-join returns rows from the first table where one or more matches are found in the second table and in order to get a hash semi join, I think, I can use IN

3: Set enable\_hashjoin to Off and Recheck the Plan



Instead of hash semi join, merge semi join is used.

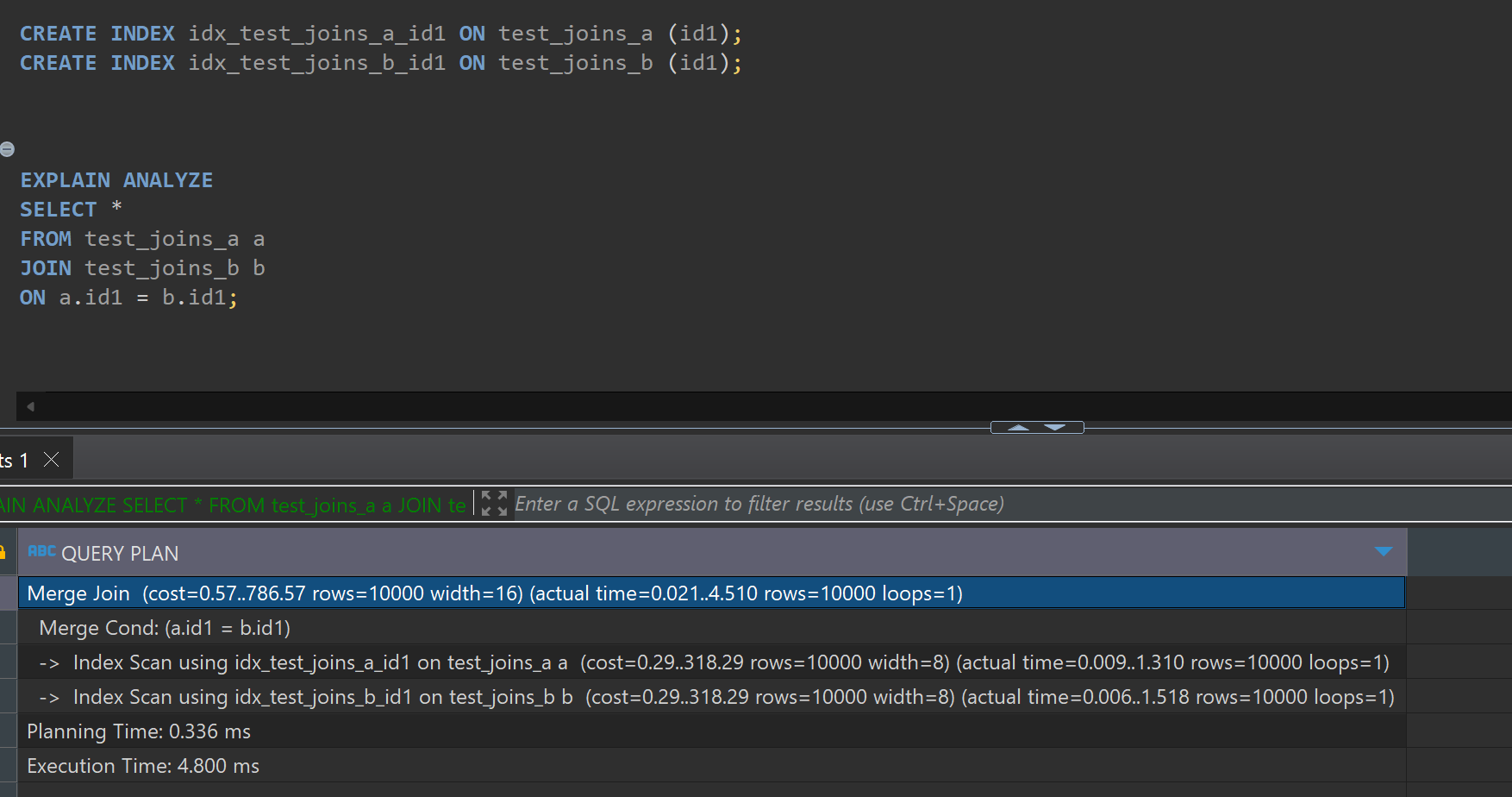


After enabling again using hash semi join.

1.3

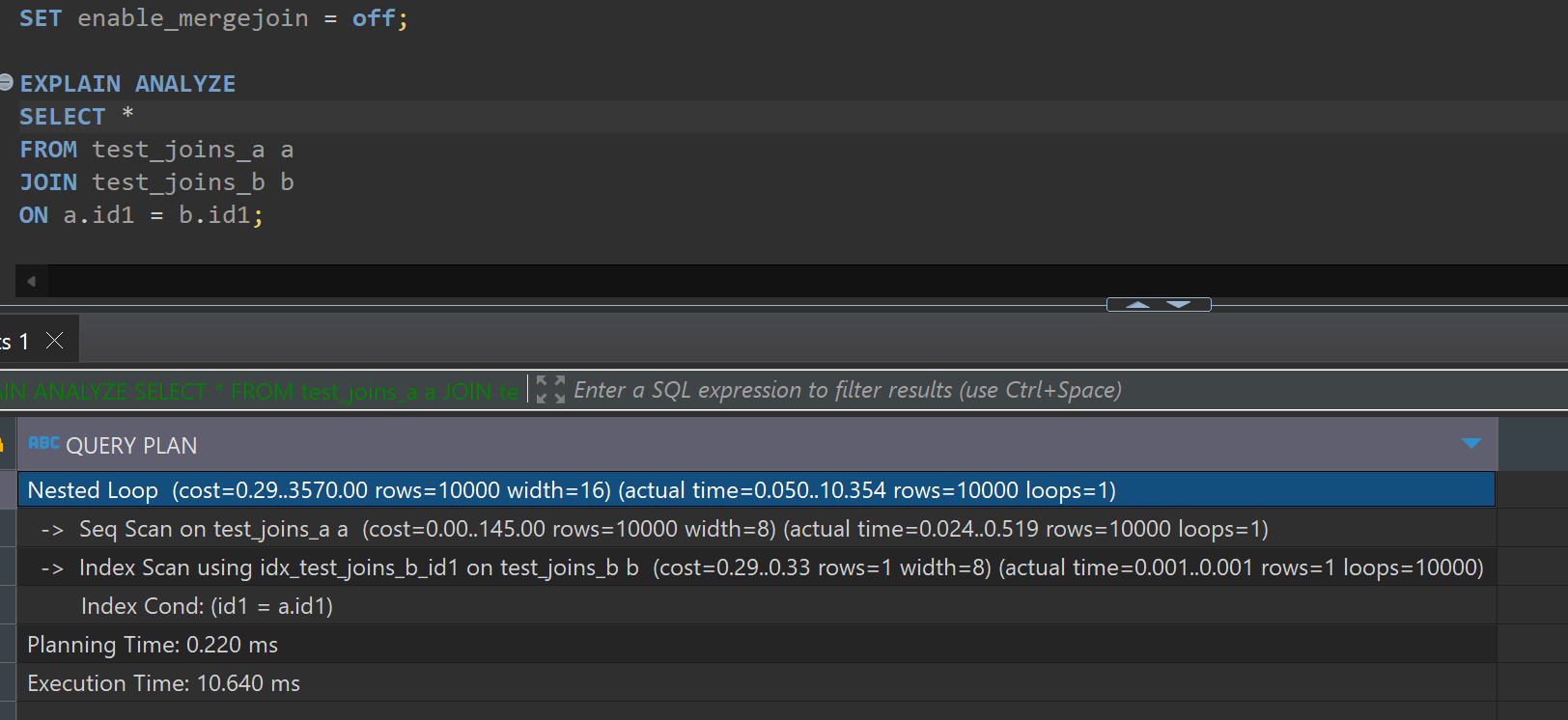
TASK 3: MERGE JOIN

1.Using Merge Join Method



To ensure PostgreSQL uses a merge join, create indexes on the join columns and then run the query.

2. Set enable\_mergejoin to off and recheck plan. Switch on enable\_mergejoin.



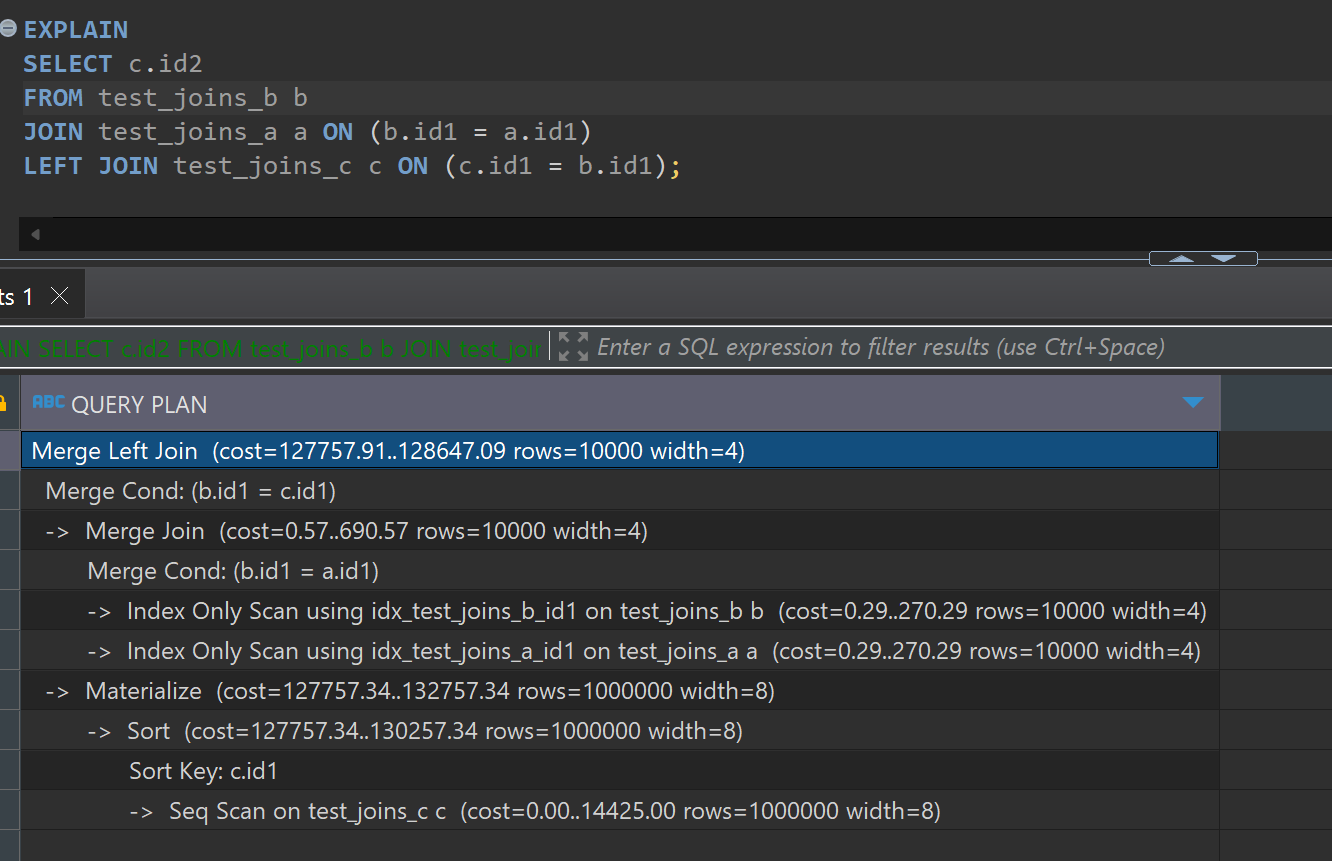
Uses nested loop, If merge is not able.

Then enabling, merge join again.

2.1

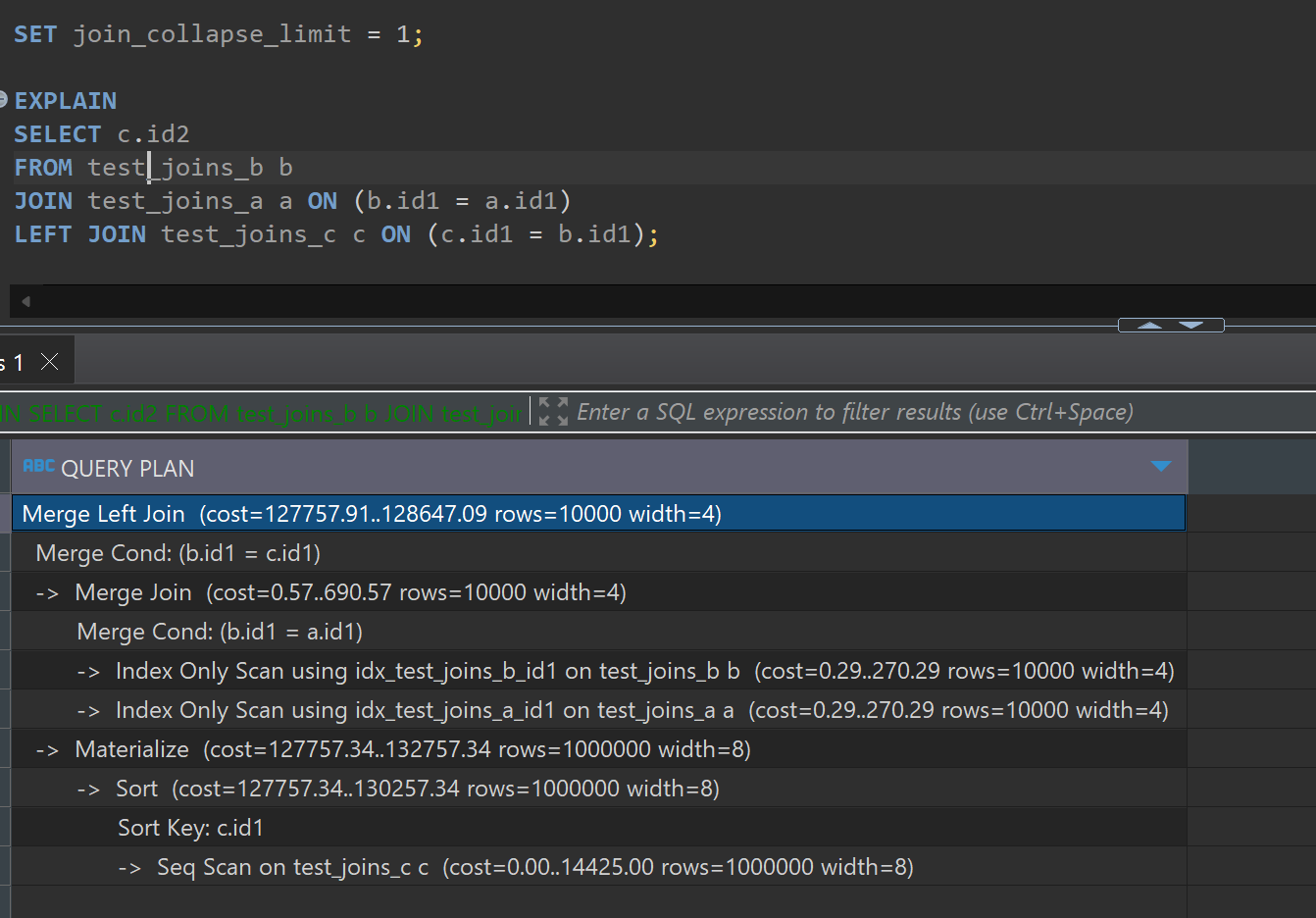
TASK 4: CHANGING JOIN ORDER

1. Create a table and populate it with sample data:
2. Check the plan



planner uses a merge join for the inner join between test\_joins\_b and test\_joins\_a because both tables have indexes on the join column id1, making sorting and merging efficient. After obtaining the result of the inner join, the planner performs a merge left join with test\_joins\_c. To do this, it first sorts test\_joins\_c on id1 and then merges it with the intermediate result. he final result returns the id2 column from test\_joins\_c.

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

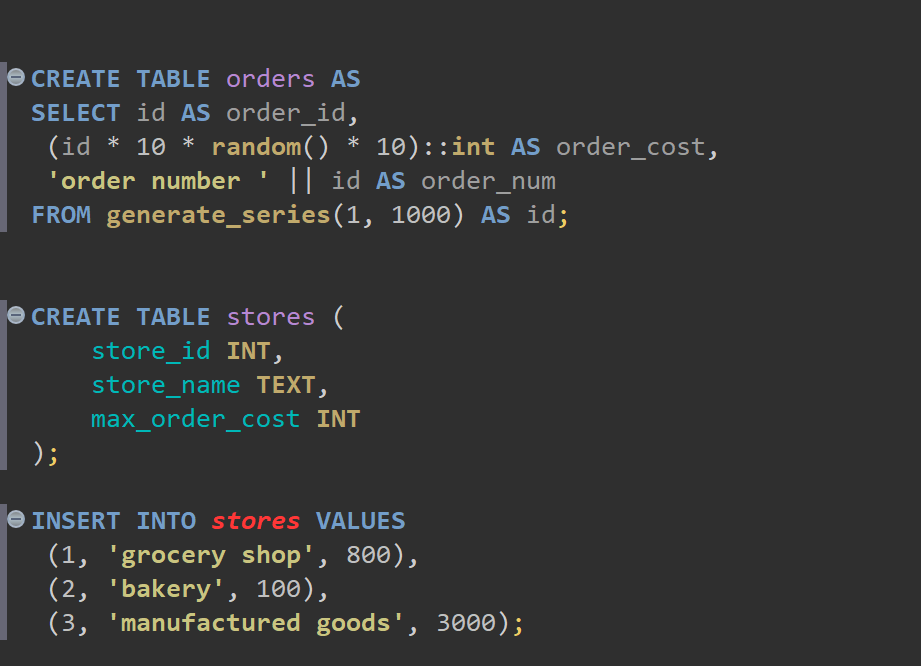


Setting join\_collapse\_limit to 1 restricts the planner to only consider one join at a time, rather than considering multiple tables together. This can influence the order in which tables are joined and the methods chosen.Despite this the planner still chose an efficient join method , which is merge join in this case because of the available indexes. The first join operation is an inner join between test\_joins\_b and test\_joins\_a using the merge join method. This join is efficient as indexes on id1 columns in both tables exists.The result of the above join is then left joined with test\_joins\_c using the merge left join method. Before this join, test\_joins\_c is sorted on id1 to be sured that merge will be efficient

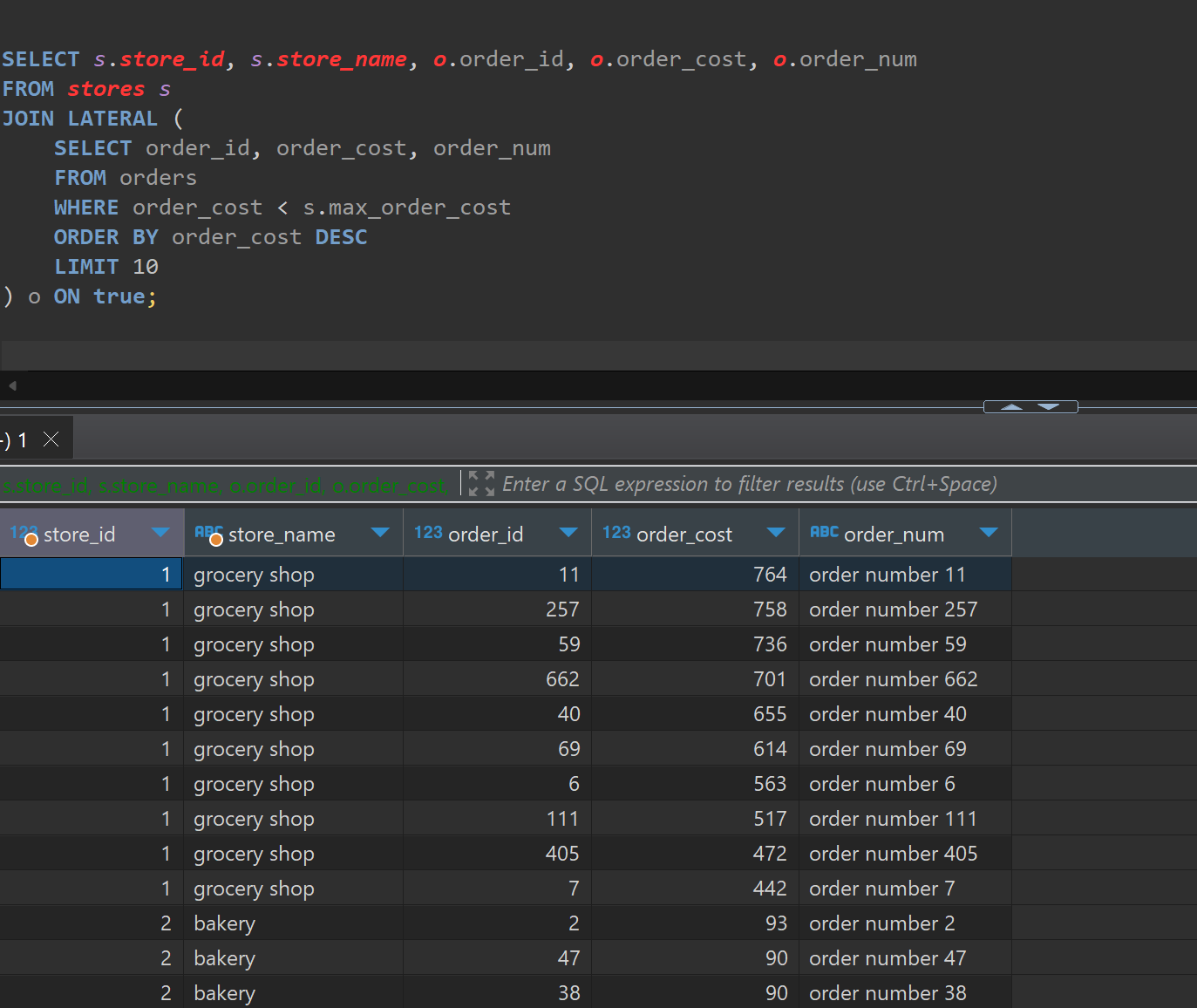
After this back to 8.

* 1. TASK 5: LATERAL JOIN

1. Create tables and populate them by data;

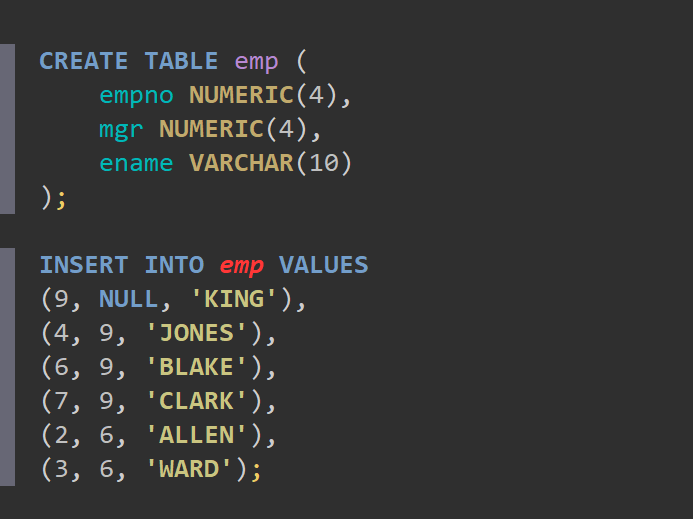


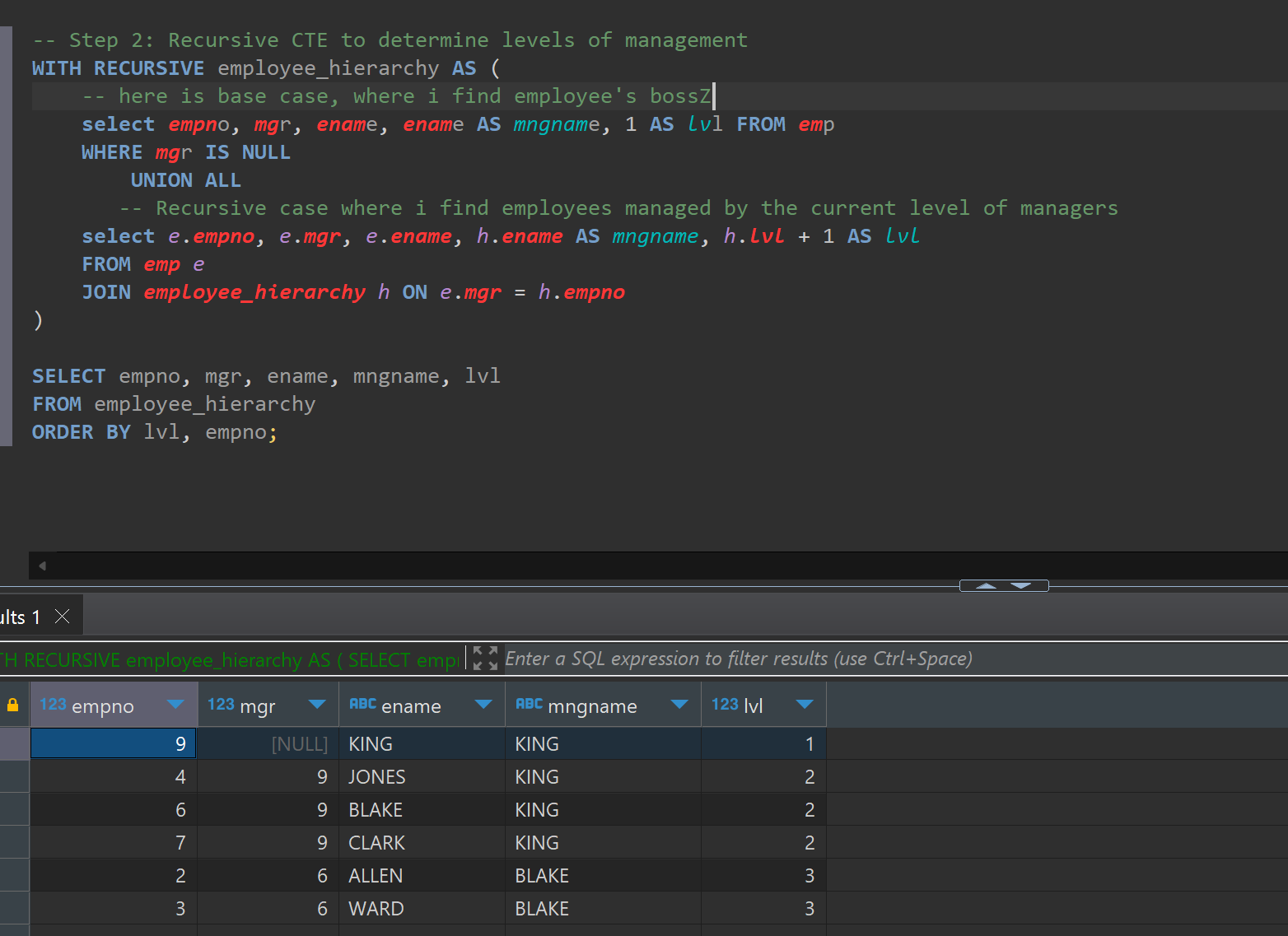
1. Create a query to find TOP 10 of orders by it cost for each store. So, on the output you should have 10 orders for each store (or less, depends on sample random data) with cost less than max\_order\_cost. Use LATERAL join.



The literal join allows the subquery to reference columns from the preceding table which are stores in this case, This enables to filter orders based on each store's max\_order\_cost. For each store, the subquery selects the top 10 orders with a cost less than the store's max\_order\_cost, ordered by order\_cost in descending order.

3.1 task 6: recursive cte





3.2 TASK 7: CHANGING DATA CTE

Create log 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

