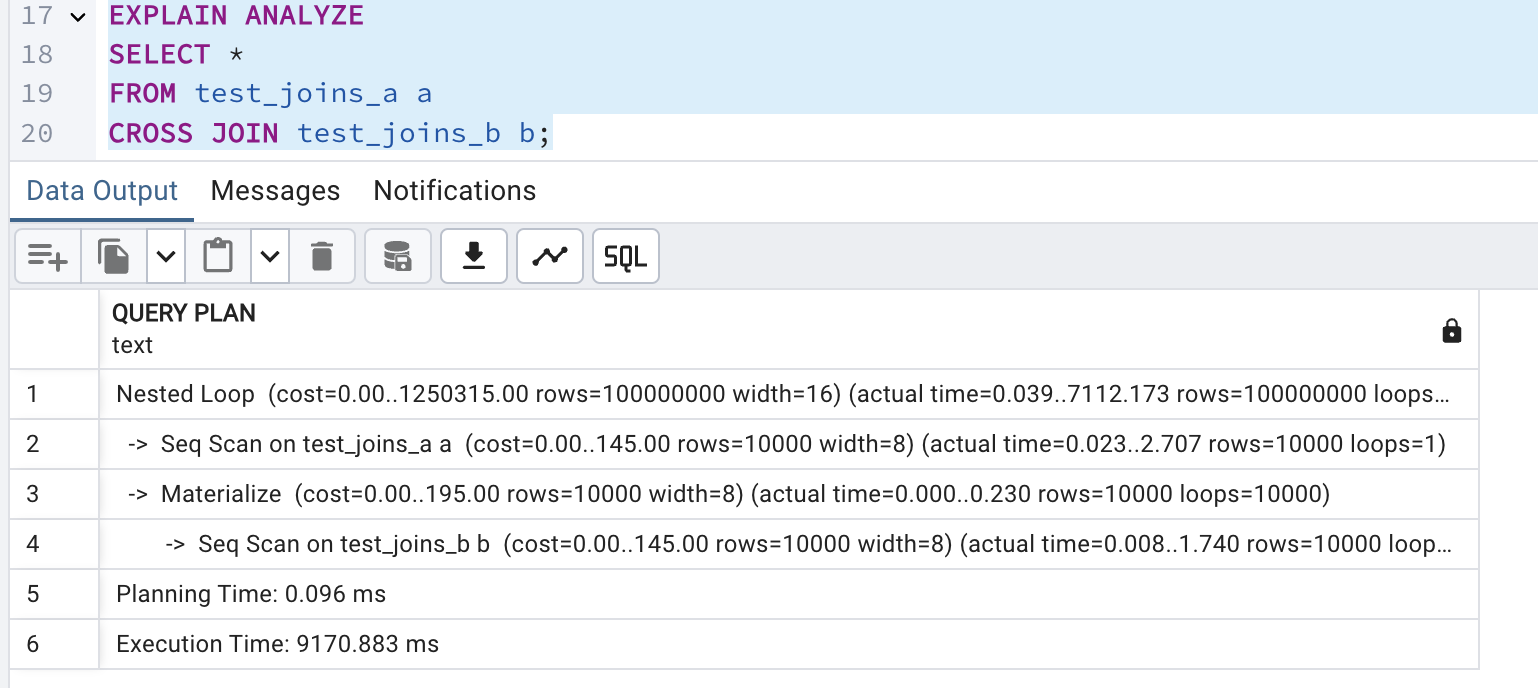
* 1. Join Methods

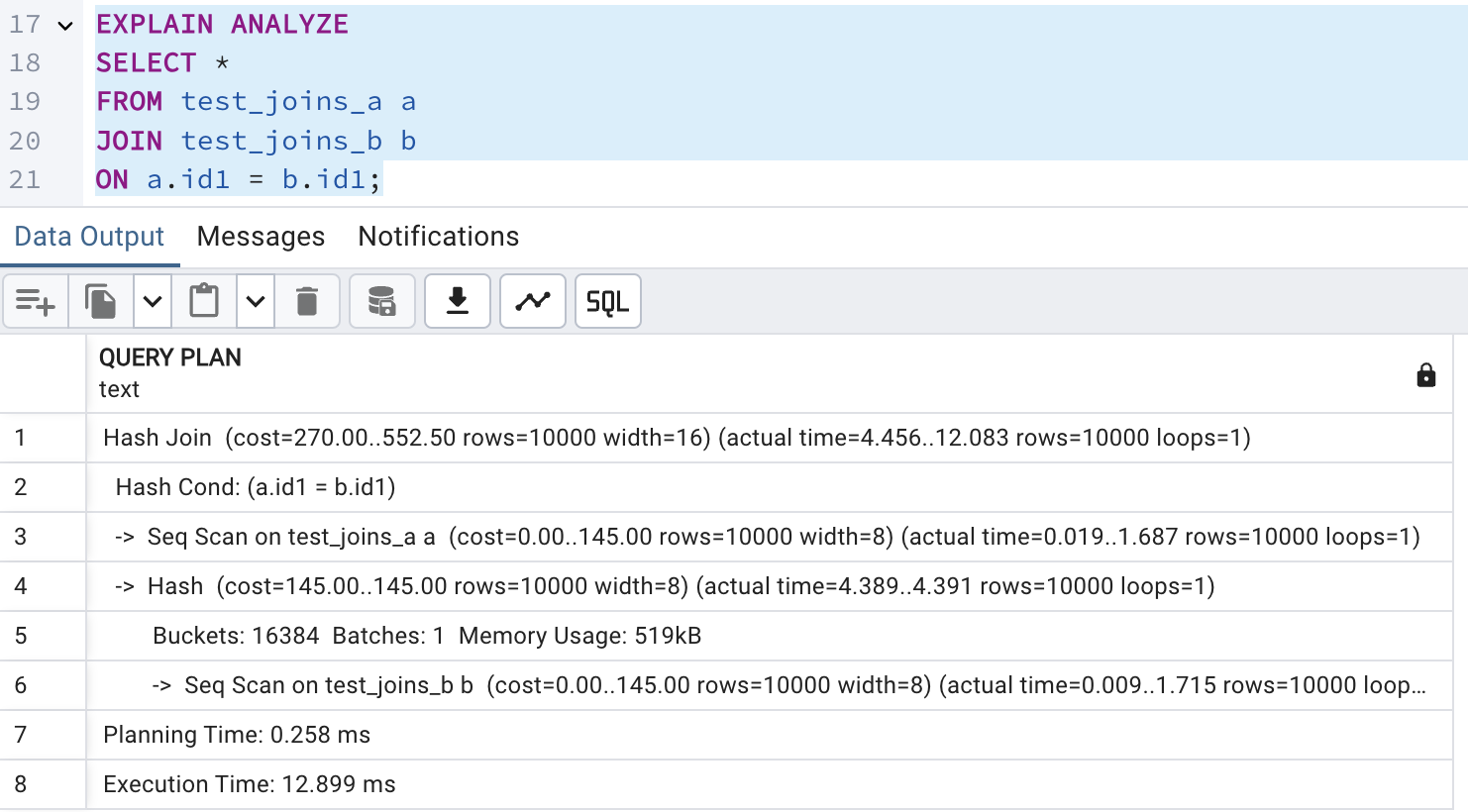
The planner selected a nested loop join because it must evaluate the condition a.id1 > b.id1 for every pair of rows. Since this is not an equality condition and no indexes exist on the tables, a nested loop Join is a natural choice. Both tables are scanned sequentially the first table, test\_joins\_a, is scanned once, and for each of its rows, the second table, test\_joins\_b, is scanned to find matching rows.



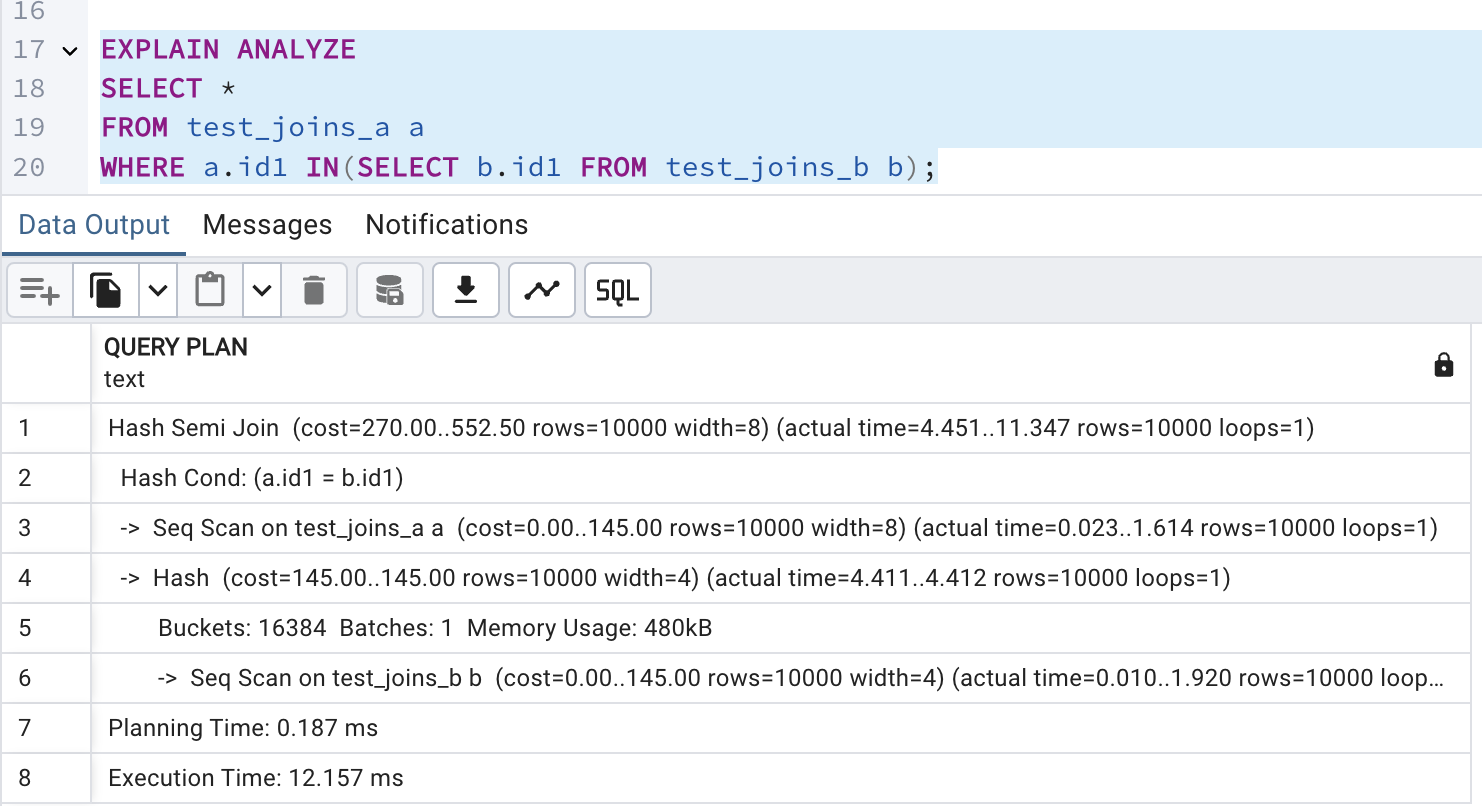
In this case, the planner opted for a Nested Loop Join again because a Cross Join requires evaluating every possible pair of rows from test\_joins\_a and test\_joins\_b. This aligns with the fundamental behavior of a nested loop Join. Both tables are scanned sequentially: test\_joins\_a is scanned once, and for each of its rows, all rows in test\_joins\_b are scanned to generate the Cartesian product.

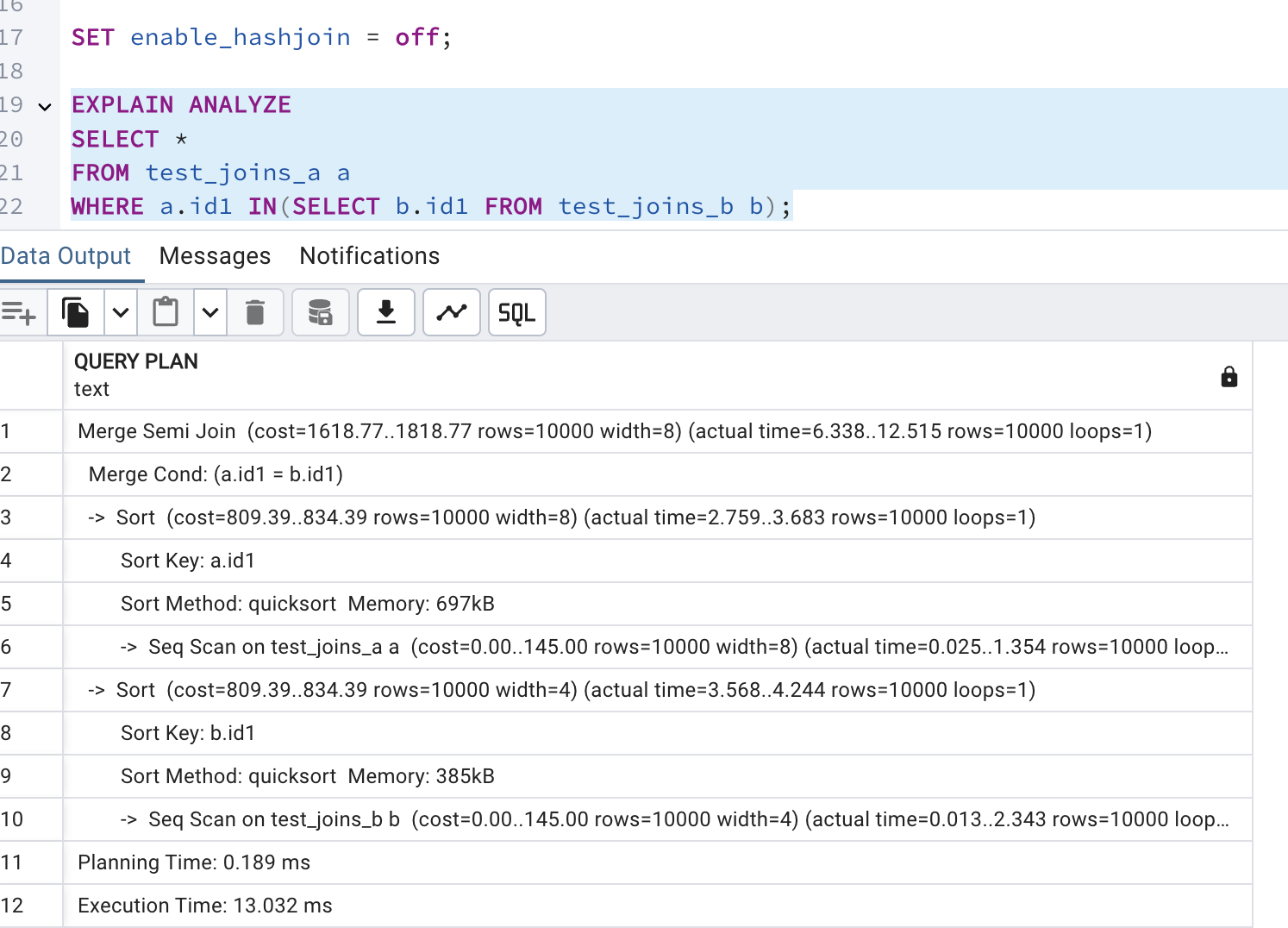
* 1. task 2

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

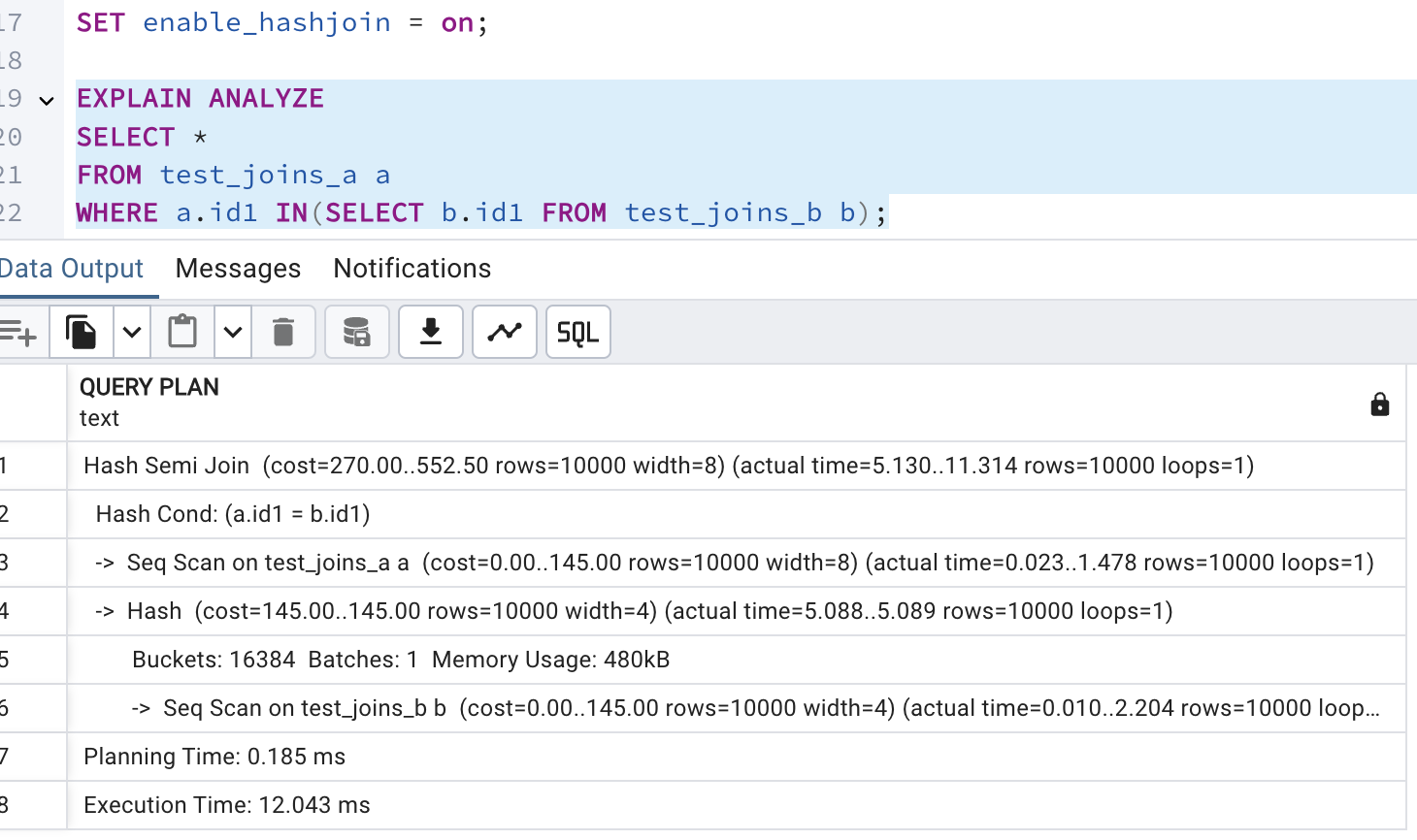
First of all, in order to ensure that a query is suitable for a Hash Join equality condition should be used.

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

A Hash 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.

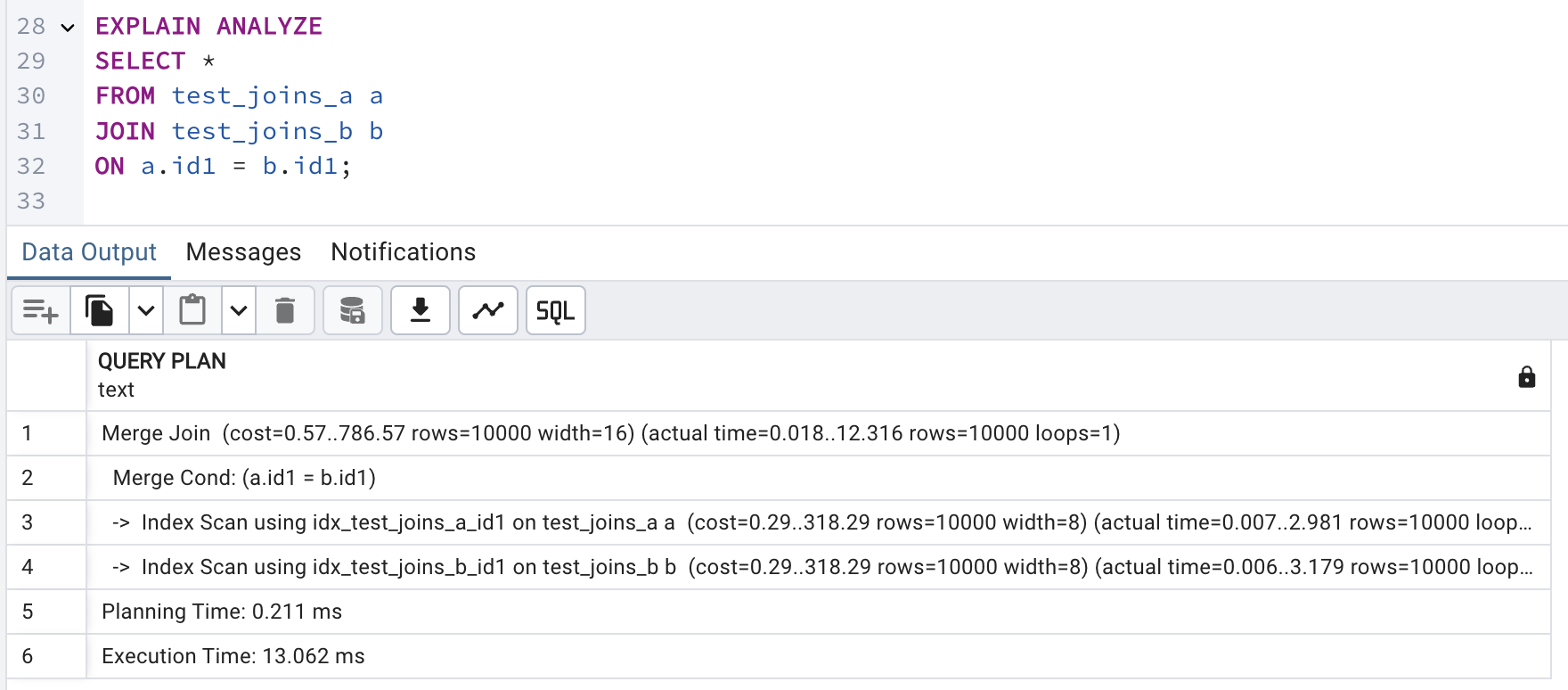


After enabling again using Hash Semi Join.

1.3

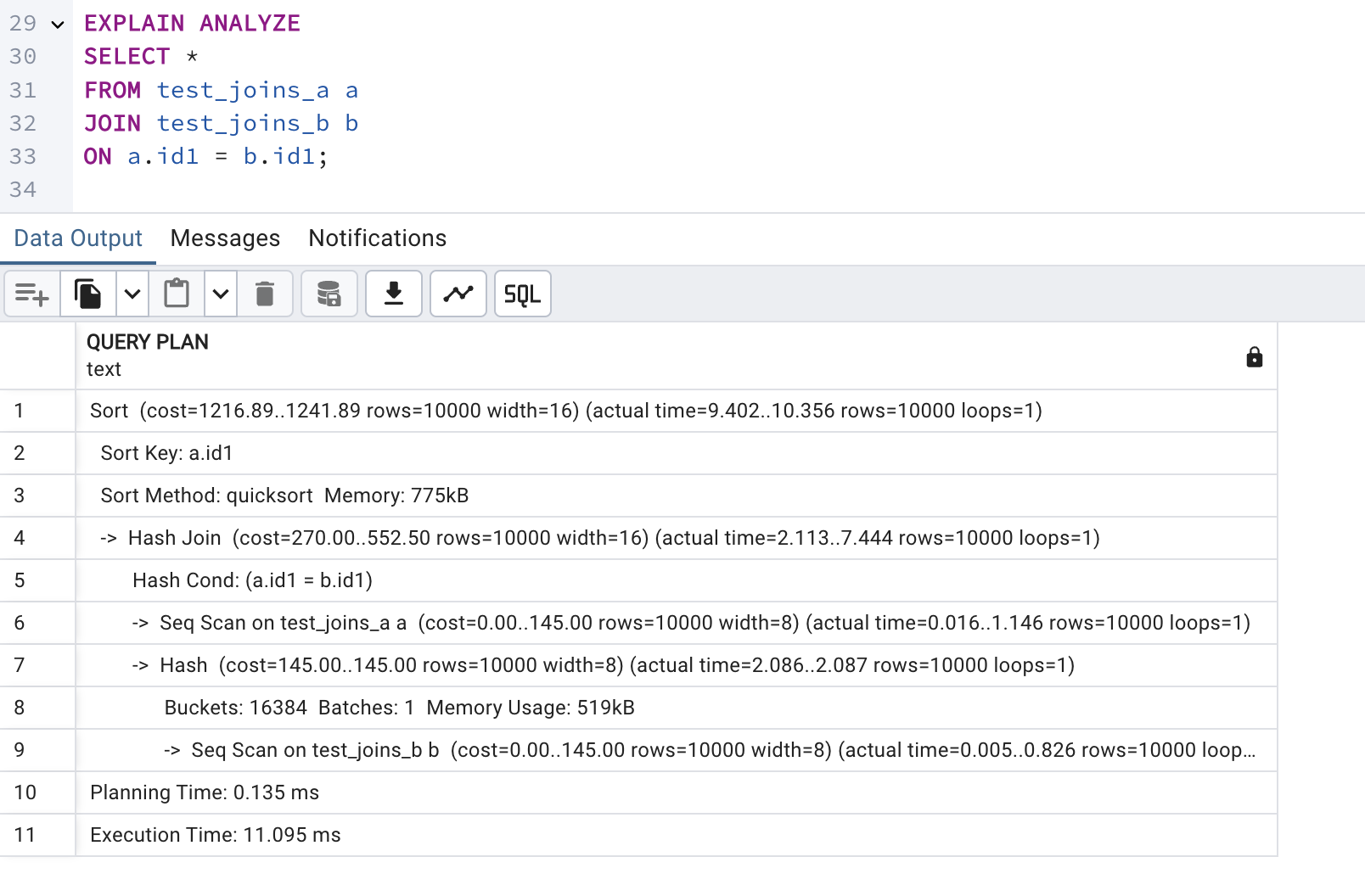
TASK 3: MERGE JOIN

1.Using Merge Join Method



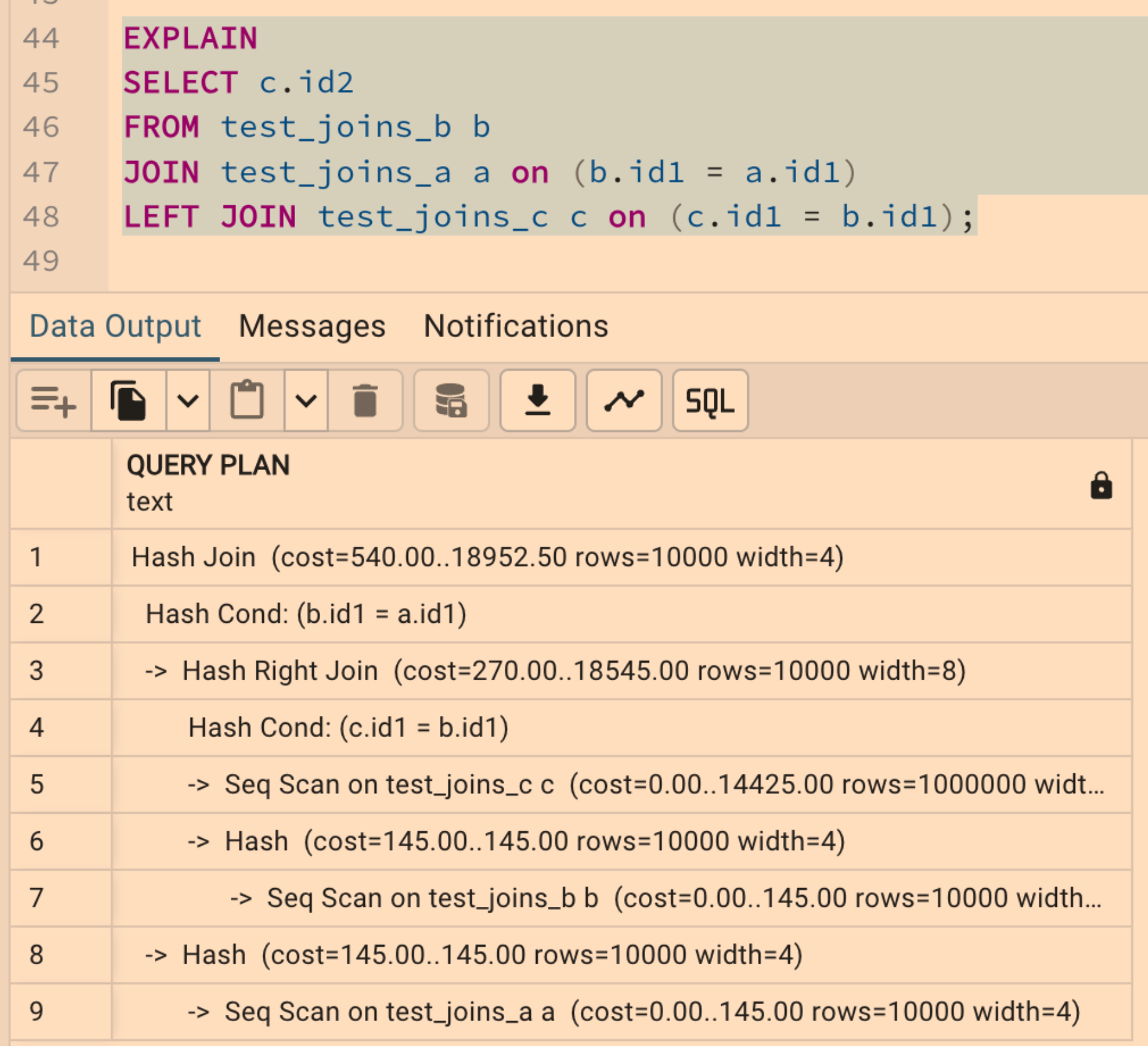
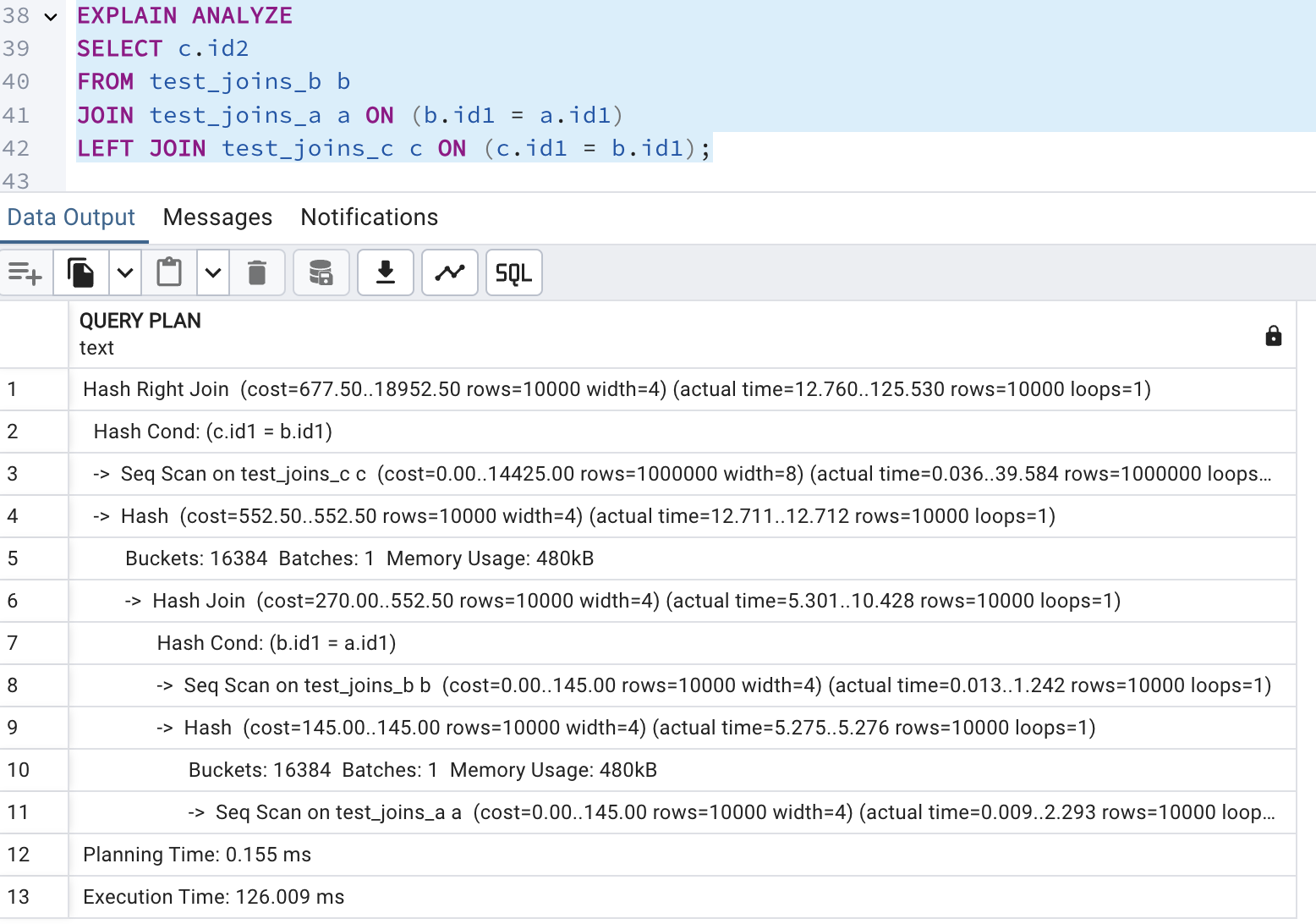
To ensure planner uses a merge join, created indexes on the join columns and then run the query.

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

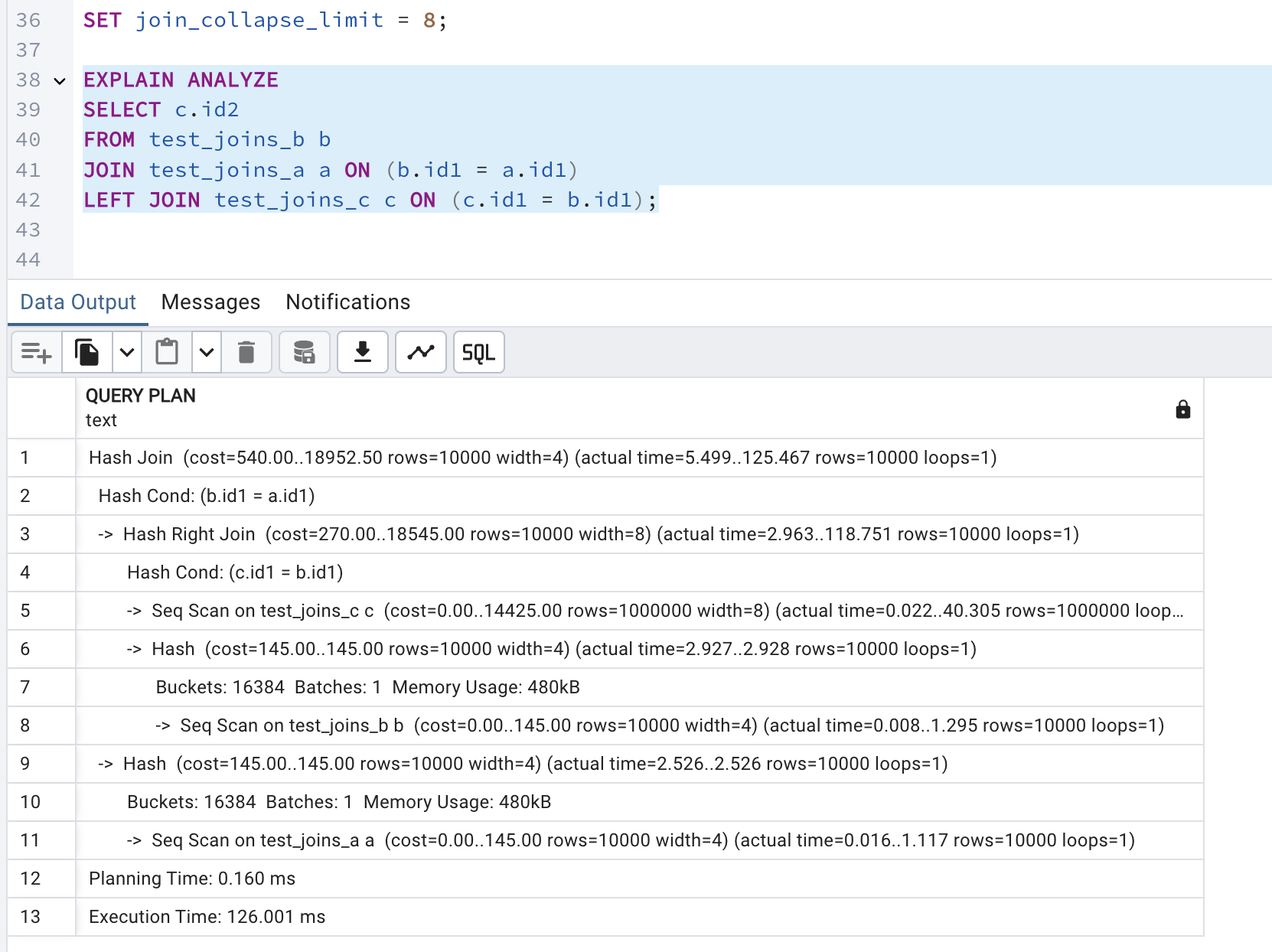


The query is trying to use a Merge Join, but the planner still chooses a Hash Join instead. This happens because SET enable\_mergejoin = off what tells PostgreSQL not to use Merge Join, so it chooses Hash Join instead. The plan shows that the table is sorted (Sort Key: a.id1), but PostgreSQL still thinks Hash Join is faster.

2.1 TASK 4: CHANGING JOIN ORDER

1. Create a table and populate it with sample data
2. Check the plan. Describe the order of tables joining:
3. Set join\_collapse\_limit = 1 a:

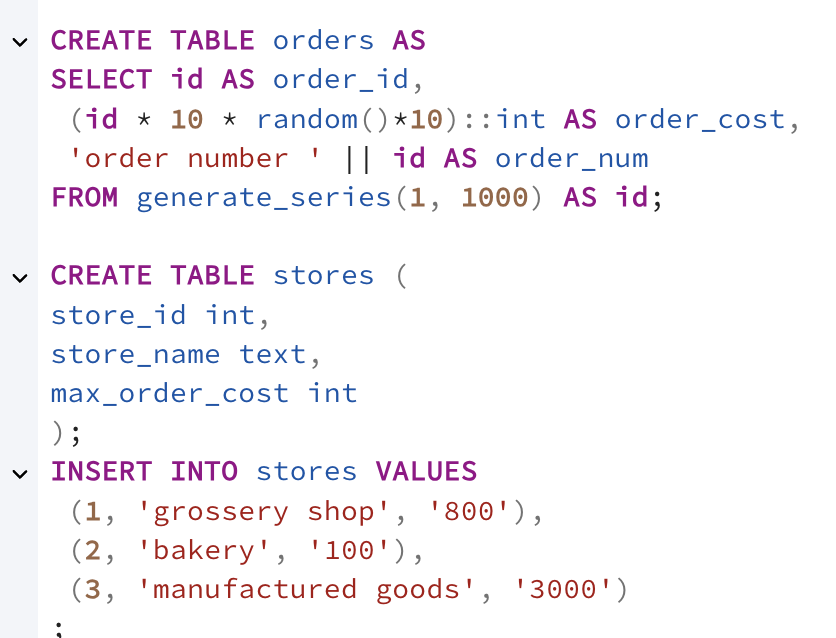
The query plan shows that PostgreSQL used Hash Joins for both joins, following the exact order written in the query. The first join between test\_joins\_b and test\_joins\_a was processed using a Hash Join, then the LEFT JOIN with test\_joins\_c was executed using a Hash Right Join. Sequential scans were performed on all tables, meaning indexes were not used for lookups.

3. Return join\_collapse\_limit = 8.The join order changed, but both plans still use Hash Joins and Sequential Scans, meaning the overall execution time remained similar.

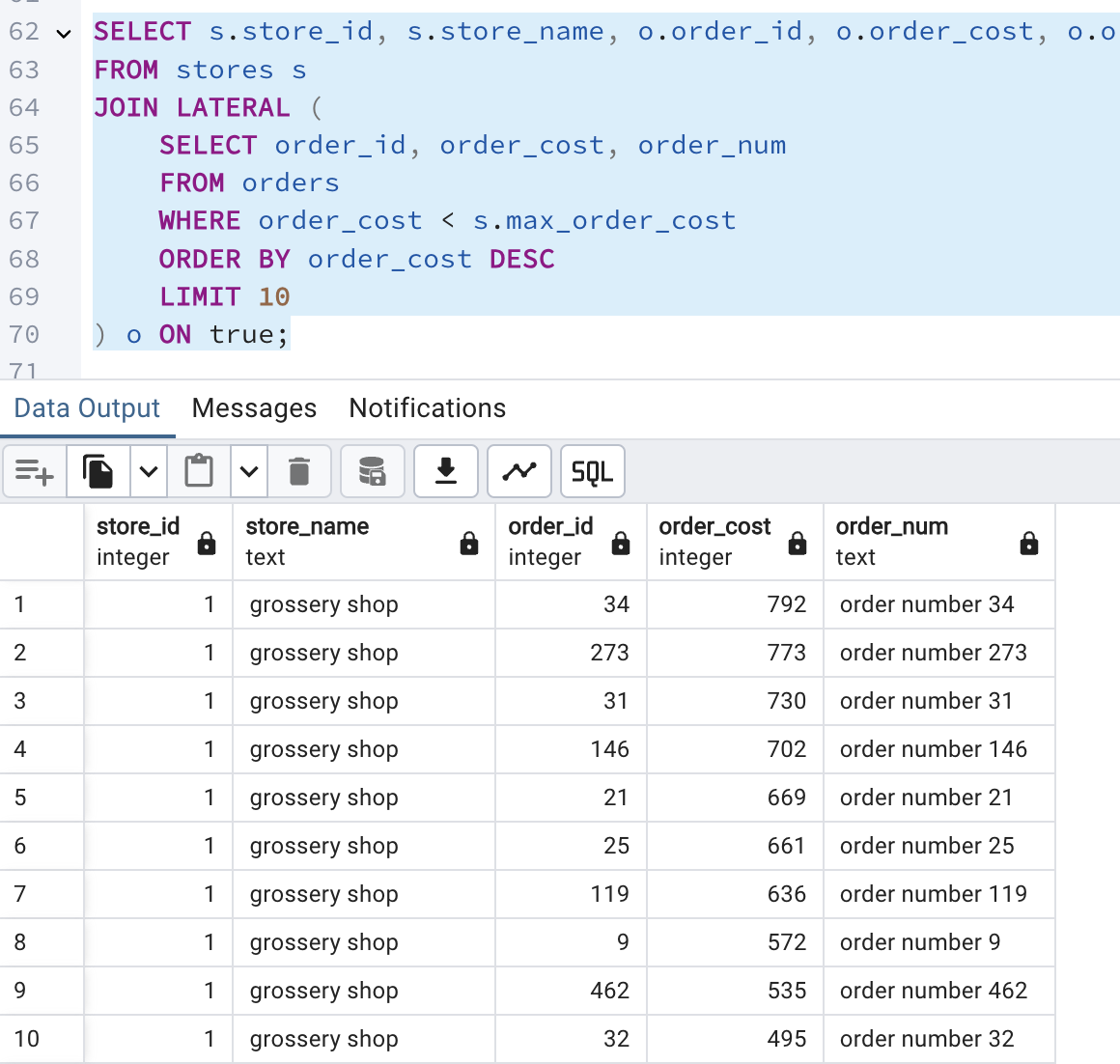
Allowing the planner to optimize (join\_collapse\_limit = 8) generally helps for larger queries where different join orders significantly impact performance.

* 1. TASK 5: LATERAL JOIN

1. Create tables and populate them by data;

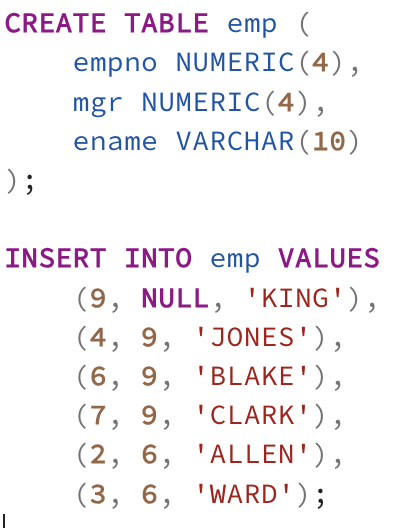


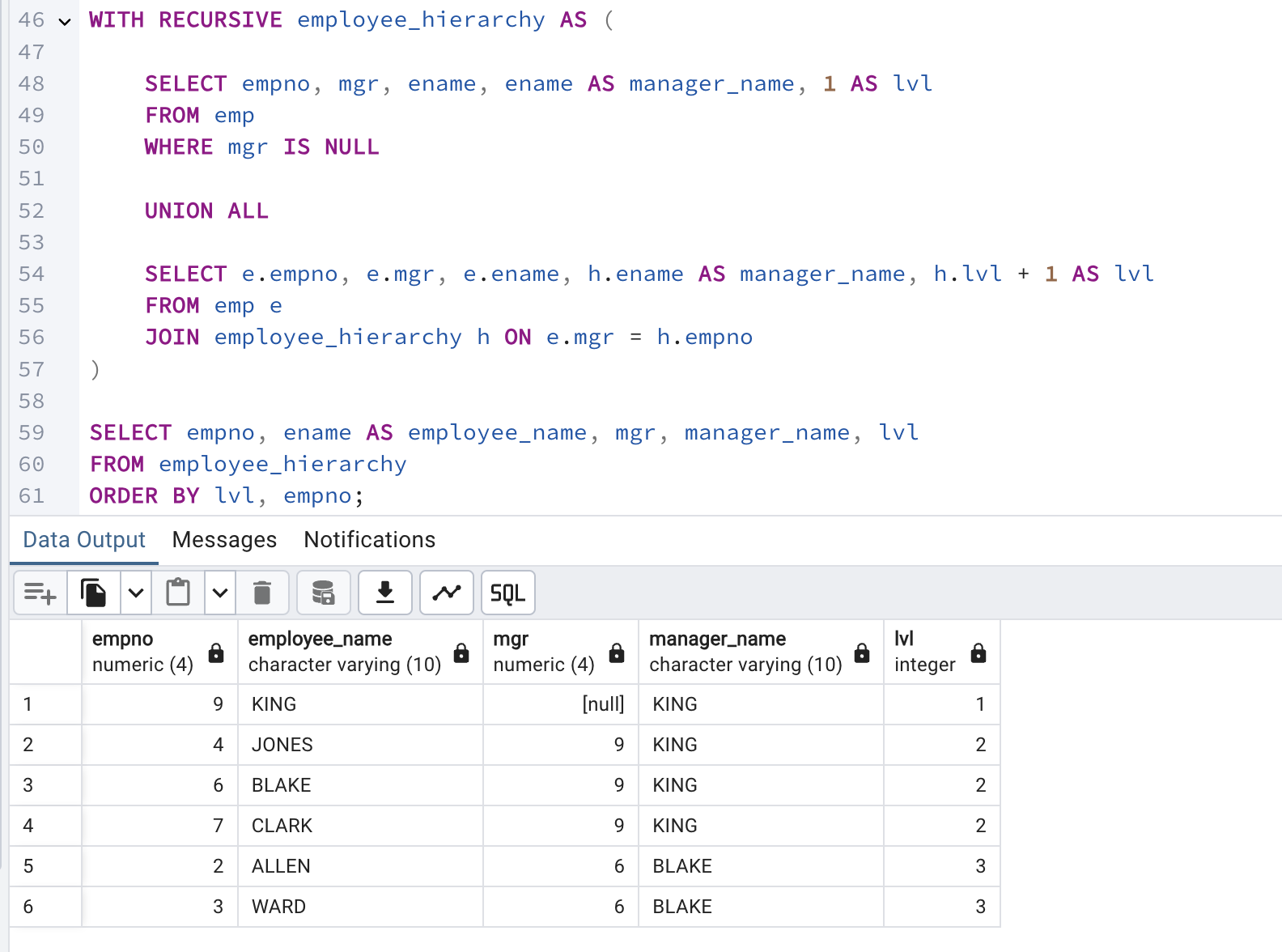
1. Create a query to find TOP 10 of orders by it cost for each store. So, on the output you should have 10 orders for each store (or less, depends on sample random data) with cost less than max\_order\_cost. Use LATERAL join.



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

