# 1. JOIN METHODS

## 1.1 TASK 1: NESTED LOOP JOIN

### 1.Create test tables and populate them with test data:

**SET** search\_path **TO** labs, public

**DROP** **TABLE** **IF** **EXISTS** test\_joins\_a, test\_joins\_b;

**CREATE** **TABLE** test\_joins\_a (id1 **int**, id2 **int**);

**CREATE** **TABLE** test\_joins\_b (id1 **int**, id2 **int**);

**INSERT** **INTO** test\_joins\_a

**SELECT** **generate\_series**(1,10\_000), 3;

**INSERT** **INTO** test\_joins\_b

**SELECT** **generate\_series**(1,10\_000), 3;

**ANALYZE**;

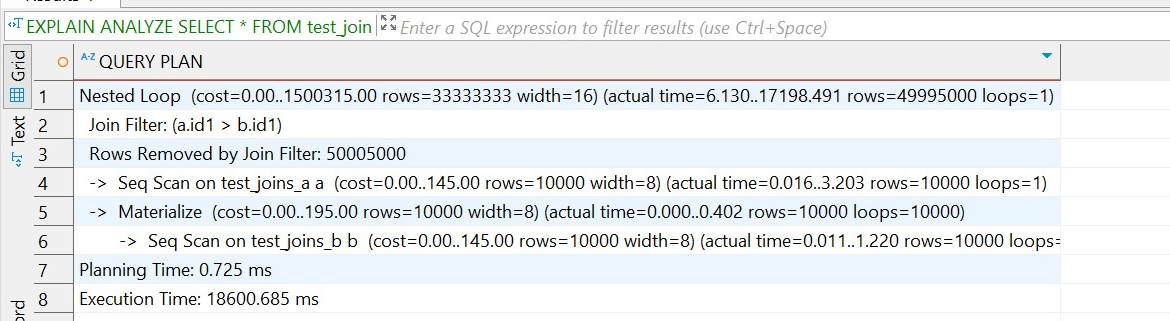
### 2. Check how NESTED LOOP JOIN method is used in below queries. Why?

-- query with inequality join

**EXPLAIN** **ANALYZE**

**SELECT** \* **FROM** test\_joins\_a a, test\_joins\_b b

**WHERE** a.id1 > b.id1;



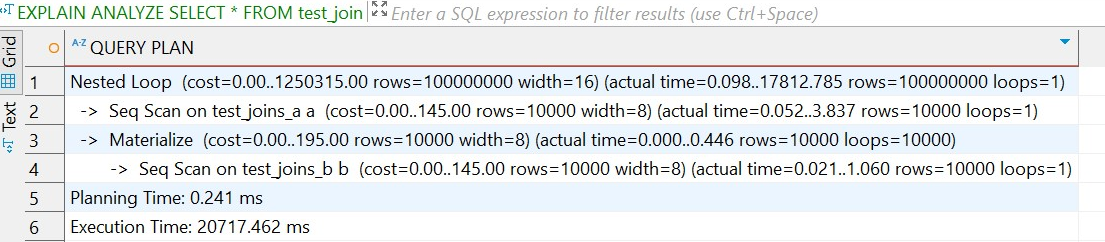
-- cross join query

**EXPLAIN** **ANALYZE**

**SELECT** \*

**FROM** test\_joins\_a a

**CROSS** **JOIN** test\_joins\_b b;



In both queries, the PostgreSQL planner chose the **Nested Loop Join** strategy due to the nature of the join conditions. In the first query, the **inequality predicate** a.id1 > b.id1 disqualifies hash and merge joins, forcing the planner to use a nested loop with a join filter. In the second query, the **CROSS JOIN** explicitly produces a Cartesian product, leaving nested loop as the only viable join algorithm. Despite the nested loop join being computationally expensive the planner used it because it was **the only compatible method** for the given logic. The execution time difference,18s and 20s, reflects the number of output rows and filter evaluations.

## 1.2 TASK 2 HASH JOIN

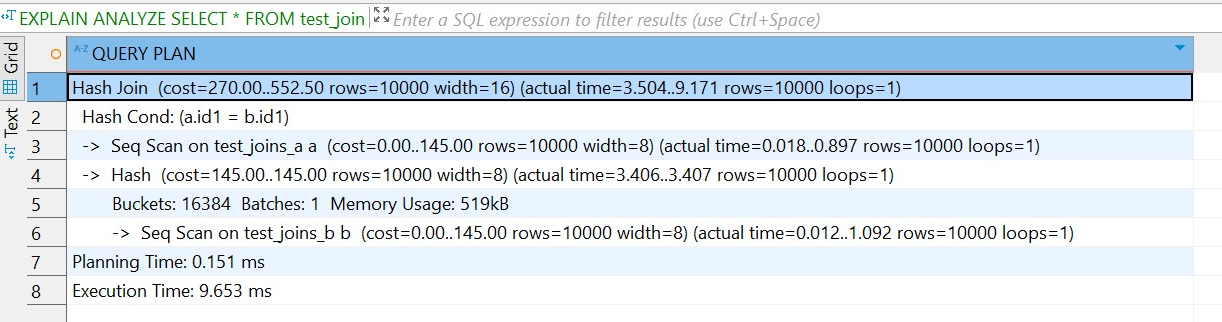
### 1. Rewrite SELECT to instruct the planner to use HASH JOIN method:

I tried forcing PostgreSQL to use HASH JOIN, bt turning off nested loops using (**SET** enable\_nestloop = **off**;) and merge joins (**SET** enable\_mergejoin = **off**;) but it didn’t work, so I had to change inequality to equality in the query, because HASH JOIN’s are used in cases where we have equality.

**EXPLAIN** **ANALYZE**

**SELECT** \* **FROM** test\_joins\_a a, test\_joins\_b b

**WHERE** a.id1 = b.id1;



So once again, this join strategy was selected because **Hash Joins are optimal for equality conditions** and can efficiently match large volumes of data without indexes. This confirms that to trigger a hash join in PostgreSQL, the join condition must be **equality-based**. The total execution time dropped to just **9.6 ms,** highlighting the performance advantage of hash joins over nested loops for such workloads.

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

For this task I didn’t turn on nestloop and merge join and proceeded:

**EXPLAIN** **ANALYZE**

**SELECT** \*

**FROM** test\_joins\_a a

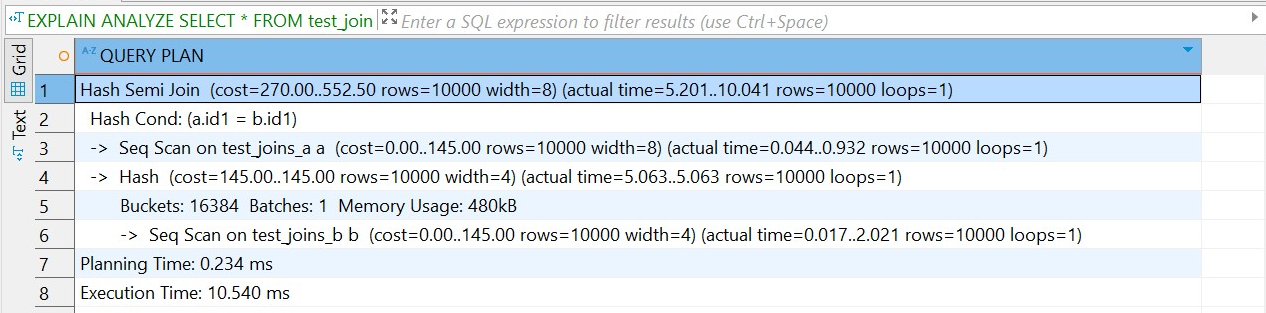
**WHERE** **EXISTS** (

**SELECT** 1

**FROM** test\_joins\_b b

**WHERE** b.id1 = a.id1

);



For this query, PostgreSQL chose this plan because it offers efficient execution when checking for the presence of matching rows, without retrieving any data from the joined table. The hash table of test\_joins\_b was built once in memory, and each row of test\_joins\_a was checked against it. The total execution time was only **10.540 ms**, demonstrating the performance advantage of this join strategy for semi-join scenarios.

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

**SET** enable\_hashjoin = **off**;

**EXPLAIN** **ANALYZE**

**SELECT** \*

**FROM** test\_joins\_a a

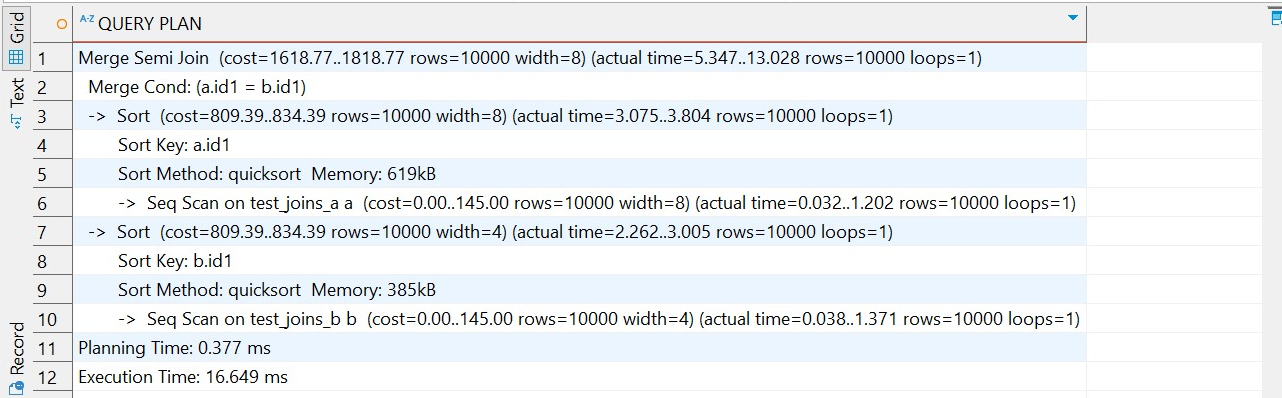
**WHERE** **EXISTS** (

**SELECT** 1

**FROM** test\_joins\_b b

**WHERE** b.id1 = a.id1

);



When enable\_hashjoin was disabled, PostgreSQL fell back to a **Merge Semi Join**. This join strategy requires sorting both tables by the join key (id1) and then performing a merge pass to detect matching rows. While still efficient for equality-based joins, it took slightly more time to execute (**16.649 ms**) than the previous **Hash Semi Join** due to the need to sort the input tables. This confirms that PostgreSQL dynamically adapts its join strategy based on the enabled planner settings and available options.

## 1.3 TASK 3: MERGE JOIN

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

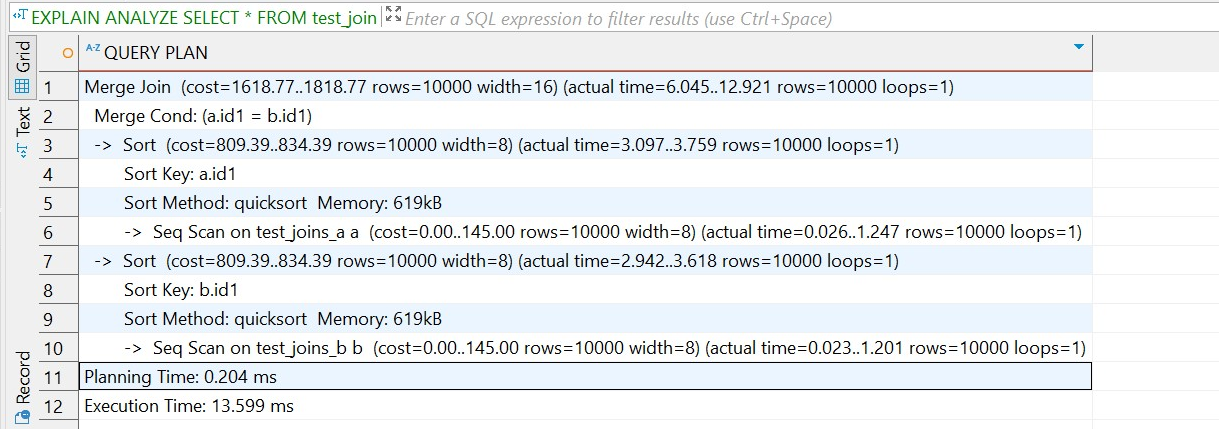
To make PostgreSQL use MERGE SORT, we have to meet few requirements. The join condition should be equality and both inputs should be sorted. Also if our tables have no indexes, PostgreSQL will most likely sort them using MERGE JOIN automatically.

**EXPLAIN** **ANALYZE**

**SELECT** \*

**FROM** test\_joins\_a a

**JOIN** test\_joins\_b b **ON** a.id1 = b.id1;



Since we met all the requirements, PostgreSQL sorted each table on and then merged the streams, showing 10 000 matching rows in 13.6 ms. The plan confirms that merge joins are favored when an equality predicate exists and the planner estimates that the combined cost of sorting plus merging is cheaper than other enabled join strategies.

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

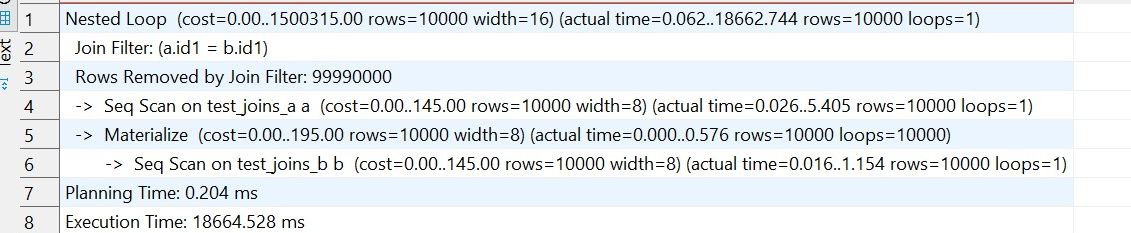
**SET** enable\_mergejoin = **off**;

**EXPLAIN** **ANALYZE**

**SELECT** \*

**FROM** test\_joins\_a a

**JOIN** test\_joins\_b b **ON** a.id1 = b.id1;



When enable\_mergejoin was disabled, PostgreSQL fell back to using a **Nested Loop Join** to evaluate the equality join a.id1 = b.id1. This resulted in a **huge performance drop** (execution time: **18.7 seconds**) due to the planner evaluating 100 million pairwise combinations. The fallback confirms that **disabling efficient join strategies can severely impact query performance,** especially when the remaining join types are computationally expensive and not optimized for large datasets.

**SET** enable\_mergejoin = **on**;

# 2. JOIN ORDER AND LATERAL JOIN

## 2.1 TASK 4: CHANGING JOIN ORDER

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

**DROP** **TABLE** **IF** **EXISTS** test\_joins\_c;

**CREATE** **TABLE** test\_joins\_c (

id1 **INT**,

id2 **INT**

);

**INSERT** **INTO** test\_joins\_c

**SELECT** **generate\_series**(1, 1000000), (**random**() \* 10)::**INT**;

**ANALYZE** test\_joins\_c;

### 

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

**EXPLAIN**

**SELECT** c.id2

**FROM** test\_joins\_b b

**JOIN** test\_joins\_a a **ON** b.id1 = a.id1

**LEFT** **JOIN** test\_joins\_c c **ON** c.id1 = b.id1;



Under the default join\_collapse\_limit = 8, PostgreSQL reordered the inner join to optimize performance. It first performed a **Merge Join** between test\_joins\_b and test\_joins\_a on b.id1 = a.id1, sorting both sides to enable an efficient streaming merge. Then it applied a **Merge Left Join** with test\_joins\_c on b.id1 = c.id1, sorting c.id1 to match the merge requirement. The planner chose this order to minimize the cost of executing the query. This behavior demonstrates that when join\_collapse\_limit allows flexibility, PostgreSQL uses cost-based optimization to determine the best join order and join methods, often using **merge joins** when sorted inputs are available or affordable to create.

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

**SET** join\_collapse\_limit = 1;

**EXPLAIN**

**SELECT** c.id2

**FROM** test\_joins\_b b

**JOIN** test\_joins\_a a **ON** b.id1 = a.id1

**LEFT** **JOIN** test\_joins\_c c **ON** c.id1 = b.id1;



After setting join\_collapse\_limit = 1, PostgreSQL did **not change** the join order or the plan structure compared to the default (join\_collapse\_limit = 8). This confirms that even with join reordering disabled, PostgreSQL **preserved the written join order**, which matched the most optimal plan. Since the inner join between b and a was written first and the planner had no reason to reorder them, the same plan was produced.

## 2.2 TASK 5: LATERAL JOIN

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

**DROP** **TABLE** **IF** **EXISTS** orders;

**CREATE** **TABLE** orders **AS**

**SELECT**

id **AS** order\_id,

(id \* 10 \* **random**() \* 10)::**int** **AS** order\_cost,

'order number ' || id **AS** order\_num

**FROM** **generate\_series**(1, 1000) **AS** id;

**DROP** **TABLE** **IF** **EXISTS** stores;

**CREATE** **TABLE** stores (

store\_id **INT**,

store\_name **TEXT**,

max\_order\_cost **INT**

);

**INSERT** **INTO** stores **VALUES**

(1, 'grossery shop', 800),

(2, 'bakery', 100),

(3, 'manufactured goods', 3000)

**ON** **CONFLICT** **DO** **NOTHING**;

### 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.

**SELECT**

*s*.store\_id,

*s*.store\_name,

o.order\_id,

o.order\_cost,

o.order\_num

**FROM** stores *s*

**LEFT** **JOIN** LATERAL (

**SELECT** \*

**FROM** orders o

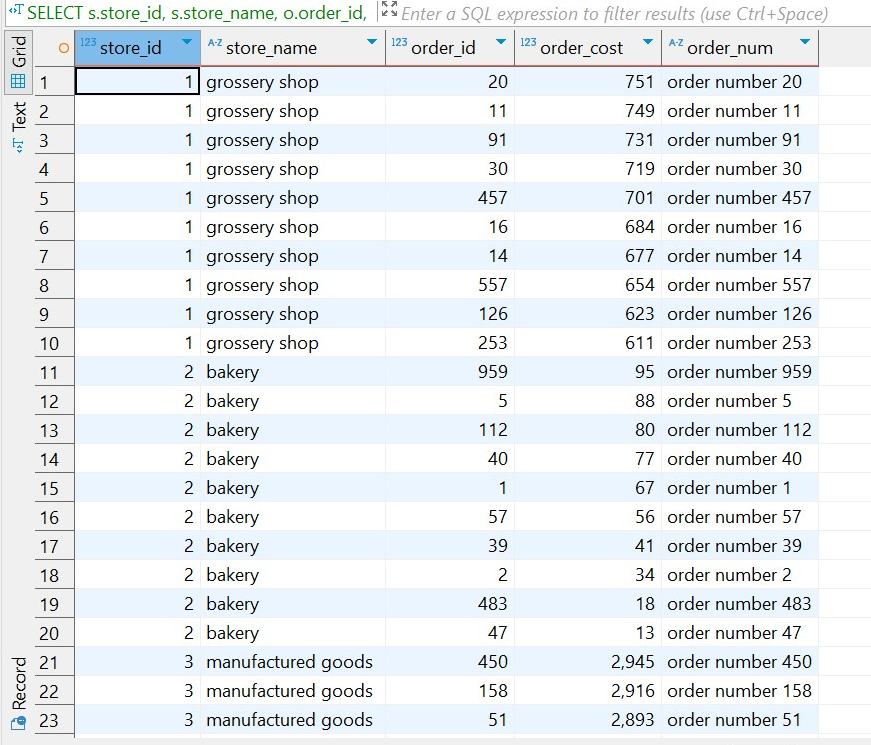
**WHERE** o.order\_cost < s.max\_order\_cost

**ORDER** **BY** o.order\_cost **DESC**

**LIMIT** 10

) o **ON** **true**;

The LATERAL subquery is evaluated separately for each row in the stores table, allowing it to access s.max\_order\_cost from the outer query. As a result, PostgreSQL efficiently returns up to 10 matching orders per store, sorted in descending order by cost. This approach is ideal for **correlated subqueries** where the inner query depends on values from the outer table.



# 3. CTES

## 3.1 TASK 6: RECURSIVE CTE

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

**WITH** **RECURSIVE** emp\_hierarchy **AS** (

**SELECT**

*e*.empno,

*e*.mgr,

*e*.ename,

*e*.ename **AS** *mngname*,

1 **AS** *lvl*

**FROM** emp *e*

**WHERE** *e*.mgr **IS** **NULL**

**UNION** **ALL**

**SELECT**

*e*.empno,

*e*.mgr,

*e*.ename,

*eh*.ename **AS** *mngname*,

*eh*.lvl + 1

**FROM** emp *e*

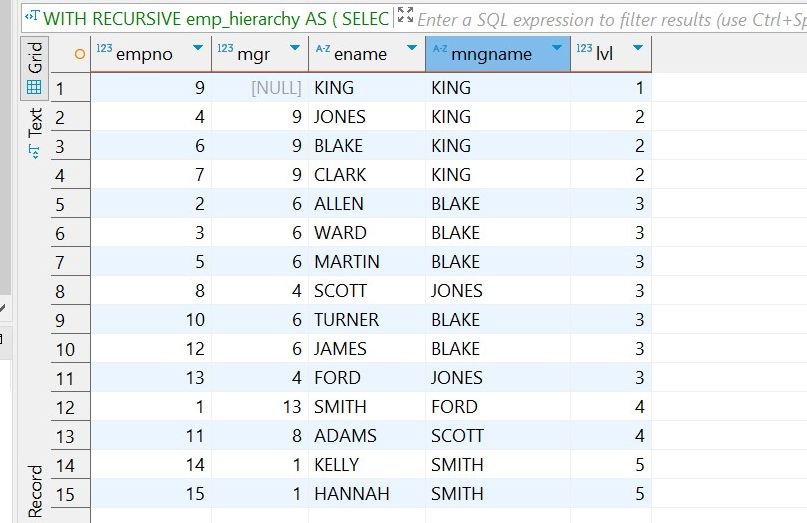
**JOIN** emp\_hierarchy *eh* **ON** *e*.mgr = *eh*.empno

)

**SELECT** \* **FROM** emp\_hierarchy

**ORDER** **BY** lvl, empno;

Using a recursive CTE, we retrieved all employees along with their manager’s name and calculated their level in the company hierarchy, starting from the president. The assigns each employee a management level (lvl) and links them to their direct manager (mngname).



## 3.2 TASK 7: CHANGING DATA CTE

### 1. Create log table for emp table:

**CREATE** **TABLE** order\_log (

log\_id **integer** **primary** **key** **generated** **always** **as** **identity**,

order\_id **integer**,

order\_cost **integer**,

order\_num **text**,

action\_type **varchar**(1) **CHECK** (action\_type **IN** ('U','D')),

log\_date **TIMESTAMPTZ** **DEFAULT** **now**()

);

### 

### 2. Update all rows for ORDER table:

**WITH** *updated\_rows* **AS** (

**UPDATE** orders

**SET** order\_cost = order\_cost / 2

**WHERE** order\_cost **BETWEEN** 100 **AND** 1000

**RETURNING** order\_id, order\_cost \* 2 **AS** old\_order\_cost, order\_num

),

log\_updates **AS** (

**INSERT** **INTO** order\_log (order\_id, order\_cost, order\_num, action\_type)

**SELECT** order\_id, old\_order\_cost, order\_num, 'U'

**FROM** updated\_rows

**RETURNING** 1

),

deleted\_rows **AS** (

**DELETE** **FROM** orders

**WHERE** order\_cost < 50

**RETURNING** order\_id, order\_cost, order\_num

),

log\_deletes **AS** (

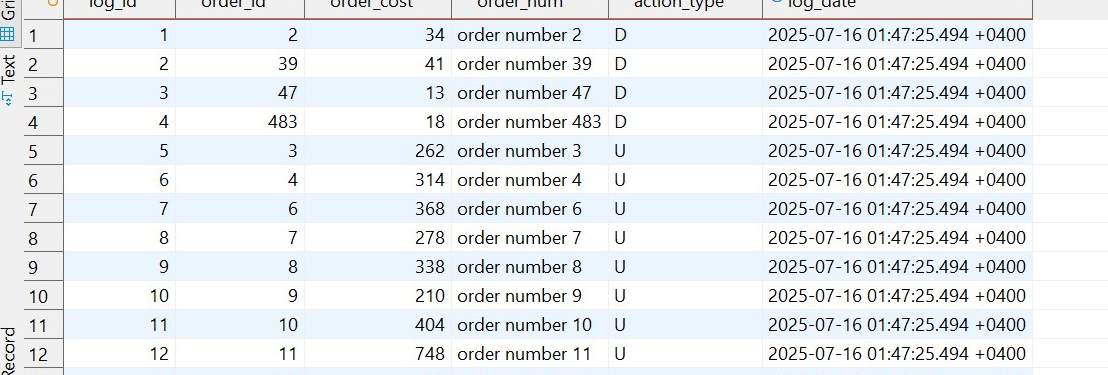
**INSERT** **INTO** order\_log (order\_id, order\_cost, order\_num, action\_type)

**SELECT** order\_id, order\_cost, order\_num, 'D'

**FROM** deleted\_rows

**RETURNING** 1

)



Using a single CTE ensures all operations are performed **atomically and consistently**, with updated rows marked as 'U' and deleted rows as 'D', preserving original values for audit purposes.