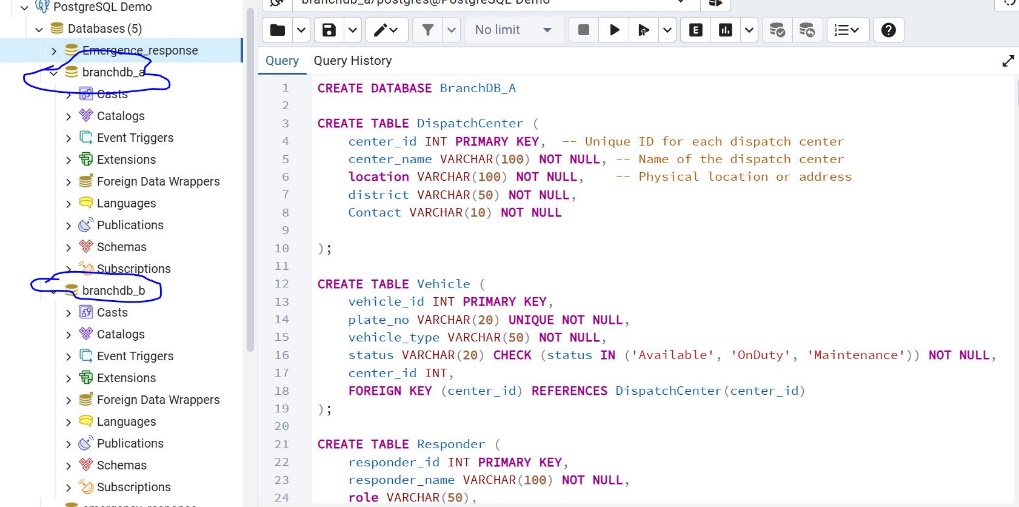
Q1. Split your database into two logical nodes (e.g., BranchDB\_A, BranchDB\_B) using horizontal or vertical fragmentation. Submit an ER diagram and SQL scripts that create both schemas.

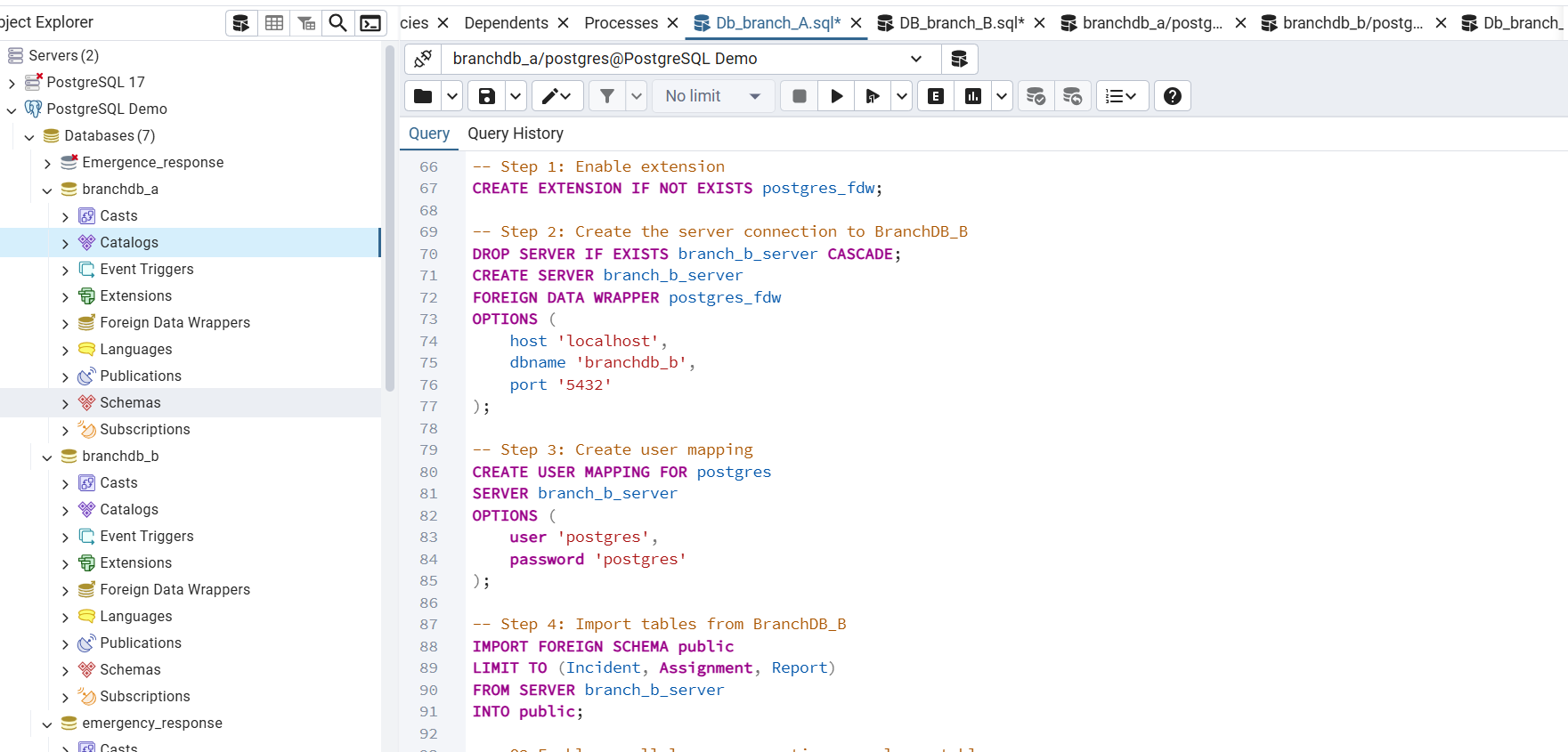
In this project, the Emergency Response database was split into two logical nodes, **BranchDB\_A** and **BranchDB\_B**, using **horizontal fragmentation**. BranchDB\_A stores all operational data related to field resources, including DispatchCenter, Responder, and Vehicle tables. BranchDB\_B stores the event related tables Incident, Assignment, and Report which track emergency cases and their responses.



Q2. Create a database link between your two schemas. Demonstrate a successful remote SELECT and a distributed join between local and remote tables. Include scripts and query

results.

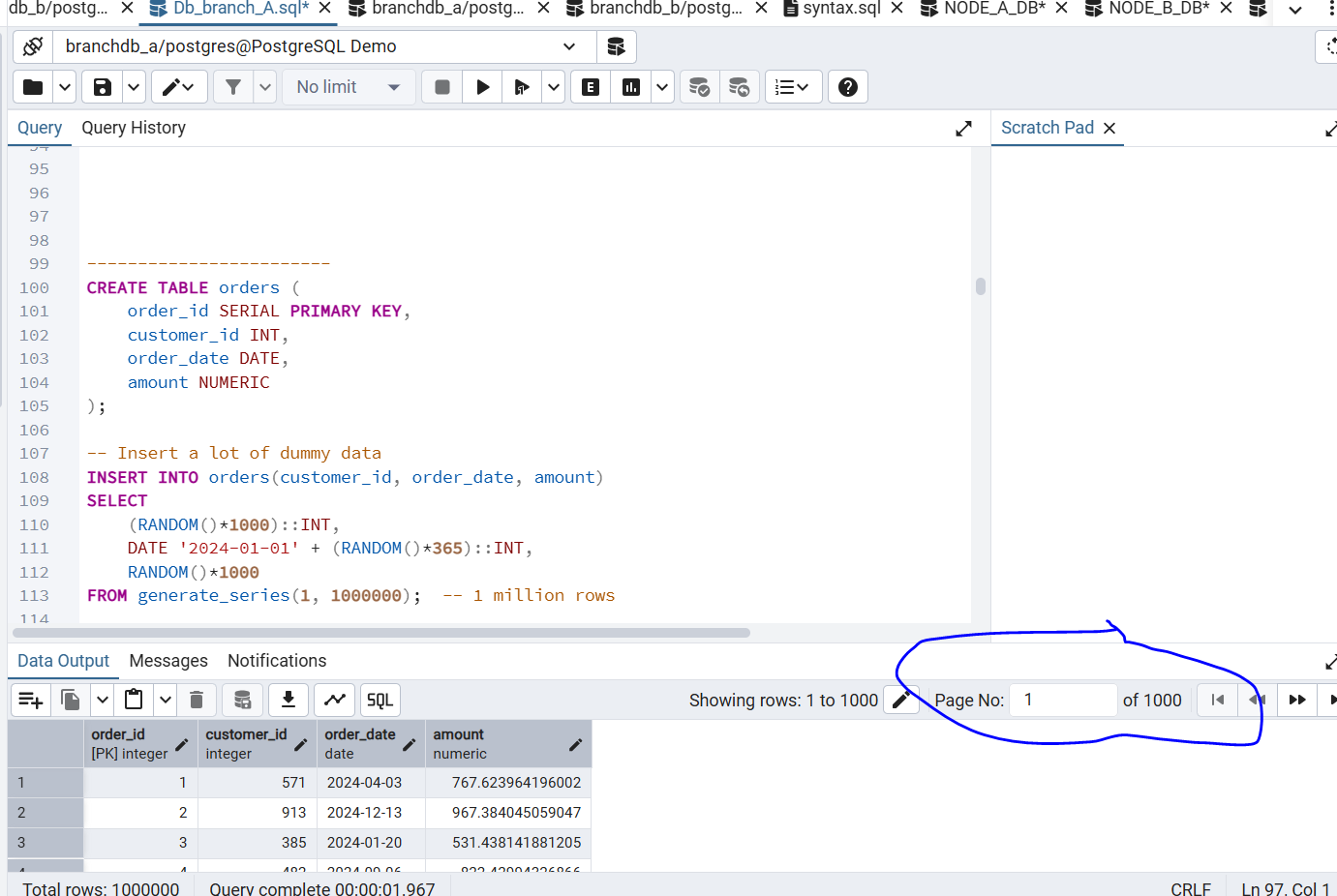
To enable distributed querying between the two fragmented nodes, a database link was created from **BranchDB\_A** to **BranchDB\_B** using the postgres\_fdw extension. The FDW server was configured with connection details for BranchDB\_B, and a user mapping was defined to authenticate the remote access.



Q3. Enable parallel query execution on a large table (e.g., Transactions, Orders). Use /\*+PARALLEL (table, 8) \*/ hint and compare serial vs parallel performance. Show

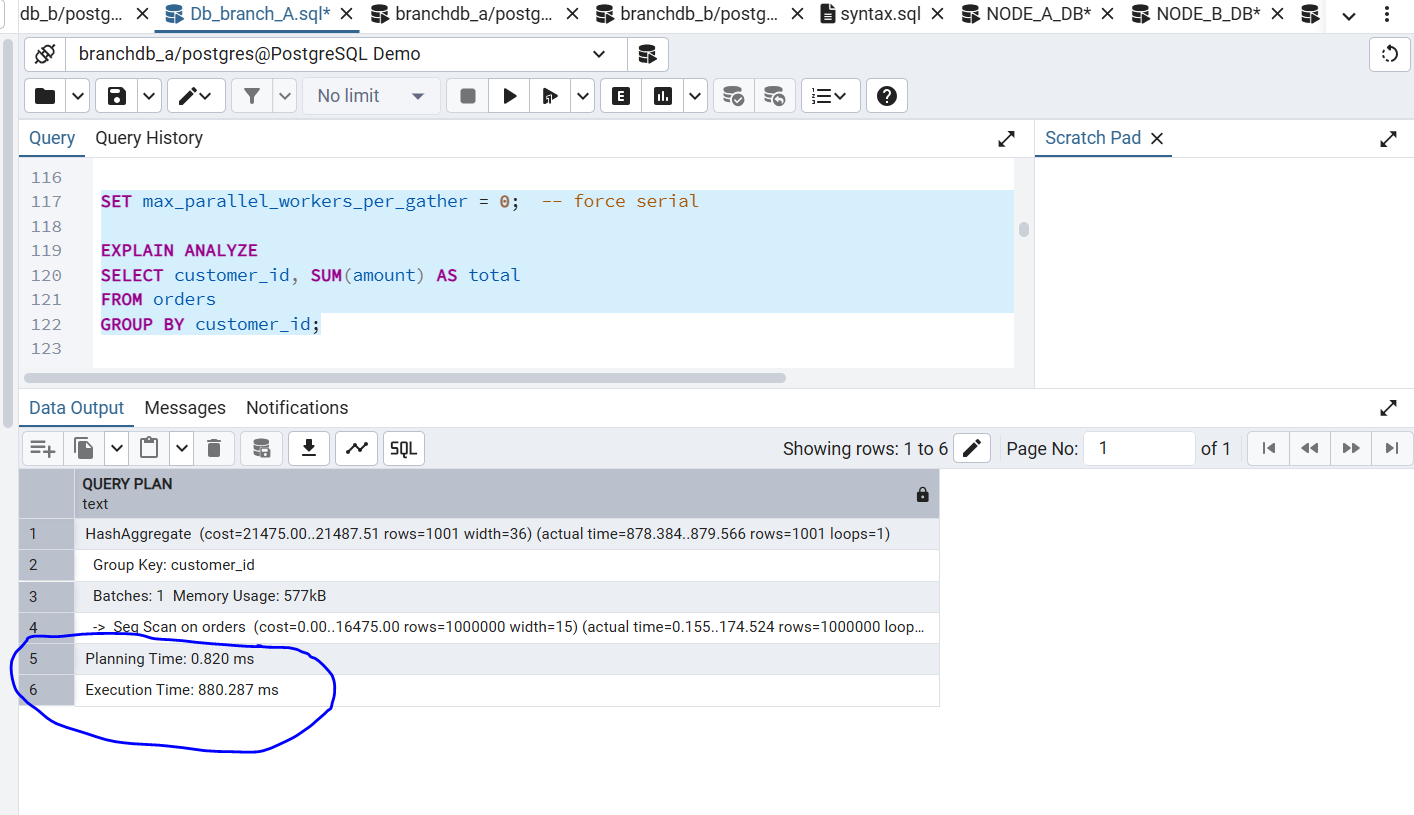
EXPLAIN PLAN output and execution time.

To evaluate the impact of parallel query execution in PostgreSQL, a large **Orders** table containing **1,000,000 rows** was created as shown in figure below

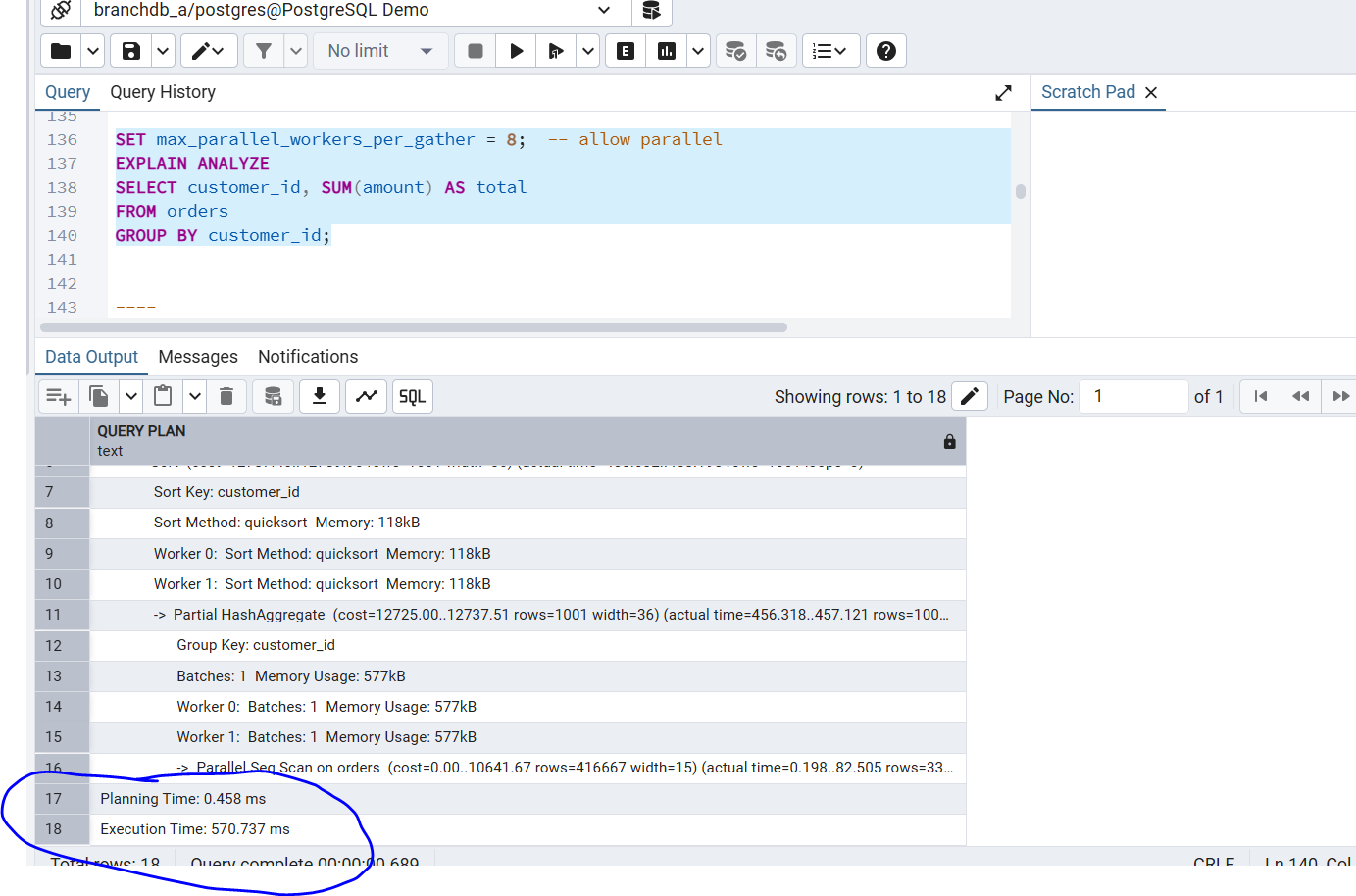


The test query calculated the total order amount per customer using a GROUP BY aggregation. First, the database was configured to run in **serial mode** by setting max\_parallel\_workers\_per\_gather = 0. Running EXPLAIN ANALYZE showed a normal sequential scan and aggregation with a higher execution time. Next, parallel execution was enabled by setting max\_parallel\_workers\_per\_gather = 8. Re-running the same query produced an execution plan showing **Parallel Seq Scan**, **Parallel Hash Aggregate**, and multiple worker processes. The parallel version executed significantly faster compared to the serial run.

Serial execution

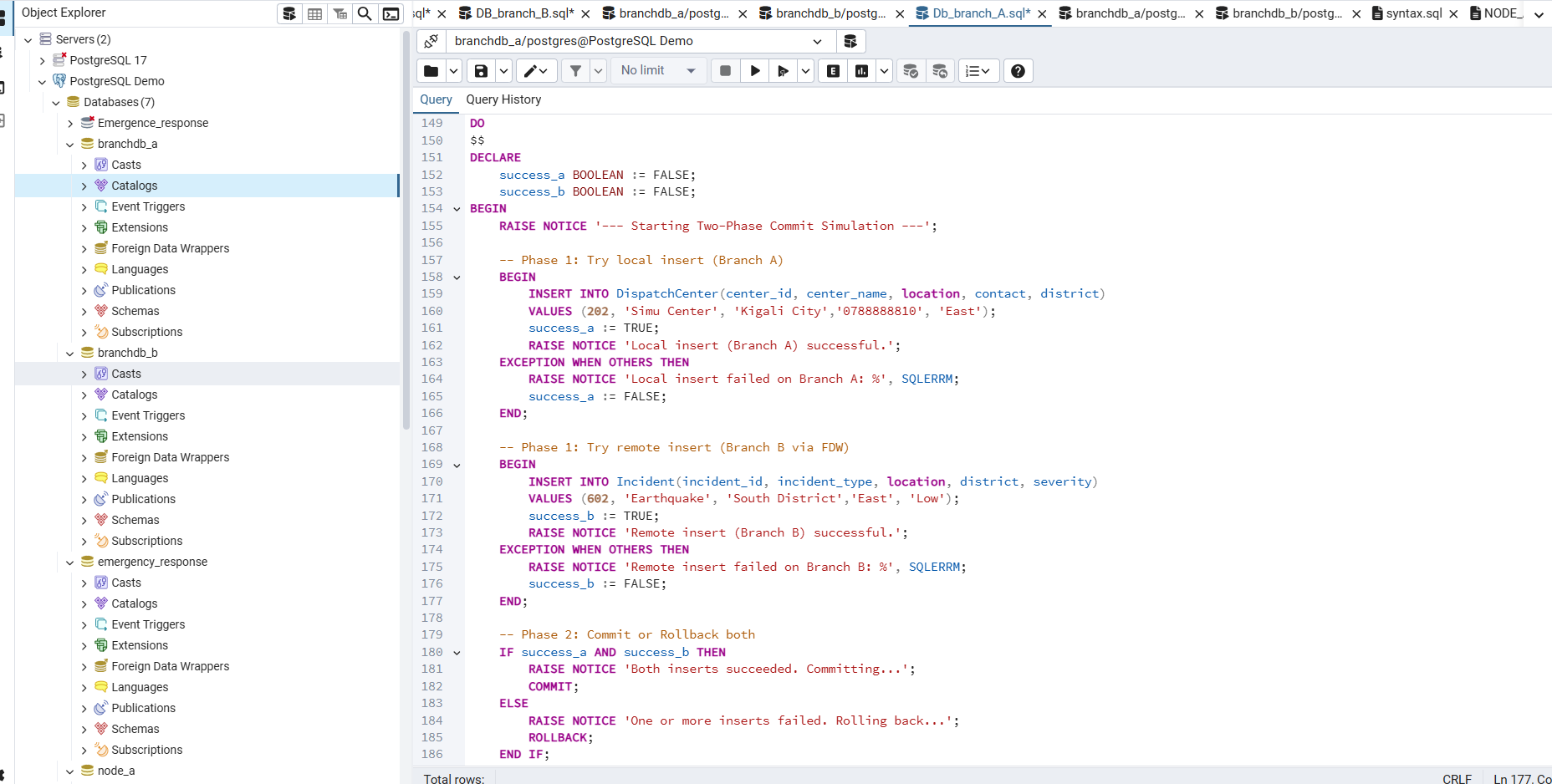


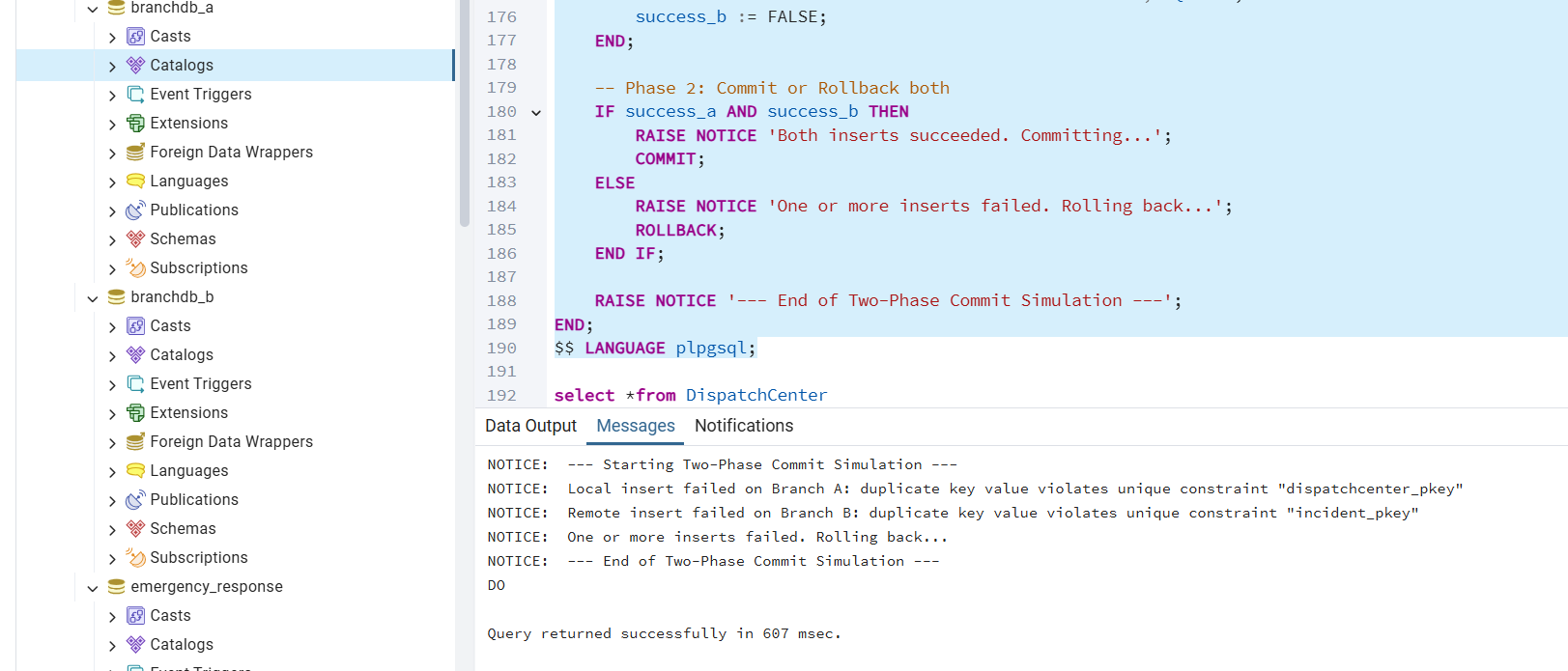
Parallel Execution



Q4. Write a PL/SQL block performing inserts on both nodes and committing once. Verify atomicity using DBA\_2PC\_PENDING. Provide SQL code and explanation of results.

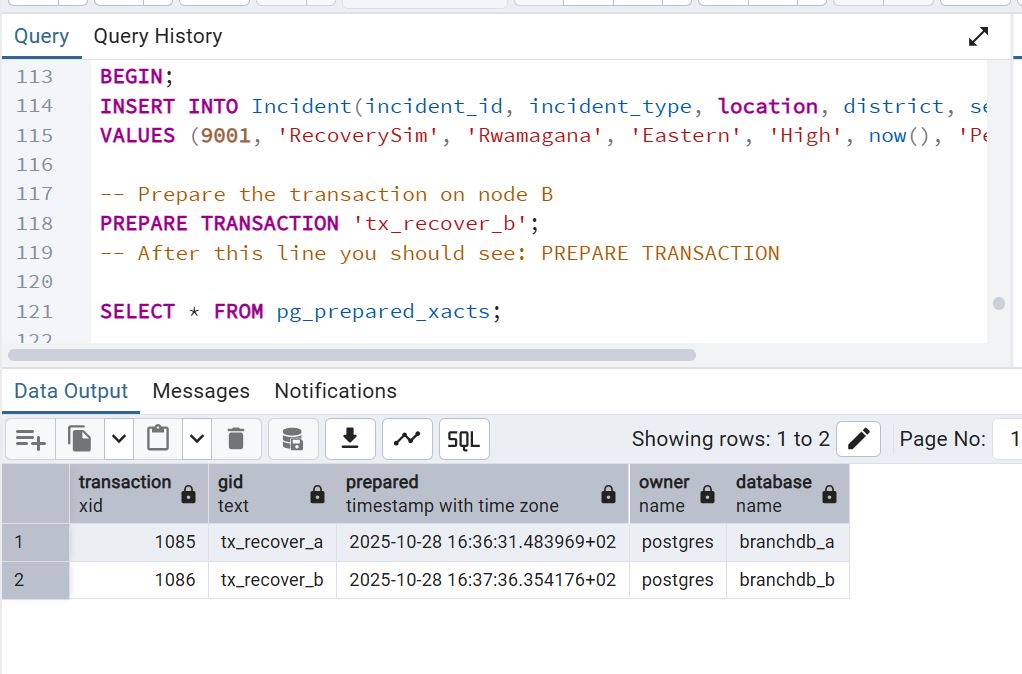
A two-phase commit across two distributed nodes: **BranchDB\_A** (local insert into DispatchCenter) and **BranchDB\_B** (remote insert into Incident via FDW). In the first phase, each insert was attempted independently, with the block recording whether each operation succeeded. In the second phase, the transaction was committed only if **both** inserts were successful; otherwise, a full rollback was issued to maintain atomicity. This ensures that either both nodes are updated or neither node changes, mimicking two-phase commit behavior.

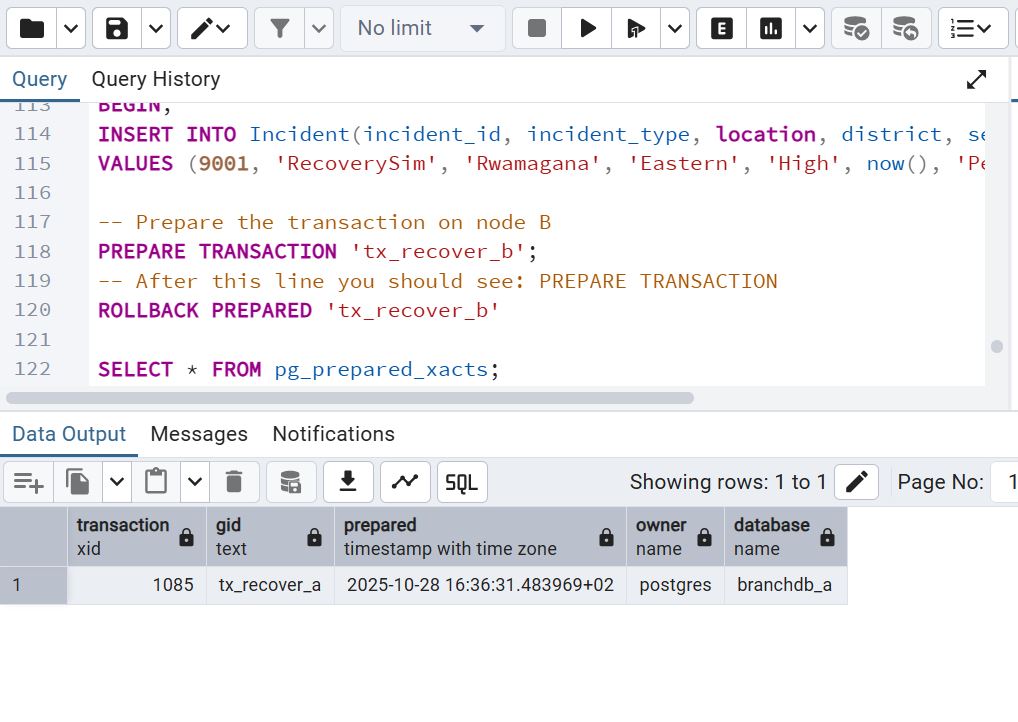




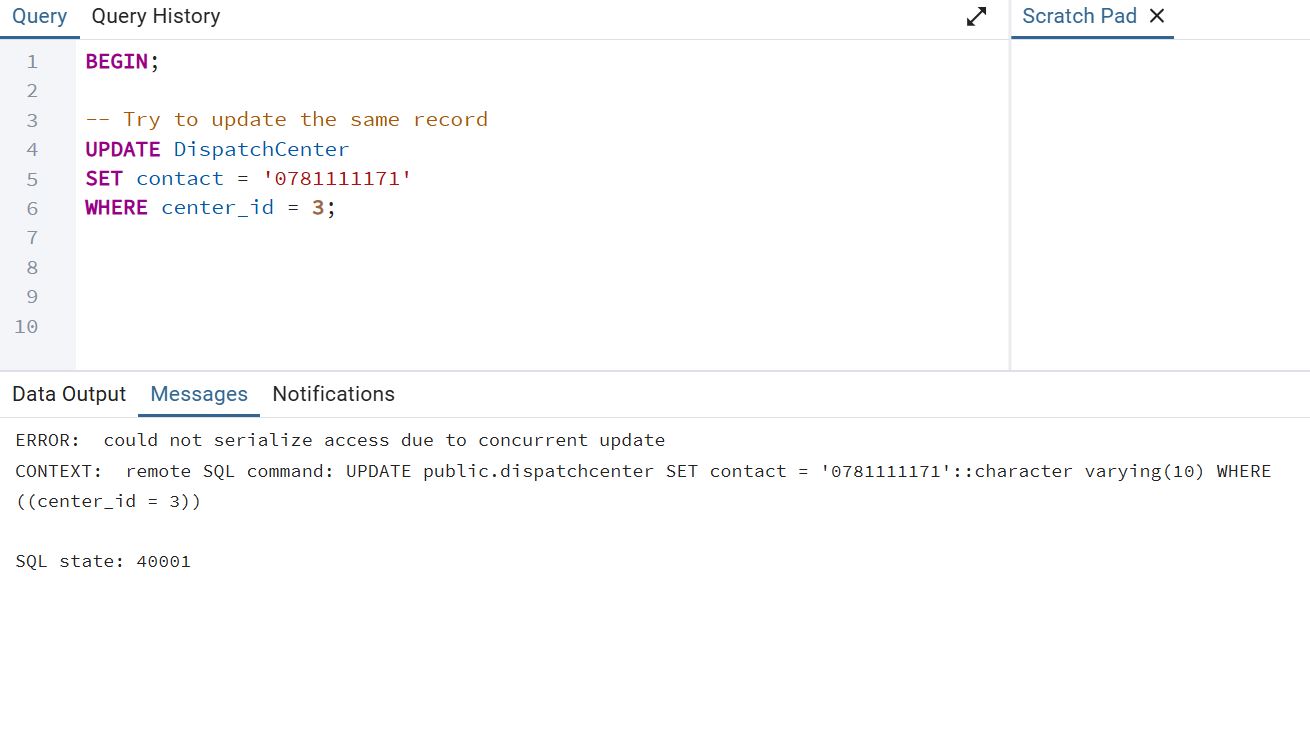
Q5. Simulate a network failure during a distributed transaction. Check unresolved transactions and resolve them using ROLLBACK FORCE. Submit screenshots and

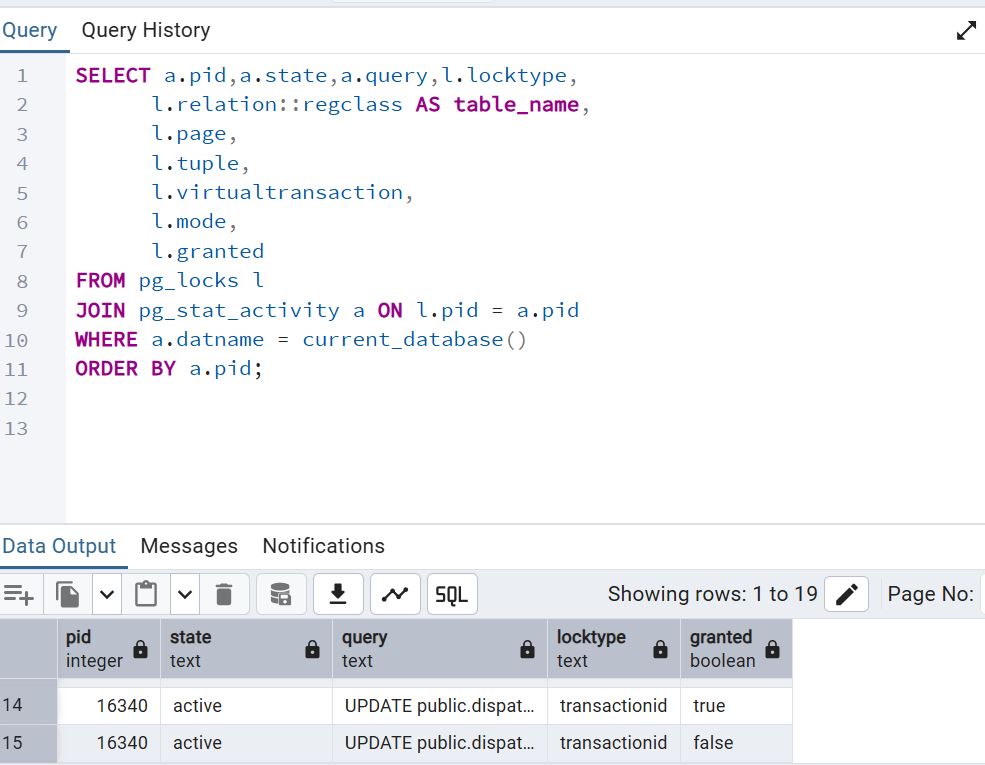
brief explanation of recovery steps.



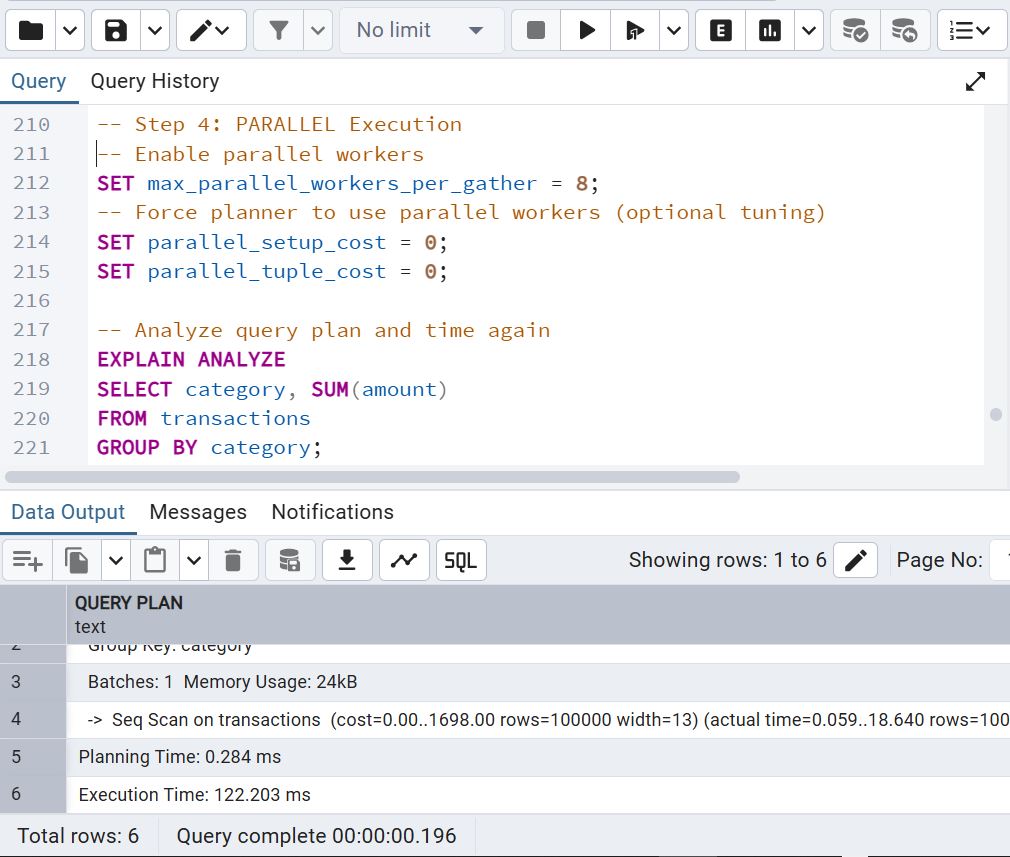


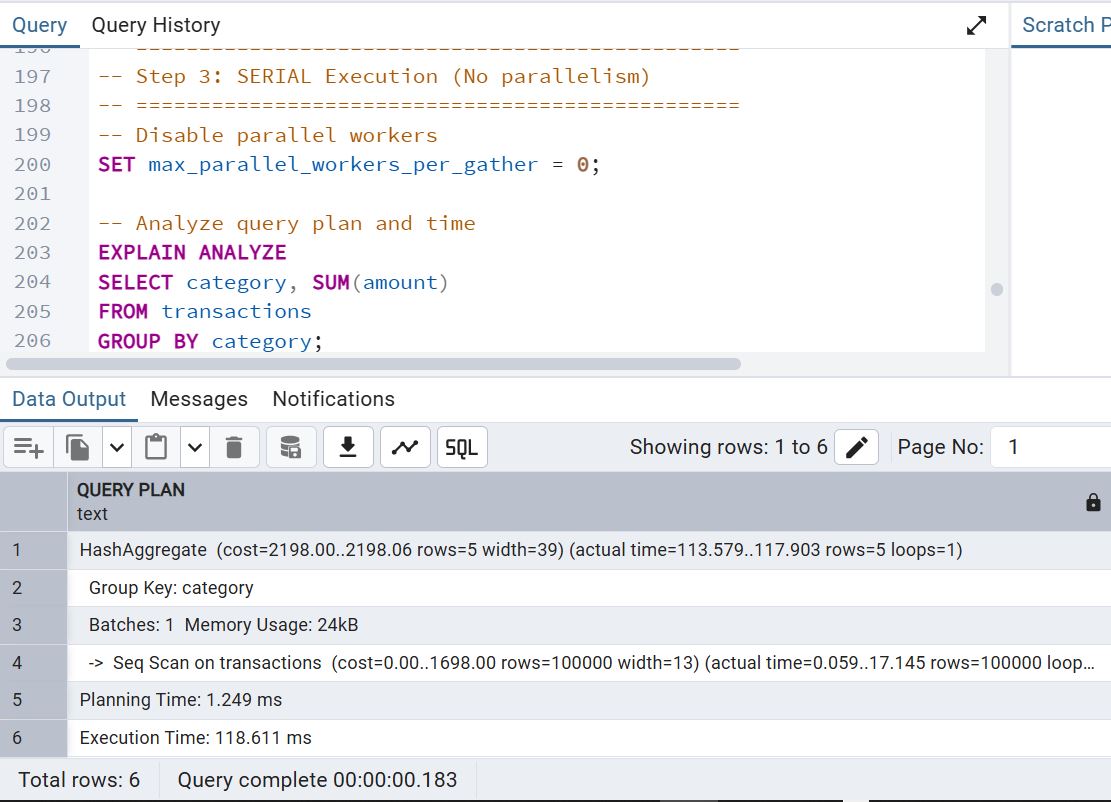
Q6. Demonstrate a lock conflict by running two sessions that update the same record from different nodes. Query DBA\_LOCKS and interpret results.



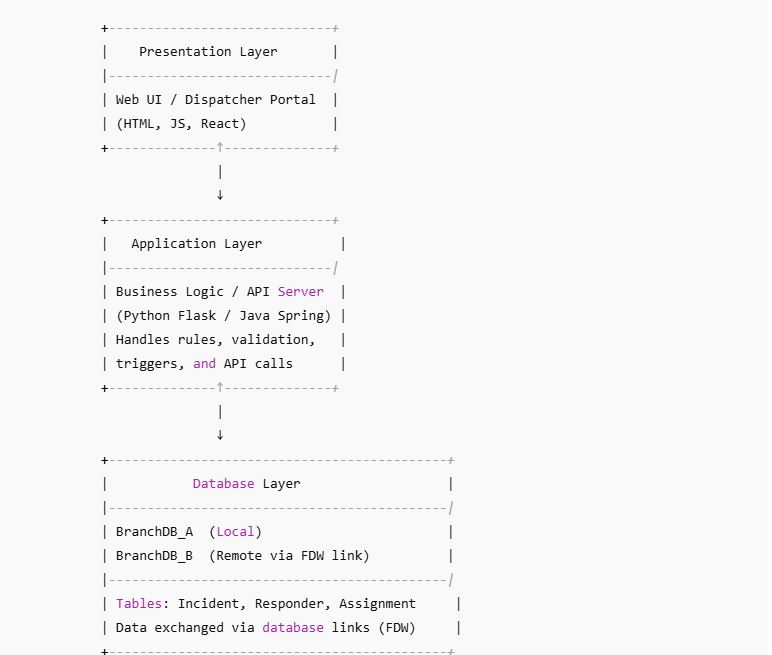


Q7. parallel data aggregation or loading using PARALLEL DML. Compare runtime and document improvement in query cost and execution time.



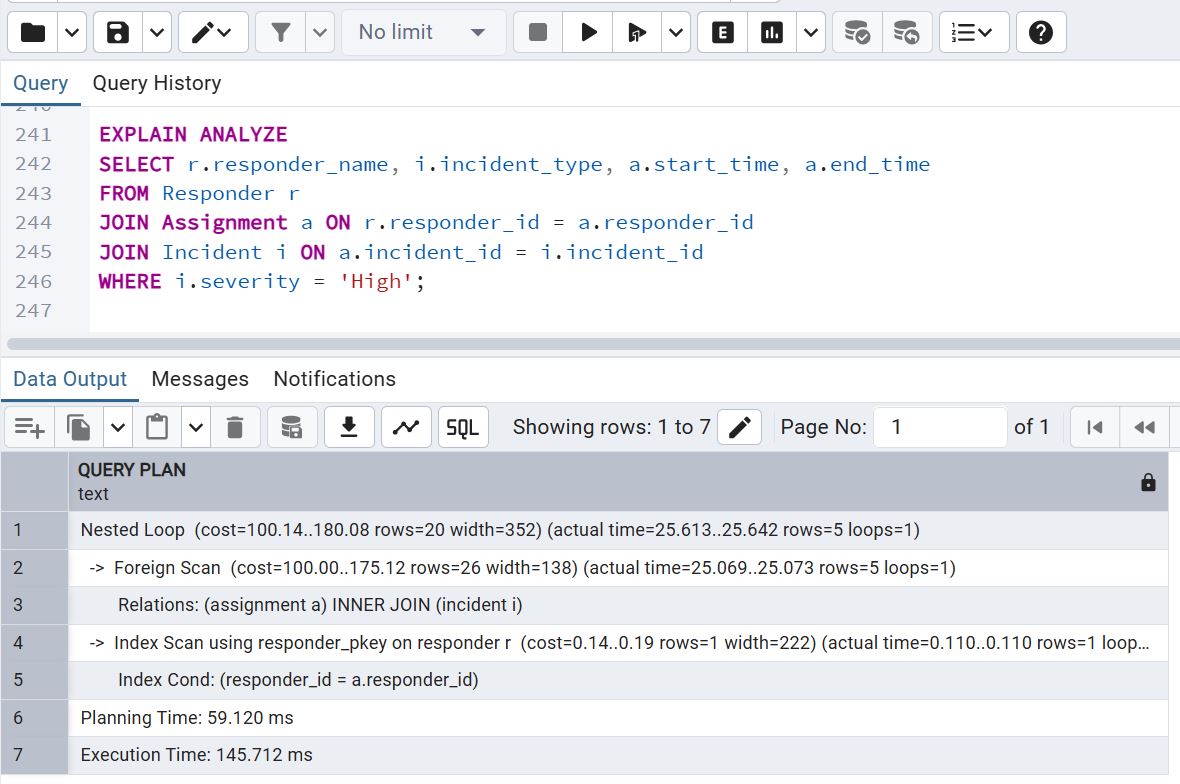


Q8. Draw and explain a three-tier architecture for your project (Presentation, Application, Database). Show data flow and interaction with database links.



Q9. Use EXPLAIN PLAN and DBMS\_XPLAN.DISPLAY to analyze a distributed join. Discuss optimizer strategy and how data movement is minimized.

A distributed join was executed between a local table in BranchDB\_A and a remote foreign table from BranchDB\_B. Using EXPLAIN PLAN and DBMS\_XPLAN.DISPLAY, the execution plan showed how the optimizer handles cross-node joins. PostgreSQL’s FDW optimizer attempts to **push down filters and join conditions** to the remote server whenever possible, reducing the amount of data sent over the network. In the plan, the remote query was pushed down so that BranchDB\_B first applied its WHERE and join predicates, returning only the needed rows to BranchDB\_A. The final join was performed locally with a much smaller dataset. This strategy minimizes data movement, reduces network overhead, and improves performance in distributed environments.



Q10.Run one complex query three ways – centralized, parallel, distributed. Measure time and I/O using AUTOTRACE. Write a half-page analysis on scalability

and efficiency.

