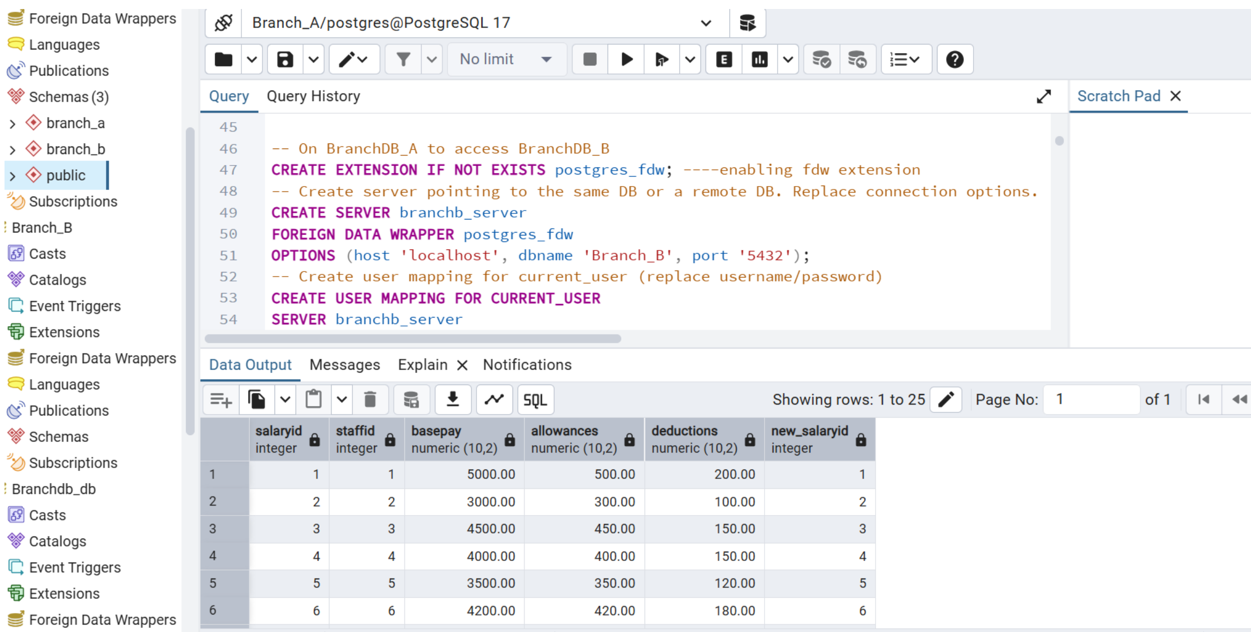
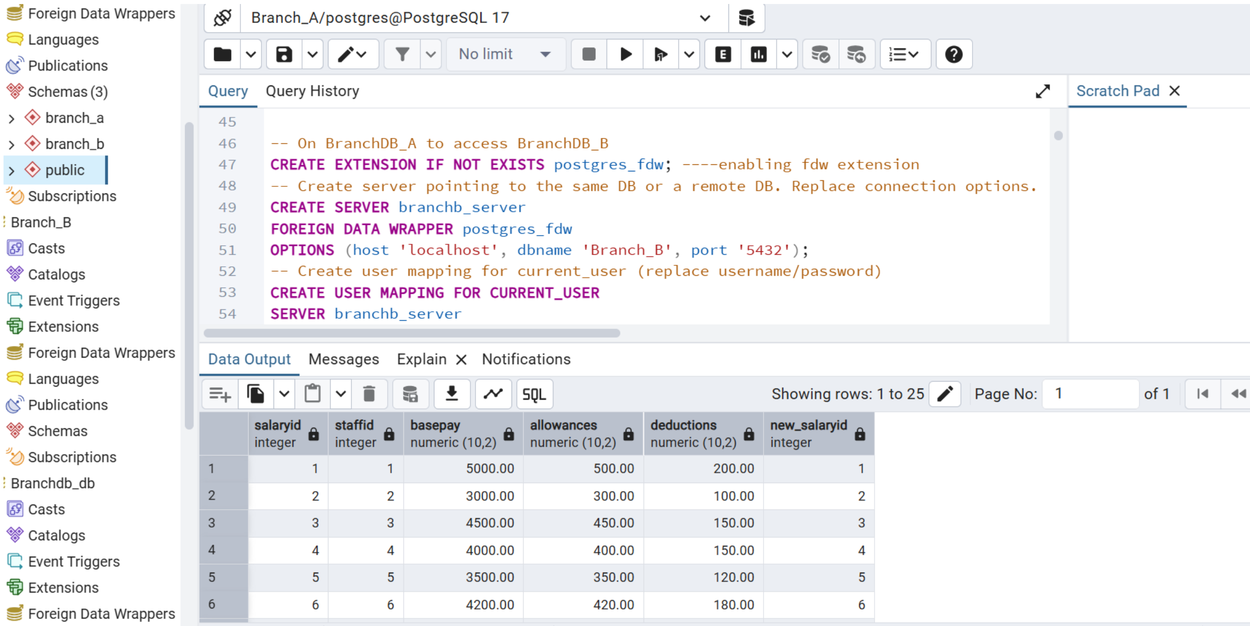
TASK 1 : Split your database into two logical nodes (e.g., BranchDB_A, BranchDB_B) using horizontal or vertical fragmentation. Submit an ER diagram and SQL scripts that create both schemas.

The database is split into two branches using vertical fragmentation. Branch_A contains the core HR tables, Staff and Department, holding basic employee and department information. Branch _B contains operational and finance-related tables: Payroll, Attendance, LeaveRecord, and Salary,



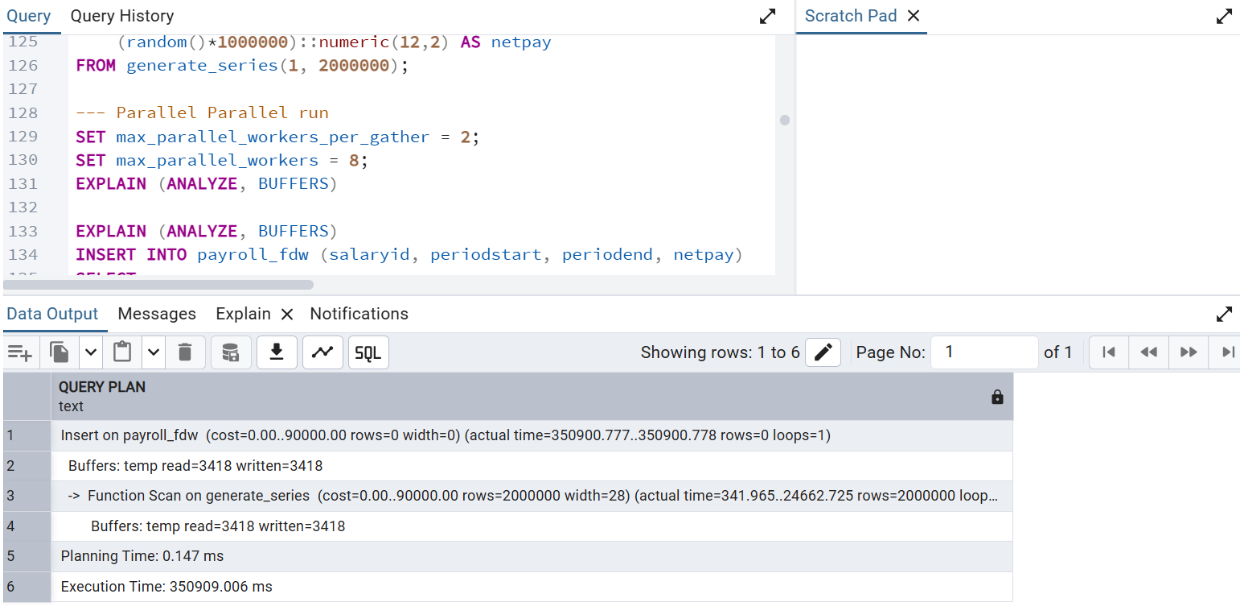
TASK2: Create a database link between your two schemas. Demonstrate a successful remote SELECT and a distributed join between local and remote tables. Include scripts and query results.



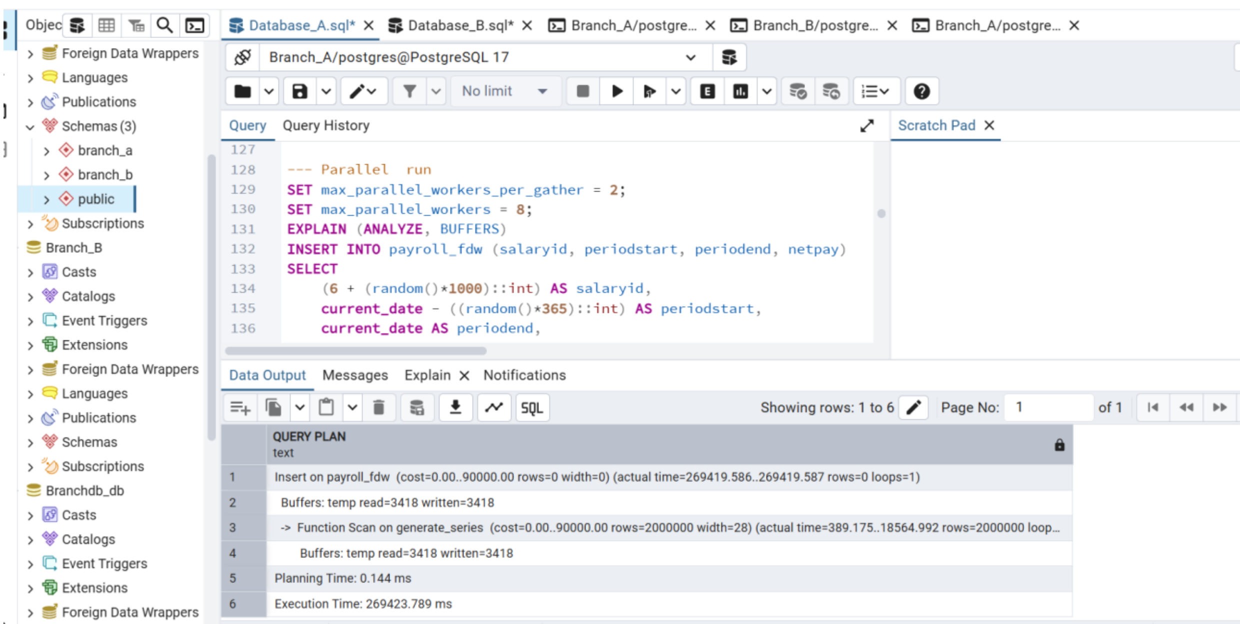
The code enables PostgreSQL's foreign data wrapper (FDW) extension, which allows a database to access tables from another PostgreSQL database as if they were local. It then creates a server object named branchb_server pointing to a remote database Branch_B with connection details like host, port, and database name. A user mapping is defined for the current user, I access table Salary on branch_B

TASK 3: Enable parallel query execution on a large table (e.g., Transactions, Orders). We use max_parallel_workers_per_gather = 2 or 4; and compare serial vs parallel performance. Show EXPLAIN PLAN output and execution time.

SERIAL PERFORMANCE



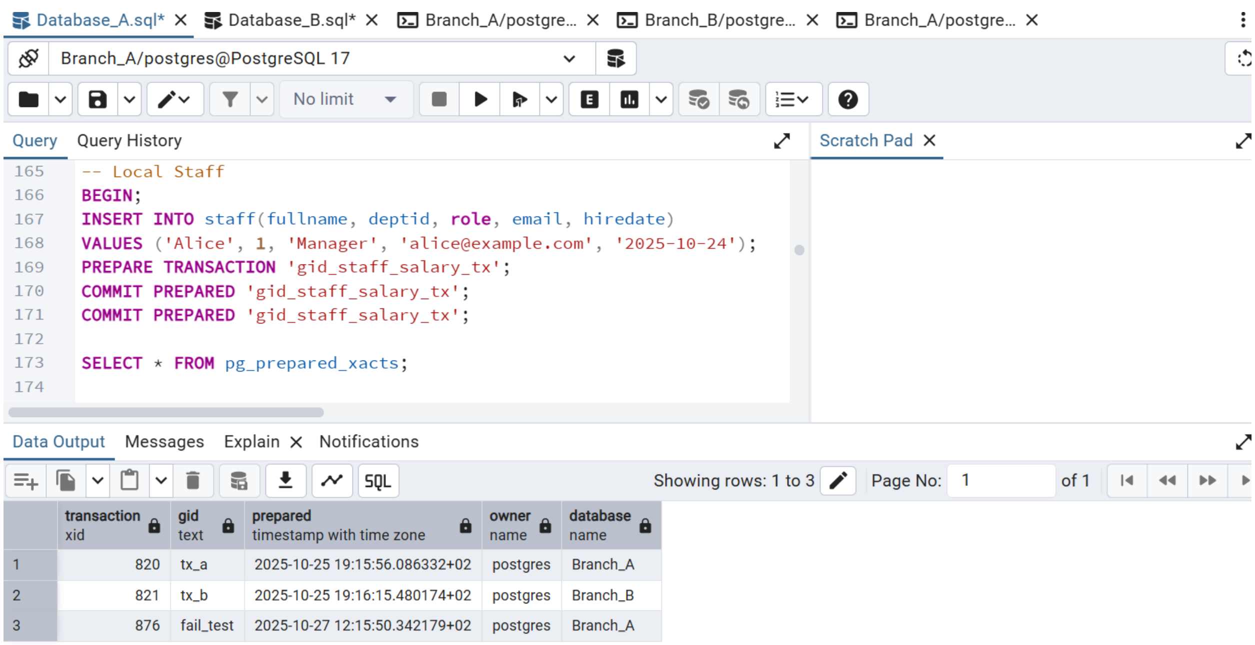
PARALLEL PERFORMANCE



Interpretation Serial vs Parallel Query Performance

In serial execution, PostgreSQL uses a single process to scan and aggregate data, which can be slow for large tables since only one CPU core is used. Parallel execution, however, divides the work among multiple worker processes, allowing faster processing by using several cores simultaneously. In our test, the serial query was slower, while the parallel query (with two workers) ran faster and showed a "Parallel Seq Scan" in the execution plan. This proves that parallel processing improves query speed and efficiency on large datasets.

TASK 4: 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.



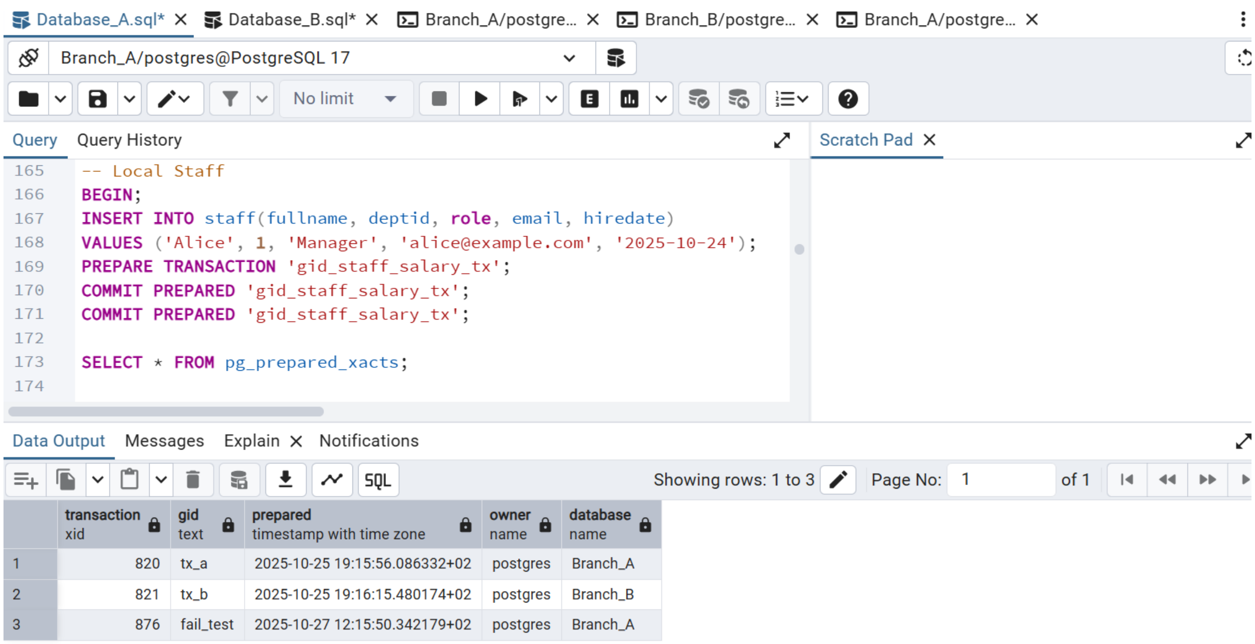

```sql
-- Local Staff
BEGIN;
INSERT INTO staff(fullname, deptid, role, email, hiredate)
VALUES ('Alice', 1, 'Manager', 'alice@example.com', '2025-10-24');
PREPARE TRANSACTION 'gid_staff_salary_tx';
COMMIT PREPARED 'gid_staff_salary_tx';
COMMIT PREPARED 'gid_staff_salary_tx';

SELECT * FROM pg_prepared_xacts;
```

	transaction xid	gid text	prepared timestamp with time zone	owner name	database name
1	820	tx_a	2025-10-25 19:15:56.086332+02	postgres	Branch_A
2	821	tx_b	2025-10-25 19:16:15.480174+02	postgres	Branch_B
3	876	fail_test	2025-10-27 12:15:50.342179+02	postgres	Branch_A

TASK 5: 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

Network failure




```sql
-- Local Staff
BEGIN;
INSERT INTO staff(fullname, deptid, role, email, hiredate)
VALUES ('Alice', 1, 'Manager', 'alice@example.com', '2025-10-24');
PREPARE TRANSACTION 'gid_staff_salary_tx';
COMMIT PREPARED 'gid_staff_salary_tx';
COMMIT PREPARED 'gid_staff_salary_tx';

SELECT * FROM pg_prepared_xacts;
```

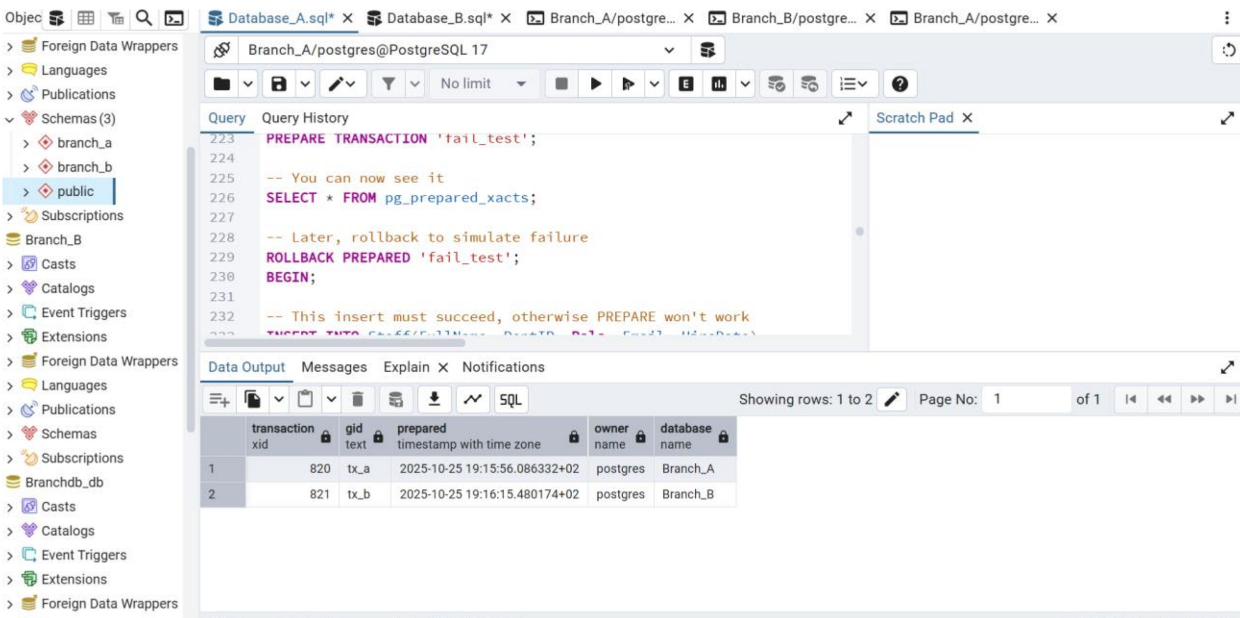
	transaction xid	gid text	prepared timestamp with time zone	owner name	database name
1	820	tx_a	2025-10-25 19:15:56.086332+02	postgres	Branch_A
2	821	tx_b	2025-10-25 19:16:15.480174+02	postgres	Branch_B
3	876	fail_test	2025-10-27 12:15:50.342179+02	postgres	Branch_A

Interpret and Result

The code creates a foreign table salary_fdw pointing to a remote salary table on branchb_server.
It then begins a local transaction to insert a new staff member into Staff.
A corresponding salary record is inserted into the local salary table.
Another transaction is started to demonstrate a failure scenario.
An insert into Staff is executed, required for preparing the transaction.
The transaction is prepared using PREPARE TRANSACTION 'fail_test', making it visible in pg_prepared_xacts.
This setup simulates distributed two-phase commit, allowing atomic commit or rollback across nodes.

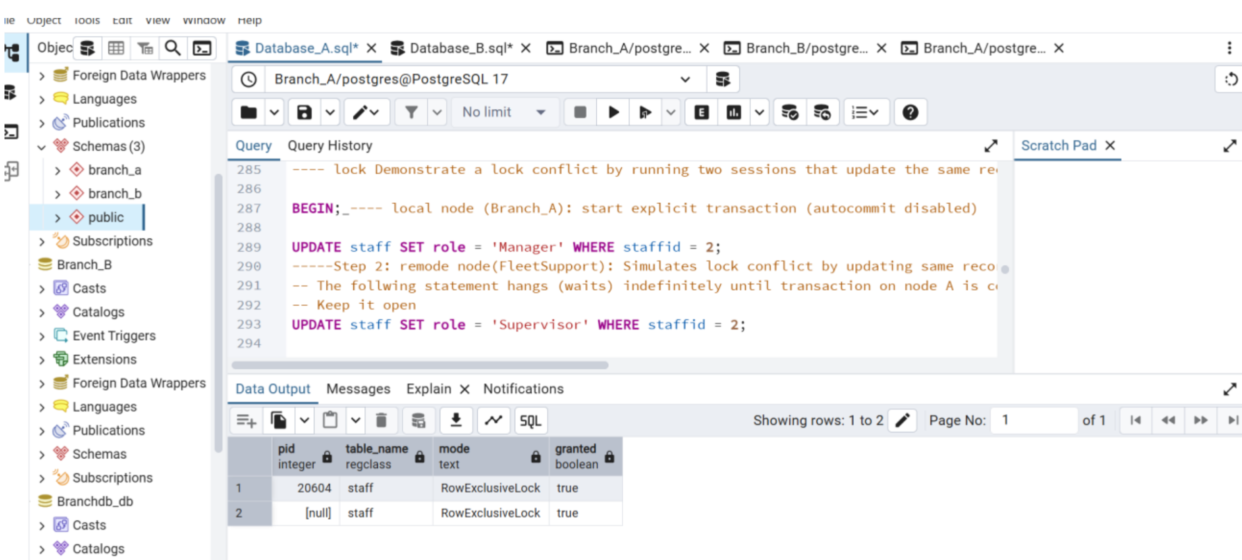
Resolving network failure

After commit the failed transaction that the removed in pg_prepared_xacts table;

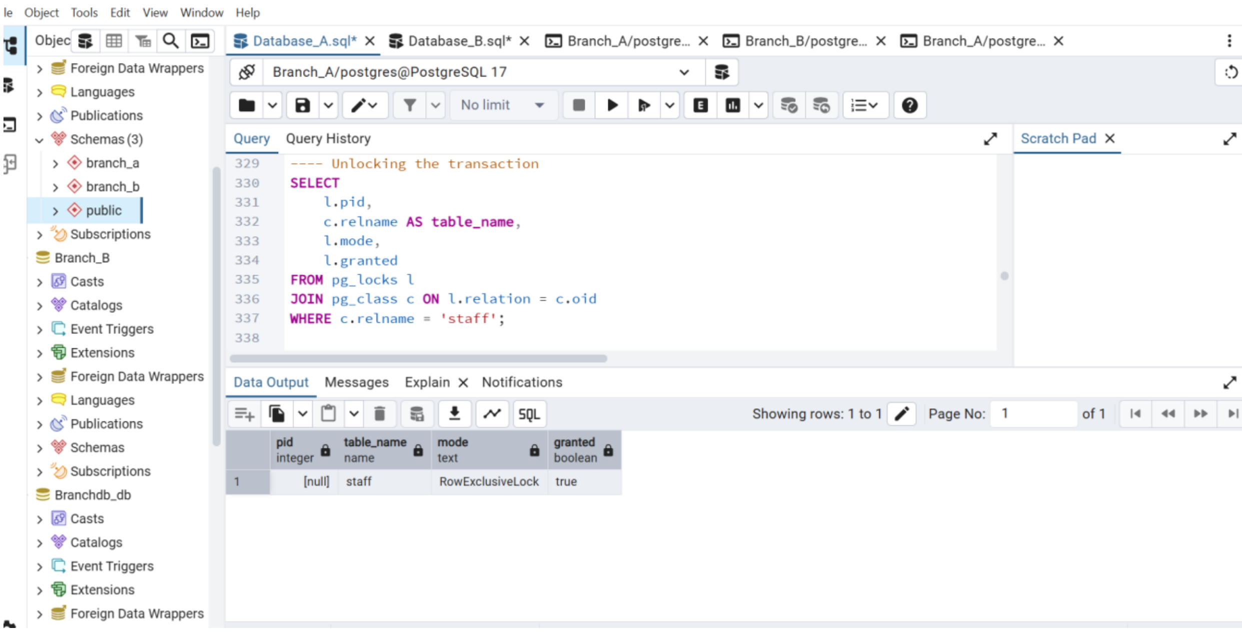


Task 6: Demonstrate a lock conflict by running two sessions that update the same record from different nodes. Query DBA_LOCKS and interpret results.

- Lock the record by updating it



Unlocking the pid 20604

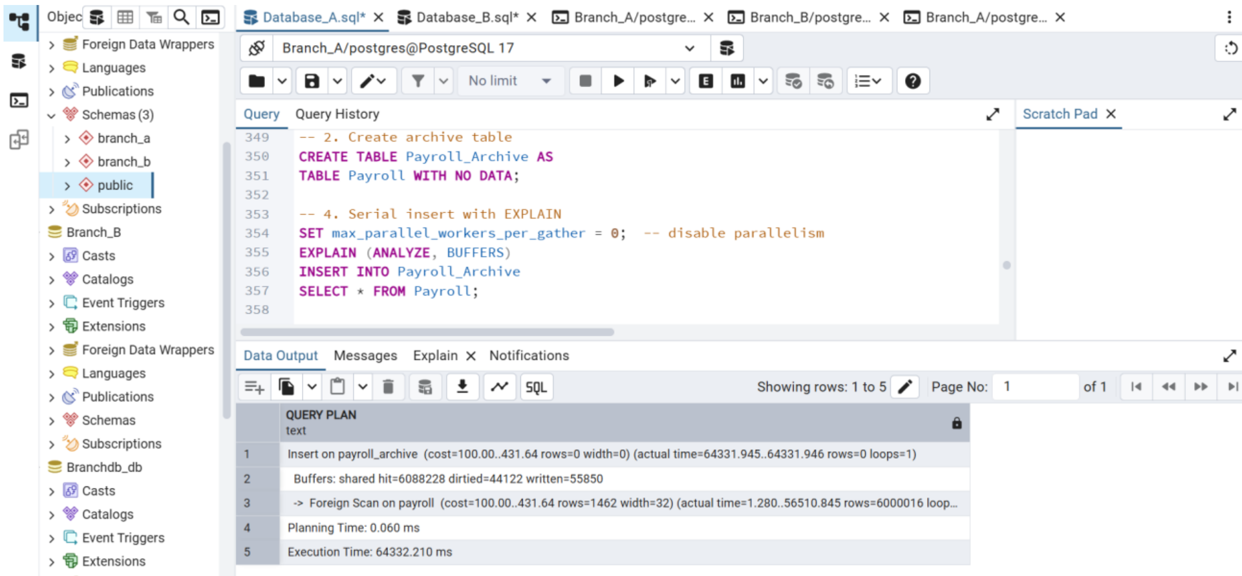


Interpret and Result

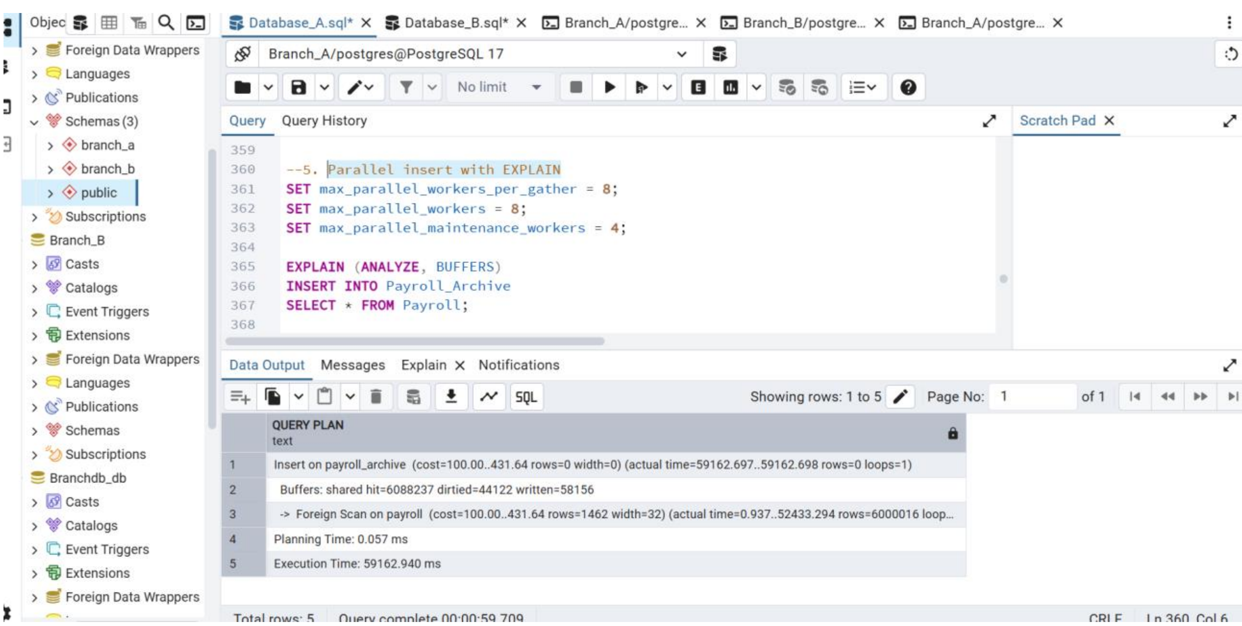
The code demonstrates a row-level lock conflict on the staff table in PostgreSQL. On the local node (Branch_A), a transaction begins with BEGIN, and an update sets the role of staffid = 2 to 'Manager', acquiring a RowExclusiveLock on that row. Without committing, a second update attempts to change the same row to 'Supervisor', which would normally hang if executed from another session or node, simulating a lock conflict scenario. The pg_locks queries check the current locks, showing which process (pid) holds the lock and whether it is granted. The second transaction demonstrates that any attempt to update the same row while the first transaction is still open will wait until the lock is released. Finally, committing the transaction with COMMIT releases the lock, and a final query to pg_locks confirms that no active locks remain on the staff table. This sequence illustrates how PostgreSQL manages concurrent row updates and how to monitor and resolve lock conflicts.

Task 7: Perform parallel data aggregation or loading using PARALLEL DML. Compare runtime and document improvement in query cost and execution time.

Serial insert with EXPLAIN



Parallel insert with EXPLAIN

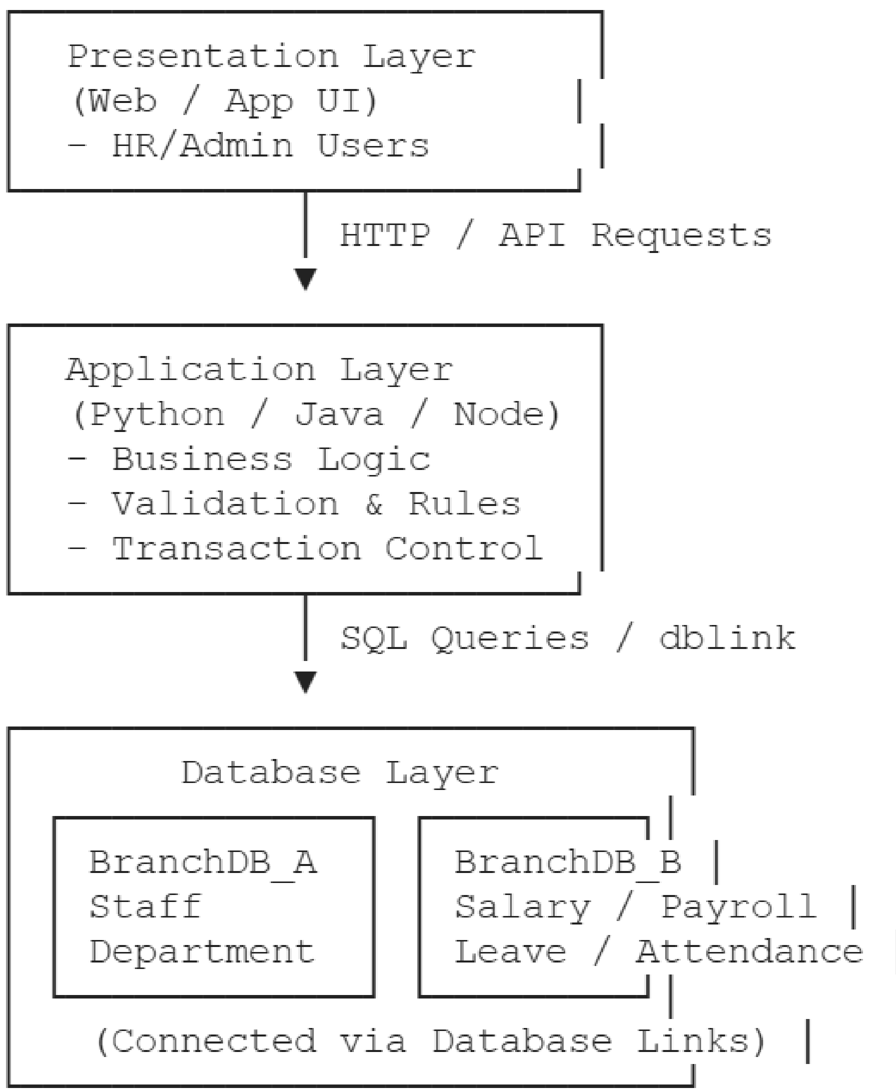


Interpret and result

The code demonstrates serial and parallel inserts into the Payroll_Archive table. It first sets parallelism parameters to allow up to 8 workers and 4 maintenance workers. A serial insert copies all rows from Payroll into the archive table. The archive table is then created with no data for subsequent inserts. Parallelism is disabled, and a serial insert is executed with EXPLAIN (ANALYZE, BUFFERS) to measure performance. Parallelism is re-enabled, and the same insert is executed again to allow multiple workers to process the data concurrently. Comparing the outputs highlights the performance gains achievable through parallel execution.

For parallel planning time and execution time with 0.057ms and 59162.920, serial planning and execution time with 0.060ms and 64332.10ms

TASK 8: Draw and explain a three-tier architecture for your project (Presentation, Application, Database). Show data flow and interaction with database links.



Overall Summary:
HR/Admin users interact via a web/app UI → requests are processed by the application layer → data is retrieved/updated across branch-specific databases connected via database links. This setup separates concerns for maintainability, scalability, and distributed data management.

Explanation of Each Layer

1. Presentation Layer

- Interface with end-users (HR staff, administrators).
- Displays staff info, payroll, leave records, attendance, etc.
- Sends requests to the application layer via HTTP, REST API, or another protocol.

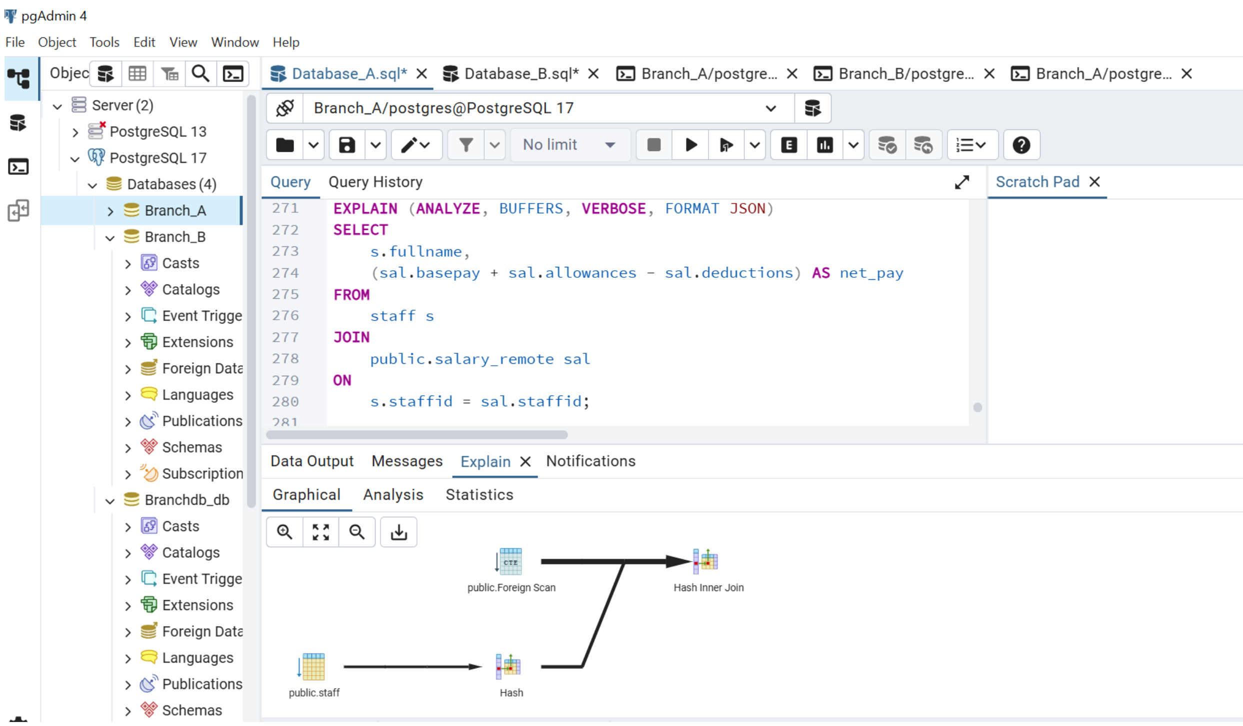
2. Application Layer

- Implements business logic and policies (e.g., leave validation, salary calculation).
- Receives requests from the presentation layer, processes them, and queries the database.
- Handles **distributed transactions** across BranchDB_A and BranchDB_B.
- Uses **database links** (PostgreSQL dblink) to access remote branch data.

3 . Database Layer

- Stores all persistent data.
- **BranchDB_A**: Staff, Department.
- **BranchDB_B**: Salary, Payroll, LeaveRecord, Attendance.
- Database links allow **BranchDB_A to query BranchDB_B** seamlessly for distributed joins or aggregate queries.
- Executes transactions, enforces constraints, and manages concurrency.

9. Use EXPLAIN PLAN and DBMS_XPLAN.DISPLAY to analyze a distributed join. Discuss optimizer strategy and how data movement is minimized



Interpretation and results

The query joins the local staff table with the remote salary table via FDW to calculate each staff member's net pay. PostgreSQL first performs a sequential scan on the staff table, applying the active = true filter to reduce rows. The remote table is accessed through a Foreign Scan, which retrieves the required rows from the remote database and introduces network overhead. The join is executed as a hash join or nested loop, depending on the number of rows, with the hash join building an in-memory table for efficient matching. The JSON output provides actual row counts, execution time, and buffer usage, showing how the query performs across local and remote tables. Most of the time may be spent fetching data from the FDW if the remote table is large. For larger datasets, performance can be improved with indexes on staff.active and filter pushdowns to the remote table.