

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,

The screenshot shows a PostgreSQL IDE interface. On the left, a sidebar lists database objects including Schemas (3), Subscriptions, and Foreign Data Wrappers. The main window displays a SQL query script for creating a foreign data wrapper and a table. The query is as follows:

```

45
46 -- On BranchDB_A to access BranchDB_B
47 CREATE EXTENSION IF NOT EXISTS postgres_fdw; ----enabling fdw extension
48 -- Create server pointing to the same DB or a remote DB. Replace connection options.
49 CREATE SERVER branchb_server
50 FOREIGN DATA WRAPPER postgres_fdw
51 OPTIONS (host 'localhost', dbname 'Branch_B', port '5432');
52 -- Create user mapping for current_user (replace username/password)
53 CREATE USER MAPPING FOR CURRENT_USER
54 SERVER branchb_server

```

Below the query editor, the 'Data Output' tab is active, showing a table with 6 rows and 7 columns. The columns are: salaryid (integer), staffid (integer), basepay (numeric (10,2)), allowances (numeric (10,2)), deductions (numeric (10,2)), and new_salaryid (integer). The data is as follows:

	salaryid integer	staffid integer	basepay numeric (10,2)	allowances numeric (10,2)	deductions numeric (10,2)	new_salaryid integer
1	1	1	5000.00	500.00	200.00	1
2	2	2	3000.00	300.00	100.00	2
3	3	3	4500.00	450.00	150.00	3
4	4	4	4000.00	400.00	150.00	4
5	5	5	3500.00	350.00	120.00	5
6	6	6	4200.00	420.00	180.00	6

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 screenshot shows a PostgreSQL IDE interface. The left sidebar displays a tree view of database objects, including Foreign Data Wrappers, Languages, Publications, Schemas (3), Subscriptions, Branch_B, Casts, Catalogs, Event Triggers, Extensions, and Foreign Data Wrappers. The main window is titled 'Branch_A/postgres@PostgreSQL 17' and contains a query editor with the following SQL code:

```

45
46 -- On BranchDB_A to access BranchDB_B
47 CREATE EXTENSION IF NOT EXISTS postgres_fdw; ----enabling fdw extension
48 -- Create server pointing to the same DB or a remote DB. Replace connection options.
49 CREATE SERVER branchb_server
50 FOREIGN DATA WRAPPER postgres_fdw
51 OPTIONS (host 'localhost', dbname 'Branch_B', port '5432');
52 -- Create user mapping for current_user (replace username/password)
53 CREATE USER MAPPING FOR CURRENT_USER
54 SERVER branchb_server

```

Below the query editor, the 'Data Output' tab is active, displaying a table with 6 rows and 7 columns. The table shows salary data for staff members, including salaryid, staffid, basepay, allowances, deductions, and new_salaryid.

	salaryid integer	staffid integer	basepay numeric (10,2)	allowances numeric (10,2)	deductions numeric (10,2)	new_salaryid integer
1	1	1	5000.00	500.00	200.00	1
2	2	2	3000.00	300.00	100.00	2
3	3	3	4500.00	450.00	150.00	3
4	4	4	4000.00	400.00	150.00	4
5	5	5	3500.00	350.00	120.00	5
6	6	6	4200.00	420.00	180.00	6

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

Query Query History Scratch Pad X

```

125 (random()*1000000)::numeric(12,2) AS netpay
126 FROM generate_series(1, 2000000);
127
128 --- Parallel Parallel run
129 SET max_parallel_workers_per_gather = 2;
130 SET max_parallel_workers = 8;
131 EXPLAIN (ANALYZE, BUFFERS)
132
133 EXPLAIN (ANALYZE, BUFFERS)
134 INSERT INTO payroll_fdw (salaryid, periodstart, periodend, netpay)
135 SELECT

```

Data Output Messages Explain X Notifications

Showing rows: 1 to 6 Page No: 1 of 1

QUERY PLAN	
text	
1	Insert on payroll_fdw (cost=0.00..90000.00 rows=0 width=0) (actual time=350900.777..350900.778 rows=0 loops=1)
2	Buffers: temp read=3418 written=3418
3	-> Function Scan on generate_series (cost=0.00..90000.00 rows=2000000 width=28) (actual time=341.965..24662.725 rows=2000000 loop...
4	Buffers: temp read=3418 written=3418
5	Planning Time: 0.147 ms
6	Execution Time: 350909.006 ms

PARALLEL PERFORMANCE

Objec Database_A.sql* Database_B.sql* Branch_A/postgre... Branch_B/postgre... Branch_A/postgre...

Branch_A/postgres@PostgreSQL 17

Query Query History Scratch Pad X

```

127
128 --- Parallel run
129 SET max_parallel_workers_per_gather = 2;
130 SET max_parallel_workers = 8;
131 EXPLAIN (ANALYZE, BUFFERS)
132 INSERT INTO payroll_fdw (salaryid, periodstart, periodend, netpay)
133 SELECT
134 (6 + (random()*1000)::int) AS salaryid,
135 current_date - ((random()*365)::int) AS periodstart,
136 current_date AS periodend,

```

Data Output Messages Explain X Notifications

Showing rows: 1 to 6 Page No: 1 of 1

QUERY PLAN	
text	
1	Insert on payroll_fdw (cost=0.00..90000.00 rows=0 width=0) (actual time=269419.586..269419.587 rows=0 loops=1)
2	Buffers: temp read=3418 written=3418
3	-> Function Scan on generate_series (cost=0.00..90000.00 rows=2000000 width=28) (actual time=389.175..18564.992 rows=2000000 loop...
4	Buffers: temp read=3418 written=3418
5	Planning Time: 0.144 ms
6	Execution Time: 269423.789 ms

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.

The screenshot shows a database IDE with a query editor and a data output window. The query editor contains a PL/SQL block that inserts a staff member and then commits the transaction. The data output window shows the results of the query, which is a table with 3 rows and 6 columns: transaction_xid, gid_text, prepared_timestamp_with_time_zone, owner_name, and database_name.

```

165 -- Local Staff
166 BEGIN;
167 INSERT INTO staff(fullname, deptid, role, email, hiredate)
168 VALUES ('Alice', 1, 'Manager', 'alice@example.com', '2025-10-24');
169 PREPARE TRANSACTION 'gid_staff_salary_tx';
170 COMMIT PREPARED 'gid_staff_salary_tx';
171 COMMIT PREPARED 'gid_staff_salary_tx';
172
173 SELECT * FROM pg_prepared_xacts;
174

```

transaction_xid	gid_text	prepared_timestamp_with_time_zone	owner_name	database_name
820	tx_a	2025-10-25 19:15:56.086332+02	postgres	Branch_A
821	tx_b	2025-10-25 19:16:15.480174+02	postgres	Branch_B
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

The screenshot shows a PostgreSQL IDE interface. The top bar displays several open tabs: Database_A.sql*, Database_B.sql*, Branch_A/postgre..., Branch_B/postgre..., and Branch_A/postgre... The main window is titled 'Branch_A/postgres@PostgreSQL 17'. Below the title bar is a toolbar with various icons for file operations, query execution, and settings. The 'Query' tab is active, showing a SQL script with the following content:

```

165 -- Local Staff
166 BEGIN;
167 INSERT INTO staff(fullname, deptid, role, email, hiredate)
168 VALUES ('Alice', 1, 'Manager', 'alice@example.com', '2025-10-24');
169 PREPARE TRANSACTION 'gid_staff_salary_tx';
170 COMMIT PREPARED 'gid_staff_salary_tx';
171 COMMIT PREPARED 'gid_staff_salary_tx';
172
173 SELECT * FROM pg_prepared_xacts;
174

```

Below the query editor is the 'Data Output' tab, which displays the results of the query. The output is a table with 6 columns: transaction xid, gid, prepared timestamp with time zone, owner name, and database name. The table contains 3 rows of data:

transaction xid	gid	prepared timestamp with time zone	owner name	database name
820	tx_a	2025-10-25 19:15:56.086332+02	postgres	Branch_A
821	tx_b	2025-10-25 19:16:15.480174+02	postgres	Branch_B
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;

The screenshot shows a PostgreSQL IDE interface. The left sidebar displays a tree view of the database structure, including schemas (branch_a, branch_b, public), subscriptions, and catalogs. The main query window is titled 'Branch_A/postgres@PostgreSQL 17' and contains the following SQL script:

```

223 PREPARE TRANSACTION 'fail_test';
224
225 -- You can now see it
226 SELECT * FROM pg_prepared_xacts;
227
228 -- Later, rollback to simulate failure
229 ROLLBACK PREPARED 'fail_test';
230 BEGIN;
231
232 -- This insert must succeed, otherwise PREPARE won't work
233 INSERT INTO public.branch_a (branch_a_id, branch_a_name) VALUES (1, 'Branch A');
  
```

The 'Data Output' pane at the bottom shows the results of the 'SELECT * FROM pg_prepared_xacts;' query:

transaction_xid	gid	text	prepared_timestamp_with_timezone	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

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

The screenshot shows a PostgreSQL IDE interface. The left sidebar displays a tree view of the database structure. The main query window is titled 'Branch_A/postgres@PostgreSQL 17' and contains the following SQL script:

```

285 ----- lock Demonstrate a lock conflict by running two sessions that update the same record
286
287 BEGIN;----- local node (Branch_A): start explicit transaction (autocommit disabled)
288
289 UPDATE staff SET role = 'Manager' WHERE staffid = 2;
290 -----Step 2: remote node(FleetSupport): Simulates lock conflict by updating same record
291 -- The following statement hangs (waits) indefinitely until transaction on node A is committed
292 -- Keep it open
293 UPDATE staff SET role = 'Supervisor' WHERE staffid = 2;
294
  
```

The 'Data Output' pane at the bottom shows the results of the 'SELECT * FROM pg_locks;' query:

pid	table_name	mode	granted
1	20604 staff	RowExclusiveLock	true
2	[null] staff	RowExclusiveLock	true

Unlocking the pid 20604

The screenshot shows a PostgreSQL IDE interface. The left sidebar displays a tree view of the database structure, including Schemas (3), Subscriptions, Branch_B, Casts, Catalogs, Event Triggers, Extensions, and Foreign Data Wrappers. The main query window shows a SQL query with line numbers 329 to 338. The query is as follows:

```

329 ---- Unlocking the transaction
330 SELECT
331     l.pid,
332     c.relname AS table_name,
333     l.mode,
334     l.granted
335 FROM pg_locks l
336 JOIN pg_class c ON l.relation = c.oid
337 WHERE c.relname = 'staff';
338

```

The Data Output window at the bottom shows the results of the query. It displays a table with the following columns: pid (integer), table_name (name), mode (text), and granted (boolean). The results show one row with pid 1, table_name [null], mode RowExclusiveLock, and granted true.

pid	table_name	mode	granted
1	[null]	RowExclusiveLock	true

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

The screenshot shows the PostgreSQL IDE interface. The left sidebar displays the database schema, including schemas (branch_a, branch_b, public), subscriptions, and databases (Branch_B, Branchdb_db). The main query editor shows the following SQL code:

```

349 -- 2. Create archive table
350 CREATE TABLE Payroll_Archive AS
351 TABLE Payroll WITH NO DATA;
352
353 -- 4. Serial insert with EXPLAIN
354 SET max_parallel_workers_per_gather = 0; -- disable parallelism
355 EXPLAIN (ANALYZE, BUFFERS)
356 INSERT INTO Payroll_Archive
357 SELECT * FROM Payroll;
358

```

The 'Data Output' tab shows the query plan for the serial insert:

Step	Operation	Cost	Time	Rows	Width	Loops
1	Insert on payroll_archive	(cost=100.00..431.64 rows=0 width=0)	(actual time=64331.945..64331.946 rows=0 loops=1)			
2	Buffers: shared hit=6088228 dirtied=44122 written=55850					
3	-> Foreign Scan on payroll	(cost=100.00..431.64 rows=1462 width=32)	(actual time=1.280..56510.845 rows=6000016 loop=1)			
4	Planning Time	0.060 ms				
5	Execution Time	64332.210 ms				

Parallel insert with EXPLAIN

The screenshot shows the PostgreSQL IDE interface. The left sidebar displays the database schema. The main query editor shows the following SQL code:

```

359
360 --5. Parallel insert with EXPLAIN
361 SET max_parallel_workers_per_gather = 8;
362 SET max_parallel_workers = 8;
363 SET max_parallel_maintenance_workers = 4;
364
365 EXPLAIN (ANALYZE, BUFFERS)
366 INSERT INTO Payroll_Archive
367 SELECT * FROM Payroll;
368

```

The 'Data Output' tab shows the query plan for the parallel insert:

Step	Operation	Cost	Time	Rows	Width	Loops
1	Insert on payroll_archive	(cost=100.00..431.64 rows=0 width=0)	(actual time=59162.697..59162.698 rows=0 loops=1)			
2	Buffers: shared hit=6088237 dirtied=44122 written=58156					
3	-> Foreign Scan on payroll	(cost=100.00..431.64 rows=1462 width=32)	(actual time=0.937..52433.294 rows=6000016 loop=1)			
4	Planning Time	0.057 ms				
5	Execution Time	59162.940 ms				

Total rows: 5 Query complete 00:00:59.709

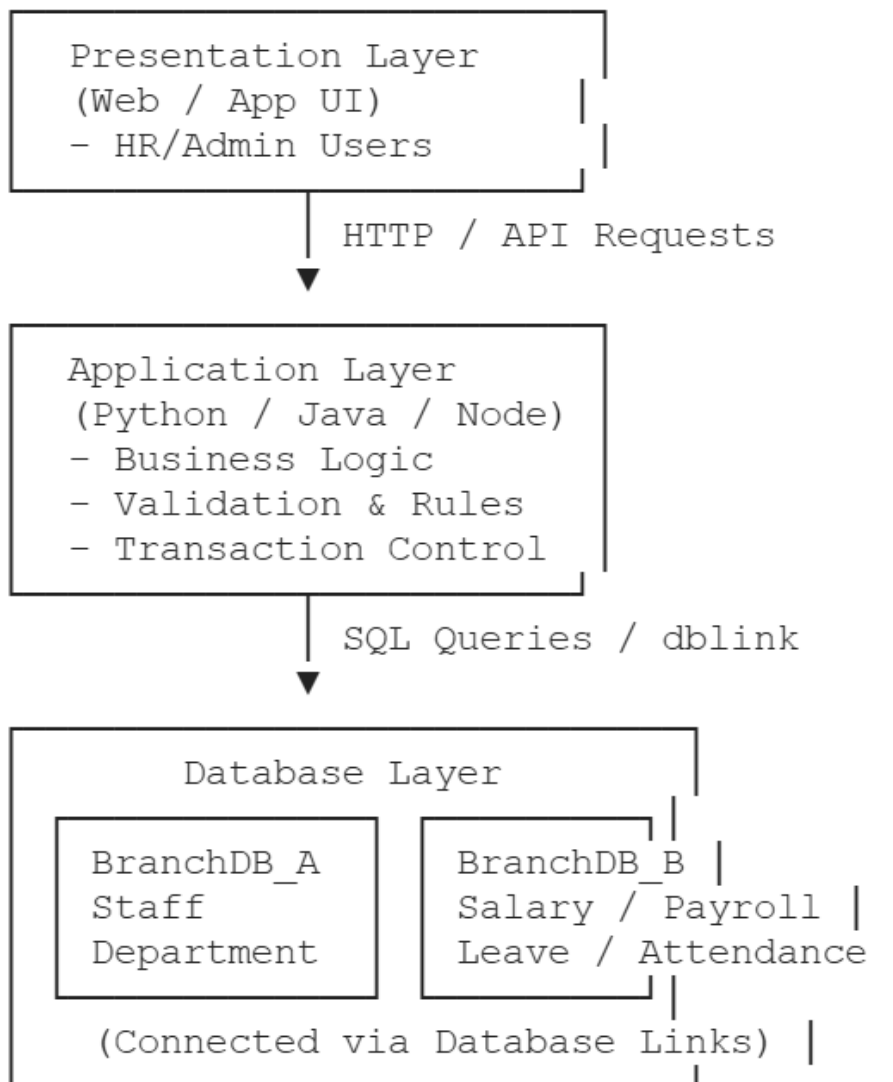
Interpret and result

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

The screenshot shows the pgAdmin 4 interface. On the left, the 'Server (2)' tree is expanded to show 'PostgreSQL 17' and 'Branch_A'. The 'Query' tab is active, displaying the following SQL query:

```
271 EXPLAIN (ANALYZE, BUFFERS, VERBOSE, FORMAT JSON)
272 SELECT
273     s.fullname,
274     (sal.basepay + sal.allowances - sal.deductions) AS net_pay
275 FROM
276     staff s
277 JOIN
278     public.salary_remote sal
279 ON
280     s.staffid = sal.staffid;
```

Below the query, the 'Explain' tab is selected, showing a graphical execution plan. The plan consists of three main components: 'public.staff', 'public.Foreign Scan', and 'Hash Inner Join'. The 'public.staff' table is scanned and its data is hashed. The 'public.Foreign Scan' retrieves data from the remote 'public.salary_remote' table. These two data streams are then joined using a 'Hash Inner Join' operation.

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.