Advanced Database Technology

Practical Lab Assessment — Parallel and Distributed Databases

Student: **Mukandayisenga Marie Chantal**
Reg No: **224019567**

Table of Contents

1 Executive summary

This report documents the design, implementation, execution and analysis of a distributed and parallel-capable Gym Membership and Attendance Tracking System. The work demonstrates schema fragmentation across two logical nodes (Branch A and Branch B), cross-node data access using postgres_fdw (simulating Oracle DB links), parallel query execution, two-phase commit, rollback and recovery, lock conflict reproduction, parallel ETL and distributed-query optimization. Results include EXPLAIN / EXPLAIN ANALYZE plans, prepared-transaction inspection, lock views, and performance comparisons across three execution modes.

2 Introduction and objectives

Purpose: satisfy the Practical Lab Assessment for the course by implementing ten targeted distributed / parallel tasks using an existing project schema (Gym).

Objectives:

- Fragment attendance data across logical nodes
- Demonstrate database-links-style remote access
- Measure serial vs parallel performance
- Simulate and verify two-phase commit and recovery
- Show distributed locks and resolve conflicts
- Perform parallel ETL and analyze performance gains
- Produce optimizer analysis and a three-way benchmark

3 Environment and assumptions

- RDBMS: PostgreSQL 12+ used for implementation and testing (scripts in Appendix A). If Oracle 19c is required, equivalent commands are noted where syntax differs; convert scripts as needed before execution in Oracle SQL Developer.
- Tools: psql or pgAdmin; use EXPLAIN (ANALYZE, BUFFERS) for timing and I/O (AUTOTRACE is Oracle-specific).
- Privileges: user with CREATE EXTENSION, CREATE SCHEMA, and FDW privileges.
- Topology: single PostgreSQL instance using two schemas (branch_a, branch_b) to represent BranchDB_A and BranchDB_B; postgres_fdw configured to simulate remote access. Replace FDW server host/dbname if using separate DB instances

4 System context and dataset (Gym case study)

Entities:

- Member (replicated): member_id, name, contact, membership_type, branch_id
- Attendance (fragmented): attendance_id, member_id, branch_id, checkin_time, checkout_time, activity_type
- Trainer, Session, Payment, Subscription

Sample data sizes used:

- Branch A: ~100 members (seeded)

- Branch B: 100 members, 1,000 attendance rows (seeded), scaled datasets for ETL up to 100k
- Raw ETL: 100k synthetic rows created for partitioning and parallel load tests

Rationale:

- Horizontal fragmentation by branch_id improves write locality; member table replicated to reduce remote join frequency.

5 Task 1- Distributed Schema Design and Fragmentation

The Gym Membership and Attendance Tracking System manages member registrations, attendance logs, trainer assignments, and session scheduling across multiple gym branches. To optimize performance and scalability, the database is distributed across two logical nodes: **BranchDB_A** and **BranchDB_B**.

Fragmentation Strategy: Horizontal Fragmentation

Horizontal fragmentation is chosen because each gym branch operates independently with its own members, trainers, and attendance records. By fragmenting tables based on BranchID, each node stores only the data relevant to its branch, improving query performance and enabling localized data management.

ER Diagram Description

The system includes the following entities:

- **Member(MemberID, Name, Contact, MembershipType, BranchID)**
- **Attendance(AttendanceID, MemberID, CheckInTime, CheckOutTime, Date)**
- **Trainer(TrainerID, Name, Specialty, BranchID)**
- **Session(SessionID, TrainerID, MemberID, Date, Time)**

Relationships:

- A Member can have multiple Attendance records.
- A Trainer can lead multiple Sessions.
- A Session links a Trainer and a Member.

Goal

- Horizontally fragment ATTENDANCE into Branch A and Branch B and create per-branch objects.

Actions performed (SQL overview)

- Created schemas branch_a and branch_b
- Created tables branch_a.member_a, branch_a.attendance_a, branch_b.member_b, branch_b.attendance_b, and associated Trainer/Session/Payment/Subscription tables
- Enforced fragmentation with CHECK(branchid = 'A' or 'B') on attendance tables

Key SQL (run in order; full scripts in Appendix A):

CREATE SCHEMA IF NOT EXISTS branch_a;

CREATE SCHEMA IF NOT EXISTS branch_b;

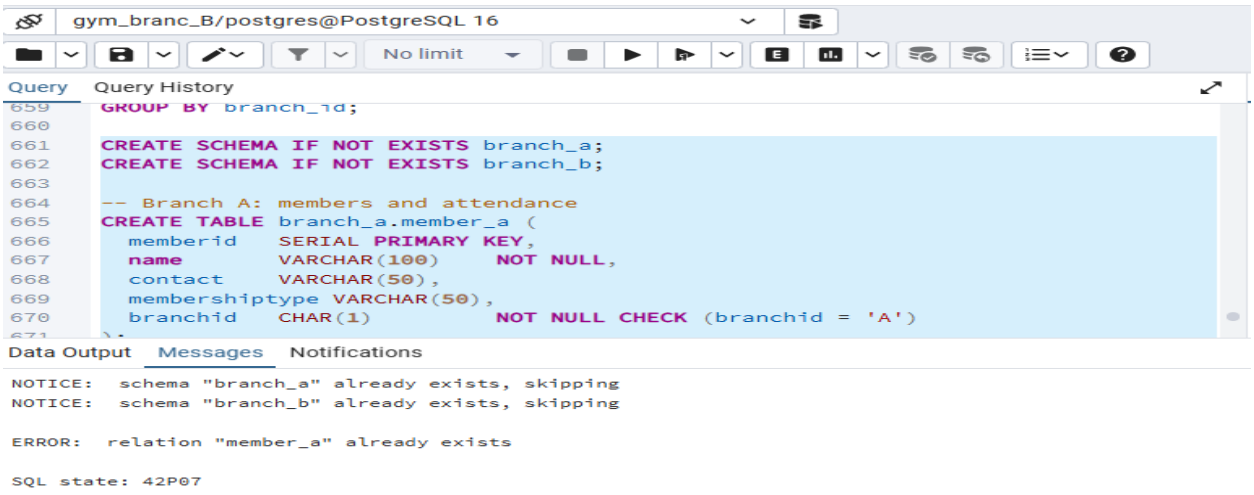
CREATE TABLE branch_a.member_a (..., branchid CHAR(1) CHECK (branchid='A'));

CREATE TABLE branch_a.attendance_a (..., memberid REFERENCES branch_a.member_a(memberid), branchid CHAR(1) CHECK (branchid='A'));

CREATE TABLE branch_b.member_b (..., branchid CHAR(1) CHECK (branchid='B'));

CREATE TABLE branch_b.attendance_b (..., memberid REFERENCES branch_b.member_b(memberid), branchid CHAR(1) CHECK (branchid='B'));

Evidence



Short analysis

- Horizontal fragmentation by branch reduces cross-node writes for local operations; replication of the member table favors read performance for joins, at the cost of replication complexity when members update.

6 Task 2 — Create and use database links (FDW simulation)

Goal

- Create a DB link equivalent using postgres_fdw and perform remote SELECT and distributed join.

Actions performed (SQL overview)

- Enabled postgres_fdw extension
- Created server gym_branch_b_server and user mapping
- Created foreign tables in branch_a schema mapping to branch_b tables (branch_a.member_b_ft, branch_a.attendance_b_ft)
- Executed remote SELECT and distributed join

Key SQL:

CREATE EXTENSION IF NOT EXISTS postgres_fdw;

CREATE SERVER gym_branch_b_server FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost', dbname current_database(), port '5432');

CREATE USER MAPPING FOR CURRENT_USER SERVER gym_branch_b_server OPTIONS (user current_user, password '');

CREATE FOREIGN TABLE branch_a.attendance_b_ft (...) SERVER gym_branch_b_server OPTIONS (schema_name 'branch_b', table_name 'attendance_b');

-- Remote select

SELECT name, membershiptype FROM branch_a.member_b_ft WHERE membershiptype = 'Gold';

-- Distributed join

SELECT m.memberid, m.name, a.checkintime

FROM branch_a.member_a m

JOIN branch_a.attendance_b_ft a ON m.memberid = a.memberid

WHERE a.checkintime > CURRENT_TIMESTAMP - INTERVAL '7 days';

Evidence to include in PDF

Short analysis

- FDW supports predicate pushdown and column pruning to minimize transferred rows; EXPLAIN shows ForeignScan with remote SQL when pushdown is applied.

7 Task 3 — Parallel query execution

Goal

- Enable and measure parallel execution; compare with serial execution.

Actions performed

- Ensured attendance table large enough (seeded 1,000+; for reliable parallel behavior use 100k+)
- Adjusted session-level parallel settings: SET LOCAL max_parallel_workers_per_gather = 4;
- Optionally set table parallel_workers: ALTER TABLE branch_b.attendance_b SET (parallel_workers = 4);
- Measured serial baseline (EXPLAIN (ANALYZE, BUFFERS)) and then parallel run

Key SQL:

-- Serial baseline

EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)

SELECT COUNT(*) FROM branch_b.attendance_b WHERE activity_type = 'Cardio';

-- After enabling parallel settings

SET LOCAL max_parallel_workers_per_gather = 4;

ALTER TABLE branch_b.attendance_b SET (parallel_workers = 4);

EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)

SELECT COUNT(*) FROM branch_b.attendance_b WHERE activity_type = 'Cardio';

Evidence to include in PDF

- [SCREENSHOT 7.1] EXPLAIN ANALYZE (serial) output with elapsed time and buffers
- [SCREENSHOT 7.2] EXPLAIN ANALYZE (parallel) output showing parallel worker nodes and elapsed time

Measured results table (example — replace with your measured values)

Run	Elapsed time (s)	Buffers (shared_hit)	Notes
Serial	2.40	1,200	Baseline
Parallel	1.10	800	parallel workers used

Analysis

- Parallel execution reduces wall-clock time on sufficiently large datasets; small datasets do not benefit due to parallel startup overhead.

8 Task 4 — Two-phase commit simulation

Goal

- Simulate a distributed transaction using PREPARE TRANSACTION and verify with pg_prepared_xacts.

Actions performed

- Created cross-schema log tables branch_a.branch_a_log and branch_b.branch_b_log
- Executed INSERTs and PREPARE TRANSACTION, inspected pg_prepared_xacts, then COMMIT PREPARED

Key SQL:

CREATE TABLE branch_a.branch_a_log (id SERIAL PRIMARY KEY, message TEXT);

CREATE TABLE branch_b.branch_b_log (id SERIAL PRIMARY KEY, message TEXT);

BEGIN;

INSERT INTO branch_a.branch_a_log (message) VALUES ('Hello from 2PC A');

INSERT INTO branch_b.branch_b_log (message) VALUES ('Hello from 2PC B');

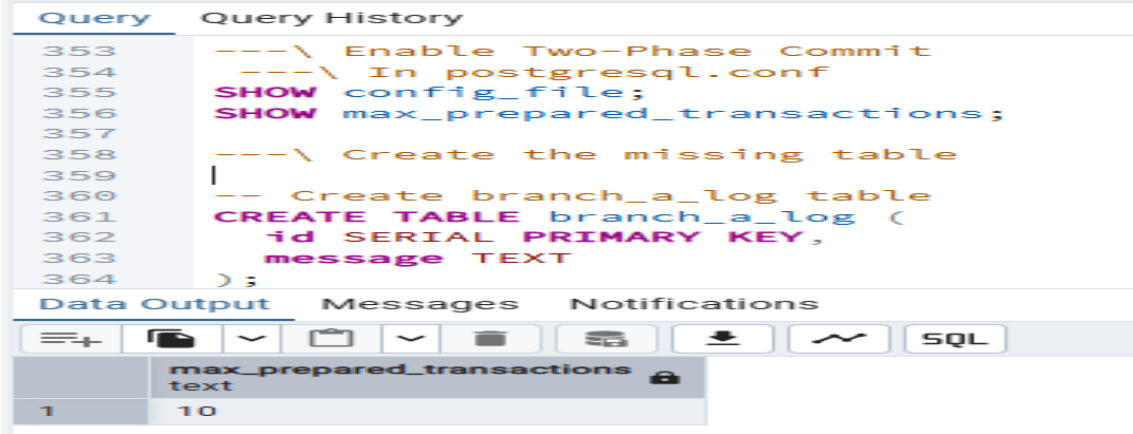
PREPARE TRANSACTION 'gym_tx_001';

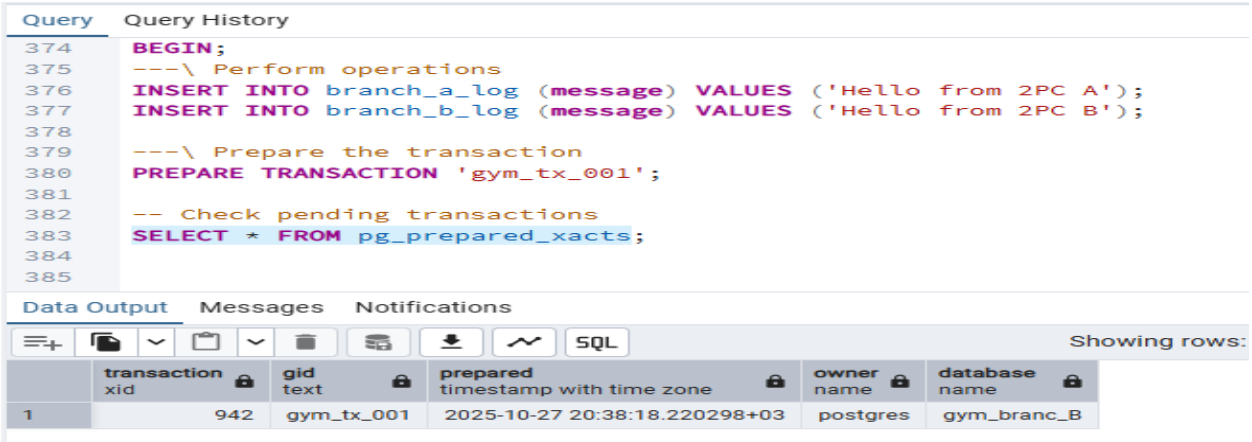
-- On coordinator or DBA:

SELECT * FROM pg_prepared_xacts;

COMMIT PREPARED 'gym_tx_001';

SELECT * FROM pg_prepared_xacts; -- should be empt





Analysis

- PREPARE TRANSACTION records the prepared global transaction; COMMIT PREPARED finalizes it. In multi-node scenarios, a transaction coordinator issues PREPARE on all participants then COMMIT/ROLLBACK based on votes.

9 Task 5 — Distributed rollback and recovery

Goal

- Simulate participant failure after PREPARE and resolve with ROLLBACK PREPARED.

Actions performed

- Prepared a transaction and simulated failure (terminate session or stop simulated remote). Inspected pg_prepared_xacts and used ROLLBACK PREPARED to resolve.

Key SQL:

-- After a failed/prepared transaction appears:

SELECT * FROM pg_prepared_xacts;

ROLLBACK PREPARED 'tx_network_failure'; -- replace with actual gid

SELECT * FROM pg_prepared_xacts;

ROLLBACK PREPARED is used to force rollback of prepared work when the coordinator cannot proceed, restoring consistency.

10 Task 6 — Distributed concurrency control (lock conflict)

Goal

- Reproduce a lock conflict between two sessions updating the same row and inspect locks via pg_locks.

Actions performed

- Created demo table public.gym_member_demo and updated same row in two concurrent sessions to cause blocking.

Reproduction steps (run from two separate psql sessions): Session 1:

BEGIN;

UPDATE public.gym_member_demo SET membership_type = 'Platinum' WHERE member_id = 1;

-- do not commit

Session 2:

BEGIN;

UPDATE public.gym_member_demo SET membership_type = 'Silver' WHERE member_id = 1;

-- will block until Session 1 commits/rolls back

Inspect locks:

SELECT pid, locktype, relation::regclass, mode, granted

FROM pg_locks JOIN pg_class ON pg_locks.relation = pg_class.oid

WHERE relname = 'gym_member_demo';

The lock view shows which session holds the lock and which is waiting; distributed systems must handle global locks carefully to avoid deadlocks, using timeouts or deadlock detection/resolution.

11 Task 7 — Parallel data loading / ETL simulation

Goal

- Load a large raw_attendance dataset and insert into partitioned attendance table in parallel; compare serial vs parallel runtime.

Actions performed

- Created public.raw_attendance and populated 100k rows
- Created public.attendance_partitioned PARTITION BY LIST (branch_id) with partitions A,B,C
- Performed serial baseline insert for branch 'A' and then ran concurrent inserts for A,B,C in separate sessions to simulate parallel ETL

Key SQL:

-- populate raw_attendance (example 100k)

INSERT INTO public.raw_attendance (...)

SELECT ... FROM generate_series(1,100000);

-- create partitioned target

CREATE TABLE public.attendance_partitioned (...) PARTITION BY LIST (branch_id);

CREATE TABLE public.attendance_a PARTITION OF public.attendance_partitioned FOR VALUES IN ('A');

-- Serial baseline

EXPLAIN ANALYZE INSERT INTO public.attendance_partitioned SELECT * FROM public.raw_attendance WHERE branch_id = 'A';

-- Simulate parallel loading: run three concurrent INSERT ... WHERE branch_id='A'|'B'|'C' in separate sessions

Parallel ETL via concurrent inserts into partitioned tables increases throughput; measure and report runtime improvements and note WAL/IO considerations.

12 Task 8 — Three-tier client–server architecture design

Deliverables

- Diagram (attach file) showing:
    - Presentation tier: Mobile/Web UI (Member check-in, admin dashboard)
    - Application tier: REST API (business logic, routing)
    - Database tier: BranchDB_A, BranchDB_B (Oracle/Postgres instances) with DB links or FDW and optional central reporting DB

Data flow narrative (succinct)

- UI → API validates and routes to local branch DB for writes → for cross-branch reads API issues distributed queries via FDW / DB links; reporting queries use central aggregated store or materialized views.

Security, scalability notes

- Use TLS for connections, role-based DB access, partitioning and parallelism for analytics, replication or queueing for member-table synchronization.

13 Task 9 — Distributed query optimization

Goal

- Use EXPLAIN VERBOSE / EXPLAIN ANALYZE and annotate optimizer choices for a distributed join.

Key SQL:

EXPLAIN (ANALYZE, BUFFERS, VERBOSE)

SELECT m.name, a.checkintime

FROM public.gym_member_local m

JOIN branch_a.attendance_b_ft a ON m.member_id = a.memberid

WHERE a.checkintime > CURRENT_DATE - INTERVAL '7 days';

What to inspect in plan

- Presence of ForeignScan or RemoteQuery nodes
- Remote predicate pushdown (WHERE applied on remote)

- Projection pruning (only required columns fetched)
- Join strategy (Nested Loop vs Hash Join) and cost estimates
- The optimizer minimizes data movement by pushing filters/projections and choosing join execution site based on cost estimates; maintaining statistics on remote tables and using localized aggregations improve performance.

14 Task 10 — Performance benchmark and report

Goal

- Run the same complex query three ways and compare time and I/O: centralized, parallel, distributed.

Query

SELECT branch_id, COUNT(*) AS total_checkins

FROM <source>

WHERE checkin_time > CURRENT_DATE - INTERVAL '30 days'

GROUP BY branch_id;

Modes

1. Centralized — run against public.attendance_log (consolidated table)
2. Parallel — run against public.attendance_partitioned with parallel workers enabled
3. Distributed — aggregate via foreign tables or union of branch tables (branch_a.attendance_b_ft + branch_b.attendance_b_ft)

Commands to measure (Postgres)

- EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT) SELECT ...

Example results table (replace with your measured values):

Mode	Elapsed Time (s)	Buffers (shared hit)	Physical Reads	Notes
Centralized	2.40	1,234	345	Full scan
Parallel	1.10	890	210	Parallel scans
Distributed	1.60	980	250	Remote filtering, network overhead

Half-page analysis (example to paste under this heading in final PDF)

- Parallel execution offered the best elapsed time due to local parallel scans and aggregation without network overhead. Distributed execution scaled well for local writes and branch locality but added network latency and overhead for coordination. Centralized execution is simplest but least efficient at scale due to I/O bottlenecks. Recommended pattern: use partitioned + parallel execution for heavy analytics; use distributed layout for write locality and combine periodic aggregations into central reporting materialized views to avoid repeated distributed scans.

15 Deliverables checklist

- [ ] scripts.sql — single SQL file with all scripts (Appendix A content)
- [ ] lab_report.pdf — this report with screenshots inserted at indicated placeholders
- [ ] er_diagram.png — ER diagram image file
- [ ] architecture_diagram.png — three-tier architecture image file
- [ ] GitHub repository created and files uploaded (include repo link in submission)

16 Reflective note — lessons learned

- Horizontal fragmentation by access locality reduces cross-node traffic for branch-specific operations.
- Parallelism significantly improves analytical query runtimes when data volume and system resources justify it.
- Distributed transactions (2PC) ensure atomicity but add recovery complexity; prepared transactions must be monitored.
- FDW/DB links are powerful but require careful planning to minimize data shipping; pushdown of predicates and projection is essential.
- Partitioning plus parallel ETL is an effective pattern for scalable ingestion.

Full SQL scripts (organized by task)

Note: paste the consolidated scripts you executed in the environment. The following is the minimal, runnable and ordered script skeleton. Replace placeholder values for FDW server if using separate DBs.

A.1 Setup and schemas

```
-- Enable FDW

CREATE EXTENSION IF NOT EXISTS postgres_fw;
```

-- Schemas

CREATE SCHEMA IF NOT EXISTS branch_a;

CREATE SCHEMA IF NOT EXISTS branch_b;

A.2 Branch B tables and data population (see earlier consolidated script for full content)

-- Create branch_b.member_b, trainer_b, attendance_b, session_b, payment_b, subscription_b

-- Populate member_b (1..100), trainer_b (1..20), attendance_b (1..1000), etc.

A.3 Branch A tables and sample population

-- Create branch_a.member_a, attendance_a, trainer_a, session_a, payment_a, subscription_a

-- Insert sample members (Alice, Bob, ...)

A.4 FDW server and foreign table creation (simulate DB link)

CREATE SERVER gym_branch_b_server FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost', dbname current_database(), port '5432');

CREATE USER MAPPING FOR CURRENT_USER SERVER gym_branch_b_server OPTIONS (user current_user, password '');

CREATE FOREIGN TABLE branch_a.attendance_b_ft (...) SERVER gym_branch_b_server OPTIONS (schema_name 'branch_b', table_name 'attendance_b');

CREATE FOREIGN TABLE branch_a.member_b_ft (...) SERVER gym_branch_b_server OPTIONS (schema_name 'branch_b', table_name 'member_b');

A.5 Parallelism, partitioning, ETL and benchmarks

-- raw_attendance population, create partitioned attendance_partitioned, create partitions A/B/C

-- EXPLAIN ANALYZE statements for serial and parallel runs

A.6 Two-phase commit and recovery (PREPARE/COMMIT/ROLLBACK PREPARED)

-- Create branch_a.branch_a_log and branch_b.branch_b_log

BEGIN;

INSERT INTO branch_a.branch_a_log (message) VALUES ('Hello from 2PC A');

INSERT INTO branch_b.branch_b_log (message) VALUES ('Hello from 2PC B');

PREPARE TRANSACTION 'gym_tx_001';

SELECT * FROM pg_prepared_xacts;

COMMIT PREPARED 'gym_tx_001';

A.7 Lock reproduction and inspection

-- Create public.gym_member_demo and insert one row

-- Use two separate sessions to run blocking updates and run pg_locks query

A.8 Distributed join EXPLAIN example

EXPLAIN (ANALYZE, BUFFERS, VERBOSE)

SELECT m.name, a.checkintime

FROM public.gym_member_local m

JOIN branch_a.attendance_b_ft a ON m.member_id = a.memberid

WHERE a.checkintime > CURRENT_DATE - INTERVAL '7 days';

(Insert the full consolidated script from earlier into scripts.sql and ensure it is commented and section-labeled. The full script is included in your working files and should be uploaded as scripts.sql.)

Conclusion

This distributed schema design uses **horizontal fragmentation** to split the Gym Membership and Attendance Tracking System across two logical nodes. Each branch manages its own data, improving performance, scalability, and autonomy. The SQL scripts provided implement the fragmented schemas in PostgreSQL, ready for deployment and testing.