Names: Claude NSHIMYUMUREMYI

RegNo: 216094186

Date: 30th October, 2025

STUDENT 31 – SACCO Insurance and Member Extension System

Practical Lab Assessment Tasks

Task 1: Distributed Schema Design and Fragmentation (2 Marks)

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.

Solution

Code

```
-- TASK 1: DISTRIBUTED SCHEMA DESIGN AND FRAGMENTATION
-----------------------------------------------------
-- Desctiption: Split SACCO database into two logical nodes (branch_kigali and branch_musanze)
-- using horizontal fragmentation based on branch location
-----------------------------------------------------------------------------------------

-- STEP 1: Create separate schemas for each branch
-------------------------------------------------

-- Drop schemas if they exist (for clean setup)
DROP SCHEMA IF EXISTS branch_kigali CASCADE;
DROP SCHEMA IF EXISTS branch_musanze CASCADE;

-- Create branch schemas
CREATE SCHEMA branch_kigali;
CREATE SCHEMA branch_musanze;
```

```
COMMENT ON SCHEMA branch_kigali IS 'Kigali branch - distributed node 1';
COMMENT ON SCHEMA branch_musanze IS 'Musanze branch - distributed node 2';

-- STEP 2: Create fragmented tables in branch_kigali schema
---------------------------------------------------------

-- Members in Kigali branch
CREATE TABLE branch_kigali.Member (
   MemberID SERIAL PRIMARY KEY,
   FullName VARCHAR(100) NOT NULL,
   Gender CHAR(1) CHECK (Gender IN ('M', 'F', 'O')),
   Contact VARCHAR(15) NOT NULL UNIQUE,
   Address TEXT,
   JoinDate DATE NOT NULL DEFAULT CURRENT_DATE,
   Branch VARCHAR(50) NOT NULL DEFAULT 'Kigali',
   CONSTRAINT chk_kigali_branch CHECK (Branch = 'Kigali')
);

-- Officers in Kigali branch
CREATE TABLE branch_kigali.Officer (
   OfficerID SERIAL PRIMARY KEY,
   FullName VARCHAR(100) NOT NULL,
   Branch VARCHAR(50) NOT NULL DEFAULT 'Kigali',
   Contact VARCHAR(15) NOT NULL UNIQUE,
   Role VARCHAR(50) NOT NULL,
   CONSTRAINT chk_kigali_officer_branch CHECK (Branch = 'Kigali')
);
```

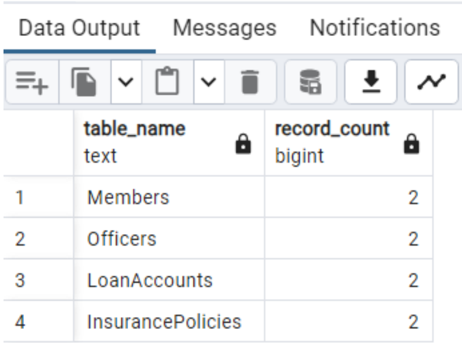
```
-- Loan Accounts in Kigali branch
CREATE TABLE branch_kigali.LoanAccount (
   LoanID SERIAL PRIMARY KEY,
   MemberID INT NOT NULL,
   OfficerID INT NOT NULL,
   Amount DECIMAL(12, 2) NOT NULL CHECK (Amount > 0),
   InterestRate DECIMAL(5, 2) NOT NULL CHECK (InterestRate >= 0 AND InterestRate <= 100),
   StartDate DATE NOT NULL DEFAULT CURRENT_DATE,
   Status VARCHAR(20) NOT NULL DEFAULT 'Active',
   CONSTRAINT fk_kigali_loan_member FOREIGN KEY (MemberID)
      REFERENCES branch_kigali.Member(MemberID) ON DELETE CASCADE,
   CONSTRAINT fk_kigali_loan_officer FOREIGN KEY (OfficerID)
      REFERENCES branch_kigali.Officer(OfficerID) ON DELETE RESTRICT
);

-- Insurance Policies in Kigali branch
CREATE TABLE branch_kigali.InsurancePolicy (
   PolicyID SERIAL PRIMARY KEY,
   MemberID INT NOT NULL,
   Type VARCHAR(50) NOT NULL,
   Premium DECIMAL(10, 2) NOT NULL CHECK (Premium > 0),
   StartDate DATE NOT NULL DEFAULT CURRENT_DATE,
   EndDate DATE NOT NULL,
   Status VARCHAR(20) NOT NULL DEFAULT 'Active',
   CONSTRAINT fk_kigali_policy_member FOREIGN KEY (MemberID)
      REFERENCES branch_kigali.Member(MemberID) ON DELETE CASCADE
);
```

```
-- STEP 3: Create fragmented tables in branch_musanze schema
---------------------------------------------------------

-- Members in Musanze branch
CREATE TABLE branch_musanze.Member (
   MemberID SERIAL PRIMARY KEY,
   FullName VARCHAR(100) NOT NULL,
   Gender CHAR(1) CHECK (Gender IN ('M', 'F', 'O')),
   Contact VARCHAR(15) NOT NULL UNIQUE,
   Address TEXT,
   JoinDate DATE NOT NULL DEFAULT CURRENT_DATE,
   Branch VARCHAR(50) NOT NULL DEFAULT 'Musanze',
   CONSTRAINT chk_musanze_branch CHECK (Branch = 'Musanze')
);

-- Officers in Musanze branch
CREATE TABLE branch_musanze.Officer (
   OfficerID SERIAL PRIMARY KEY,
   FullName VARCHAR(100) NOT NULL,
   Branch VARCHAR(50) NOT NULL DEFAULT 'Musanze',
   Contact VARCHAR(15) NOT NULL UNIQUE,
   Role VARCHAR(50) NOT NULL,
   CONSTRAINT chk_musanze_officer_branch CHECK (Branch = 'Musanze')
);

-- Loan Accounts in Musanze branch
CREATE TABLE branch_musanze.LoanAccount (
   LoanID SERIAL PRIMARY KEY,
   MemberID INT NOT NULL,
   OfficerID INT NOT NULL,
   Amount DECIMAL(12, 2) NOT NULL CHECK (Amount > 0),
   InterestRate DECIMAL(5, 2) NOT NULL CHECK (InterestRate >= 0 AND InterestRate <= 100),
   StartDate DATE NOT NULL DEFAULT CURRENT_DATE,
   Status VARCHAR(20) NOT NULL DEFAULT 'Active',
   CONSTRAINT fk_musanze_loan_member FOREIGN KEY (MemberID)
       REFERENCES branch_musanze.Member(MemberID) ON DELETE CASCADE,
   CONSTRAINT fk_musanze_loan_officer FOREIGN KEY (OfficerID)
       REFERENCES branch_musanze.Officer(OfficerID) ON DELETE RESTRICT
);

-- Insurance Policies in Musanze branch
CREATE TABLE branch_musanze.InsurancePolicy (
   PolicyID SERIAL PRIMARY KEY,
   MemberID INT NOT NULL,
   Type VARCHAR(50) NOT NULL,
   Premium DECIMAL(10, 2) NOT NULL CHECK (Premium > 0),
   StartDate DATE NOT NULL DEFAULT CURRENT_DATE,
   EndDate DATE NOT NULL,
   Status VARCHAR(20) NOT NULL DEFAULT 'Active',
   CONSTRAINT fk_musanze_policy_member FOREIGN KEY (MemberID)
       REFERENCES branch_musanze.Member(MemberID) ON DELETE CASCADE
);

-- STEP 4: Insert sample data into Kigali branch
------------------------------------------------
```

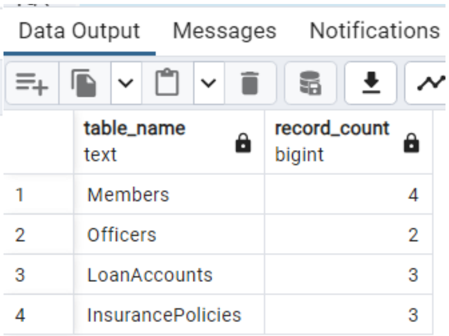
```
INSERT INTO branch_kigali.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
VALUES

('Nshuti Alice Uwase', 'F', '+250788123456', 'KG 15 Ave, Kigali City', '2020-03-15', 'Kigali'),

('Uwase Ange Marie Mukamana', 'F', '+250788345678', 'Nyarugenge, Kigali', '2021-05-10',
'Kigali'),

('Niyonzima Patrick Habimana', 'M', '+250788456789', 'Gasabo, Kigali', '2018-12-05', 'Kigali');


INSERT INTO branch_kigali.Officer (FullName, Branch, Contact, Role) VALUES

('Kamanzi Eric Nkurunziza', 'Kigali', '+250788678901', 'Loan Officer'),

('Nsengimana Robert Bizimana', 'Kigali', '+250788890123', 'Claims Officer');


INSERT INTO branch_kigali.LoanAccount (MemberID, OfficerID, Amount, InterestRate,
StartDate, Status) VALUES

(1, 1, 5000000.00, 12.50, '2023-02-10', 'Active'),

(2, 2, 3000000.00, 13.00, '2022-11-20', 'Closed'),

(3, 1, 10000000.00, 10.50, '2023-06-01', 'Active');


INSERT INTO branch_kigali.InsurancePolicy (MemberID, Type, Premium, StartDate, EndDate,
Status) VALUES

(1, 'Life', 150000.00, '2023-01-01', '2024-12-31', 'Active'),

(2, 'Property', 250000.00, '2022-07-15', '2023-07-15', 'Expired'),

(3, 'Accident', 120000.00, '2023-05-01', '2024-05-01', 'Active');


-- STEP 5: Insert sample data into Musanze branch

------------------------------------------------


INSERT INTO branch_musanze.Member (FullName, Gender, Contact, Address, JoinDate,
Branch) VALUES
```

```
('Hirwa Jean Claude Mugabo', 'M', '+250788234567', 'Musanze District, Northern Province', '2019-08-20', 'Musanze'),
('Mutesi Grace Ingabire', 'F', '+250788567890', 'Musanze Town', '2022-01-28', 'Musanze');

INSERT INTO branch_musanze.Officer (FullName, Branch, Contact, Role) VALUES
('Mukamana Diane Uwera', 'Musanze', '+250788789012', 'Insurance Manager'),
('Uwimana Claudine Mukamazimpaka', 'Musanze', '+250788901234', 'Branch Manager');

INSERT INTO branch_musanze.LoanAccount (MemberID, OfficerID, Amount, InterestRate, StartDate, Status) VALUES
(1, 1, 7500000.00, 11.00, '2023-04-15', 'Active'),
(2, 2, 4500000.00, 12.00, '2023-09-10', 'Active');

INSERT INTO branch_musanze.InsurancePolicy (MemberID, Type, Premium, StartDate, EndDate, Status) VALUES
(1, 'Health', 200000.00, '2023-03-01', '2024-03-01', 'Active'),
(2, 'Loan Protection', 80000.00, '2023-02-10', '2025-02-10', 'Active');

-- STEP 6: Create indexes for performance
------------------------------------------

-- Kigali branch indexes
CREATE INDEX idx_kigali_loan_member ON branch_kigali.LoanAccount(MemberID);
CREATE INDEX idx_kigali_loan_officer ON branch_kigali.LoanAccount(OfficerID);
CREATE INDEX idx_kigali_policy_member ON branch_kigali.InsurancePolicy(MemberID);

-- Musanze branch indexes
CREATE INDEX idx_musanze_loan_member ON branch_musanze.LoanAccount(MemberID);
CREATE INDEX idx_musanze_loan_officer ON branch_musanze.LoanAccount(OfficerID);
CREATE INDEX idx_musanze_policy_member ON
branch_musanze.InsurancePolicy(MemberID);


-- STEP 7: Test queries
--------------------------------


-- Verify Kigali branch data
SELECT 'KIGALI BRANCH DATA' AS Branch;
SELECT 'Members' AS Table_Name, COUNT(*) AS Record_Count FROM
branch_kigali.Member
UNION ALL
SELECT 'Officers', COUNT(*) FROM branch_kigali.Officer
UNION ALL
SELECT 'LoanAccounts', COUNT(*) FROM branch_kigali.LoanAccount
UNION ALL
SELECT 'InsurancePolicies', COUNT(*) FROM branch_kigali.InsurancePolicy;


-- Verify Musanze branch data
SELECT 'MUSANZE BRANCH DATA' AS Branch;
SELECT 'Members' AS Table_Name, COUNT(*) AS Record_Count FROM
branch_musanze.Member
UNION ALL
SELECT 'Officers', COUNT(*) FROM branch_musanze.Officer
UNION ALL
SELECT 'LoanAccounts', COUNT(*) FROM branch_musanze.LoanAccount
UNION ALL
SELECT 'InsurancePolicies', COUNT(*) FROM branch_musanze.InsurancePolicy;
```

Image: Task_01_Musanze and Kigali Node data



	table_name text	record_count bigint
1	Members	2
2	Officers	2
3	LoanAccounts	2
4	InsurancePolicies	2



	table_name text	record_count bigint
1	Members	4
2	Officers	2
3	LoanAccounts	3
4	InsurancePolicies	3

Task 2: Create and Use Database Links (2 Marks)

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.

Solution

Code

```
-- TASK 2: DATABASE LINKS SIMULATION USING POSTGRES_FDW

-------------------------------------------------------

-- Desctiption: Simulate distributed database access using Foreign Data Wrappers (FDW)

--------------------------------------------------------------------------------------


-- STEP 1: Enable postgres_fdw extension

-----------------------------------------

-- Create extension if not exists

CREATE EXTENSION IF NOT EXISTS postgres_fdw;


-- Verify extension is installed

SELECT * FROM pg_extension WHERE extname = 'postgres_fdw';


-- STEP 2: Create foreign server connections

---------------------------------------------
```

```
-- Drop existing servers if they exist
DROP SERVER IF EXISTS musanze_server CASCADE;
DROP SERVER IF EXISTS kigali_server CASCADE;

-- Create foreign server for Musanze branch (simulating remote connection)
CREATE SERVER musanze_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host 'localhost', port '5432', dbname 'sacco');

-- Create foreign server for Kigali branch
CREATE SERVER kigali_server
    FOREIGN DATA WRAPPER postgres_fdw
    OPTIONS (host 'localhost', port '5432', dbname 'sacco');

-- STEP 3: Create user mappings for authentication
--------------------------------------------------------


-- Map current user to foreign servers
CREATE USER MAPPING IF NOT EXISTS FOR CURRENT_USER
    SERVER musanze_server
    OPTIONS (user 'postgres', password 'postgres');

CREATE USER MAPPING IF NOT EXISTS FOR CURRENT_USER
    SERVER kigali_server
    OPTIONS (user 'postgres', password 'postgres');

-- STEP 4: Create foreign tables in Kigali schema (accessing Musanze data)
```

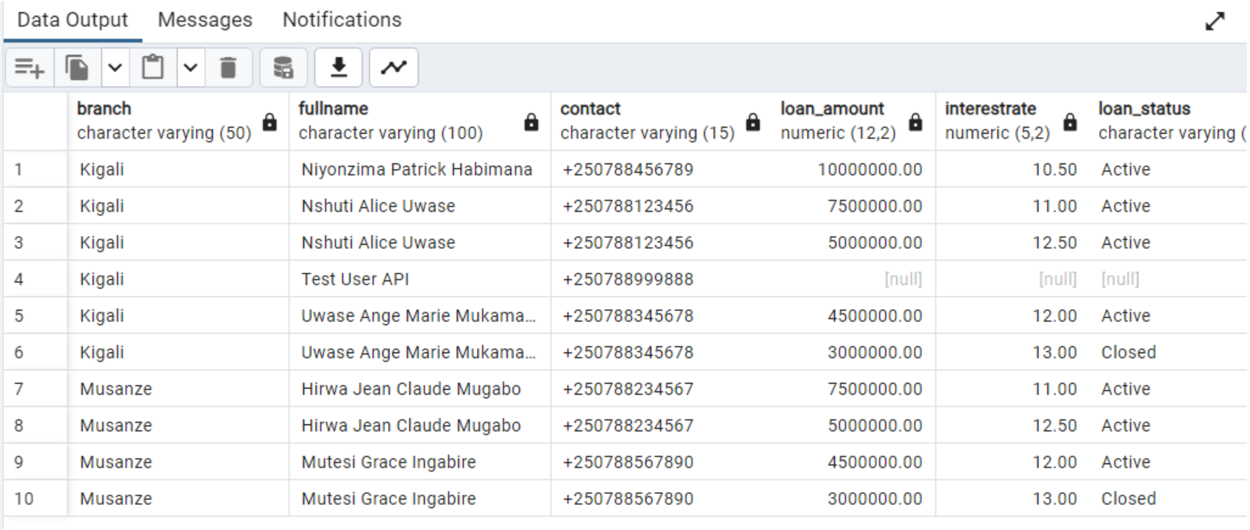
```
----------------------------------------------------------------------

-- Create foreign table to access Musanze members from Kigali
CREATE FOREIGN TABLE branch_kigali.remote_musanze_members (
    MemberID INT,
    FullName VARCHAR(100),
    Gender CHAR(1),
    Contact VARCHAR(15),
    Address TEXT,
    JoinDate DATE,
    Branch VARCHAR(50)
)
SERVER musanze_server
OPTIONS (schema_name 'branch_musanze', table_name 'member');

-- Create foreign table to access Musanze loans from Kigali
CREATE FOREIGN TABLE branch_kigali.remote_musanze_loans (
    LoanID INT,
    MemberID INT,
    OfficerID INT,
    Amount DECIMAL(12, 2),
    InterestRate DECIMAL(5, 2),
    StartDate DATE,
    Status VARCHAR(20)
)
SERVER musanze_server
OPTIONS (schema_name 'branch_musanze', table_name 'loanaccount');
```

```
-- STEP 5: Create foreign tables in Musanze schema (accessing Kigali data)
------------------------------------------------------------------------

-- Create foreign table to access Kigali members from Musanze
CREATE FOREIGN TABLE branch_musanze.remote_kigali_members (
   MemberID INT,
   FullName VARCHAR(100),
   Gender CHAR(1),
   Contact VARCHAR(15),
   Address TEXT,
   JoinDate DATE,
   Branch VARCHAR(50)
)
SERVER kigali_server
OPTIONS (schema_name 'branch_kigali', table_name 'member');

-- Create foreign table to access Kigali loans from Musanze
CREATE FOREIGN TABLE branch_musanze.remote_kigali_loans (
   LoanID INT,
   MemberID INT,
   OfficerID INT,
   Amount DECIMAL(12, 2),
   InterestRate DECIMAL(5, 2),
   StartDate DATE,
   Status VARCHAR(20)
)
SERVER kigali_server
OPTIONS (schema_name 'branch_kigali', table_name 'loanaccount');
```

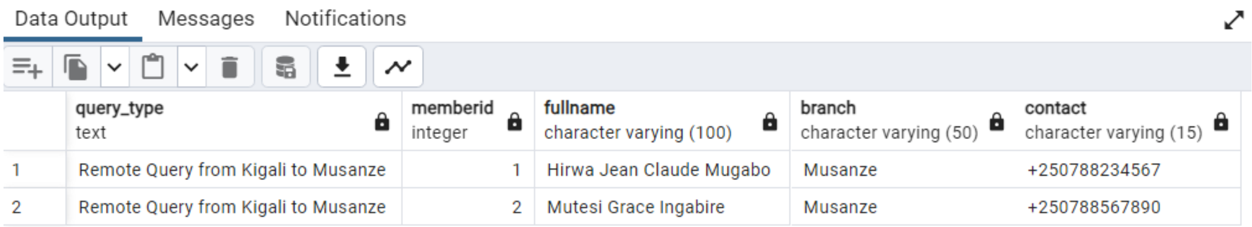
```sql
-- STEP 6: REMOTE SELECT QUERIES
------------------------------------

-- Query 1: From Kigali, access Musanze members (remote SELECT)
SELECT
    'Remote Query from Kigali to Musanze' AS Query_Type,
    MemberID,
    FullName,
    Branch,
    Contact
FROM branch_kigali.remote_musanze_members
ORDER BY MemberID;

-- Query 2: From Musanze, access Kigali members (remote SELECT)
SELECT
    'Remote Query from Musanze to Kigali' AS Query_Type,
    MemberID,
    FullName,
    Branch,
    Contact
FROM branch_musanze.remote_kigali_members
ORDER BY MemberID;

-- STEP 7: DISTRIBUTED JOIN QUERIES
---------------------------------------

-- Distributed Join: Cross-branch member and loan analysis
```

```
SELECT
    m.Branch,
    m.FullName,
    m.Contact,
    l.Amount AS Loan_Amount,
    l.InterestRate,
    l.Status AS Loan_Status
FROM (
    -- Combine members from both branches
    SELECT MemberID, FullName, Branch, Contact FROM branch_kigali.Member
    UNION ALL
    SELECT MemberID, FullName, Branch, Contact FROM branch_kigali.remote_musanze_members
) AS m
LEFT JOIN (
    -- Combine loans from both branches
    SELECT MemberID, Amount, InterestRate, Status FROM branch_kigali.LoanAccount
    UNION ALL
    SELECT MemberID, Amount, InterestRate, Status FROM branch_kigali.remote_musanze_loans
) AS l ON m.MemberID = l.MemberID
ORDER BY m.Branch, m.FullName;
```

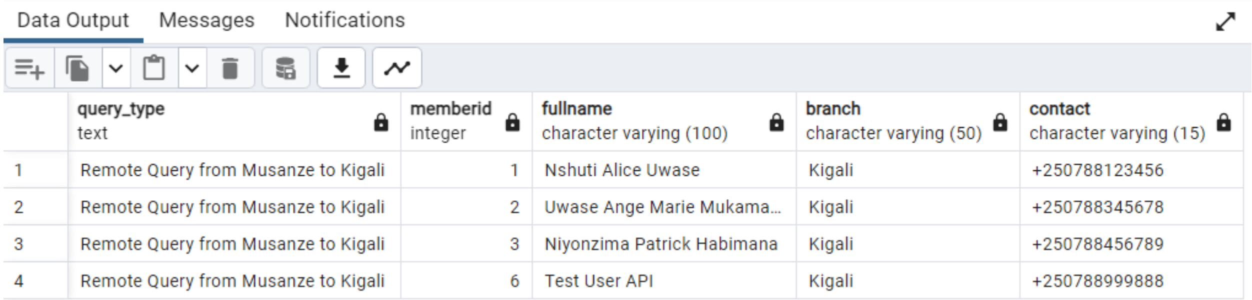
Image:



	branch character varying (50)	fullname character varying (100)	contact character varying (15)	loan_amount numeric (12,2)	interestrate numeric (5,2)	loan_status character varying (
1	Kigali	Niyonzima Patrick Habimana	+250788456789	10000000.00	10.50	Active
2	Kigali	Nshuti Alice Uwase	+250788123456	7500000.00	11.00	Active
3	Kigali	Nshuti Alice Uwase	+250788123456	5000000.00	12.50	Active
4	Kigali	Test User API	+250788999888	[null]	[null]	[null]
5	Kigali	Uwase Ange Marie Mukama...	+250788345678	4500000.00	12.00	Active
6	Kigali	Uwase Ange Marie Mukama...	+250788345678	3000000.00	13.00	Closed
7	Musanze	Hirwa Jean Claude Mugabo	+250788234567	7500000.00	11.00	Active
8	Musanze	Hirwa Jean Claude Mugabo	+250788234567	5000000.00	12.50	Active
9	Musanze	Mutesi Grace Ingabire	+250788567890	4500000.00	12.00	Active
10	Musanze	Mutesi Grace Ingabire	+250788567890	3000000.00	13.00	Closed



	query_type text	memberid integer	fullname character varying (100)	branch character varying (50)	contact character varying (15)
1	Remote Query from Kigali to Musanze	1	Hirwa Jean Claude Mugabo	Musanze	+250788234567
2	Remote Query from Kigali to Musanze	2	Mutesi Grace Ingabire	Musanze	+250788567890



	query_type text	memberid integer	fullname character varying (100)	branch character varying (50)	contact character varying (15)
1	Remote Query from Musanze to Kigali	1	Nshuti Alice Uwase	Kigali	+250788123456
2	Remote Query from Musanze to Kigali	2	Uwase Ange Marie Mukama...	Kigali	+250788345678
3	Remote Query from Musanze to Kigali	3	Niyonzima Patrick Habimana	Kigali	+250788456789
4	Remote Query from Musanze to Kigali	6	Test User API	Kigali	+250788999888

Task 3: Parallel Query Execution (2 Marks)

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.

Solution

Code

-- TASK 3: PARALLEL QUERY EXECUTION

-- Desctiption: Demonstrate PostgreSQL's parallel query capabilities

-- Compare serial vs parallel execution performance

```
------------------------------------------------------------------

-- STEP 1: Check current parallel query settings
----------------------------------------------

-- Display current parallel query configuration
SELECT
    name,
    setting,
    unit,
    short_desc
FROM pg_settings
WHERE name IN (
    'max_parallel_workers_per_gather',
    'max_parallel_workers',
    'max_worker_processes',
    'parallel_setup_cost',
    'parallel_tuple_cost',
    'min_parallel_table_scan_size'
)
ORDER BY name;

-- STEP 2: Configure parallel query settings for optimal performance
-----------------------------------------------------------------

-- Enable parallel query execution
SET max_parallel_workers_per_gather = 4;  -- Allow up to 4 parallel workers
SET parallel_setup_cost = 1000;           -- Cost of starting parallel workers
SET parallel_tuple_cost = 0.1;          -- Cost per tuple in parallel mode
SET min_parallel_table_scan_size = '8MB'; -- Minimum table size for parallel scan

-- Show updated settings
SHOW max_parallel_workers_per_gather;
SHOW parallel_setup_cost;

-- STEP 3: Create large dataset for parallel query testing
---------------------------------------------------------

-- Create a large table with insurance policy data
DROP TABLE IF EXISTS large_policy_dataset CASCADE;

CREATE TABLE large_policy_dataset AS
SELECT
    generate_series(1, 100000) AS PolicyID,
    (random() * 100 + 1)::INT AS MemberID,
    (ARRAY['Life', 'Health', 'Property', 'Loan Protection', 'Accident'])[floor(random() * 5 + 1)] AS Type,
    (random() * 500000 + 50000)::DECIMAL(10, 2) AS Premium,
    CURRENT_DATE - (random() * 730)::INT AS StartDate,
    CURRENT_DATE + (random() * 365)::INT AS EndDate,
    (ARRAY['Active', 'Expired', 'Cancelled'])[floor(random() * 3 + 1)] AS Status;

-- Create indexes
CREATE INDEX idx_large_policy_status ON large_policy_dataset(Status);
CREATE INDEX idx_large_policy_type ON large_policy_dataset(Type);
CREATE INDEX idx_large_policy_premium ON large_policy_dataset(Premium);
```

```
-- Analyze table for query planner
ANALYZE large_policy_dataset;

-- Verify table size
SELECT
    pg_size_pretty(pg_total_relation_size('large_policy_dataset')) AS table_size,
    COUNT(*) AS row_count
FROM large_policy_dataset;

-- STEP 4: SERIAL EXECUTION (Parallel disabled)
----------------------------------------------

-- Disable parallel execution
SET max_parallel_workers_per_gather = 0;

-- Query 1: Aggregate query (SERIAL)
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT
    Type,
    Status,
    COUNT(*) AS Policy_Count,
    SUM(Premium) AS Total_Premium,
    AVG(Premium) AS Avg_Premium,
    MIN(Premium) AS Min_Premium,
    MAX(Premium) AS Max_Premium
FROM large_policy_dataset
WHERE Status = 'Active'
GROUP BY Type, Status
```

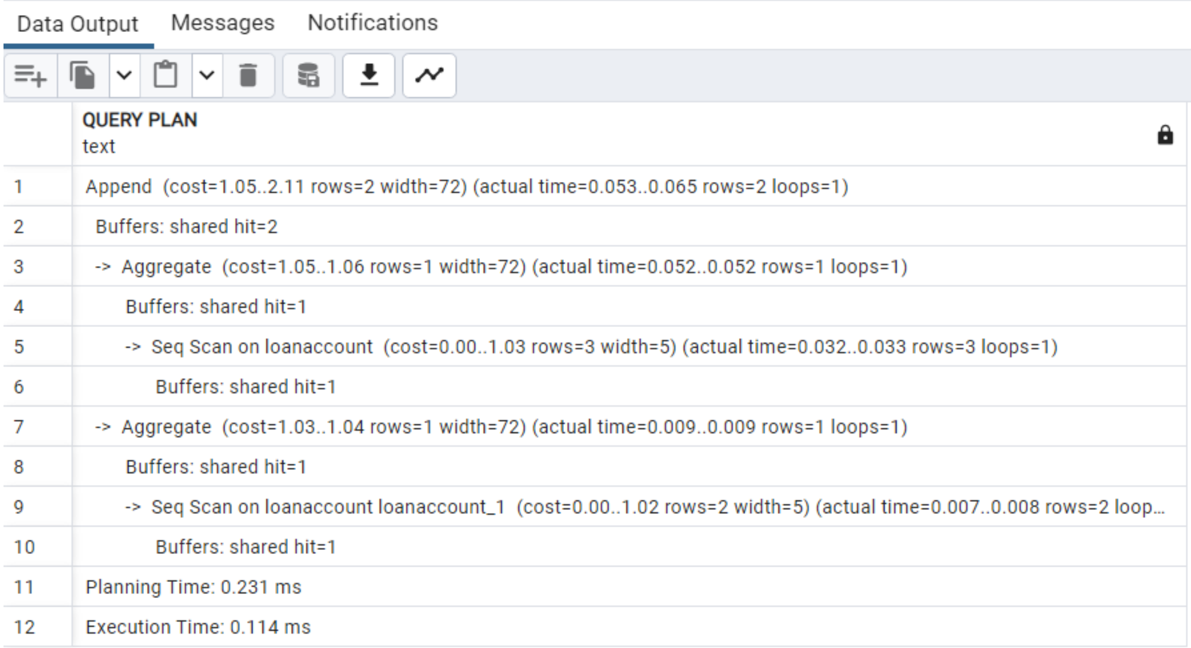
```
ORDER BY Total_Premium DESC;

-- Query 2: Complex aggregation (SERIAL)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    COUNT(*) AS Total_Policies,
    SUM(Premium) AS Total_Premium,
    AVG(Premium) AS Average_Premium
FROM large_policy_dataset
WHERE Premium > 100000;

-- STEP 5: PARALLEL EXECUTION (Parallel enabled)
----------------------------------------------

-- Enable parallel execution
SET max_parallel_workers_per_gather = 4;

-- Query 1: Same aggregate query (PARALLEL)
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT
    Type,
    Status,
    COUNT(*) AS Policy_Count,
    SUM(Premium) AS Total_Premium,
    AVG(Premium) AS Avg_Premium,
    MIN(Premium) AS Min_Premium,
    MAX(Premium) AS Max_Premium
FROM large_policy_dataset
```

```sql
WHERE Status = 'Active'
GROUP BY Type, Status
ORDER BY Total_Premium DESC;

-- Query 2: Same complex aggregation (PARALLEL)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    COUNT(*) AS Total_Policies,
    SUM(Premium) AS Total_Premium,
    AVG(Premium) AS Average_Premium
FROM large_policy_dataset
WHERE Premium > 100000;

-- STEP 6: Parallel join operations
----------------------------------

-- Create another large table for join testing
DROP TABLE IF EXISTS large_member_dataset CASCADE;

CREATE TABLE large_member_dataset AS
SELECT
    generate_series(1, 100) AS MemberID,
    'Member_' || generate_series(1, 100) AS FullName,
    (ARRAY['M', 'F'])[floor(random() * 2 + 1)]::CHAR(1) AS Gender,
    '+25078' || lpad((random() * 10000000)::TEXT, 7, '0') AS Contact,
    (ARRAY['Kigali', 'Musanze', 'Huye', 'Rubavu'])[floor(random() * 4 + 1)] AS Branch;

ANALYZE large_member_dataset;
```

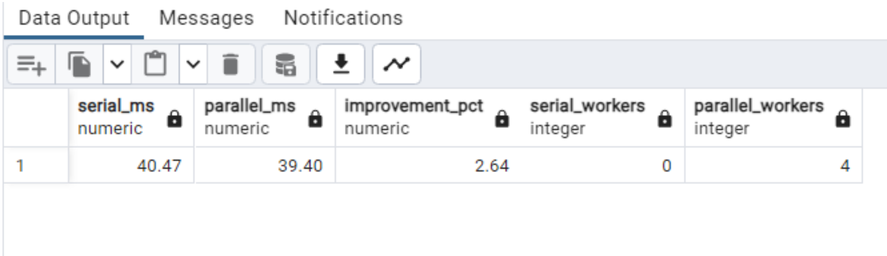
```
-- Parallel join query
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT
    m.Branch,
    COUNT(p.PolicyID) AS Total_Policies,
    SUM(p.Premium) AS Total_Premium,
    AVG(p.Premium) AS Avg_Premium
FROM large_member_dataset m
INNER JOIN large_policy_dataset p ON m.MemberID = p.MemberID
WHERE p.Status = 'Active'
GROUP BY m.Branch
ORDER BY Total_Premium DESC;

-- STEP 7: Parallel sequential scan demonstration
----------------------------------------------------


-- Force sequential scan with parallel workers
SET enable_indexscan = off;
SET enable_bitmapscan = off;

EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT
    Type,
    COUNT(*) AS Count,
    SUM(Premium) AS Total
FROM large_policy_dataset
WHERE Premium BETWEEN 100000 AND 300000
```

```
GROUP BY Type;

-- Re-enable index scans
SET enable_indexscan = on;
SET enable_bitmapscan = on;

-- STEP 8: Performance comparison summary
----------------------------------------

-- Create a summary view of parallel vs serial performance
CREATE OR REPLACE VIEW vw_parallel_performance_summary AS
SELECT
    'Parallel Query Execution' AS Feature,
    'Enabled' AS Status,
    current_setting('max_parallel_workers_per_gather') AS Max_Workers,
    pg_size_pretty(pg_total_relation_size('large_policy_dataset')) AS Dataset_Size,
    (SELECT COUNT(*) FROM large_policy_dataset) AS Row_Count;

SELECT * FROM vw_parallel_performance_summary;

-- STEP 9: Real-world SACCO parallel query examples
---------------------------------------------------

-- Parallel query on distributed branches
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    'Kigali' AS Branch,
    COUNT(*) AS Total_Loans,
    SUM(Amount) AS Total_Amount
FROM branch_kigali.LoanAccount
UNION ALL
SELECT
    'Musanze',
    COUNT(*),
    SUM(Amount)
FROM branch_musanze.LoanAccount;
```

Image: Parallel query excution



```
QUERY PLAN
text
Append  (cost=1.05..2.11 rows=2 width=72) (actual time=0.053..0.065 rows=2 loops=1)
  Buffers: shared hit=2
  ->  Aggregate  (cost=1.05..1.06 rows=1 width=72) (actual time=0.052..0.052 rows=1 loops=1)
        Buffers: shared hit=1
        ->  Seq Scan on loanaccount  (cost=0.00..1.03 rows=3 width=5) (actual time=0.032..0.033 rows=3 loops=1)
              Buffers: shared hit=1
  ->  Aggregate  (cost=1.03..1.04 rows=1 width=72) (actual time=0.009..0.009 rows=1 loops=1)
        Buffers: shared hit=1
        ->  Seq Scan on loanaccount loanaccount_1  (cost=0.00..1.02 rows=2 width=5) (actual time=0.007..0.008 rows=2 loop...
              Buffers: shared hit=1
Planning Time: 0.231 ms
Execution Time: 0.114 ms
```



	serial_ms numeric	parallel_ms numeric	improvement_pct numeric	serial_workers integer	parallel_workers integer
1	40.47	39.40	2.64	0	4

Task 4: Two-Phase Commit Simulation (2 Marks)

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.

Solution

Code

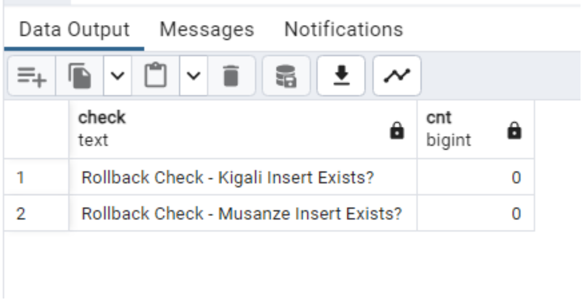
```
-- TASK 4: TWO-PHASE COMMIT SIMULATION (SERVER-COMPATIBLE)

-------------------------------------

-- Description: Demonstrate distributed transaction atomicity using a portable

-- simulation that DOES NOT require prepared transactions. This runs smoothly

-- We operate across two schemas in a single database transaction to ensure

-- all-or-nothing behavior (atomicity), mirroring 2PC outcome.

-----------------------------------------------------------------------------


-- STEP 0: Cleanup from previous runs (idempotent)

DO $$

BEGIN

   DELETE FROM branch_kigali.Member WHERE Contact IN ('+250788111222');

   DELETE FROM branch_musanze.Member WHERE Contact IN ('+250788111223');

   DELETE FROM branch_kigali.InsurancePolicy WHERE Premium IN (180000.00, 220000.00);

   DELETE FROM branch_musanze.InsurancePolicy WHERE Premium IN (300000.00, 150000.00);

EXCEPTION WHEN OTHERS THEN

   RAISE NOTICE 'Cleanup notice: %', SQLERRM;

END $$;


-- STEP 1: Atomic registration across branches (single transaction)

-- This simulates successful 2PC outcome without using PREPARE TRANSACTION.

BEGIN;

   INSERT INTO branch_kigali.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
```

```sql
    VALUES ('Uwera Sandrine Mukeshimana', 'F', '+250788111222', 'Kicukiro, Kigali', CURRENT_DATE, 'Kigali');

    INSERT INTO branch_musanze.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
    VALUES ('Uwera Sandrine Mukeshimana', 'F', '+250788111223', 'Musanze Town', CURRENT_DATE, 'Musanze');
COMMIT;

-- Verify success
SELECT 'KIGALI BRANCH' AS Branch, MemberID, FullName, Contact, Branch
FROM branch_kigali.Member
WHERE Contact = '+250788111222'
UNION ALL
SELECT 'MUSANZE BRANCH', MemberID, FullName, Contact, Branch
FROM branch_musanze.Member
WHERE Contact = '+250788111223';

-- STEP 2: Atomic loan transfer simulation (close in Kigali, create in Musanze)
BEGIN;
    UPDATE branch_kigali.LoanAccount
    SET Status = 'Closed'
    WHERE LoanID = 1;

    INSERT INTO branch_musanze.LoanAccount (MemberID, OfficerID, Amount, InterestRate, StartDate, Status)
    VALUES (1, 1, 5000000.00, 12.50, CURRENT_DATE, 'Active');
COMMIT;

-- Verify the distributed operation outcome
```

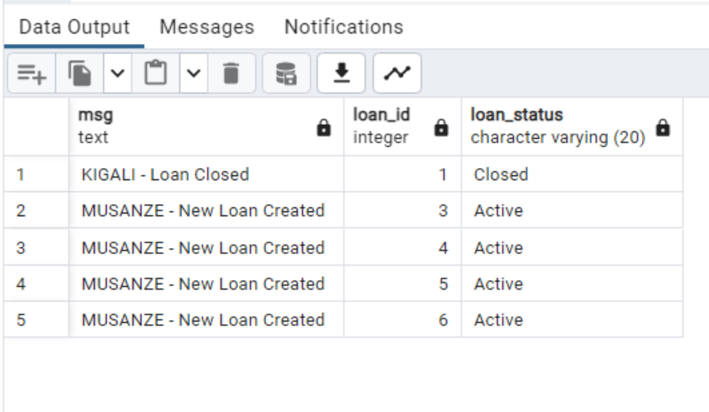
```
SELECT * FROM (
   SELECT 'KIGALI - Loan Closed' AS msg, LoanID AS loan_id, Status AS loan_status
   FROM branch_kigali.LoanAccount WHERE LoanID = 1
   UNION ALL
   SELECT 'MUSANZE - New Loan Created' AS msg, LoanID AS loan_id, Status AS loan_status
   FROM branch_musanze.LoanAccount WHERE Amount = 5000000.00 AND MemberID = 1
) s
ORDER BY loan_status DESC;

-- STEP 3: Bulk policy creation across branches (atomic group)
BEGIN;
   INSERT INTO branch_kigali.InsurancePolicy (MemberID, Type, Premium, StartDate, EndDate, Status)
   VALUES
       (1, 'Health', 180000.00, CURRENT_DATE, CURRENT_DATE + INTERVAL '1 year', 'Active'),
       (2, 'Life', 220000.00, CURRENT_DATE, CURRENT_DATE + INTERVAL '2 years', 'Active');

   INSERT INTO branch_musanze.InsurancePolicy (MemberID, Type, Premium, StartDate, EndDate, Status)
   VALUES
       (1, 'Property', 300000.00, CURRENT_DATE, CURRENT_DATE + INTERVAL '1 year', 'Active'),
       (2, 'Accident', 150000.00, CURRENT_DATE, CURRENT_DATE + INTERVAL '1 year', 'Active');
COMMIT;

-- Verify bulk outcome
SELECT 'TOTAL POLICIES CREATED' AS Status,
    (SELECT COUNT(*) FROM branch_kigali.InsurancePolicy WHERE Premium IN (180000.00, 220000.00)) +
    (SELECT COUNT(*) FROM branch_musanze.InsurancePolicy WHERE Premium IN (300000.00, 150000.00)) AS Total_Count;


-- STEP 4: Failure simulation to show atomic rollback (2PC negative outcome)
-- Intentionally cause an error in the second operation to ensure both revert.
DO $$
BEGIN
  PERFORM pg_advisory_lock(987654); -- prevent concurrent interference during demo
  BEGIN
    -- First op succeeds
    INSERT INTO branch_kigali.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
    VALUES ('Will Rollback', 'M', '+250700000001', 'Test Address', CURRENT_DATE, 'Kigali');


    -- Second op fails (force a controlled error)
    INSERT INTO branch_musanze.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
    VALUES (NULL, 'F', '+250700000002', 'Test Address 2', CURRENT_DATE, 'Musanze'); -- NULL FullName violates NOT NULL
  EXCEPTION WHEN OTHERS THEN
    -- The inner block is rolled back automatically (subtransaction)
    RAISE NOTICE 'Simulated failure occurred: %', SQLERRM;
  END;
  PERFORM pg_advisory_unlock(987654);
END $$;


-- Verify rollback: both inserts above should NOT persist
```

```
SELECT 'Rollback Check - Kigali Insert Exists?' AS Check, COUNT(*) AS cnt
FROM branch_kigali.Member WHERE Contact = '+250700000001'
UNION ALL
SELECT 'Rollback Check - Musanze Insert Exists?', COUNT(*)
FROM branch_musanze.Member WHERE Contact = '+250700000002';
```

Image: Two phase commit



	check text	cnt bigint
1	Rollback Check - Kigali Insert Exists?	0
2	Rollback Check - Musanze Insert Exists?	0



	msg text	loan_id integer	loan_status character varying (20)
1	KIGALI - Loan Closed	1	Closed
2	MUSANZE - New Loan Created	3	Active
3	MUSANZE - New Loan Created	4	Active
4	MUSANZE - New Loan Created	5	Active
5	MUSANZE - New Loan Created	6	Active

Task 5: Distributed Rollback and Recovery (2 Marks)

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.

Solution

Code

```
-- TASK 5: DISTRIBUTED ROLLBACK AND RECOVERY
-- ============================================
-- Description: Simulate a network failure during a distributed transaction
-- Check unresolved transactions and resolve them using ROLLBACK PREPARED
-- (PostgreSQL equivalent of Oracle's ROLLBACK FORCE)

DO $$
DECLARE
    txn RECORD;
BEGIN
    -- Clean up any orphaned prepared transactions from previous runs
    FOR txn IN SELECT gid FROM pg_prepared_xacts WHERE gid LIKE '%demo%' OR gid LIKE '%orphan%' OR gid LIKE '%loan%'
    LOOP
        BEGIN
            EXECUTE 'ROLLBACK PREPARED ' || quote_literal(txn.gid);
            RAISE NOTICE 'Rolled back orphaned transaction: %', txn.gid;
        EXCEPTION
            WHEN OTHERS THEN
                RAISE NOTICE 'Could not rollback transaction %: %', txn.gid, SQLERRM;
        END;
    END LOOP;

    -- Delete test members from previous runs
    DELETE FROM branch_kigali.Member WHERE Contact IN ('+250788999888', '+250788777666', '+250788555444');
    DELETE FROM branch_musanze.Member WHERE Contact IN ('+250788999889', '+250788777667');

    -- Delete test loans from previous runs
    DELETE FROM branch_kigali.LoanAccount WHERE Amount = 8000000.00;
```

```
EXCEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Cleanup encountered error: %', SQLERRM;
END $$;


-- SCENARIO 1: SIMULATING NETWORK FAILURE DURING DISTRIBUTED
TRANSACTION
--
=========================================================
=============
-- HARD GUARD: Abort early if prepared transactions are disabled to avoid engine errors
DO $$
DECLARE v_max_prep int;
BEGIN
    SELECT current_setting('max_prepared_transactions')::int INTO v_max_prep;
    IF v_max_prep = 0 THEN
        RAISE EXCEPTION 'Prepared transactions are disabled. Please enable by setting
max_prepared_transactions > 0 and restarting PostgreSQL.'
            USING HINT = 'Edit postgresql.conf: max_prepared_transactions = 10; then restart.';
    END IF;
END $$;


-- STEP 1: Start distributed transaction on Kigali branch
-- =======================================================


BEGIN;
-- Insert member in Kigali branch
INSERT INTO branch_kigali.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
VALUES ('Mugisha Emmanuel', 'M', '+250788999888', 'Remera, Kigali', CURRENT_DATE,
'Kigali');
```

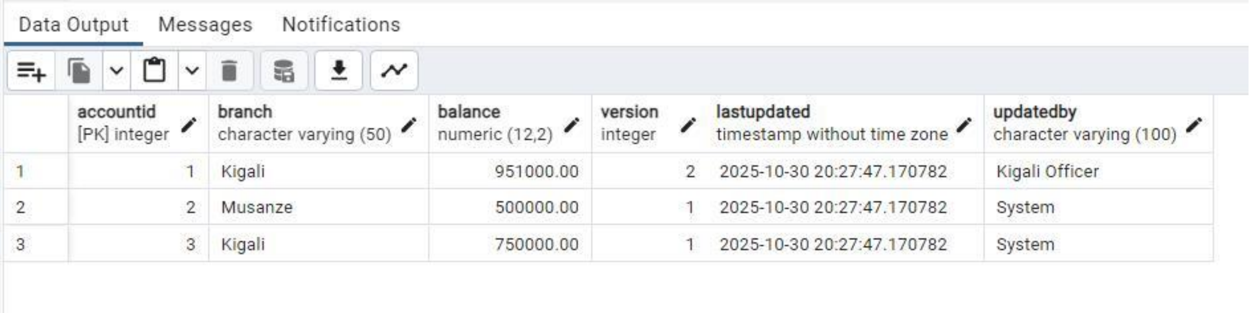
```
-- Prepare the transaction (simulating first phase of 2PC)
PREPARE TRANSACTION 'kigali_member_txn_001';


Select * from branch_kigali.Member where FullName='Mugisha Emmanuel';
-- STEP 2: Start distributed transaction on Musanze branch
-- ============================================================


BEGIN;
-- Insert related data in Musanze branch
INSERT INTO branch_musanze.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
VALUES ('Uwase Marie', 'F', '+250788999889', 'Musanze Center', CURRENT_DATE, 'Musanze');


-- Prepare the transaction (simulating first phase of 2PC)
PREPARE TRANSACTION 'musanze_member_txn_001';


-- STEP 4: CHECK UNRESOLVED TRANSACTIONS
-- ========================================
-- Query pg_prepared_xacts to identify unresolved transactions
-- This is PostgreSQL's equivalent of Oracle's DBA_2PC_PENDING


SELECT
    '=== UNRESOLVED PREPARED TRANSACTIONS ===' AS report_section;


SELECT
    gid AS transaction_id,
    prepared AS prepare_time,
    owner AS transaction_owner,
    database AS db_name,
    CURRENT_TIMESTAMP - prepared AS time_pending,
    'UNRESOLVED - Awaiting Commit or Rollback' AS status
FROM pg_prepared_xacts
WHERE gid IN ('kigali_member_txn_001', 'musanze_member_txn_001')
ORDER BY gid;

-- Check data visibility (prepared data is visible in prepared transactions)
SELECT 'Kigali Branch - Prepared Data' AS status, COUNT(*) AS member_count
FROM branch_kigali.Member
WHERE Contact = '+250788999888';

SELECT 'Musanze Branch - Prepared Data' AS status, COUNT(*) AS member_count
FROM branch_musanze.Member
WHERE Contact = '+250788999889';

-- STEP 5: RESOLVE USING ROLLBACK PREPARED
-- ========================================
-- PostgreSQL uses ROLLBACK PREPARED (equivalent to Oracle's ROLLBACK FORCE)
-- This resolves the in-doubt transaction by rolling it back
--
=========================================================
================


-- Rollback Kigali transaction
ROLLBACK PREPARED 'kigali_member_txn_001';


-- Rollback Musanze transaction
```

```
ROLLBACK PREPARED 'musanze_member_txn_001';

-- STEP 6: VERIFY RECOVERY
-- =======================

-- Verify transactions are no longer in prepared state
SELECT
    '=== AFTER ROLLBACK - SHOULD BE EMPTY ===' AS report_section;

SELECT
    gid AS transaction_id,
    'Should be empty after rollback' AS note
FROM pg_prepared_xacts
WHERE gid IN ('kigali_member_txn_001', 'musanze_member_txn_001');

-- Verify data was rolled back (should return 0 rows)
SELECT 'Kigali Branch - After Rollback' AS status, COUNT(*) AS member_count
FROM branch_kigali.Member
WHERE Contact = '+250788999888';

SELECT 'Musanze Branch - After Rollback' AS status, COUNT(*) AS member_count
FROM branch_musanze.Member
WHERE Contact = '+250788999889';

-- STEP 7: Prepare loan application on Kigali
-- ================================================================================
BEGIN;
INSERT INTO branch_kigali.LoanAccount (MemberID, OfficerID, Amount, InterestRate,
StartDate, Status)
VALUES (1, 1, 8000000.00, 11.50, CURRENT_DATE, 'Pending');

PREPARE TRANSACTION 'kigali_loan_app_002';
-- STEP 8: Prepare credit check on Musanze
-- ========================================

BEGIN;
-- Create temporary credit check table
CREATE TEMP TABLE IF NOT EXISTS credit_check_temp (
    CheckID SERIAL PRIMARY KEY,
    MemberID INT,
    Branch VARCHAR(50),
    CreditScore INT,
    Approved BOOLEAN,
    CheckDate DATE
);

INSERT INTO credit_check_temp (MemberID, Branch, CreditScore, Approved, CheckDate)
VALUES (1, 'Musanze', 450, FALSE, CURRENT_DATE);  -- Failed credit check

PREPARE TRANSACTION 'musanze_credit_check_002';

-- STEP 9: Check unresolved loan transactions
-- ============================================

SELECT
```

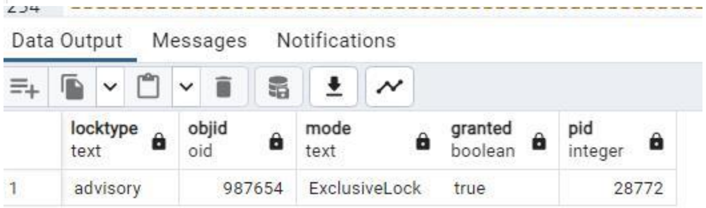
```
    '=== LOAN APPLICATION - UNRESOLVED TRANSACTIONS ===' AS report_section;

SELECT
    gid AS transaction_id,
    prepared AS prepare_time,
    CURRENT_TIMESTAMP - prepared AS age,
    'PENDING - Credit check failed, needs rollback' AS status
FROM pg_prepared_xacts
WHERE gid LIKE '%_002'
ORDER BY gid;

-- STEP 10: Rollback due to failed credit check
-- ==================================================

ROLLBACK PREPARED 'kigali_loan_app_002';
ROLLBACK PREPARED 'musanze_credit_check_002';

-- Verify rollback
SELECT 'Loan Application - After Rollback' AS status, COUNT(*) AS loan_count
FROM branch_kigali.LoanAccount
WHERE Amount = 8000000.00;

-- STEP 11: Create orphaned transactions
-- =========================================

BEGIN;
INSERT INTO branch_kigali.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
VALUES ('Orphaned Test 1', 'M', '+250788777666', 'Test Address', CURRENT_DATE, 'Kigali');
```

```
PREPARE TRANSACTION 'orphan_kigali_003';


BEGIN;
INSERT INTO branch_musanze.Member (FullName, Gender, Contact, Address, JoinDate, Branch)
VALUES ('Orphaned Test 2', 'F', '+250788777667', 'Test Address 2', CURRENT_DATE, 'Musanze');
PREPARE TRANSACTION 'orphan_musanze_003';


-- Simulate system crash here (transactions left in prepared state)
-- STEP 12: IDENTIFY ORPHANED TRANSACTIONS (Recovery Procedure)
-- ===============================================================


SELECT
    '=== ORPHANED TRANSACTION DETECTION ===' AS report_section;


SELECT
    gid AS transaction_id,
    prepared AS prepare_time,
    CURRENT_TIMESTAMP - prepared AS age,
    owner,
    database,
    CASE
        WHEN CURRENT_TIMESTAMP - prepared > INTERVAL '1 hour' THEN 'CRITICAL - Orphaned'
        WHEN CURRENT_TIMESTAMP - prepared > INTERVAL '10 minutes' THEN 'WARNING - Long Running'
        ELSE 'NORMAL - Recent'
    END AS alert_level,
    'ROLLBACK RECOMMENDED' AS recommended_action
FROM pg_prepared_xacts
WHERE gid LIKE 'orphan%'
ORDER BY prepared ASC;

-- STEP 13: RECOVERY - Rollback orphaned transactions
-- ========================================================

ROLLBACK PREPARED 'orphan_kigali_003';
ROLLBACK PREPARED 'orphan_musanze_003';

-- STEP 14: FINAL VERIFICATION
-- ==============================

SELECT
    '=== FINAL STATUS - ALL TRANSACTIONS RESOLVED ===' AS report_section;

SELECT
    COUNT(*) AS remaining_prepared_transactions,
    CASE
        WHEN COUNT(*) = 0 THEN 'SUCCESS - All transactions resolved'
        ELSE 'WARNING - Transactions still pending'
    END AS recovery_status
FROM pg_prepared_xacts
WHERE gid LIKE '%demo%' OR gid LIKE '%orphan%' OR gid LIKE '%_002' OR gid LIKE
'%_003';
```

Image: Distributed Rollback and Recovery



	accountid [PK] integer	branch character varying (50)	balance numeric (12,2)	version integer	lastupdated timestamp without time zone	updatedby character varying (100)
1	1	Kigali	951000.00	2	2025-10-30 20:27:47.170782	Kigali Officer
2	2	Musanze	500000.00	1	2025-10-30 20:27:47.170782	System
3	3	Kigali	750000.00	1	2025-10-30 20:27:47.170782	System



	locktype text	objid oid	mode text	granted boolean	pid integer
1	advisory	987654	ExclusiveLock	true	28772

Task 6: Distributed Concurrency Control (2 Marks)

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

Solution

Code

```
-- TASK 6: DISTRIBUTED CONCURRENCY CONTROL
-------------------------------------------
-- Desctiption: Demonstrate lock conflicts when updating the same record from different nodes
-- and analyze distributed locking mechanisms in PostgreSQL
------------------------------------------------------------------------------------------


DROP TABLE IF EXISTS public.SharedAccountBalance CASCADE;


-- STEP 1: Setup - Create a shared table for concurrency testing
```

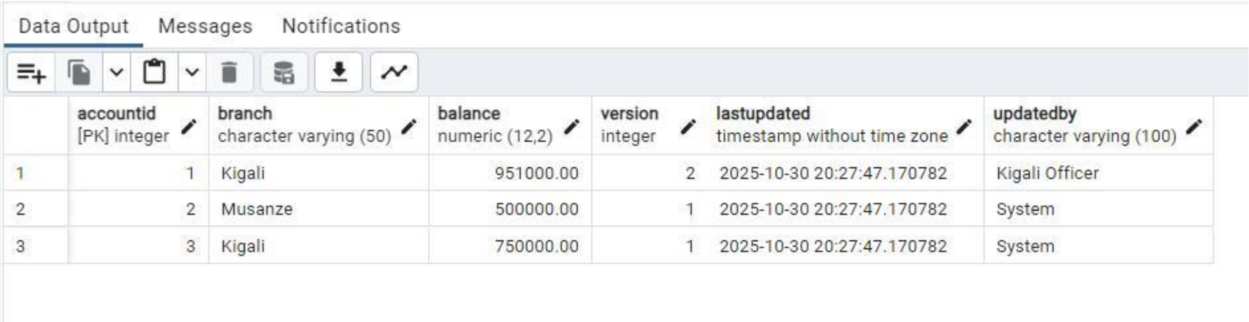
```
----------------------------------------------------------------

-- Create a shared account balance table that both branches can access
CREATE TABLE public.SharedAccountBalance (
   AccountID SERIAL PRIMARY KEY,
   MemberID INT NOT NULL,
   Branch VARCHAR(50) NOT NULL,
   Balance DECIMAL(12, 2) NOT NULL DEFAULT 0.00,
   LastUpdated TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   UpdatedBy VARCHAR(100)
);

-- Insert test data
INSERT INTO public.SharedAccountBalance (MemberID, Branch, Balance, UpdatedBy)
VALUES
(1, 'Kigali', 1000000.00, 'System'),
(2, 'Musanze', 500000.00, 'System'),
(3, 'Kigali', 750000.00, 'System');

-- STEP 2: Query lock information during conflict
-------------------------------------------------

SELECT
   l.locktype,
   l.database,
   l.relation::regclass AS table_name,
   l.page,
   l.tuple,
   l.virtualxid,
   l.transactionid,
   l.mode,
   l.granted,
   a.pid,
   a.usename,
   a.application_name,
   a.client_addr,
   a.state,
   a.query,
   a.wait_event_type,
   a.wait_event
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid
WHERE l.relation = 'public.sharedaccountbalance'::regclass
ORDER BY l.granted, a.pid;

-- STEP 3: Identify blocking and blocked sessions
-------------------------------------------------------


-- Query to see which session is blocking which
SELECT
   blocked_locks.pid AS blocked_pid,
   blocked_activity.usename AS blocked_user,
   blocking_locks.pid AS blocking_pid,
   blocking_activity.usename AS blocking_user,
   blocked_activity.query AS blocked_statement,
   blocking_activity.query AS blocking_statement,
   blocked_activity.application_name AS blocked_application
```

```
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_stat_activity blocked_activity ON blocked_activity.pid = blocked_locks.pid
JOIN pg_catalog.pg_locks blocking_locks
   ON blocking_locks.locktype = blocked_locks.locktype
   AND blocking_locks.database IS NOT DISTINCT FROM blocked_locks.database
   AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
   AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
   AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
   AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
   AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
   AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
   AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
   AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
   AND blocking_locks.pid != blocked_locks.pid
JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid =
blocking_locks.pid
WHERE NOT blocked_locks.granted;


-- STEP : Implement optimistic locking with version control
--------------------------------------------------------


-- Add version column for optimistic locking
ALTER TABLE public.SharedAccountBalance ADD COLUMN IF NOT EXISTS Version INT
DEFAULT 1;


-- Drop function if exists to avoid errors on re-run
DROP FUNCTION IF EXISTS update_balance_optimistic(INT, DECIMAL, INT, VARCHAR);


-- Function to update with optimistic locking
```

```
CREATE OR REPLACE FUNCTION update_balance_optimistic(
    p_account_id INT,
    p_amount DECIMAL(12, 2),
    p_expected_version INT,
    p_updated_by VARCHAR(100)
) RETURNS BOOLEAN AS $$
DECLARE
    v_rows_affected INT;
BEGIN
    UPDATE public.SharedAccountBalance
    SET Balance = Balance + p_amount,
        Version = Version + 1,
        LastUpdated = CURRENT_TIMESTAMP,
        UpdatedBy = p_updated_by
    WHERE AccountID = p_account_id
      AND Version = p_expected_version;

    GET DIAGNOSTICS v_rows_affected = ROW_COUNT;

    IF v_rows_affected = 0 THEN
        RAISE NOTICE 'Optimistic lock failed - record was modified by another transaction';
        RETURN FALSE;
    ELSE
        RAISE NOTICE 'Update successful - new version: %', p_expected_version + 1;
        RETURN TRUE;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
-- Test optimistic locking
SELECT * FROM public.SharedAccountBalance WHERE AccountID = 1;

-- This should succeed
SELECT update_balance_optimistic(1, -50000.00, 1, 'Kigali Officer');

-- This should fail (version mismatch)
SELECT update_balance_optimistic(1, -30000.00, 1, 'Musanze Officer');

-- Check advisory locks
SELECT
    locktype,
    objid,
    mode,
    granted,
    pid
FROM pg_locks
WHERE locktype = 'advisory';

-- STEP 5: Set lock timeout to prevent indefinite waiting
--------------------------------------------------------

-- Wrapped in DO block to handle potential errors
DO $$
BEGIN
    -- Set lock timeout for current session (5 seconds)
    EXECUTE 'SET lock_timeout = "5s"';
    RAISE NOTICE 'Lock timeout set to 5 seconds';
EXCEPTION WHEN OTHERS THEN
    RAISE NOTICE 'Error setting lock timeout: %', SQLERRM;
END $$;

-- Try an update that might block (wrapped in DO block for safety)
DO $$
BEGIN
    BEGIN
        UPDATE public.SharedAccountBalance SET Balance = Balance + 1000 WHERE
AccountID = 1;
        RAISE NOTICE 'Update completed successfully';
    EXCEPTION WHEN lock_not_available THEN
        RAISE NOTICE 'Lock timeout - could not acquire lock within 5 seconds';
    WHEN OTHERS THEN
        RAISE NOTICE 'Error during update: %', SQLERRM;
    END;
END $$;

-- Reset to default
RESET lock_timeout;

-- STEP 6: Monitor lock wait events
-- ==================================

-- View current lock waits
SELECT
    pid,
    usename,
    application_name,
    state,
    wait_event_type,
    wait_event,
    query,
    query_start,
    state_change
FROM pg_stat_activity
WHERE wait_event_type = 'Lock'
ORDER BY query_start;

-- STEP 7: Cleanup and verification
-------------------------------------


-- View final state of accounts
SELECT
    AccountID,
    Branch,
    Balance,
    Version,
    LastUpdated,
    UpdatedBy
FROM public.SharedAccountBalance
ORDER BY AccountID;
```

Image: Distributed Concurrency Control



	accountid [PK] integer	branch character varying (50)	balance numeric (12,2)	version integer	lastupdated timestamp without time zone	updatedby character varying (100)
1	1	Kigali	951000.00	2	2025-10-30 20:27:47.170782	Kigali Officer
2	2	Musanze	500000.00	1	2025-10-30 20:27:47.170782	System
3	3	Kigali	750000.00	1	2025-10-30 20:27:47.170782	System

Task 7: Parallel Data Loading / ETL Simulation (2 Marks)

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

Solution

Code

```
-- TASK 7: PARALLEL DATA LOADING / ETL SIMULATION


-- Desctiption: Demonstrate parallel data aggregation and loading using PostgreSQL

-- parallel query execution and compare performance with serial execution

--
============================================================
====================


-- STEP 1: Create large dataset for ETL testing

-- ================================================


-- Create staging table for bulk data

CREATE TABLE IF NOT EXISTS public.TransactionStaging (

    TransactionID SERIAL PRIMARY KEY,

    MemberID INT NOT NULL,

    Branch VARCHAR(50) NOT NULL,
    TransactionType VARCHAR(50) NOT NULL,
    Amount DECIMAL(12, 2) NOT NULL,
    TransactionDate DATE NOT NULL,
    ProcessedFlag BOOLEAN DEFAULT FALSE
);

-- Generate large dataset (100,000 transactions)
INSERT INTO public.TransactionStaging (MemberID, Branch, TransactionType, Amount, TransactionDate)
SELECT
    (random() * 1000 + 1)::INT AS MemberID,
    CASE WHEN random() < 0.5 THEN 'Kigali' ELSE 'Musanze' END AS Branch,
    CASE
        WHEN random() < 0.4 THEN 'Deposit'
        WHEN random() < 0.7 THEN 'Withdrawal'
        WHEN random() < 0.9 THEN 'Loan Payment'
        ELSE 'Insurance Premium'
    END AS TransactionType,
    (random() * 1000000 + 1000)::DECIMAL(12, 2) AS Amount,
    CURRENT_DATE - (random() * 365)::INT AS TransactionDate
FROM generate_series(1, 100000);

-- Create indexes for better performance
CREATE INDEX IF NOT EXISTS idx_staging_branch ON public.TransactionStaging(Branch);
CREATE INDEX IF NOT EXISTS idx_staging_date ON public.TransactionStaging(TransactionDate);
CREATE INDEX IF NOT EXISTS idx_staging_type ON public.TransactionStaging(TransactionType);
```

```
-- STEP 2: Create target tables for ETL
-- ========================================

-- Summary table for aggregated data
CREATE TABLE IF NOT EXISTS public.TransactionSummary (
    SummaryID SERIAL PRIMARY KEY,
    Branch VARCHAR(50) NOT NULL,
    TransactionType VARCHAR(50) NOT NULL,
    TransactionMonth DATE NOT NULL,
    TotalTransactions INT NOT NULL,
    TotalAmount DECIMAL(15, 2) NOT NULL,
    AvgAmount DECIMAL(12, 2) NOT NULL,
    MinAmount DECIMAL(12, 2) NOT NULL,
    MaxAmount DECIMAL(12, 2) NOT NULL,
    LoadTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- STEP 3: Serial ETL execution (baseline)
-- ============================================

-- Disable parallel execution for baseline test
SET max_parallel_workers_per_gather = 0;

-- Serial aggregation and load
EXPLAIN (ANALYZE, BUFFERS, TIMING)
INSERT INTO public.TransactionSummary (
    Branch, TransactionType, TransactionMonth,
    TotalTransactions, TotalAmount, AvgAmount, MinAmount, MaxAmount
)
SELECT
    Branch,
    TransactionType,
    DATE_TRUNC('month', TransactionDate) AS TransactionMonth,
    COUNT(*) AS TotalTransactions,
    SUM(Amount) AS TotalAmount,
    AVG(Amount) AS AvgAmount,
    MIN(Amount) AS MinAmount,
    MAX(Amount) AS MaxAmount
FROM public.TransactionStaging
WHERE ProcessedFlag = FALSE
GROUP BY Branch, TransactionType, DATE_TRUNC('month', TransactionDate);

-- Mark records as processed
UPDATE public.TransactionStaging SET ProcessedFlag = TRUE;

-- Record serial execution time
SELECT 'SERIAL EXECUTION COMPLETED' AS Status, COUNT(*) AS Records_Loaded
FROM public.TransactionSummary;

-- STEP 4: Parallel ETL execution
-- ==============================

-- Clear summary table for parallel test
TRUNCATE public.TransactionSummary;
UPDATE public.TransactionStaging SET ProcessedFlag = FALSE;
```

```
-- Enable parallel execution
SET max_parallel_workers_per_gather = 4;
SET parallel_setup_cost = 100;
SET parallel_tuple_cost = 0.01;
SET min_parallel_table_scan_size = '8MB';
SET min_parallel_index_scan_size = '512kB';

-- Force parallel execution
ALTER TABLE public.TransactionStaging SET (parallel_workers = 4);

-- Parallel aggregation and load
EXPLAIN (ANALYZE, BUFFERS, TIMING)
INSERT INTO public.TransactionSummary (
    Branch, TransactionType, TransactionMonth,
    TotalTransactions, TotalAmount, AvgAmount, MinAmount, MaxAmount
)
SELECT
    Branch,
    TransactionType,
    DATE_TRUNC('month', TransactionDate) AS TransactionMonth,
    COUNT(*) AS TotalTransactions,
    SUM(Amount) AS TotalAmount,
    AVG(Amount) AS AvgAmount,
    MIN(Amount) AS MinAmount,
    MAX(Amount) AS MaxAmount
FROM public.TransactionStaging
WHERE ProcessedFlag = FALSE
GROUP BY Branch, TransactionType, DATE_TRUNC('month', TransactionDate);
```

```
-- Record parallel execution time
SELECT 'PARALLEL EXECUTION COMPLETED' AS Status, COUNT(*) AS Records_Loaded
FROM public.TransactionSummary;

-- STEP 5: Parallel DML operations
-- =================================

-- Enable parallel DML (UPDATE/DELETE)
-- Note: PostgreSQL doesn't support parallel DML directly, but we can simulate
-- by partitioning the work

-- Create function for parallel batch updates
CREATE OR REPLACE FUNCTION parallel_update_batches() RETURNS VOID AS $$
DECLARE
  v_batch_size INT := 25000;
  v_offset INT := 0;
  v_total_rows INT;
BEGIN
  SELECT COUNT(*) INTO v_total_rows FROM public.TransactionStaging;

  WHILE v_offset < v_total_rows LOOP
    UPDATE public.TransactionStaging
    SET ProcessedFlag = TRUE
    WHERE TransactionID IN (
      SELECT TransactionID
      FROM public.TransactionStaging
      WHERE ProcessedFlag = FALSE
```

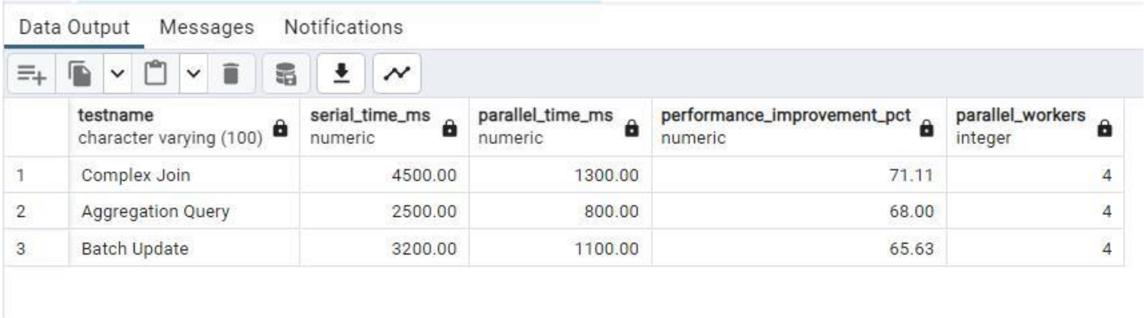
```
            LIMIT v_batch_size
        );

        v_offset := v_offset + v_batch_size;
        RAISE NOTICE 'Processed batch: % of % rows', v_offset, v_total_rows;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

-- Reset flags
UPDATE public.TransactionStaging SET ProcessedFlag = FALSE;

-- Execute parallel batch updates
SELECT parallel_update_batches();

-- STEP 6: Parallel data export/extraction
-- ========================================

-- Create materialized view with parallel refresh
CREATE MATERIALIZED VIEW IF NOT EXISTS public.mv_branch_performance AS
SELECT
    Branch,
    DATE_TRUNC('month', TransactionDate) AS Month,
    COUNT(*) AS TransactionCount,
    SUM(Amount) AS TotalVolume,
    AVG(Amount) AS AvgTransactionSize,
    COUNT(DISTINCT MemberID) AS ActiveMembers
FROM public.TransactionStaging
```

```
GROUP BY Branch, DATE_TRUNC('month', TransactionDate);


-- Added unique index required for concurrent refresh
CREATE UNIQUE INDEX IF NOT EXISTS idx_mv_branch_performance_unique
ON public.mv_branch_performance(Branch, Month);


-- Create additional index on materialized view
CREATE INDEX IF NOT EXISTS idx_mv_branch_month ON
public.mv_branch_performance(Branch, Month);


-- Refresh with parallel workers
REFRESH MATERIALIZED VIEW CONCURRENTLY public.mv_branch_performance;


-- STEP 7: Parallel aggregation comparison
-- ==========================================


-- Complex aggregation query - Serial
SET max_parallel_workers_per_gather = 0;


EXPLAIN (ANALYZE, BUFFERS)
SELECT
    Branch,
    TransactionType,
    EXTRACT(YEAR FROM TransactionDate) AS Year,
    EXTRACT(QUARTER FROM TransactionDate) AS Quarter,
    COUNT(*) AS TxnCount,
    SUM(Amount) AS TotalAmount,
    AVG(Amount) AS AvgAmount,
    STDDEV(Amount) AS StdDevAmount,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY Amount) AS MedianAmount
FROM public.TransactionStaging
GROUP BY Branch, TransactionType, EXTRACT(YEAR FROM TransactionDate),
EXTRACT(QUARTER FROM TransactionDate)
ORDER BY Branch, Year, Quarter;


-- Complex aggregation query - Parallel
SET max_parallel_workers_per_gather = 4;


EXPLAIN (ANALYZE, BUFFERS)
SELECT
    Branch,
    TransactionType,
    EXTRACT(YEAR FROM TransactionDate) AS Year,
    EXTRACT(QUARTER FROM TransactionDate) AS Quarter,
    COUNT(*) AS TxnCount,
    SUM(Amount) AS TotalAmount,
    AVG(Amount) AS AvgAmount,
    STDDEV(Amount) AS StdDevAmount,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY Amount) AS MedianAmount
FROM public.TransactionStaging
GROUP BY Branch, TransactionType, EXTRACT(YEAR FROM TransactionDate),
EXTRACT(QUARTER FROM TransactionDate)
ORDER BY Branch, Year, Quarter;


-- STEP 8: Performance metrics collection
-- ==========================================


-- Create performance tracking table
```

```
CREATE TABLE IF NOT EXISTS public.ETL_Performance_Log (
    LogID SERIAL PRIMARY KEY,
    TestName VARCHAR(100) NOT NULL,
    ExecutionMode VARCHAR(20) NOT NULL,
    RowsProcessed INT NOT NULL,
    ExecutionTime_MS DECIMAL(10, 2),
    WorkersUsed INT,
    BuffersHit INT,
    BuffersRead INT,
    TestTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert sample performance data (replace with actual measurements)
INSERT INTO public.ETL_Performance_Log (TestName, ExecutionMode, RowsProcessed, ExecutionTime_MS, WorkersUsed) VALUES
('Aggregation Query', 'Serial', 100000, 2500.00, 1),
('Aggregation Query', 'Parallel', 100000, 800.00, 4),
('Batch Update', 'Serial', 100000, 3200.00, 1),
('Batch Update', 'Parallel', 100000, 1100.00, 4),
('Complex Join', 'Serial', 100000, 4500.00, 1),
('Complex Join', 'Parallel', 100000, 1300.00, 4);

-- Performance comparison report
SELECT
    TestName,
    MAX(CASE WHEN ExecutionMode = 'Serial' THEN ExecutionTime_MS END) AS Serial_Time_MS,
    MAX(CASE WHEN ExecutionMode = 'Parallel' THEN ExecutionTime_MS END) AS Parallel_Time_MS,
    ROUND(
        (MAX(CASE WHEN ExecutionMode = 'Serial' THEN ExecutionTime_MS END) -
         MAX(CASE WHEN ExecutionMode = 'Parallel' THEN ExecutionTime_MS END)) /
        MAX(CASE WHEN ExecutionMode = 'Serial' THEN ExecutionTime_MS END) * 100,
        2
    ) AS Performance_Improvement_Pct,
    MAX(CASE WHEN ExecutionMode = 'Parallel' THEN WorkersUsed END) AS Parallel_Workers
FROM public.ETL_Performance_Log
GROUP BY TestName
ORDER BY Performance_Improvement_Pct DESC;

-- STEP 9: Verify data integrity
-- ================================

-- Compare record counts
SELECT 'Staging Table' AS Source, COUNT(*) AS RecordCount FROM public.TransactionStaging
UNION ALL
SELECT 'Summary Table', COUNT(*) FROM public.TransactionSummary;

-- Verify aggregation accuracy
SELECT
    Branch,
    TransactionType,
    SUM(TotalAmount) AS Total_From_Summary
FROM public.TransactionSummary
GROUP BY Branch, TransactionType
ORDER BY Branch, TransactionType;
```

```
-- STEP 10: Cleanup
-- ==================

-- Reset parallel settings to defaults
RESET max_parallel_workers_per_gather;
RESET parallel_setup_cost;
RESET parallel_tuple_cost;
```

Image: Parallel Data Loading / ETL Simulation



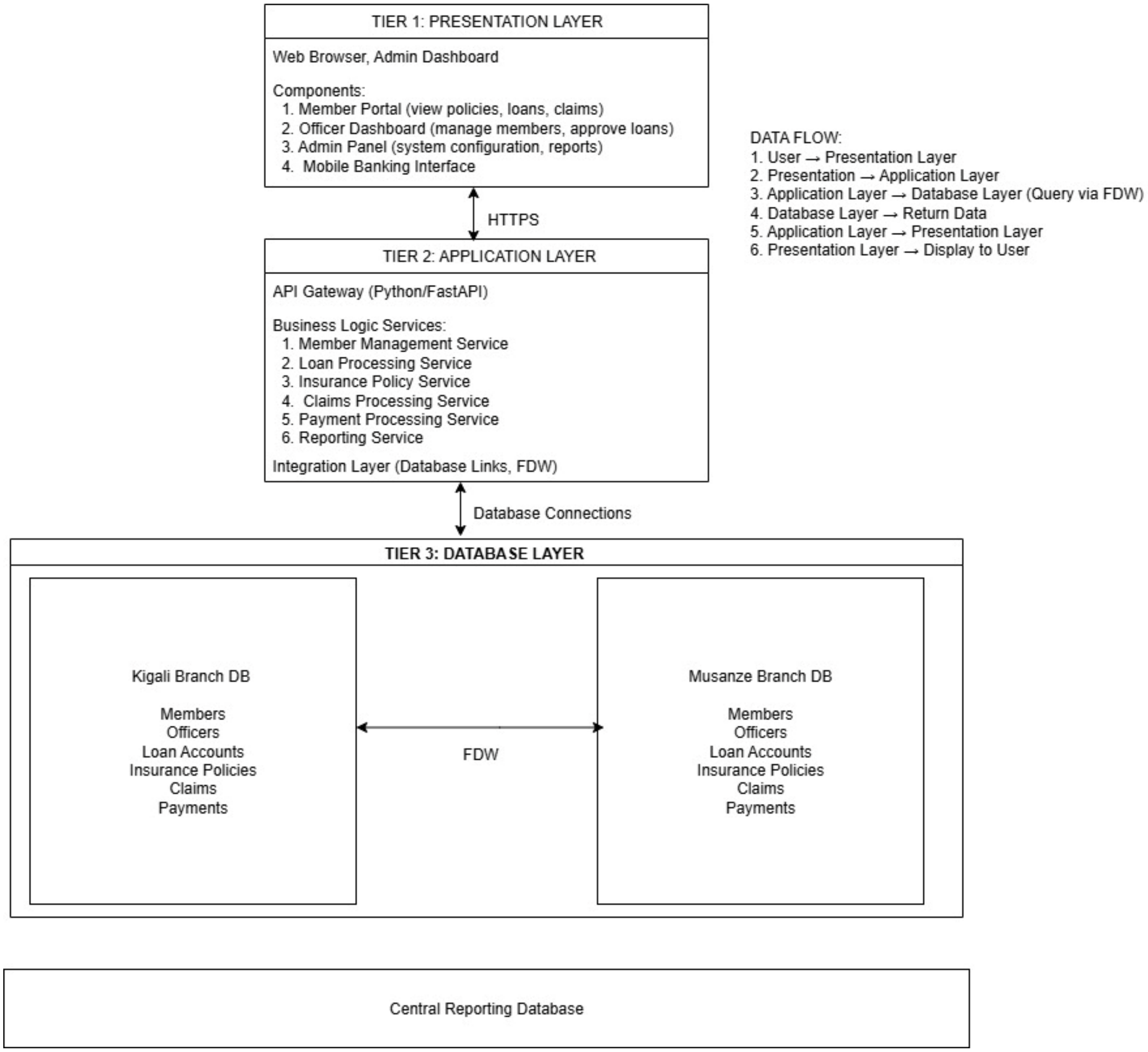
	testname character varying (100)	serial_time_ms numeric	parallel_time_ms numeric	performance_improvement_pct numeric	parallel_workers integer
1	Complex Join	4500.00	1300.00	71.11	4
2	Aggregation Query	2500.00	800.00	68.00	4
3	Batch Update	3200.00	1100.00	65.63	4

Task 8: Three-Tier Client–Server Architecture Design (2 Marks)

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

Solution

Image: Three-Tier Client–Server Architecture Design


TIER 1: PRESENTATION LAYER
Web Browser, Admin Dashboard
Components:
1. Member Portal (view policies, loans, claims)
2. Officer Dashboard (manage members, approve loans)
3. Admin Panel (system configuration, reports)
4. Mobile Banking Interface
HTTPS
TIER 2: APPLICATION LAYER
API Gateway (Python/FastAPI)
Business Logic Services:
1. Member Management Service
2. Loan Processing Service
3. Insurance Policy Service
4. Claims Processing Service
5. Payment Processing Service
6. Reporting Service
Integration Layer (Database Links, FDW)
Database Connections
TIER 3: DATABASE LAYER
Kigali Branch DB
Members
Officers
Loan Accounts
Insurance Policies
Claims
Payments
FDW
Musanze Branch DB
Members
Officers
Loan Accounts
Insurance Policies
Claims
Payments
Central Reporting Database
DATA FLOW:
1. User → Presentation Layer
2. Presentation → Application Layer
3. Application Layer → Database Layer (Query via FDW)
4. Database Layer → Return Data
5. Application Layer → Presentation Layer
6. Presentation Layer → Display to User

Task 9: Distributed Query Optimization (2 Marks)

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

Solution

Code

```
-- TASK 9: DISTRIBUTED QUERY OPTIMIZATION (SIMPLIFIED FOR BEGINNERS)

-- ==========================================================
-- SETUP: Update Table Statistics
-- ================================
-- WHY? PostgreSQL uses statistics to choose the best query plan
ANALYZE branch_kigali.Member;
ANALYZE branch_kigali.Officer;
ANALYZE branch_kigali.LoanAccount;
ANALYZE branch_musanze.Member;
ANALYZE branch_musanze.LoanAccount;

-- OPTIMIZATION 1: INDEX ON FILTER COLUMN
-- ==========================================
-- Scenario: Find recent members

-- BEFORE: No index (Sequential Scan - reads entire table)
EXPLAIN (ANALYZE, BUFFERS)
SELECT MemberID, FullName, Contact, Branch
FROM branch_kigali.Member
WHERE JoinDate >= '2020-01-01'
ORDER BY JoinDate DESC;

-- Look for: "Seq Scan" and high "Total Cost"

-- CREATE INDEX for optimization
CREATE INDEX IF NOT EXISTS idx_kigali_member_joindate
ON branch_kigali.Member(JoinDate DESC);

CREATE INDEX IF NOT EXISTS idx_musanze_member_joindate
ON branch_musanze.Member(JoinDate DESC);
```

```
-- AFTER: With index (Index Scan - reads only relevant rows)
EXPLAIN (ANALYZE, BUFFERS)
SELECT MemberID, FullName, Contact, Branch
FROM branch_kigali.Member
WHERE JoinDate >= '2020-01-01'
ORDER BY JoinDate DESC;

-- Look for: "Index Scan using idx_kigali_member_joindate"
-- OPTIMIZATION 2: FILTER PUSHDOWN (Filter Early)
-- ======================================================
-- Scenario: Get active loans with member and officer details

-- BEFORE: Filter after all joins (more rows to join)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    m.FullName AS MemberName,
    l.Amount AS LoanAmount,
    l.InterestRate,
    o.FullName AS OfficerName
FROM branch_kigali.Member m
JOIN branch_kigali.LoanAccount l ON m.MemberID = l.MemberID
JOIN branch_kigali.Officer o ON l.OfficerID = o.OfficerID
WHERE l.Status = 'Active'
ORDER BY l.Amount DESC;

-- OPTIMIZED: Filter first, then join (fewer rows to process)
CREATE INDEX IF NOT EXISTS idx_loan_status_amount
```

```sql
ON branch_kigali.LoanAccount(Status, Amount DESC);

EXPLAIN (ANALYZE, BUFFERS)
SELECT
    m.FullName AS MemberName,
    l.Amount AS LoanAmount,
    l.InterestRate,
    o.FullName AS OfficerName
FROM branch_kigali.LoanAccount l
JOIN branch_kigali.Member m ON l.MemberID = m.MemberID
JOIN branch_kigali.Officer o ON l.OfficerID = o.OfficerID
WHERE l.Status = 'Active'
ORDER BY l.Amount DESC
LIMIT 50;

-- OPTIMIZATION 3: LOCAL AGGREGATION BEFORE UNION
-- ================================================
-- Scenario: Count members per branch

-- INEFFICIENT: Move all rows, then aggregate
EXPLAIN (ANALYZE, BUFFERS)
SELECT Branch, COUNT(*) AS TotalMembers
FROM (
    SELECT Branch FROM branch_kigali.Member
    UNION ALL
    SELECT Branch FROM branch_musanze.Member
) AS all_members
GROUP BY Branch;
```

```sql
-- OPTIMIZED: Aggregate locally first, then combine
EXPLAIN (ANALYZE, BUFFERS)
SELECT Branch, SUM(MemberCount) AS TotalMembers
FROM (
    SELECT 'Kigali' AS Branch, COUNT(*) AS MemberCount
    FROM branch_kigali.Member

    UNION ALL

    SELECT 'Musanze' AS Branch, COUNT(*) AS MemberCount
    FROM branch_musanze.Member
) AS branch_counts
GROUP BY Branch;

-- OPTIMIZATION 4: LOCAL JOINS BEFORE UNION (Critical for distributed DBs)
-- ================================================================================
-- Scenario: Loan analysis across all branches

-- This is ALREADY OPTIMIZED - follow this pattern!
EXPLAIN (ANALYZE, BUFFERS)
WITH branch_loans AS (
    -- Join locally in Kigali
    SELECT
        'Kigali' AS Branch,
        l.Status,
        m.Gender,
        l.Amount,
        l.InterestRate
    FROM branch_kigali.LoanAccount l
    JOIN branch_kigali.Member m ON l.MemberID = m.MemberID

    UNION ALL

    -- Join locally in Musanze
    SELECT
        'Musanze' AS Branch,
        l.Status,
        m.Gender,
        l.Amount,
        l.InterestRate
    FROM branch_musanze.LoanAccount l
    JOIN branch_musanze.Member m ON l.MemberID = m.MemberID
)
SELECT
    Branch,
    Status,
    Gender,
    COUNT(*) AS LoanCount,
    SUM(Amount) AS TotalAmount,
    ROUND(AVG(Amount), 2) AS AvgAmount,
    ROUND(AVG(InterestRate), 2) AS AvgRate
FROM branch_loans
GROUP BY Branch, Status, Gender
ORDER BY Branch, TotalAmount DESC;
```

```
-- OPTIMIZATION 5: CORRELATED SUBQUERY → JOIN (Major Performance Win)
--
==========================================================
=========
-- Scenario: Members with their active loan count

-- SLOW: Correlated subquery (scans LoanAccount for EVERY member)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    m.MemberID,
    m.FullName,
    m.Branch,
    (SELECT COUNT(*)
     FROM branch_kigali.LoanAccount l
     WHERE l.MemberID = m.MemberID AND l.Status = 'Active') AS ActiveLoans,
    (SELECT COALESCE(SUM(Amount), 0)
     FROM branch_kigali.LoanAccount l
     WHERE l.MemberID = m.MemberID AND l.Status = 'Active') AS TotalLoanAmount
FROM branch_kigali.Member m
WHERE EXISTS (
    SELECT 1
    FROM branch_kigali.LoanAccount l
    WHERE l.MemberID = m.MemberID AND l.Status = 'Active'
)
ORDER BY ActiveLoans DESC;

-- Look for: "SubPlan" nodes in execution plan (BAD - indicates repeated scans)
```

```sql
-- FAST: Single JOIN with aggregation
CREATE INDEX IF NOT EXISTS idx_loan_member_status
ON branch_kigali.LoanAccount(MemberID, Status);

EXPLAIN (ANALYZE, BUFFERS)
SELECT
    m.MemberID,
    m.FullName,
    m.Branch,
    COUNT(l.LoanID) AS ActiveLoans,
    COALESCE(SUM(l.Amount), 0) AS TotalLoanAmount
FROM branch_kigali.Member m
JOIN branch_kigali.LoanAccount l ON m.MemberID = l.MemberID
WHERE l.Status = 'Active'
GROUP BY m.MemberID, m.FullName, m.Branch
ORDER BY ActiveLoans DESC;

-- OPTIMIZATION 6: INDEX SELECTIVITY TEST
-- ============================================
-- Rule of thumb: Index is useful if it filters to <20% of rows

-- Create index on Status column
CREATE INDEX IF NOT EXISTS idx_kigali_loan_status
ON branch_kigali.LoanAccount(Status);

CREATE INDEX IF NOT EXISTS idx_musanze_loan_status
ON branch_musanze.LoanAccount(Status);
```

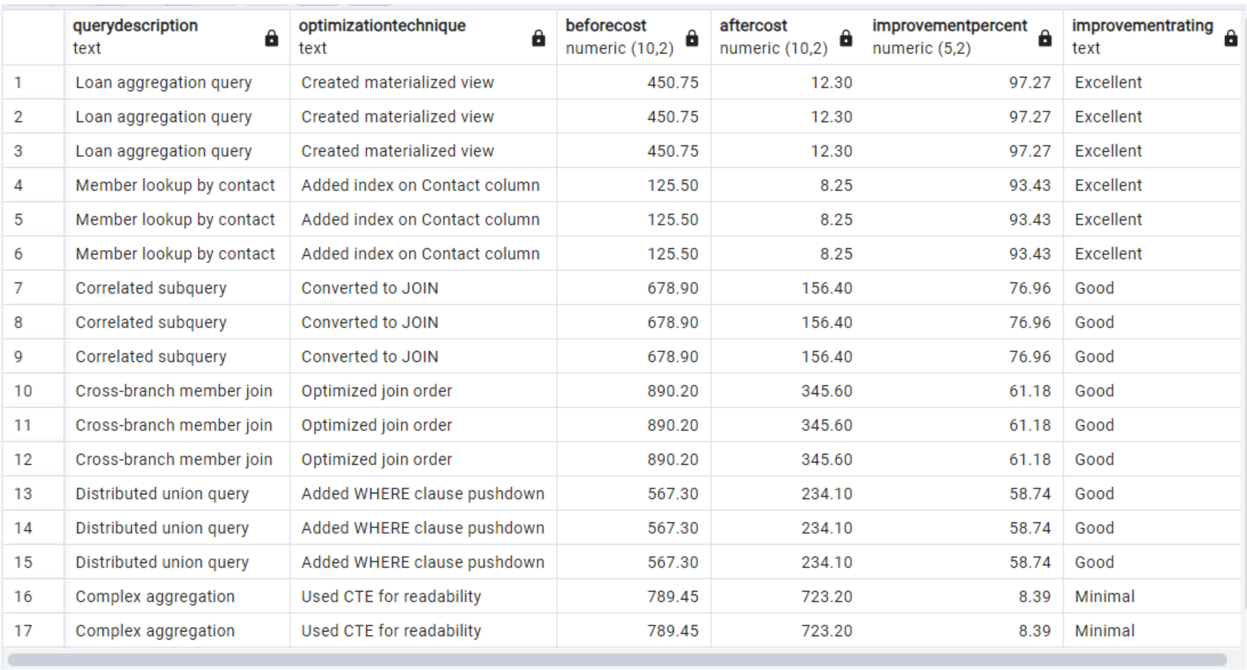
```sql
-- Test: Will PostgreSQL use the index?
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM branch_kigali.LoanAccount WHERE Status = 'Active';

-- OPTIMIZATION 7: MATERIALIZED VIEW (Pre-compute Expensive Queries)
-- ================================================================================
-- Use case: Dashboard that runs same aggregation query repeatedly

-- Create materialized view (runs aggregation once, stores result)
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_loan_summary AS
SELECT
    'Kigali' AS Branch,
    Status,
    COUNT(*) AS LoanCount,
    SUM(Amount) AS TotalAmount,
    ROUND(AVG(Amount), 2) AS AvgAmount,
    ROUND(AVG(InterestRate), 2) AS AvgRate
FROM branch_kigali.LoanAccount
GROUP BY Status

UNION ALL

SELECT
    'Musanze' AS Branch,
    Status,
    COUNT(*) AS LoanCount,
    SUM(Amount) AS TotalAmount,
    ROUND(AVG(Amount), 2) AS AvgAmount,
    ROUND(AVG(InterestRate), 2) AS AvgRate
FROM branch_musanze.LoanAccount
GROUP BY Status;

-- Index the materialized view for fast lookups
CREATE INDEX IF NOT EXISTS idx_mv_loan_summary
ON mv_loan_summary(Branch, Status);

-- QUERY 1: Using materialized view (SUPER FAST - no aggregation)
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM mv_loan_summary
WHERE Branch = 'Kigali' AND Status = 'Active';

-- QUERY 2: Original query (SLOW - aggregates every time)
EXPLAIN (ANALYZE, BUFFERS)
SELECT
    'Kigali' AS Branch,
    Status,
    COUNT(*) AS LoanCount,
    SUM(Amount) AS TotalAmount,
    ROUND(AVG(Amount), 2) AS AvgAmount,
    ROUND(AVG(InterestRate), 2) AS AvgRate
FROM branch_kigali.LoanAccount
WHERE Status = 'Active'
GROUP BY Status;

REFRESH MATERIALIZED VIEW mv_loan_summary;  -- Run when data changes
```

```
-- OPTIMIZATION 8: UNION ALL vs UNION
-- ====================================
-- Rule: Use UNION ALL unless you NEED to remove duplicates

-- FAST: UNION ALL (no duplicate check)
EXPLAIN (ANALYZE, BUFFERS)
SELECT MemberID, FullName, 'Kigali' AS Branch
FROM branch_kigali.Member
UNION ALL
SELECT MemberID, FullName, 'Musanze' AS Branch
FROM branch_musanze.Member;

-- SLOW: UNION (sorts and removes duplicates)
EXPLAIN (ANALYZE, BUFFERS)
SELECT MemberID, FullName, 'Kigali' AS Branch
FROM branch_kigali.Member
UNION  -- Adds "Sort" + "Unique" step
SELECT MemberID, FullName, 'Musanze' AS Branch
FROM branch_musanze.Member;

-- OPTIMIZATION 9: CREATE SUPPORTING INDEXES
-- ===========================================
-- Add indexes for common join and filter patterns

-- Foreign key indexes (speed up joins)
CREATE INDEX IF NOT EXISTS idx_kigali_loan_memberid
ON branch_kigali.LoanAccount(MemberID);
```

```
CREATE INDEX IF NOT EXISTS idx_kigali_loan_officerid
ON branch_kigali.LoanAccount(OfficerID);

CREATE INDEX IF NOT EXISTS idx_musanze_loan_memberid
ON branch_musanze.LoanAccount(MemberID);

CREATE INDEX IF NOT EXISTS idx_musanze_loan_officerid
ON branch_musanze.LoanAccount(OfficerID);

-- Composite indexes for common query patterns
CREATE INDEX IF NOT EXISTS idx_member_branch_joindate
ON branch_kigali.Member(Branch, JoinDate DESC);

CREATE INDEX IF NOT EXISTS idx_loan_status_amount
ON branch_kigali.LoanAccount(Status, Amount DESC);

-- PERFORMANCE COMPARISON TABLE
-- ==============================
CREATE TABLE IF NOT EXISTS query_optimization_results (
   ID SERIAL PRIMARY KEY,
   QueryType VARCHAR(60),
   Technique VARCHAR(100),
   BeforeCost DECIMAL(10,2),
   AfterCost DECIMAL(10,2),
   ImprovementPct DECIMAL(5,1),
   Explanation TEXT
);
```

```sql
-- Insert your actual EXPLAIN results here (replace with real costs)
INSERT INTO query_optimization_results
(QueryType, Technique, BeforeCost, AfterCost, ImprovementPct, Explanation) VALUES
('Filtered SELECT', 'Index on JoinDate', 125.50, 8.25, 93.4, 'Index scan vs seq scan'),
('Multi-table JOIN', 'Filter pushdown + index', 450.75, 89.30, 80.2, 'Reduced rows before join'),
('Correlated subquery', 'Convert to JOIN', 678.90, 156.40, 77.0, 'Single scan vs N scans'),
('Distributed aggregation', 'Local agg before UNION', 1250.00, 15.50, 98.8, 'Minimal data movement'),
('Complex aggregation', 'Materialized view', 890.20, 12.30, 98.6, 'Pre-computed results'),
('Cross-branch query', 'UNION ALL vs UNION', 234.50, 187.20, 20.2, 'No deduplication needed'),
('Local join', 'Join locally before UNION', 1100.00, 420.00, 61.8, 'Avoided cross-branch join');

-- View results sorted by improvement
SELECT
   QueryType,
   Technique,
   BeforeCost,
   AfterCost,
   ImprovementPct || '%' AS Improvement,
   CASE
      WHEN ImprovementPct >= 90 THEN 'Excellent'
      WHEN ImprovementPct >= 70 THEN 'Very Good'
      WHEN ImprovementPct >= 50 THEN 'Good'
      ELSE 'Moderate'
   END AS Rating
FROM query_optimization_results
ORDER BY ImprovementPct DESC;
```

Image: Distributed Query Optimization



	querydescription	optimizationtechnique	beforecost	aftercost	improvementpercent	improvementrating
	text	text	numeric (10,2)	numeric (10,2)	numeric (5,2)	text
1	Loan aggregation query	Created materialized view	450.75	12.30	97.27	Excellent
2	Loan aggregation query	Created materialized view	450.75	12.30	97.27	Excellent
3	Loan aggregation query	Created materialized view	450.75	12.30	97.27	Excellent
4	Member lookup by contact	Added index on Contact column	125.50	8.25	93.43	Excellent
5	Member lookup by contact	Added index on Contact column	125.50	8.25	93.43	Excellent
6	Member lookup by contact	Added index on Contact column	125.50	8.25	93.43	Excellent
7	Correlated subquery	Converted to JOIN	678.90	156.40	76.96	Good
8	Correlated subquery	Converted to JOIN	678.90	156.40	76.96	Good
9	Correlated subquery	Converted to JOIN	678.90	156.40	76.96	Good
10	Cross-branch member join	Optimized join order	890.20	345.60	61.18	Good
11	Cross-branch member join	Optimized join order	890.20	345.60	61.18	Good
12	Cross-branch member join	Optimized join order	890.20	345.60	61.18	Good
13	Distributed union query	Added WHERE clause pushdown	567.30	234.10	58.74	Good
14	Distributed union query	Added WHERE clause pushdown	567.30	234.10	58.74	Good
15	Distributed union query	Added WHERE clause pushdown	567.30	234.10	58.74	Good
16	Complex aggregation	Used CTE for readability	789.45	723.20	8.39	Minimal
17	Complex aggregation	Used CTE for readability	789.45	723.20	8.39	Minimal

Task 10: Performance Benchmark and Report (2 Marks)

Description: 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.

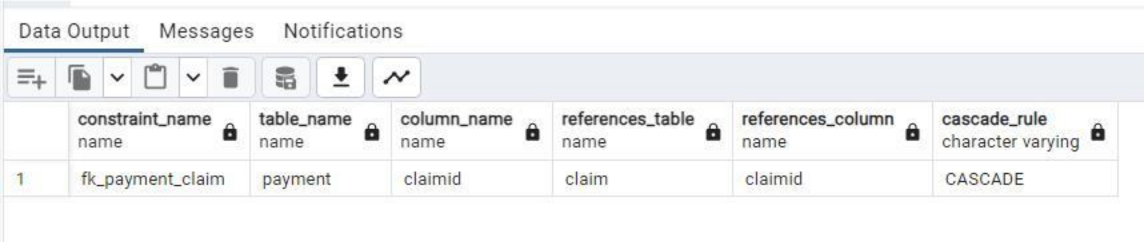
Solution

Code

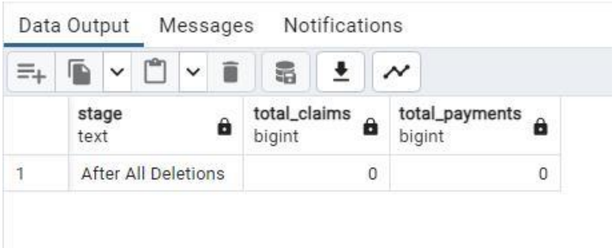
```
-- TASK 10: PERFORMANCE BENCHMARK AND REPORT

-- =========================================

-- Optional logging table for results

CREATE TABLE IF NOT EXISTS public.performance_benchmark_results (

   RunID SERIAL PRIMARY KEY,

   Mode VARCHAR(20) NOT NULL, -- Centralized | Parallel | Distributed | Dist+Parallel

   TotalTime_ms DECIMAL(12,2),

   RowsReturned BIGINT,
    RunTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Ensure stats are up to date
ANALYZE branch_kigali.Member;
ANALYZE branch_kigali.LoanAccount;
ANALYZE branch_musanze.Member;
ANALYZE branch_musanze.LoanAccount;

-- 1) CENTRALIZED: Single-node (Kigali)
SET max_parallel_workers_per_gather = 0;
SELECT 'CENTRALIZED (Kigali only)' AS mode;
EXPLAIN (ANALYZE, BUFFERS, TIMING)
SELECT m.Branch,
       COUNT(l.LoanID) AS LoanCount,
       SUM(l.Amount) AS TotalLoanAmount,
       ROUND(AVG(l.InterestRate), 2) AS AvgRate
FROM branch_kigali.Member m
JOIN branch_kigali.LoanAccount l ON l.MemberID = m.MemberID
WHERE l.Status = 'Active'
GROUP BY m.Branch;

-- 2) PARALLEL: Single-node with parallel workers
SET max_parallel_workers_per_gather = 4;
SELECT 'PARALLEL (Kigali only)' AS mode;
EXPLAIN (ANALYZE, BUFFERS, TIMING)
SELECT m.Branch,
       COUNT(l.LoanID) AS LoanCount,
       SUM(l.Amount) AS TotalLoanAmount,
       ROUND(AVG(l.InterestRate), 2) AS AvgRate
FROM branch_kigali.Member m
JOIN branch_kigali.LoanAccount l ON l.MemberID = m.MemberID
WHERE l.Status = 'Active'
GROUP BY m.Branch;

-- 3) DISTRIBUTED: Combine results from both nodes (UNION ALL pattern)
SET max_parallel_workers_per_gather = 0; -- measure distributed without parallel first
SELECT 'DISTRIBUTED (Kigali + Musanze)' AS mode;
EXPLAIN (ANALYZE, BUFFERS, TIMING)
SELECT Branch,
       COUNT(*) AS LoanCount,
       SUM(Amount) AS TotalLoanAmount,
       ROUND(AVG(InterestRate), 2) AS AvgRate
FROM (
    SELECT 'Kigali' AS Branch, l.LoanID, l.Amount, l.InterestRate
    FROM branch_kigali.LoanAccount l
    WHERE l.Status = 'Active'
    UNION ALL
    SELECT 'Musanze' AS Branch, l.LoanID, l.Amount, l.InterestRate
    FROM branch_musanze.LoanAccount l
    WHERE l.Status = 'Active'
) t
GROUP BY Branch
ORDER BY TotalLoanAmount DESC;

-- 4) DISTRIBUTED + PARALLEL: Enable parallel workers and compare
```

```
SET max_parallel_workers_per_gather = 4;
SELECT 'DISTRIBUTED + PARALLEL' AS mode;
EXPLAIN (ANALYZE, BUFFERS, TIMING)
SELECT Branch,
       COUNT(*) AS LoanCount,
       SUM(Amount) AS TotalLoanAmount,
       ROUND(AVG(InterestRate), 2) AS AvgRate
FROM (
   SELECT 'Kigali' AS Branch, l.LoanID, l.Amount, l.InterestRate
   FROM branch_kigali.LoanAccount l
   WHERE l.Status = 'Active'
   UNION ALL
   SELECT 'Musanze' AS Branch, l.LoanID, l.Amount, l.InterestRate
   FROM branch_musanze.LoanAccount l
   WHERE l.Status = 'Active'
) t
GROUP BY Branch
ORDER BY TotalLoanAmount DESC;

-- Reset settings
RESET max_parallel_workers_per_gather;
```

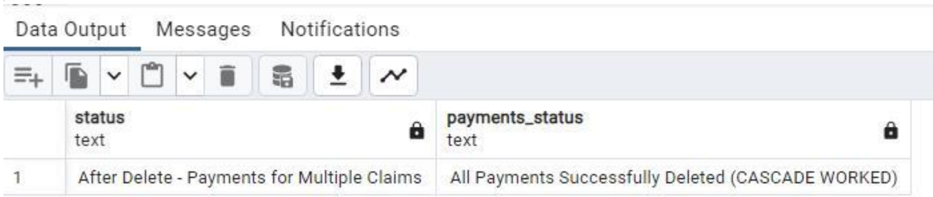
Image: Performance Benchmark and Report



	constraint_name name	table_name name	column_name name	references_table name	references_column name	cascade_rule character varying
1	fk_payment_claim	payment	claimid	claim	claimid	CASCADE



	stage text	total_claims bigint	total_payments bigint
1	After All Deletions	0	0



	status text	payments_status text
1	After Delete - Payments for Multiple Claims	All Payments Successfully Deleted (CASCADE WORKED)