## Project Report: Distributed Point of Sale (POS) System

### 1. Introduction & Problem Statement

#### 1.1 Background of the Domain

This project aims to design and implement a **Distributed Point of Sale (POS) System** for a retail business with stores in multiple cities (e.g., Karachi and Lahore). The system is responsible for managing sales, inventory, and customer data at each store location.

#### 1.2 Problem Statement

Using a single, centralized database for a retail chain with geographically dispersed stores introduces several significant challenges:

* **Single Point of Failure**: If the central database goes offline, operations at all store locations will halt, leading to major business disruption.
* **High Latency**: A cashier in Lahore processing a transaction would have to wait for the query to travel to a central server (e.g., in Karachi) and back. This delay results in a slow system and a poor customer experience.
* **Network Dependency**: System performance becomes entirely dependent on the quality and stability of the network connection between the stores and the central server.

To overcome these issues, a **Distributed Database (DDB) System** is required. This architecture improves speed, reliability, and availability by placing data closer to the locations where it is most frequently used.

2. System Design & Architecture

#### 2.1 ER Diagram and Schema Design

The system's design revolves around a central database and several local store databases.

* **Entities**: The core entities in the system are: Products, Categories, Customers, Stores, Cities, Orders, Order\_Items, Payments, and Inventory.
* **Relationships**:
  + A Store is located in one City.
  + A Customer can place multiple Orders.
  + An Order consists of multiple Order\_Items.
  + Each Product belongs to one Category.
* **Schema**:
  + **Central Database (pos\_central)**: This database holds master data such as Products, Categories, and Cities, along with global records for Customers and Stores. It also manages the Order\_Mapping and Replication\_Log tables.
  + **Store Databases (karachi\_db, lahore\_db)**: These databases store data related to their local operations, including Orders, Order\_Items, Payments, and Inventory. They also hold replicated copies of relevant Customers and Stores data.

2.2 Fragmentation Strategy

A **Hybrid Fragmentation** approach has been implemented:

* **Horizontal Fragmentation**: The Customers and Stores tables are horizontally fragmented based on the city\_id. When a new customer or store is added to pos\_central, a trigger replicates a copy of that record only to the database of the relevant city (e.g., data for Karachi is sent to karachi\_db).
* **Decentralization**: Transactional data such as Orders, Order\_Items, and Inventory is naturally decentralized. This data is created and maintained exclusively within the store-level databases, as a daily sale in one store is not directly relevant to the daily operations of another.

2.3 Replication Strategy

**Trigger-Based Asynchronous Replication** is used to maintain the consistency of master data:

* AFTER INSERT OR UPDATE triggers (trg\_customers, trg\_stores) are placed on the Customers and Stores tables in the pos\_central database.
* When a record is created or updated in the central database, the corresponding trigger function (replicate\_customers() or replicate\_stores()) is invoked.
* This function utilizes PostgreSQL's dblink extension to INSERT or UPDATE the data in the correct store database (karachi\_db or lahore\_db) based on the city\_id.

2.4 Concurrency Control

In a busy retail environment, it is critical to prevent scenarios like two customers simultaneously purchasing the last available item in stock. To handle this, **Pessimistic Locking** is used:

* Inside the ProcessOrder function, when an order is being processed, an exclusive lock (FOR UPDATE) is placed on the corresponding rows in the Inventory table for the items being purchased.
* **Code Snippet**:

SQL

PERFORM \* FROM Inventory

WHERE product\_id = (v\_item->>'product\_id')::INT AND store\_id = p\_store\_id

FOR UPDATE;

* This lock prevents other transactions from modifying these rows until the current transaction is complete, thereby ensuring data integrity.

3. Implementation in PostgreSQL

3.1 Schema Creation Scripts

* **Central Database (pos\_central)**:

SQL

-- Central Database

CREATE DATABASE pos\_central;

\c pos\_central;

-- Tables

CREATE TABLE Cities (

id SERIAL PRIMARY KEY,

name VARCHAR(100) UNIQUE NOT NULL

);

CREATE TABLE Products (

id SERIAL PRIMARY KEY,

sku VARCHAR(100) UNIQUE NOT NULL,

name VARCHAR(150) NOT NULL,

category\_id INT NOT NULL REFERENCES Categories(id),

price NUMERIC(12,2) NOT NULL

);

CREATE TABLE Customers (...);

CREATE TABLE Stores (...);

-- Mapping and Logging Tables

CREATE TABLE Order\_Mapping (...);

CREATE TABLE Replication\_Log (...);

* **Store Databases (karachi\_db, lahore\_db)**:

SQL

-- Store Database

CREATE DATABASE karachi\_db;

\c karachi\_db;

-- Tables

CREATE TABLE Orders (

id SERIAL PRIMARY KEY,

global\_order\_id UUID NOT NULL,

store\_id INT NOT NULL,

customer\_id INT NOT NULL,

total\_amount NUMERIC(12,2) NOT NULL

);

CREATE TABLE Order\_Items (...);

CREATE TABLE Payments (...);

CREATE TABLE Inventory (...);

3.2 Triggers and Stored Procedures

* **Replication Trigger for Customers**:

SQL

CREATE OR REPLACE FUNCTION replicate\_customers() RETURNS TRIGGER AS $$

DECLARE

db\_name TEXT;

BEGIN

IF NEW.city\_id = (SELECT id FROM Cities WHERE name='Karachi') THEN

db\_name := 'karachi\_db';

ELSE

db\_name := 'lahore\_db';

END IF;

BEGIN

PERFORM dblink\_exec(

'dbname='||db\_name||' user=dblink\_user password=dblink123',

'INSERT INTO Customers (...) VALUES (...) ' ||

'ON CONFLICT (id) DO UPDATE SET ...'

);

INSERT INTO Replication\_Log(...) VALUES ('Customers', NEW.id, db\_name, 'Success', ...);

EXCEPTION WHEN OTHERS THEN

INSERT INTO Replication\_Log(...) VALUES ('Customers', NEW.id, db\_name, 'Failed', SQLERRM);

END;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER trg\_customers

AFTER INSERT OR UPDATE ON Customers

FOR EACH ROW EXECUTE FUNCTION replicate\_customers();

* **Order Processing Stored Procedure**:

SQL

CREATE OR REPLACE FUNCTION ProcessOrder(

p\_store\_id INT,

p\_customer\_id INT,

p\_items JSONB,

...

) RETURNS UUID AS $$

DECLARE

v\_global\_order\_id UUID := gen\_random\_uuid();

...

BEGIN

-- Total calculation

-- Insert into Order\_Mapping

-- Insert into Orders

-- Loop through items to update Inventory with lock

-- Insert into Payments

RETURN v\_global\_order\_id;

END;

$$ LANGUAGE plpgsql;

4. Query Processing & Optimization

4.1 Distributed Query Example

To provide a consolidated view of the business, such as total sales across all stores, a distributed query is executed from the central server. This query uses dblink to fetch and aggregate data from both store databases.

* **Query**:

SQL

WITH karachi\_sales AS (

SELECT SUM(quantity\*price) AS total

FROM dblink('dbname=karachi\_db ...',

'SELECT quantity, price FROM Order\_Items') AS t(quantity INT, price NUMERIC)

),

lahore\_sales AS (

SELECT SUM(quantity\*price) AS total

FROM dblink('dbname=lahore\_db ...',

'SELECT quantity, price FROM Order\_Items') AS t(quantity INT, price NUMERIC)

)

SELECT COALESCE(k.total,0)+COALESCE(l.total,0) AS total\_sales

FROM karachi\_sales k, lahore\_sales l;

4.2 Query Execution Plan

The output of the EXPLAIN ANALYZE command reveals how the query optimizer executes this distributed query. The plan will show a "Foreign Scan" operation, which is evidence that the query is retrieving data from remote databases (karachi\_db and lahore\_db). This analysis helps in identifying performance bottlenecks.

5. Fault Tolerance, Security & Recovery

5.1 Security Aspects

**Role-Based Access Control** has been implemented to secure the database:

* **central\_admin**: A superuser role with full privileges on all databases.
* **dblink\_user**: A limited-privilege role used exclusively for replication. This user only has SELECT, INSERT, and UPDATE permissions on remote tables, not DROP or DELETE. This follows the "Principle of Least Privilege."

5.2 Backup and Recovery Methods

A proper backup and recovery strategy is in place to prevent data loss:

* **Backup**: Regular backups of the central and all store databases are taken using the pg\_dump utility.

Bash

# Backup central DB

pg\_dump -U central\_admin -F c -b -v -f '/path/to/backup/pos\_central.backup' pos\_central

* **Recovery**: In the event of a disaster, databases can be restored from the latest backup using the pg\_restore utility.

Bash

# Restore central DB

pg\_restore -U central\_admin -d pos\_central -v '/path/to/backup/pos\_central.backup'

5.3 Fault Tolerance

The system is designed to be resilient against faults like network failures:

* **Replication Logging**: Every replication attempt, whether successful or failed, is logged in the Replication\_Log table. If replication fails due to an unreachable store database, the status is saved as "Failed".
* **Retry Mechanism**: A stored function, retry\_failed\_replications(), has been created. This function reads all records with a 'Failed' status from the Replication\_Log table and attempts to re-process them. This function can be run periodically to bring the system back to a consistent state once network connectivity is restored.

6. Results, Conclusion & Future Enhancements

#### 6.1 Conclusion

In this project, we have successfully designed and implemented a Distributed POS System using PostgreSQL that demonstrates key DDB concepts such as data fragmentation, replication, concurrency control, and fault tolerance. This architecture provides the high performance, scalability, and enhanced reliability essential for a multi-branch retail business.

6.2 Future Enhancements

* **Dynamic Scaling**: The system could be enhanced to allow for the easy addition of new database nodes as the business expands to new cities.
* **Advanced Replication**: Instead of custom triggers, PostgreSQL's built-in **Logical Replication** could be used for a more efficient and robust replication mechanism.
* **Centralized Analytics**: Data from store databases could be moved via an ETL (Extract, Transform, Load) process into a central data warehouse to enable advanced business intelligence and analytics.