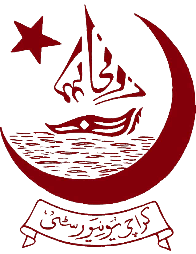
****

Distributed Database Systems

**Project Report**

*Topic: Distributed Retail Point-of-Sale (POS) System*

**Submitted by:**

**Arham Sharif EB21102022**

**Subhan Ali EB22210006125**

**Bisma Imran EB22210006029**

Course: **Distributed Database Systems**Course No : BSCS 606   
Group No : 09  
Instructor: **Dr. Taha Shabbir**  
Department: **Department of Computer Science (UBIT)**  
Institute: **University of Karachi**Submission Date: 30 October 2025

**Table of Contents**

[Project Report: Distributed Point of Sale (POS) System 3](#_Toc211800591)

[1. Introduction & Problem Statement 3](#_Toc211800592)

[1.1 Background of the Domain 3](#_Toc211800593)

[1.2 Problem Statement 3](#_Toc211800594)

[2. System Design & Architecture 4](#_Toc211800595)

[2.1 ER Diagram and Schema Design 4](#_Toc211800596)

[2.2 Fragmentation Strategy 5](#_Toc211800597)

[2.3 Replication Strategy 6](#_Toc211800598)

[2.4 Concurrency Control 7](#_Toc211800599)

[3. Implementation in PostgreSQL 8](#_Toc211800600)

[3.1 Schema Creation Scripts 8](#_Toc211800601)

[3.2 Triggers and Stored Procedures 9](#_Toc211800602)

[3.3 Working System 12](#_Toc211800603)

[4. Query Processing & Optimization 15](#_Toc211800604)

[4.1 Distributed Query Example 15](#_Toc211800605)

[4.2 Query Execution Plan 15](#_Toc211800606)

[5. Fault Tolerance, Security & Recovery 17](#_Toc211800607)

[5.1 Security Aspects 17](#_Toc211800608)

[5.2 Backup and Recovery Methods 17](#_Toc211800609)

[5.3 Fault Tolerance 17](#_Toc211800610)

[6. Results, Conclusion & Future Enhancements 19](#_Toc211800611)

[6.1 Performance Evaluation 19](#_Toc211800612)

[6.2 Lesson Learned 19](#_Toc211800613)

[6.3 Conclusion 19](#_Toc211800614)

[6.4 Future Enhancements 19](#_Toc211800615)

# 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.

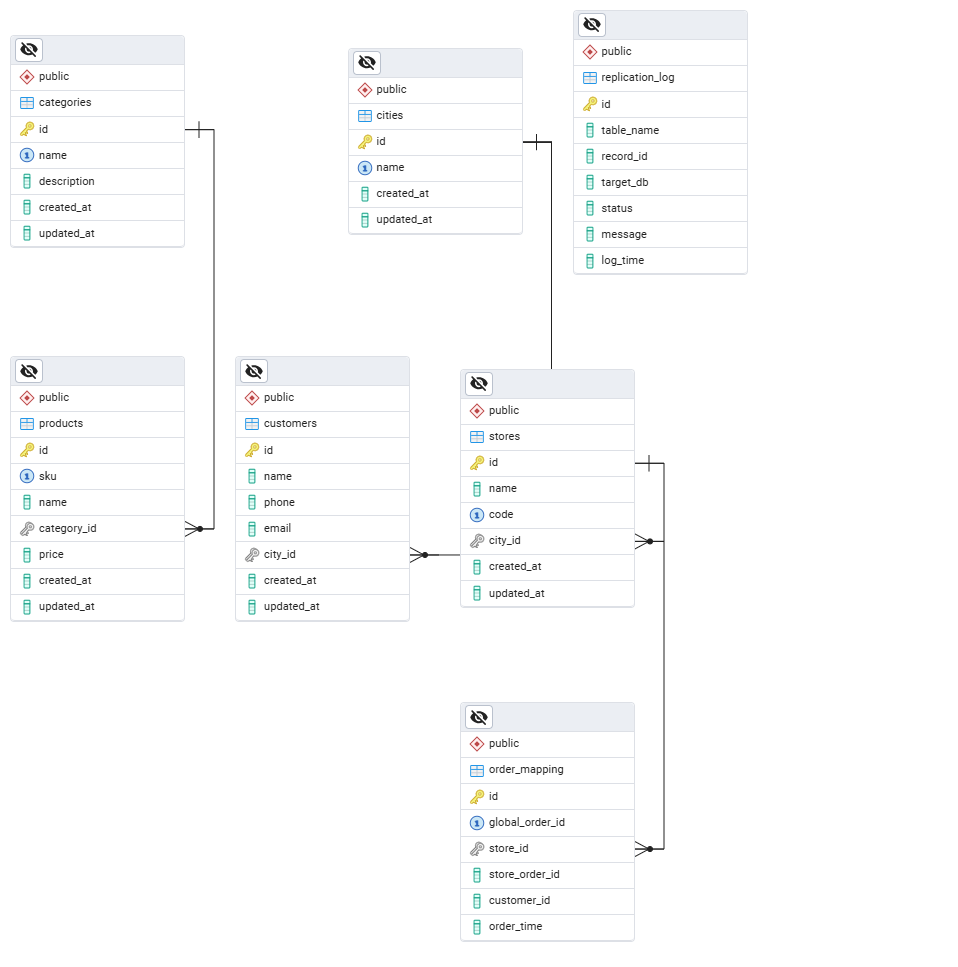


Fig 2.1 (a) – pos\_central

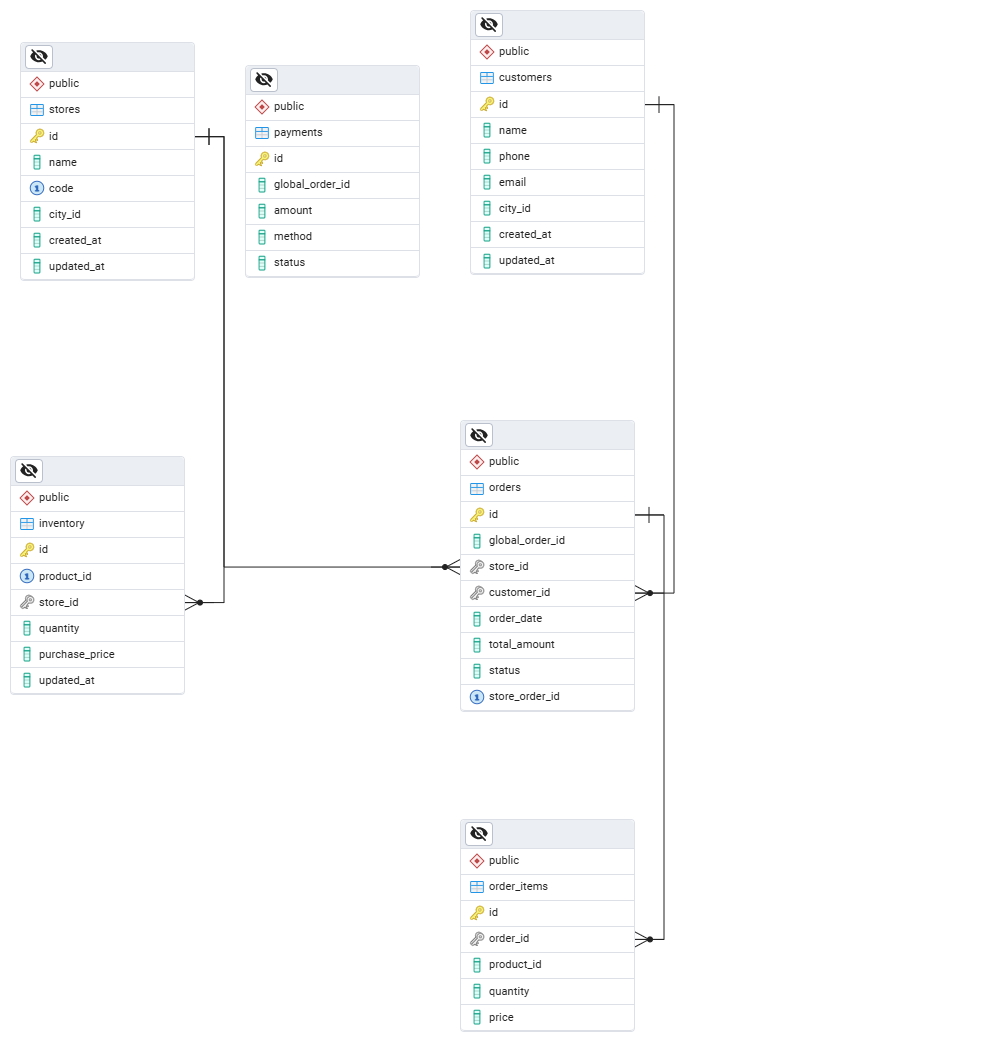


Fig 2.1 (b) – karachi\_db or lahore\_db

### 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.

| **Table** | **Fragmentation Type** | **Fragment Key / Strategy** | **Location / DB** |
| --- | --- | --- | --- |
| Customers | Horizontal | city\_id | Central → Store DBs |
| Stores | Horizontal | city\_id | Central → Store DBs |
| Orders | Local | Store-level only | Store DB only |
| Order\_Items | Local | Store-level only | Store DB only |
| Payments | Local | Store-level only | Store DB only |
| Inventory | Local | Store-level only | Store DB only |

Fig 2.2 – Fragmentation Table

### 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.

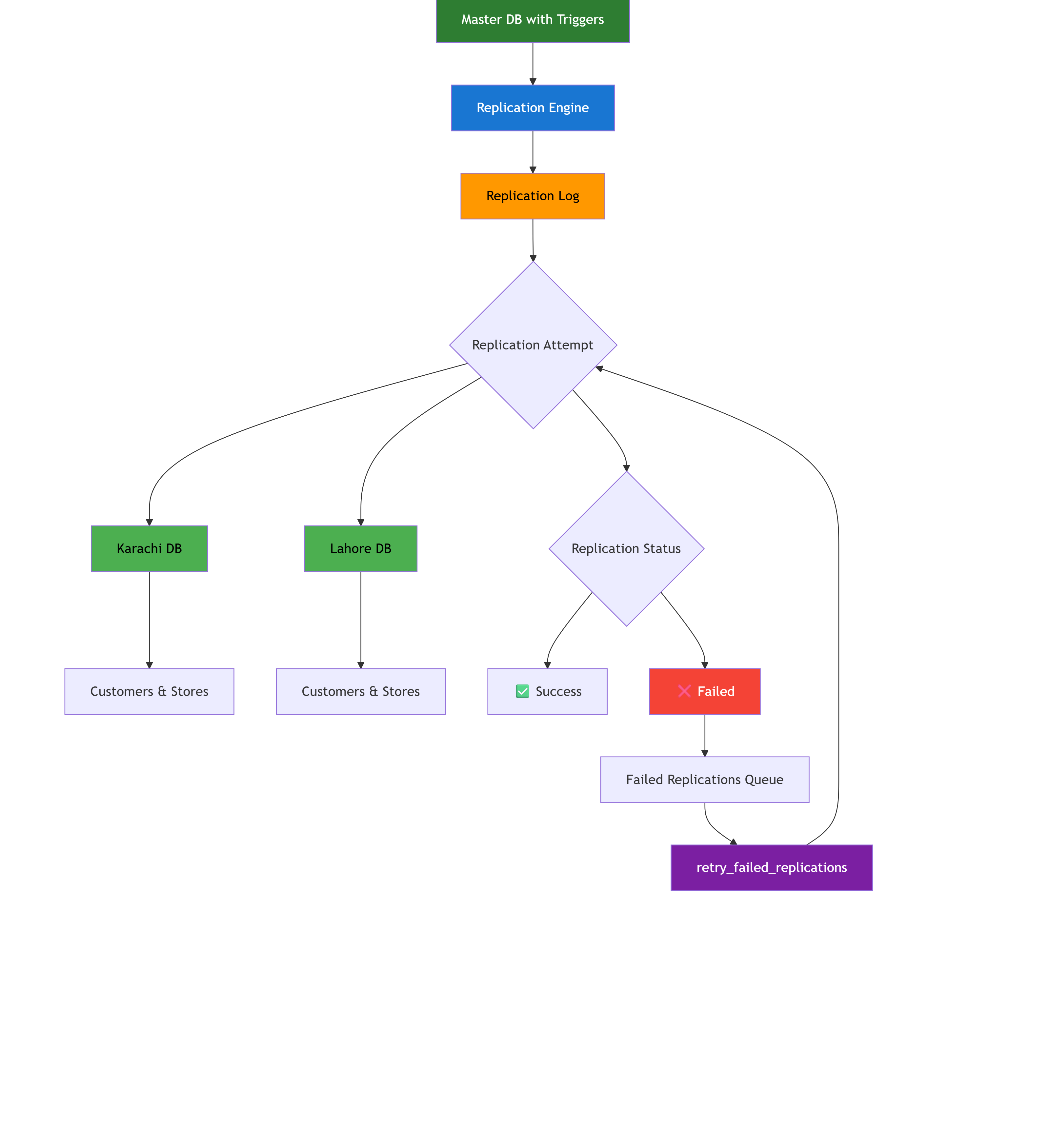


Fig 2.3 – Replication Flow

### 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**:

-- Lock inventory row

        PERFORM 1 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.

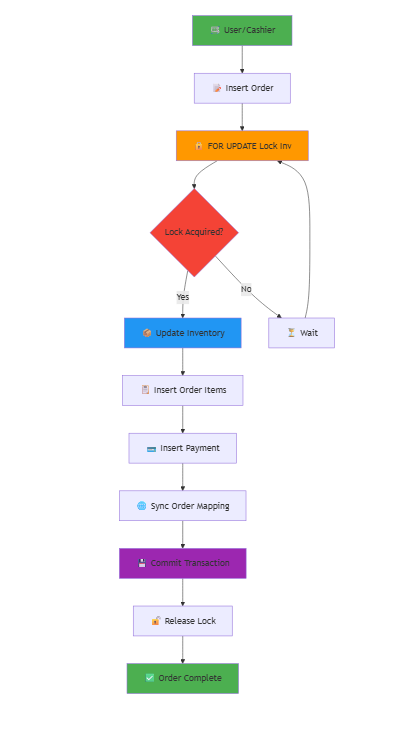


Fig 2.4 – ProcessOrder Flow with Locks

## 3. Implementation in PostgreSQL

### 3.1 Schema Creation Scripts

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

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

-- 1. CENTRAL DATABASE

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

CREATE DATABASE pos\_central;

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

-- Tables

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

-- Cities

CREATE TABLE Cities (

    id SERIAL PRIMARY KEY,

    name VARCHAR(100) UNIQUE NOT NULL,

    created\_at TIMESTAMPTZ DEFAULT now(),

    updated\_at TIMESTAMPTZ DEFAULT now()

);

-- Categories

CREATE TABLE Categories (…);

-- Products

CREATE TABLE Products (…);

-- Stores

CREATE TABLE Stores (…);

-- Customers

CREATE TABLE Customers (…);

-- Order Mapping

CREATE TABLE Order\_Mapping (…);

-- Replication Logging

CREATE TABLE Replication\_Log (…);

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

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

-- 2. STORE DATABASES

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

CREATE DATABASE karachi\_db;

CREATE DATABASE lahore\_db;

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

-- Tables for store DBs (same for both)

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

CREATE TABLE Customers (…);

CREATE TABLE Stores (…);

CREATE TABLE Orders (…);

CREATE TABLE Order\_Items (…);

CREATE TABLE Payments (…);

CREATE TABLE Inventory (…);

### 3.2 Triggers and Stored Procedures

* **Replication Trigger for Customers**:

-- Customers replication

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(

            'host=localhost port=5433 dbname='||db\_name||' user=dblink\_user password=dblink123',

            'INSERT INTO Customers (id,name,phone,email,city\_id,created\_at,updated\_at) VALUES ('||

            NEW.id||','''||NEW.name||''','''||COALESCE(NEW.phone,'')||''','''||COALESCE(NEW.email,'')||''','||

            NEW.city\_id||','''||NEW.created\_at||''','''||NEW.updated\_at||''') '||

            'ON CONFLICT (id) DO UPDATE SET name=EXCLUDED.name, phone=EXCLUDED.phone, email=EXCLUDED.email, city\_id=EXCLUDED.city\_id, updated\_at=EXCLUDED.updated\_at'

        );

        INSERT INTO Replication\_Log(table\_name, record\_id, target\_db, status, message)

        VALUES ('Customers', NEW.id, db\_name, 'Success', 'Replicated successfully');

    EXCEPTION WHEN OTHERS THEN

        INSERT INTO Replication\_Log(table\_name, record\_id, target\_db, status, message)

        VALUES ('Customers', NEW.id, db\_name, 'Failed', SQLERRM);

    END;

    RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE TRIGGER trg\_customers

AFTER INSERT OR UPDATE ON Customers

FOR EACH ROW EXECUTE FUNCTION replicate\_customers();

* **Order Processing Stored Procedure**:

CREATE OR REPLACE FUNCTION ProcessOrder(

    p\_store\_id INT,

    p\_customer\_id INT,

    p\_items JSONB,

    p\_payment\_method VARCHAR

) RETURNS UUID AS $$

DECLARE

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

    v\_total\_amount NUMERIC := 0;

    v\_item JSONB;

    v\_order\_id INT;

    v\_store\_order\_id INT;

BEGIN

    -- Calculate total amount

    FOR v\_item IN SELECT value FROM jsonb\_array\_elements(p\_items)

    LOOP

        v\_total\_amount := v\_total\_amount + (v\_item->>'quantity')::NUMERIC \* (v\_item->>'price')::NUMERIC;

    END LOOP;

    -- Get next store\_order\_id (local to this store)

    SELECT COALESCE(MAX(store\_order\_id),0)+1 INTO v\_store\_order\_id FROM Orders WHERE store\_id = p\_store\_id;

    -- Insert into Orders table (local)

    INSERT INTO Orders(global\_order\_id, store\_id, customer\_id, total\_amount, status, order\_date, store\_order\_id)

    VALUES (v\_global\_order\_id, p\_store\_id, p\_customer\_id, v\_total\_amount, 'Pending', now(), v\_store\_order\_id);

    SELECT id INTO v\_order\_id FROM Orders WHERE global\_order\_id = v\_global\_order\_id;

    -- Insert order items and update inventory

    FOR v\_item IN SELECT value FROM jsonb\_array\_elements(p\_items)

    LOOP

        -- Lock inventory row

        PERFORM 1 FROM Inventory

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

        FOR UPDATE;

        -- Insert order item

        INSERT INTO Order\_Items(order\_id, product\_id, quantity, price)

        VALUES (v\_order\_id, (v\_item->>'product\_id')::INT, (v\_item->>'quantity')::INT, (v\_item->>'price')::NUMERIC);

        -- Update inventory

        UPDATE Inventory

        SET quantity = quantity - (v\_item->>'quantity')::INT,

            updated\_at = now()

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

    END LOOP;

    -- Insert payment

    INSERT INTO Payments(global\_order\_id, amount, method, status)

    VALUES (v\_global\_order\_id, v\_total\_amount, p\_payment\_method, 'Paid');

    -- Insert mapping into central database using dblink

    PERFORM dblink\_exec(

        'host=localhost port=5433 dbname=pos\_central user=dblink\_user password=dblink123',

        'INSERT INTO Order\_Mapping (store\_id, store\_order\_id, customer\_id, global\_order\_id, order\_time) VALUES ('||

        p\_store\_id||','||v\_store\_order\_id||','||p\_customer\_id||','''||v\_global\_order\_id||''',now())'

    );

    RETURN v\_global\_order\_id;

END;

$$ LANGUAGE plpgsql;

### 3.3 Working System

* **Create Table**:

CREATE TABLE Customers (

    id SERIAL PRIMARY KEY,

    name VARCHAR(150) NOT NULL,

    phone VARCHAR(20),

    email VARCHAR(150),

    city\_id INT NOT NULL REFERENCES Cities(id),

    created\_at TIMESTAMPTZ DEFAULT now(),

    updated\_at TIMESTAMPTZ DEFAULT now()

);

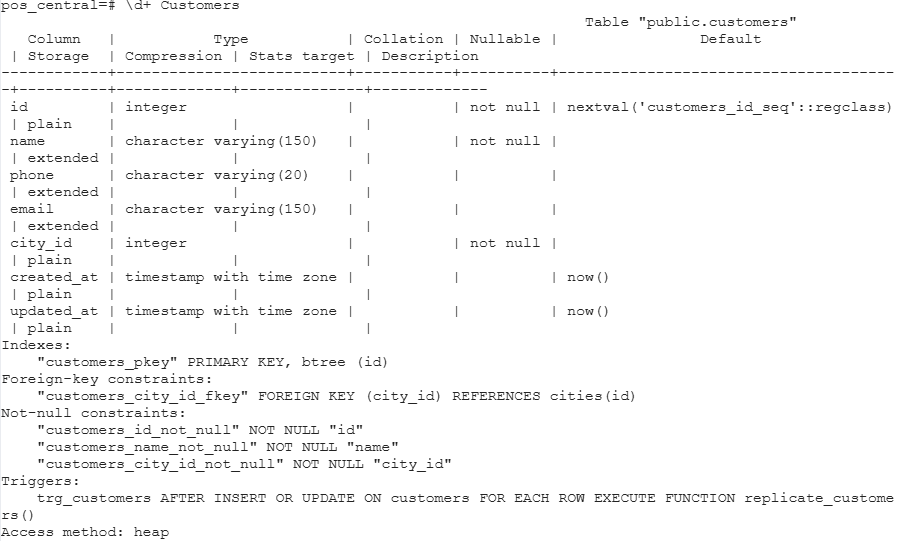


Fig 3.3 (a) – Table Details

* **Process Order**:

SELECT ProcessOrder(

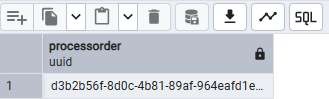
    (SELECT id FROM Stores WHERE code='KHI001'),

    (SELECT id FROM Customers WHERE name='Arham Sharif'),

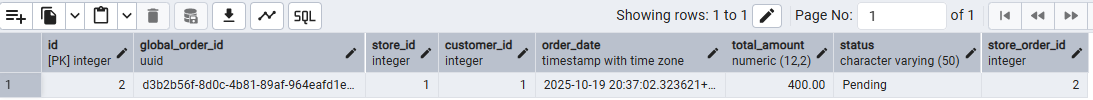
    '[{"product\_id":1,"quantity":2,"price":150.00},{"product\_id":3,"quantity":1,"price":100.00}]'::JSONB,

    'Cash'

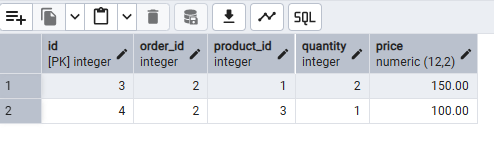
);



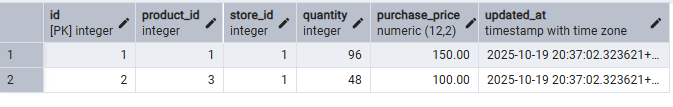
* UUID



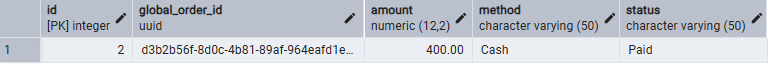
* Order



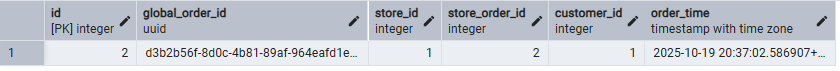
* Order Items



* Inventory



* Payment



* Order Mapping

Fig 3.3 (b) – Order Details

* **Replication Logs**:

SELECT \* FROM public.replication\_log;

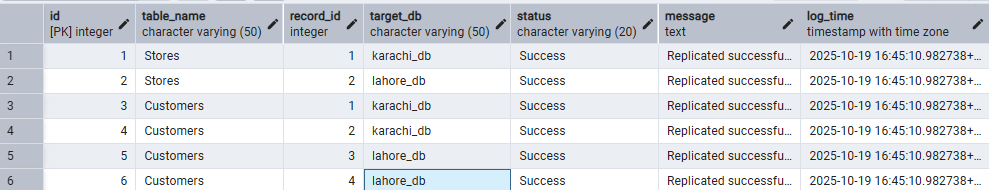


Fig 3.3 (c) – Replication Log Table

## 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**:

WITH karachi\_sales AS (

    SELECT SUM(quantity\*price) AS total

    FROM dblink('host=localhost port=5433 dbname=karachi\_db user=dblink\_user password=dblink123',

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

),

lahore\_sales AS (

    SELECT SUM(quantity\*price) AS total

    FROM dblink('host=localhost port=5433 dbname=lahore\_db user=dblink\_user password=dblink123',

                '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.

| **Step** | **Operation** | **Estimated Cost** | **Actual Time** | **Rows** | **Buffers** | **Description** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | Nested Loop | 35.03..35.08 | 155.500..155.502 ms | 1 | 4 | Combines results from karachi\_sales and lahore\_sales CTEs to calculate total sales. |
| 2 | CTE karachi\_sales Aggregate | 17.50..17.52 | 81.953..81.954 ms | 1 | 2 | Summing quantity\*price from Karachi store via dblink. |
| 3 | Function Scan on dblink t (Karachi) | 0.00..10.00 | 81.419..81.420 ms | 2 | 2 | Fetches order items from Karachi store database. |
| 4 | CTE lahore\_sales Aggregate | 17.50..17.52 | 73.534..73.534 ms | 1 | 2 | Summing quantity\*price from Lahore store via dblink. |
| 5 | Function Scan on dblink t\_1 (Lahore) | 0.00..10.00 | 73.531..73.531 ms | 0 | 2 | Fetches order items from Lahore store database (no rows found). |
| 6 | CTE Scan on karachi\_sales | 0.00..0.02 | 81.958..81.959 ms | 1 | 2 | Scans intermediate aggregated result from Karachi CTE. |
| 7 | CTE Scan on lahore\_sales | 0.00..0.02 | 73.537..73.537 ms | 1 | 2 | Scans intermediate aggregated result from Lahore CTE. |
| - | Planning Time | - | 0.197 ms | - | - | Time PostgreSQL spent planning query execution. |
| - | Total Execution Time | - | 155.560 ms | - | - | Total time for entire distributed query. |

Fig 4.2 – Query Plan Table

**Optimization Notes:**

* Queries use **Foreign Scan** to retrieve only required columns.
* Can use indexes on product\_id and order\_id for faster lookups.

## 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.

-- Backup central DB

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

-- Backup store DBs

\! pg\_dump -U central\_admin -F c -b -v -f '/path/to/backup/karachi\_db.backup' karachi\_db

\! pg\_dump -U central\_admin -F c -b -v -f '/path/to/backup/lahore\_db.backup' lahore\_db

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

-- Restore central DB

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

-- Restore store DBs

\! pg\_restore -U central\_admin -d karachi\_db -v '/path/to/backup/karachi\_db.backup'

\! pg\_restore -U central\_admin -d lahore\_db -v '/path/to/backup/lahore\_db.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 Performance Evaluation

The system processes local store transactions efficiently with minimal delay due to local inventory updates and pessimistic locking. Distributed queries using dblink correctly fetch and aggregate sales data from multiple stores, performing well for small datasets. Replication logs maintain consistency, and retry mechanisms handle network or replication failures effectively.

### 6.2 Lesson Learned

This project highlighted key insights about distributed databases. Trigger-based replication works but requires careful handling for failed cases. Pessimistic locking ensures inventory accuracy, and stored procedures simplify complex transactions. dblink is suitable for small datasets but may slow down as the number of stores grows. Proper design, replication, and concurrency control are critical for reliable distributed systems.

### 6.3 Conclusion

We successfully designed and implemented a Distributed POS System using PostgreSQL, demonstrating key concepts such as data fragmentation, replication, concurrency control, and fault tolerance. The system provides high performance, scalability, and enhanced reliability, making it suitable for multi-branch retail operations.

### 6.4 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.