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

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.

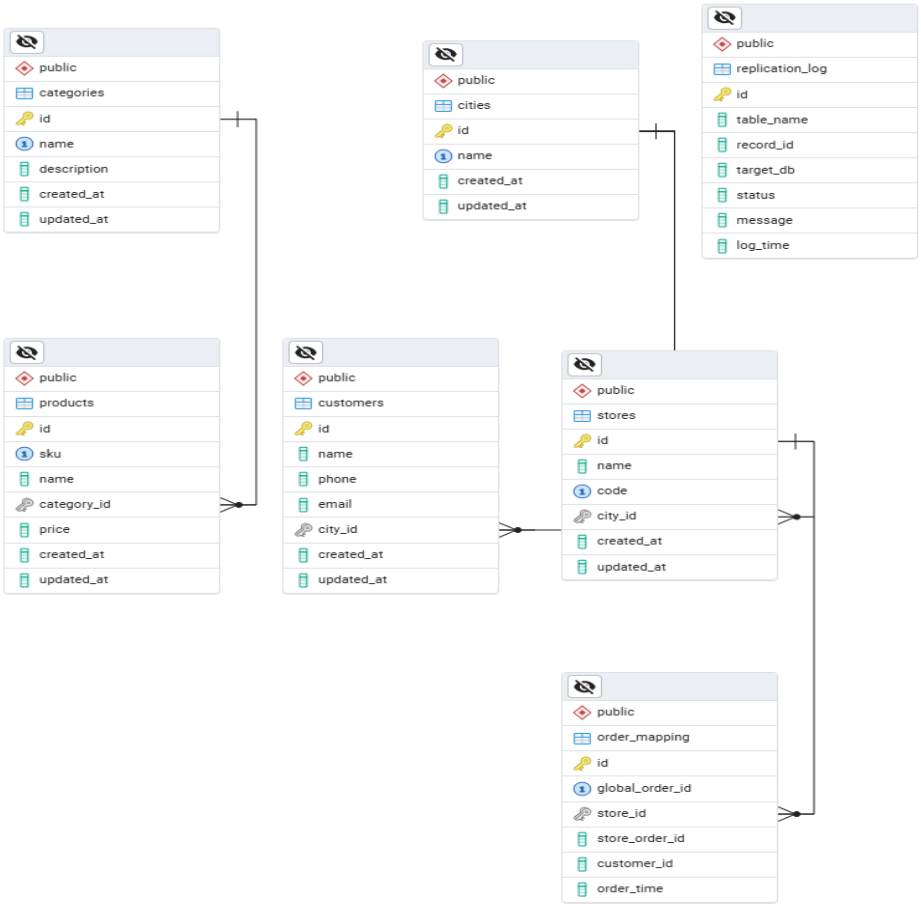


Fig 2.1 (a) – pos_central

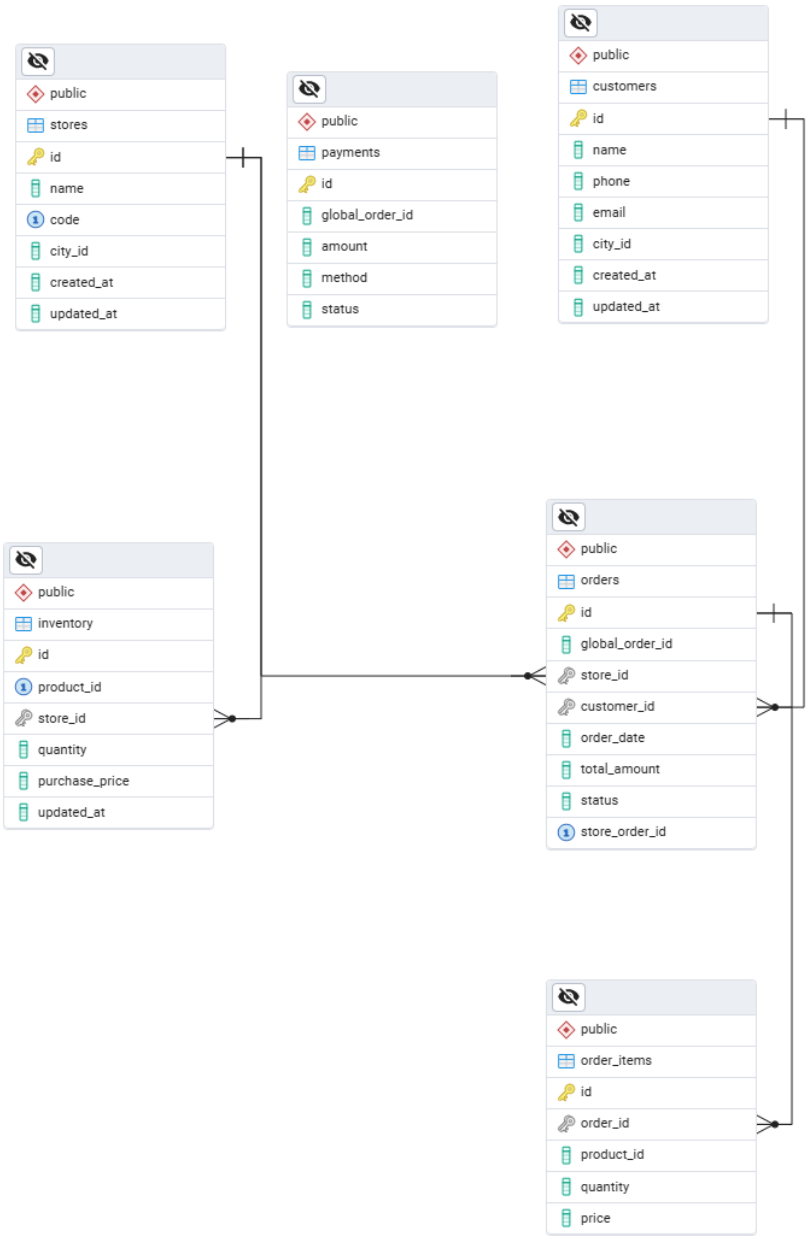


Fig 2.1 (b) – karachi_db or lahore_db

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.

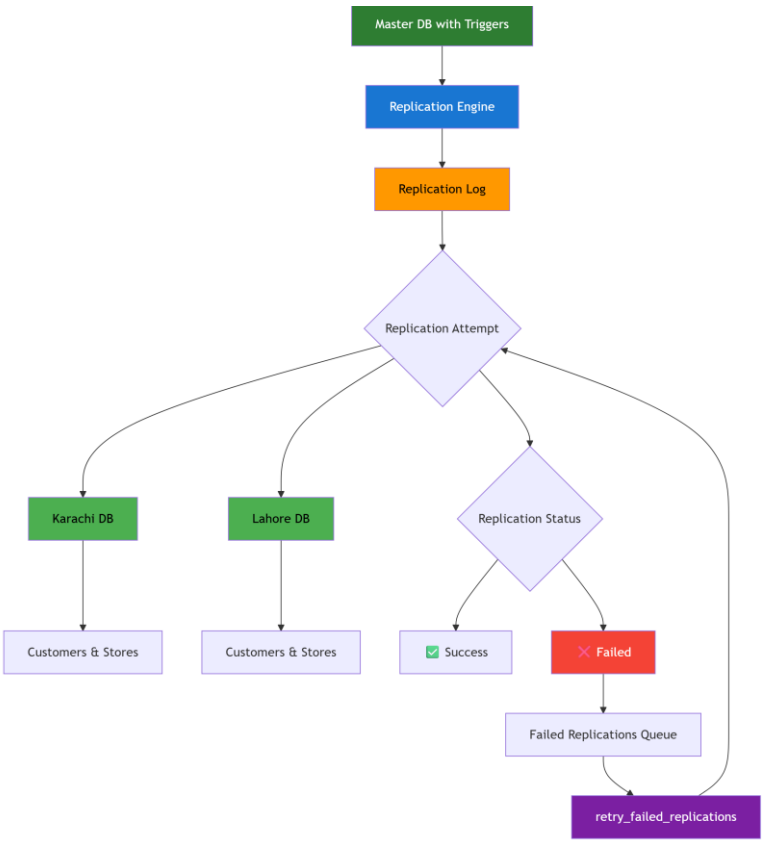


Fig 2.3 – Replication Flow

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.

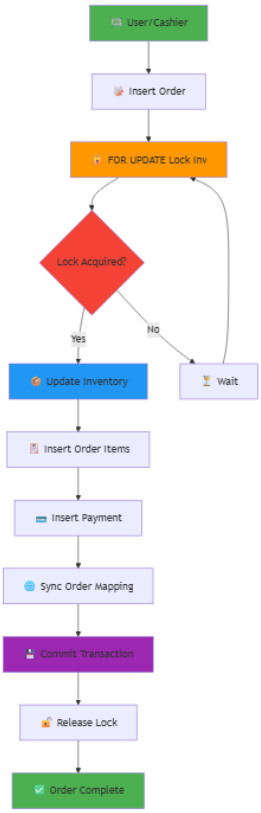


Fig 2.4 – ProcessOrder Flow with Locks

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()
);
```

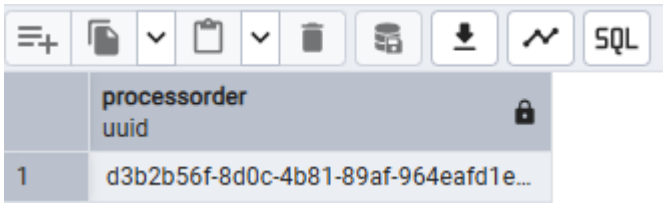
```
pos_central=# \d+ Customers
                                                                    Table "public.customers"
   Column   |           Type           | Collation | Nullable |                 Default                  | Storage  | Compression | Stats target | Description
------------+--------------------------+-----------+----------+------------------------------------------+----------+-------------+--------------+-------------
 id         | integer                  |           | not null | nextval('customers_id_seq'::regclass)    | plain    |             |              |
 name       | character varying(150)   |           | not null |                                          | extended |             |              |
 phone      | character varying(20)    |           |          |                                          | extended |             |              |
 email      | character varying(150)   |           |          |                                          | extended |             |              |
 city_id    | integer                  |           | not null |                                          | plain    |             |              |
 created_at | timestamp with time zone |           |          | now()                                    | plain    |             |              |
 updated_at | timestamp with time zone |           |          | now()                                    | plain    |             |              |
Indexes:
    "customers_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "customers_city_id_fkey" FOREIGN KEY (city_id) REFERENCES cities(id)
Not-null constraints:
    "customers_id_not_null" NOT NULL "id"
    "customers_name_not_null" NOT NULL "name"
    "customers_city_id_not_null" NOT NULL "city_id"
Triggers:
    trg_customers AFTER INSERT OR UPDATE ON customers FOR EACH ROW EXECUTE FUNCTION replicate_custome
rs()
Access method: heap
```

Fig 3.3 (a) – Table Details

- **Process Order**:

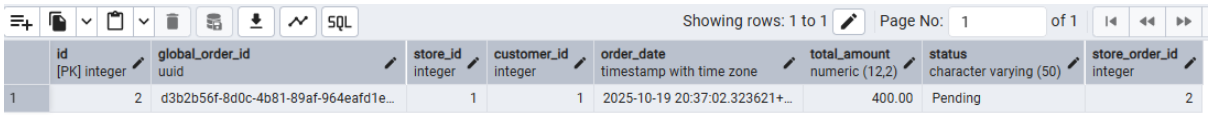
```
SELECT ProcessOrder(
    (SELECT id FROM Stores WHERE code='KHI001'),
    (SELECT id FROM Customers WHERE name='Arham Sharif'),
    '[{"product_id":1,"quantity":2,"price":150.00},{"product_id":3,"qua
ntity":1,"price":100.00}]'::JSONB,
    'Cash'
```

```
);
```



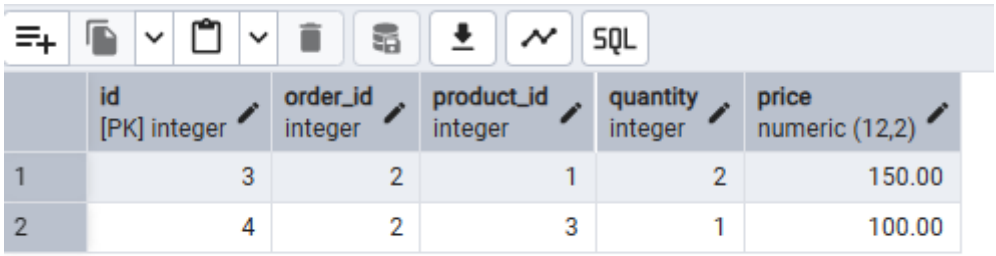
	processorder uuid
1	d3b2b56f-8d0c-4b81-89af-964eafd1e...

■ UUID



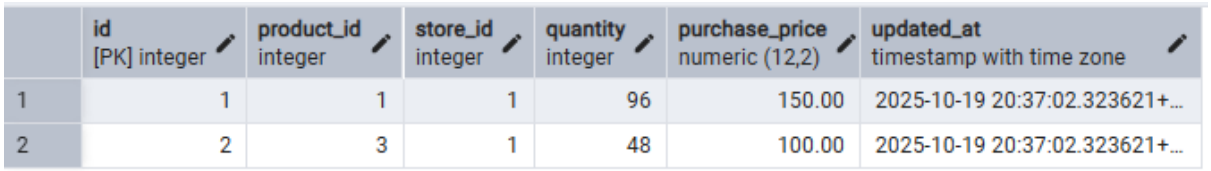
	id [PK] integer	global_order_id uuid	store_id integer	customer_id integer	order_date timestamp with time zone	total_amount numeric (12,2)	status character varying (50)	store_order_id integer
1	2	d3b2b56f-8d0c-4b81-89af-964eafd1e...	1	1	2025-10-19 20:37:02.323621+...	400.00	Pending	2

■ Order



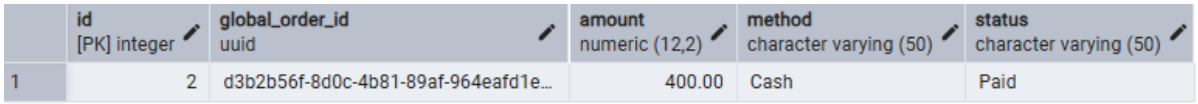
	id [PK] integer	order_id integer	product_id integer	quantity integer	price numeric (12,2)
1	3	2	1	2	150.00
2	4	2	3	1	100.00

■ Order Items



	id [PK] integer	product_id integer	store_id integer	quantity integer	purchase_price numeric (12,2)	updated_at timestamp with time zone
1	1	1	1	96	150.00	2025-10-19 20:37:02.323621+...
2	2	3	1	48	100.00	2025-10-19 20:37:02.323621+...

■ Inventory



	id [PK] integer	global_order_id uuid	amount numeric (12,2)	method character varying (50)	status character varying (50)
1	2	d3b2b56f-8d0c-4b81-89af-964eafd1e...	400.00	Cash	Paid

■ Payment

	id [PK] integer	global_order_id uuid	store_id integer	store_order_id integer	customer_id integer	order_time timestamp with time zone
1	2	d3b2b56f-8d0c-4b81-89af-964eafd1e...	1	2	1	2025-10-19 20:37:02.586907+...

■ Order Mapping

Fig 3.3 (b) – Order Details

- **Replication Logs**:

```
SELECT * FROM public.replication_log;
```

	id [PK] integer	table_name character varying (50)	record_id integer	target_db character varying (50)	status character varying (20)	message text	log_time timestamp with time zone
1	1	Stores	1	karachi_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...
2	2	Stores	2	lahore_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...
3	3	Customers	1	karachi_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...
4	4	Customers	2	karachi_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...
5	5	Customers	3	lahore_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...
6	6	Customers	4	lahore_db	Success	Replicated successfu...	2025-10-19 16:45:10.982738+...

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.

Step	Operation	Estimated Cost	Actual Time	Rows	Buffers	Description
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.