Optimizing Retail Operations: A Comprehensive Database System for Order Management, Sales Analytics, and Customer Insights.

1st Rakesh Nandan Anantharam
Engineering Science - Data Science
University at Buffalo - SUNY)
Buffalo, United States
ranantha@buffalo.edu

2nd Afreen
Engineering Science - Data Science
University at Buffalo - SUNY)
Buffalo, United States
afreen@buffalo.edu

3rd Kousalya Kethavath
Engineering Science - Data Science
University at Buffalo - SUNY
Buffalo, United States
kousalya@buffalo.edu

***Abstract*—This project proposes a relational database system to optimize data management for a global retail operation, addressing scalability, data integrity, security, and advanced analytics. Key entities include Orders, Products, Customers, Shipments, and Sales, linked through a normalized schema adhering to BCNF. Enhanced with unique identifiers using Python's Faker library, the system enables efficient operations, complex reporting, and data-driven decision-making. Replacing Excel-based management, it ensures scalability, multi-user access, and seamless customer service.**

***Index Terms*—Predictive Modeling, Database System, Data Organization,Data Integrity,Data Analysis,Functional Dependencies, Sports Data Management,Excel vs. Database,Researches**

I. Introduction

As global retail operations expand, the complexity and volume of data generated daily increase exponentially. Managing this data using traditional tools like Excel becomes inefficient, leading to challenges such as limited scalability, lack of data integrity, and difficulties in multi-user collaboration. To address these issues, this project proposes a robust relational database system designed to efficiently handle large-scale datasets while ensuring data consistency, security, and advanced analytics capabilities.

The database is built on a normalized schema adhering to BCNF principles, incorporating key entities such as Orders, Products, Customers, Shipments, and Sales. By leveraging structured relationships and constraints, the system optimizes data management and supports complex querying for reporting and decision-making. Using real-world data from the "Global Superstore" dataset, enriched with unique identifiers generated through Python's Faker library, the database ensures relational integrity and enhanced usability.

This system provides a scalable solution for managing retail operations across diverse markets, empowering departments such as sales, marketing, supply chain, and customer service to make informed decisions. By replacing Excel-based management, the proposed database significantly improves operational efficiency, customer satisfaction, and strategic planning, making it an indispensable tool for modern retail enterprises.

II. Problem Statement

The proposed database system aims to optimize the management of a large-scale global retail operation, handling orders, customers, products, shipments, sales, and market data. As the business grows, so does the complexity and volume of the data being generated daily. This includes order details for thousands of products across multiple regions, detailed customer information, shipments, and sales transactions that must be tracked efficiently. By utilizing a database system, we seek to address the following challenges:

- **Data Volume and Scalability:** Excel has limitations in handling very large datasets, leading to slower performance and potential file corruption. A database system, on the other hand, can handle millions of records efficiently and scale as the business grows.
- **Data Integrity and Consistency:** Excel lacks the ability to enforce relationships and constraints between data. A database system ensures data integrity by implementing relationships through primary and foreign keys, reducing the risk of duplicate or inconsistent data entries.
- **Concurrent Access:** Excel is not built for multi-user access and cannot efficiently support simultaneous editing by multiple team members. A database allows multiple users to access, query, and update data simultaneously while maintaining data consistency and security.
- **Data Security:** Excel files can be easily manipulated or deleted, leading to data loss. A database provides robust security features, such as user access controls and backups, to protect the data.
- **Complex Queries and Reporting:** Running complex queries and generating reports based on multiple conditions or data sets is cumbersome in Excel. A database system, with its SQL querying capabilities, can easily generate advanced reports and insights with better performance.

Overall, using a database offers numerous advantages over Excel in terms of data organization, integrity, efficiency, scalability, concurrency, collaboration, security, and advanced anal-

ysis capabilities, making it the preferred choice for managing and analyzing retail data.

III. Data Source

For this project, we used a dataset from a real-world source, the "Global Superstore" dataset. In order to enhance the dataset and make it more suitable for our relational database model, we generated additional unique identifiers for certain columns using Python's Faker library. These additional attributes, such as 'Sales_ID', 'Address_ID', 'Market_ID' and 'Shipment_ID', Category_ID, Status were not originally present in the dataset but were necessary to facilitate better relational mapping and schema design.

We created a Python script to transform the data by adding unique IDs to several columns. The process involved using Faker to generate unique UUIDs for the specified columns, which ensured that each record had a unique identifier. This approach enabled us to create essential data attributes that are typically required in databases to ensure relational integrity and facilitate efficient query execution.

IV. Importance of the Proposed Database System in Optimizing Retail Operations

A. Target Users

The primary users of the proposed database system will be various departments within the company that manage day-to-day operations and strategic decisions. The target users can be broken down into several categories:

- **Sales and Marketing Teams:** These users will utilize the database to analyze customer purchasing patterns, monitor sales performance across different products and regions, and tailor marketing campaigns based on customer segments.
- **Supply Chain and Logistics Teams:** These teams will rely on the database to track shipments, monitor delivery times, and optimize shipping costs based on the order and shipment data stored in the system.
- **Customer Support and Service Teams:** The database will enable customer service teams to quickly retrieve customer information, including order history, shipment status, and previous interactions, allowing them to resolve issues efficiently.
- **Executive Leadership and Management:** Management will use the database for high-level reporting, including sales trends, profit analysis, and customer satisfaction metrics. This data helps in making informed business decisions and long-term strategy planning.

B. Real-Life Scenario:

Imagine a large retail company that operates in various regions worldwide. The company deals with thousands of customers, products, and shipments every day. As the company expands, it becomes challenging to manage the growing volume of orders, track customer preferences, and handle shipping logistics across different markets.

A store manager in the New York region needs to optimize inventory to meet increasing customer demand while reducing shipping costs. Meanwhile, the marketing team is preparing for a major sales campaign and wants to identify which customer segments are most profitable in specific regions like Europe and Asia. The customer service department, on the other hand, aims to provide faster and more accurate responses to customer inquiries regarding order statuses and delivery timelines.

Through this database project, A retail company can achieve several objectives:

- **Store Managers** Store Managers can better manage inventory by analyzing historical sales and shipment data to predict product demand in specific markets.
- **Marketing Teams** can identify key customer segments and regions to target their campaigns based on previous sales performance.
- **Customer Service** Representatives can quickly access customer orders, shipping statuses, and preferences to provide accurate real-time updates.

The insights from this database will improve operational efficiency, help tailor marketing strategies, optimize inventory management, and enhance customer satisfaction across the entire business.

C. Database Administration

The database will be administered by the company's IT and Data Management Teams. These teams will be responsible for ensuring the database's availability, maintaining its security, performing regular backups, and optimizing performance as the database scales with increasing data.

1) IT Team::

- **Use Case:** They will handle server management, user permissions, database updates, and ensure the database is running smoothly with minimal downtime.
- **Real-Life Scenario:** The IT team needs to add new users to the database and assign different levels of access depending on the user's role within the company (e.g., read-only for some users, full access for others).

2) Data Analysts and Database Developers::

- **Use Case:** They will be responsible for developing complex queries, managing database schema changes, and providing support for advanced data extraction needs.
- **Real-Life Scenario:** The data analysts need to create custom SQL queries to extract monthly sales performance, filtering by specific product categories, regions, or customer segments, providing detailed reports to the management.

V. Design of the Database System

A. Database Schema

The database was created by initially importing the .csv dataset into a mega table to consolidate the data. From this mega table, the relevant columns were extracted and inserted into their respective relations, ensuring proper normalization and adherence to the defined schema. This process streamlined the data organization and ensured consistency across all tables.

1) Orders Table:

- **Schema:** Orders(Order_ID: VARCHAR(15), Order_Date: DATE, Customer_ID: VARCHAR(10), Order_Priority: VARCHAR(20)) **Information stored:** This table stores information about individual orders, such as the date, customer, product, and order priority.
- **Primary Key:** Order_ID ensures that each order for a specific customer is unique.
- **Foreign Key:** Customer_ID references Customers(Customer_ID) ON DELETE CASCADE to link the order to a customer and to ensure that when a customer is deleted, all related orders are automatically removed, maintaining data integrity.
- **Relationships:** One customer can place many orders (relationship with Customers). An order can include one or more products (relationship with Products, linked through Order Details).
- **Not Null:** Customer_ID **Default Values:** Order_Date with default value 'NULL' Order_Priority with default value 'Low'

2) Order Details Table:

- **Schema:** Order_Details(Row_ID: INT, Order_ID: VARCHAR(15), Product_ID: VARCHAR(20), Quantity: INT) **Information stored:** This table provides detailed information about the individual products in each order, including the quantity of each item.
- **Primary Key:** Row_ID - Uniquely identifies each row in the order details.
- **Foreign Key:** Order_ID references Orders(Order_ID) ON DELETE CASCADE to link the details to a specific order and ensures that when an order is deleted, all associated records in related tables are automatically deleted.. Product_ID references Products(Product_ID) ON DELETE ON DELETE CASCADE to prevent orphaned records if products are deleted or implement soft deletes.
- **Relationships:**Links multiple products to an order (relationship between Orders and Products through this table).
- **Not Null:** Order_ID, Product_ID **Default Values:** Quantity with default value 1 and checks if Quantity ¿ 0

3) Products Table:

- **Schema:** Products(Product_ID: VARCHAR(20), Product_Name: VARCHAR(255), Category_ID: UUID) **Information stored:** This table holds information about products, including the product name and the category id which identifies the category-subcategory combination they belong to.
- **Primary Key:** Product_ID - Uniquely identifies each product.
- **Foreign Key:** Category_ID references Categories(Category_ID) ON DELETE CASCADE to link each product to its category-subcategory combination, ensuring that deleting a category also removes all associated products, thus maintaining referential integrity.
- **Relationships:** Each product belongs to one category-subcategory combination (relationship with Categories).
- **Not Null:** Product_Name, Category_ID

4) Customers Table:

- **Schema:** Customers(Customer_ID: VARCHAR(10), Customer_Name: VARCHAR(100), Segment: VARCHAR(50), Address_ID: UUID, Market_ID: UUID) **Information stored:** This table stores customer information, such as names, segments (e.g., Consumer, Corporate), addresses, and market regions.
- **Primary Key:** Customer_ID - Uniquely identifies each customer.
- **Foreign Key:** Address_ID references Address(Address_ID) ON DELETE CASCADE to link customers to their addresses and ensures that when an address is deleted, all associated records are automatically removed Market_ID references Market(Market_ID) ON DELETE NO ACTION to preserve the association between Market and Customers tables. Ensures customer associations are preserved, even if the market is no longer active. Avoids losing the relationship between Customers and Market, which is important for historical data and analysis.
- **Relationships:** Each customer can place multiple orders (relationship with Orders). Each customer has one address (relationship with Address). An order can include one or more products (relationship with Products, linked through Order Details).
- **Not Null:** Customer_Name **Default Values:** Segment with default value 'Consumer'

5) Shipping Table:

- **Schema:** Shipping(Shipment_ID: UUID, Shipping_Cost: DECIMAL(10, 2), Ship_Mode: VARCHAR(20), Ship_Date: DATE, Order_ID: VARCHAR(15), Address_ID: UUID) **Information stored:** TThis table records shipment-related information, including shipping costs, shipping modes, and dates.
- **Primary Key:** Shipment_ID - Uniquely identifies each shipment.
- **Foreign Key:** Order_ID references Orders(Order_ID) ON DELETE CASCADE to link the shipment to a specific order and ensures that when an order is deleted, all associated records in the related tables are automatically removed. Address_ID references Address(Address_ID) to link the shipment to a delivery address.
- **Relationships:** Each shipment is linked to one order (relationship with Orders). A shipment has a delivery address (relationship with Address).
- **Not Null:** Order_ID **Default Values:** Ship_Mode with default values 'Standard Class' Ship_Date with default values 'NULL'

6) Sales Table:

- **Schema:** Sales(Sales_ID: UUID, Order_ID: VARCHAR(15), Product_ID: VARCHAR(20), Sales: DECI-

MAL(10, 2), Profit: DECIMAL(10, 2))
Information stored: This table contains information about the sales transactions, including the sales amount and profit for each order.
- **Primary Key:** Sales_ID - Uniquely identifies each sales transaction.
- **Foreign Key:** Order_ID references Orders(Order_ID) ON DELETE CASCADE to link the sale to a specific order and ensures that when an order is deleted, all related records are automatically removed Product_ID references Products(Product_ID) ON DELETE CASCADE to prevent orphaned records if products are deleted or implement soft deletes.
- **Relationships:** Each sale is associated with one order (relationship with Orders). A sale can include multiple products (relationship between Sales and Products).
- **Not Null:** Order_ID, Product_ID

7) Address Table:
- **Schema:** Address(Address_ID: UUID, City: VARCHAR(50), State: VARCHAR(50), Country: VARCHAR(50), Postal_Code: VARCHAR(10), Region: VARCHAR(50))
Information stored: This table holds detailed address information for customers and shipments.
- **Primary Key:** Address_ID - Uniquely identifies each address.
- **Relationships:** A customer has one address (relationship with Customers). A shipment is linked to one address (relationship with Shipping).

8) Market Table:
- **Schema:** Market(Market_ID: UUID, Market: VARCHAR(50), Region: VARCHAR(50)) Status: VARCHAR(10)
Information stored: This table holds information about different market regions.
- **Primary Key:** Market_ID - Uniquely identifies each market.
- **Relationships:** A customer belongs to one market (relationship with Customers).
- **Not Null:** Market_Name **Default Values:** Status with default value 'Active'

9) Market Table:
- **Schema:** Categories (Category_ID : UUID, Category : VARCHAR(50), Sub_Category : VARCHAR(50))
Information stored: This table holds information about different categories and their corresponding subcategories.
- **Primary Key:** Category_ID - Uniquely identifies each category-subcategory pair.
- **Relationships:** Products are associated with a category-subcategory combination (relationship with Products).
- **Not Null, Unique:** Category, Sub_Category

B. Functional Dependencies, BCNF Analysis, and Decomposition

1) Orders Table: .
- **Schema:** Orders(Order_ID, Order_Date, Customer_ID, Order_Priority)
- **Functional Dependencies:**
  - Order_ID → Order_Date, Customer_ID, Order_Priority
  - Customer_ID → (No transitive dependency in this table)
- **Primary Key:** Order_ID
- **Analysis:** The functional dependency Order_ID → Order_Date, Customer_ID, Order_Priority ensures that all attributes are functionally dependent on the primary key. There are no other dependencies that violate BCNF.
- **Conclusion:** This table is already in BCNF.

2) Order Details Table: .
- **Schema:** Order_Details(Row_ID, Order_ID, Product_ID, Quantity)
- **Functional Dependencies:**
  - Row_ID → Order_ID, Product_ID, Quantity
- **Primary Key:** Row_ID
- **Analysis:** The functional dependency Row_ID → Order_ID, Product_ID, Quantity ensures all attributes depend solely on the primary key. No other functional dependencies exist that violate BCNF.
- **Conclusion:** This table is in BCNF.

3) Products Table: .
- **Schema:** Products(Product_ID, Product_Name, Category_ID)
- **Functional Dependencies:**
  - Product_ID → Product_Name, Category_ID
- **Primary Key:** Product_ID
- **Analysis:** The dependency Product_ID → Product_Name, Category_ID ensures that all attributes depend only on the primary key.
- **Conclusion:** This table is in BCNF.

4) Customers Table: .
- **Schema:** Customers(Customer_ID, Customer_Name, Segment, Address_ID, Market_ID)
- **Functional Dependencies:**
  - Customer_ID → Customer_Name, Segment, Address_ID, Market_ID
  - Address_ID → (No dependencies in this table)
- **Primary Key:** Customer_ID
- **Analysis:** All attributes are functionally dependent on the primary key Customer_ID. No transitive dependencies exist in this table.
- **Conclusion:** This table is in BCNF.

5) Shipping Table: .
- **Schema:** Shipping(Shipment_ID, Shipping_Cost, Ship_Mode, Ship_Date, Order_ID, Address_ID)
- **Functional Dependencies:**
  - Shipment_ID → Shipping_Cost, Ship_Mode, Ship_Date, Order_ID, Address_ID
- **Primary Key:** Shipment_ID

- **Analysis:** The functional dependency Shipment_ID → Shipping_Cost, Ship_Mode, Ship_Date, Order_ID, Address_ID ensures all attributes are determined by the primary key.
- **Conclusion:** This table is in BCNF.

6) Sales Table: .

- **Schema:** Sales(Sales_ID, Order_ID, Product_ID, Sales, Profit)
- **Functional Dependencies:**
  - Sales_ID → Order_ID, Product_ID, Sales, Profit
- **Primary Key:** Sales_ID
- **Analysis:** The functional dependency Sales_ID → Order_ID, Product_ID, Sales, Profit ensures that all attributes depend on the primary key.
- **Conclusion:** TThis table is in BCNF.

7) Address Table:

- **Schema:** Address(Address_ID, City, State, Country, Postal_Code, Region)
- **Functional Dependencies:**
  - Address_ID → City, State, Country, Postal_Code, Region
- **Primary Key:** Address_ID
- **Analysis:** The functional dependency Address_ID → City, State, Country, Postal_Code, Region ensures that all attributes are determined by the primary key.
- **Conclusion:** This table is in BCNF.

8) Market Table:

- **Schema:** Market(Market_ID, Market, Region, Status)
- **Functional Dependencies:**
  - Market_ID → Market, Region, Status
- **Primary Key:** Market_ID
- **Analysis:** All attributes are determined by the primary key Market_ID, and no other functional dependencies exist.
- **Conclusion:** This table is in BCNF.

9) Categories Table:

- **Schema:** Categories(Category_ID, Category, Sub_Category)
- **Functional Dependencies:**
  - Category_ID → Category, Sub_Category
- **Primary Key:** Category_ID
- **Analysis:** All attributes depend on the primary key Category_ID, and no additional dependencies exist that violate BCNF.
- **Conclusion:** This table is in BCNF.

C. Final Schema After BCNF Transformation

No decompositions were needed as all relations already satisfy BCNF. The functional dependencies ensure data consistency, and the schema is organized to eliminate redundancy and anomalies.

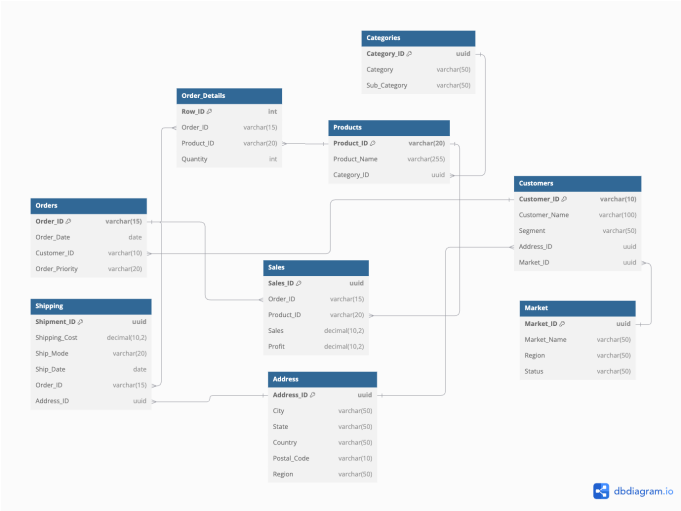


Fig. 1. ER Diagram

D. E/R diagram

VI. Problem Handling for Larger Datasets

When working with large datasets, several common challenges arise, especially in a relational database system. Below are the issues we faced:

A. Common Problems:

- Slow Query Performance: Queries involving joins and aggregation (e.g., GROUP BY, SUM, JOIN) tend to slow down when there are large amounts of data.
- Inefficient Join Operations: With multiple large tables, join operations can become resource-intensive, especially when foreign keys are not indexed properly.
- Inefficient Join Operations: With multiple large tables, join operations can become resource-intensive, especially when foreign keys are not indexed properly.
- Scalability: As the size of the data increases, some queries take significantly longer to execute, particularly those with nested subqueries or WHERE clauses on large text-based columns.

B. Solutions Adopted:

- Indexing: We applied indexing on commonly queried columns, especially those used in join conditions or filtering (WHERE clause). For example:
  - Category_ID in Products table and Product_ID in Order_Details
  - Customer_ID in the Customers and Orders tables.
  - Shipping_Cost and Address_ID in Shipping table.
- These indexes helped speed up query execution by reducing the amount of data scanned during joins and filtering.
- Partitioning: For tables that grow quickly (e.g., Orders, Sales), partitioning can help improve query performance. For example, partitioning Orders by Order_Date allows the system to scan only relevant partitions rather than the entire table.

- Optimizing Subqueries: We attempted to optimize queries with subqueries by replacing them with more efficient JOIN operations when possible, or using indexed views, CTE in PostgreSQL.

VII. SQL Queries

A. Insert query

Insert a new product into the Products table:

```sql
INSERT INTO Products (Product_ID, Product_Name, Category_ID)
VALUES ('TEC-AC-10003030', 'Plantronics CS510 - Over-the-Head monaural Wireless Headset System', '5b355c8e-ad32-4b82-8835-8c500fc0f5d0');
SELECT * FROM Products
WHERE Product_ID = 'TEC-AC-10003030';
```

	product_id [PK] character varying (20)	product_name character varying (255)	category_id uuid
1	TEC-AC-10003030	Plantronics CS510 - Over-the-Head monaural Wireless Headset Syste...	5b355c8e-ad32-4b82-8835-8c500fc0f5d0

Fig. 2.

B. Deletion Query

Delete an order with a specific Order_ID: Remove a customer who is no longer active:

```sql
DELETE FROM Orders
WHERE Order_ID = 'IN-2013-71247';
SELECT * FROM Orders
WHERE Order_ID = 'IN-2013-71247';
```

order_id [PK] character varying (15)	order_date date	customer_id character varying (10)	order_priority character varying (20)

Fig. 3.

```sql
DELETE FROM Customers
WHERE Customer_ID = 'RH-19495';
SELECT * FROM Customers
WHERE Customer_ID = 'RH-19495';
```

customer_id [PK] character varying (20)	customer_name character varying (100)	segment character varying (50)	address_id uuid	market_id uuid

Fig. 4.

C. update query

Update the shipping cost for a particular shipment:[fig 5]

```sql
UPDATE Shipping
SET Shipping_Cost = 950.00
WHERE Shipment_ID = '105480c3-d0ae-4128-922f-35b8960013bb';

SELECT * FROM Shipping
WHERE Shipment_ID = '105480c3-d0ae-4128-922f-35b8960013bb';
```

shipment_id [PK] uuid	shipping_cost numeric (10,2)	ship_mode character varying (20)	ship_date date	order_id character varying (15)	address_id uuid
105480c3-d0ae-4128-922f-35b8960013bb	950.00	First Class	2013-01-30	ES-2013-1579342	80ef45cb-222b-4f7d-8dcd-74cd704ed0e2

Fig. 5.

D. Markets with Sales:

To identify underperforming markets with the lowest sales, enabling businesses to evaluate market performance and target areas for potential growth and improvement.[fig 6]

```sql
SELECT M.Market_Name,
       SUM(S.Sales) AS total_sales
FROM Sales S
JOIN Orders O ON S.Order_ID = O.Order_ID
JOIN Customers C ON O.Customer_ID = C.Customer_ID
JOIN Market M ON C.Market_ID = M.Market_ID
GROUP BY M.Market_Name
ORDER BY total_sales ASC;
```

	market_name character varying (50)	total_sales numeric
1	Canada	39809.82
2	Africa	423873.63
3	EMEA	450605.99
4	US	1420827.90
5	LATAM	1610009.60
6	EU	1698066.37
7	APAC	2355969.54

Fig. 6.

E. top 10 products

Identifying the top 10 products with the highest total quantity ordered helps businesses ensure popular items remain in stock while enabling more effective marketing strategies to further drive sales growth.[fig 7]

```sql
SELECT P.Product_Name, SUM(OD.Quantity) AS total_ordered
FROM Order_Details OD
JOIN Products P ON OD.Product_ID = P.Product_ID
GROUP BY P.Product_Name
ORDER BY total_ordered DESC
LIMIT 10;
```

	product_name character varying (255)	total_ordered bigint
1	Eldon File Cart, Single Width	222
2	Apple Smart Phone, Full Size	160
3	Smead Lockers, Industrial	158
4	Smead File Cart, Single Width	143
5	Rogers File Cart, Single Width	143
6	Hon Executive Leather Armchair, Adjustable	142
7	Hewlett Copy Machine, Color	134
8	Office Star Executive Leather Armchair, Adjustable	134
9	Nokia Smart Phone, Full Size	134
10	Harbour Creations Executive Leather Armchair, Adjustab...	133

Fig. 7.

F. highest profits

Identifying products with the highest profits helps businesses recognize their most profitable items, enabling informed decisions on pricing and targeted marketing to maximize revenue and drive business growth.[fig 8]

Query Query History

```sql
SELECT Product_Name, total_sales, total_profit
FROM (
    SELECT P.Product_Name, SUM(S.Sales) AS total_sales, SUM(S.Profit) AS total_profit
    FROM Sales S
    JOIN Products P ON S.Product_ID = P.Product_ID
    GROUP BY P.Product_Name
) AS SalesSummary
WHERE total_sales > 40000
ORDER BY total_sales DESC;
```

Data Output Messages Notifications

	product_name character varying (255)	total_sales numeric	total_profit numeric
1	Apple Smart Phone, Full Size	81905.95	5560.72
2	Motorola Smart Phone, Full Size	72341.39	17352.64
3	Cisco Smart Phone, Full Size	68745.07	15294.96
4	Nokia Smart Phone, Full Size	65032.78	8598.70
5	Hon Executive Leather Armchair, Adjustable	50744.77	5987.73
6	Harbour Creations Executive Leather Armchair, Adjustab...	48197.47	11035.14
7	Office Star Executive Leather Armchair, Adjustable	47846.46	4377.57
8	Canon imageCLASS 2200 Advanced Copier	47599.87	18479.96
9	Samsung Smart Phone, VoIP	45406.34	3903.87
10	Samsung Smart Phone, Cordless	44180.46	-1808.37
11	Nokia Smart Phone, with Caller ID	43220.01	8640.19
12	Cisco Smart Phone, with Caller ID	41513.72	10070.22
13	SAFCO Executive Leather Armchair, Black	41097.33	6975.27

Fig. 8.

G. Average Shipping Cost by Ship Mode

To identify the most economical shipping methods, improving profitability while ensuring timely deliveries and enhancing customer satisfaction with reliable prices.[fig 9]

Query Query History

```sql
SELECT Product_Name, total_sales, total_profit
FROM (
    SELECT P.Product_Name, SUM(S.Sales) AS total_sales, SUM(S.Profit) AS total_profit
    FROM Sales S
    JOIN Products P ON S.Product_ID = P.Product_ID
    GROUP BY P.Product_Name
) AS SalesSummary
WHERE total_sales > 40000
ORDER BY total_sales DESC;
```

Data Output Messages Notifications

	product_name character varying (255)	total_sales numeric	total_profit numeric
1	Apple Smart Phone, Full Size	81905.95	5560.72
2	Motorola Smart Phone, Full Size	72341.39	17352.64
3	Cisco Smart Phone, Full Size	68745.07	15294.96
4	Nokia Smart Phone, Full Size	65032.78	8598.70
5	Hon Executive Leather Armchair, Adjustable	50744.77	5987.73
6	Harbour Creations Executive Leather Armchair, Adjustab...	48197.47	11035.14
7	Office Star Executive Leather Armchair, Adjustable	47846.46	4377.57
8	Canon imageCLASS 2200 Advanced Copier	47599.87	18479.96
9	Samsung Smart Phone, VoIP	45406.34	3903.87
10	Samsung Smart Phone, Cordless	44180.46	-1808.37
11	Nokia Smart Phone, with Caller ID	43220.01	8640.19
12	Cisco Smart Phone, with Caller ID	41513.72	10070.22
13	SAFCO Executive Leather Armchair, Black	41097.33	6975.27

Fig. 9.

Using ALTER:[fig 10]

H. Common Table Expression (CTE) Query

Critical Orders Requiring Immediate Attention

Purpose: Lists orders with a ”Critical” priority to prioritize customer service efforts. [fig 11]

I. Product Complaints by Customers

Purpose: Identifies customers associated with products experiencing negative profit (potential complaints or returns). [fig 12]

```sql
ALTER TABLE Market
RENAME COLUMN Status TO Market_status;

SELECT *
FROM Market LIMIT 5;
```

ta Output Messages Notifications

market_id [PK] uuid	market_name character varying (50)	region character varying (50)	market_status character varying (10)
f1c99d43-c65c-4a7f-a446-b74630bcdf1d	LATAM	Central	Active
c5690a25-1e4b-4779-b868-0d02307155...	Africa	Africa	Active
5446a4af-1bd2-465e-8ad2-8705b3aaa2...	LATAM	North	Active
e7745f59-8c5c-4328-bf33-f8504049d195	US	East	Active
fd92252a-a48d-47ff-b9b2-2b900525aa33	EMEA	EMEA	Active

Fig. 10.

```sql
WITH CriticalOrders AS (
    SELECT
        O.Order_ID,
        O.Order_Date,
        C.Customer_Name,
        C.Segment,
        A.City,
        A.State
    FROM Orders O
    JOIN Customers C ON O.Customer_ID = C.Customer_ID
    JOIN Address A ON C.Address_ID = A.Address_ID
    WHERE O.Order_Priority = 'Critical'
)
SELECT *
FROM CriticalOrders
WHERE State = 'California';
```

ata Output Messages Notifications

order_id character varying (15)	order_date date	customer_name character varying (100)	segment character varying (50)	city character varying (50)	state character varying (50)
IN-2012-77409	2012-11-01	Emily Phan	Consumer	San Diego	California
IN-2012-28367	2012-11-28	Carlos Soltero	Consumer	Los Angeles	California
IN-2014-77276	2014-08-08	Anna Häberlin	Corporate	San Diego	California
IN-2013-58509	2013-11-04	Joseph Holt	Consumer	San Francisco	California
US-2012-142475	2012-04-20	Joseph Holt	Consumer	San Francisco	California
ES-2011-4805316	2011-07-31	Emily Phan	Consumer	San Diego	California
CA-2012-116638	2012-01-28	Joseph Holt	Consumer	San Francisco	California
ES-2014-2846503	2014-08-23	Richard Bierner	Consumer	San Diego	California
ES-2012-4635011	2012-07-12	Carlos Soltero	Consumer	Los Angeles	California
IN-2012-23054	2012-09-15	Tamara Manning	Consumer	San Francisco	California

Total rows: 50 of 50 Query complete 00:00:00.093 Ln 70, Col 1

Fig. 11.

```sql
SELECT C.Customer_Name, P.Product_Name, S.Profit
FROM Sales S
JOIN Products P ON S.Product_ID = P.Product_ID
JOIN Orders O ON S.Order_ID = O.Order_ID
JOIN Customers C ON O.Customer_ID = C.Customer_ID
WHERE S.Profit < 0
ORDER BY S.Profit ASC;
```

Data Output Messages Notifications

	customer_name character varying (100)	product_name character varying (255)	profit numeric (10,2)
46	Michael Moore	Cisco Smart Phone, Cordless	-1303.06
47	Tracy Hopkins	Barricks Computer Table, Fully Assembled	-1294.35
48	Ralph Arnett	Cuisinart Stove, Silver	-1293.79
49	Anthony Witt	HP Wireless Fax, Laser	-1268.04
50	Ellis Ballard	Nokia Smart Phone, Full Size	-1242.98
51	Dan Campbell	GBC DocuBind TL300 Electric Binding System	-1237.85
52	Tom Ashbrook	Office Star Swivel Stool, Adjustable	-1235.05
53	Bruce Geld	Chromcraft Round Table, Fully Assembled	-1218.38
54	Justin Deggeller	Lesro Training Table, Adjustable Height	-1195.24
55	Peter Fuller	Hoover Upright Vacuum With Dirt Cup	-1181.28
56	Richard Bierner	Cisco Smart Phone, Cordless	-1172.74
57	Stuart Van	Bevis Training Table, Fully Assembled	-1150.83
58	Irene Maddox	Bush Library with Doors, Traditional	-1147.65

Total rows: 1000 of 12524 Query complete 00:00:00.142 Ln 77, Col 1

Fig. 12.

VIII. Query Execution Analysis

A. Identification of Problematic Queries

When handling large datasets, some queries can become problematic, especially when they join multiple tables without optimization. These queries can be slow due to the absence of indexes on frequently used columns in join or filter conditions. Below, we identified and explained the most problematic queries from our dataset:

B. Query Execution Plan for Problematic Queries

- SCAN
  - Scanning a large table, such as Categories or Orders, can be costly in terms of performance, particularly if these tables have a significant number of records.
  - The solution is to create indexes on the columns used in join and filter conditions (Category_ID, Product_ID, Customer_ID, etc.).
- SEARCH USING INTEGER PRIMARY KEY:
  - Whenever a table is joined on indexed columns, the database engine will use the indexes to search for relevant rows efficiently. This step is usually fast due to the indexed search on primary key or foreign key columns.
- SCALAR SUBQUERY:
  - A scalar subquery can be slow if it scans large tables like Orders and Sales without indexing. Optimizing this subquery by indexing Customer_ID in Orders and Sales can significantly improve performance.
- JOIN:
  - When joining large tables without indexes, a full table scan is performed. Using indexes on foreign key columns like Order_ID, Customer_ID, Product_ID, etc., helps avoid full scans, improving query performance.

C. Execution Plan Analysis:

1) Using Join and Group By Query:

- **Optimization Focus:** Indexing is essential to reduce the cost of scans and subqueries. By indexing foreign keys and frequently filtered columns, we reduce disk I/O and speed up the query execution time.
- Products Sold by Sub-Category. [fig 13]
- **Potential Problems:**
  - Full table scan: This query requires scanning multiple large tables (Categories, Products, Order_Details) and performing a GROUP BY operation. It can become slow when the dataset grows.
  - Join Efficiency: The JOIN operation on non-indexed columns (e.g., Category_ID and Product_ID) could cause performance issues.
- **Optimizations:**
  - Optimization Strategy: Create indexes on Category_ID in Products and Product_ID in Order_Details to speed up the join and aggregation process.[fig 14]

```sql
SELECT
    C.Sub_Category,
    SUM(OD.Quantity) AS Total_Quantity_Sold
FROM Categories C
JOIN Products P ON C.Category_ID = P.Category_ID
JOIN Order_Details OD ON P.Product_ID = OD.Product_ID
GROUP BY C.Sub_Category
ORDER BY Total_Quantity_Sold DESC;
```

Data Output Messages Notifications

	sub_category character varying (50)	total_quantity_sold bigint
1	Binders	21391
2	Storage	16906
3	Art	16285
4	Paper	12794
5	Chairs	12306
6	Phones	11863
7	Furnishings	11219
8	Accessories	10924
9	Labels	9329
10	Supplies	8533
11	Fasteners	8387
12	Envelopes	8357
13	Bookcases	8307
14	Copiers	7454
15	Appliances	6078
16	Machines	4906

Total rows: 17 of 17 Query complete 00:00:00.127 Ln 38, Col 1

Fig. 13.

```sql
CREATE INDEX IF NOT EXISTS idx_products_category_id ON Products (Category_ID);
CREATE INDEX IF NOT EXISTS idx_order_details_product_id ON Order_Details (Product_ID);
SELECT
    C.Sub_Category,
    SUM(OD.Quantity) AS Total_Quantity_Sold
FROM Categories C
JOIN Products P ON C.Category_ID = P.Category_ID
JOIN Order_Details OD ON P.Product_ID = OD.Product_ID
GROUP BY C.Sub_Category
ORDER BY Total_Quantity_Sold DESC;
```

Data Output Messages Notifications

	sub_category character varying (50)	total_quantity_sold bigint
1	Binders	21391
2	Storage	16906
3	Art	16285
4	Paper	12794
5	Chairs	12306
6	Phones	11863
7	Furnishings	11219
8	Accessories	10924

Total rows: 17 of 17 Query complete 00:00:00.110 Ln 99, Col 1

Fig. 14.

- **Benefits of Indexing:** With the appropriate indexes on Category_ID in Products and Product_ID in Order_Details, the DBMS can efficiently process the join and group operations, drastically reducing the overall time to retrieve and aggregate the data.
- **Improvement in Query Performance:** The combination of these two indexes allows the query to bypass unnecessary row scans and directly access the required data, improving query execution speed. As a result, the

queriess performance improves from linear to logarithmic time complexity, especially for large datasets.

2) Using Sub query, Top Customers by Market:

- **Purpose:** Identifies key customers in each market for personalized campaigns or loyalty programs. [fig 15]

```sql
SELECT M.Market_Name, C.Customer_Name, T.Total_Sales
FROM Customers C
JOIN Market M ON C.Market_ID = M.Market_ID
JOIN (
    SELECT O.Customer_ID, SUM(S.Sales) AS Total_Sales
    FROM Orders O
    JOIN Sales S ON O.Order_ID = S.Order_ID
    GROUP BY O.Customer_ID
) T ON C.Customer_ID = T.Customer_ID
ORDER BY M.Market_Name, T.Total_Sales DESC;
```

Data Output Messages Notifications

	market_name character varying (50)	customer_name character varying (100)	total_sales numeric
1	Africa	Barry Weirich	9027.48
2	Africa	Ritsa Hightower	8753.47
3	Africa	Liz Thompson	8234.99
4	Africa	Stuart Van	7948.35
5	Africa	Pauline Johnson	7865.03
6	Africa	Liz Carlisle	7193.65
7	Africa	Christopher Conant	6957.23
8	Africa	Michelle Tran	6822.92
9	Africa	Russell D'Ascenzo	6523.39
10	Africa	Paul Van Hugh	6417.03
11	Africa	Steve Carroll	6344.79
12	Africa	Aaron Hawkins	6320.02
13	Africa	Liz MacKendrick	6280.47
14	Africa	Benjamin Patterson	6190.77

Total rows: 1000 of 1587 Query complete 00:00:00.254 Ln 41, Col 1

Fig. 15.

- **Potential Problems:** This query involves multiple joins and a subquery. The subquery is calculating the total sales for each customer, which can be costly if the Orders and Sales tables are large. Additionally, the final join with Customers and Market requires indexes on Customer_ID, Market_ID, and Order_ID.
  - Subquery in JOIN: The subquery (T) could be a performance bottleneck due to the aggregation (SUM(S.Sales)) on a potentially large dataset.
  - Sorting by Total_Sales: Sorting after aggregation may increase execution time when dealing with large result sets.
- **Optimizations:**
  - Create indexes on Customer_ID in Customers, Orders. [fig 16]
- **Benefits of Indexing:** The combination of indexing Customer_ID in both Orders and Customers tables ensures that the joins are fast and efficient, particularly in large

```sql
CREATE INDEX IF NOT EXISTS idx_customers_customer_id ON Customers (Customer_ID);
CREATE INDEX IF NOT EXISTS idx_orders_customer_id ON Orders (Customer_ID);
SELECT M.Market_Name, C.Customer_Name, T.Total_Sales
FROM Customers C
JOIN Market M ON C.Market_ID = M.Market_ID
JOIN (
    SELECT O.Customer_ID, SUM(S.Sales) AS Total_Sales
    FROM Orders O
    JOIN Sales S ON O.Order_ID = S.Order_ID
    GROUP BY O.Customer_ID
) T ON C.Customer_ID = T.Customer_ID
ORDER BY M.Market_Name, T.Total_Sales DESC;
```

Data Output Messages Notifications

	market_name character varying (50)	customer_name character varying (100)	total_sales numeric
1	Africa	Barry Weirich	9027.48
2	Africa	Ritsa Hightower	8753.47
3	Africa	Liz Thompson	8234.99
4	Africa	Stuart Van	7948.35
5	Africa	Pauline Johnson	7865.03
6	Africa	Liz Carlisle	7193.65
7	Africa	Christopher Conant	6957.23
8	Africa	Michelle Tran	6822.92
9	Africa	Russell D'Ascenzo	6523.39
10	Africa	Paul Van Hugh	6417.03

Total rows: 1000 of 1587 Query complete 00:00:00.159 Ln 112, Col 1

Fig. 16.

datasets where scanning the entire table would be time-consuming.
- **Improvement in Query Performance:** These indexes allow the DBMS to perform the necessary joins efficiently, reducing the time complexity of the query. This results in faster query execution and optimized resource usage when working with large data volumes.

3) High-Cost Shipping Orders:

- **Purpose:** Highlights orders with high shipping costs for cost analysis and optimization. [fig 17]

```sql
SELECT O.Order_ID, S.Shipping_Cost, A.City, A.State, A.Country, S.Ship_Mode
FROM Shipping S
JOIN Orders O ON S.Order_ID = O.Order_ID
JOIN Address A ON S.Address_ID = A.Address_ID
WHERE S.Shipping_Cost > 600
ORDER BY S.Shipping_Cost DESC;
```

Data Output Messages Notifications

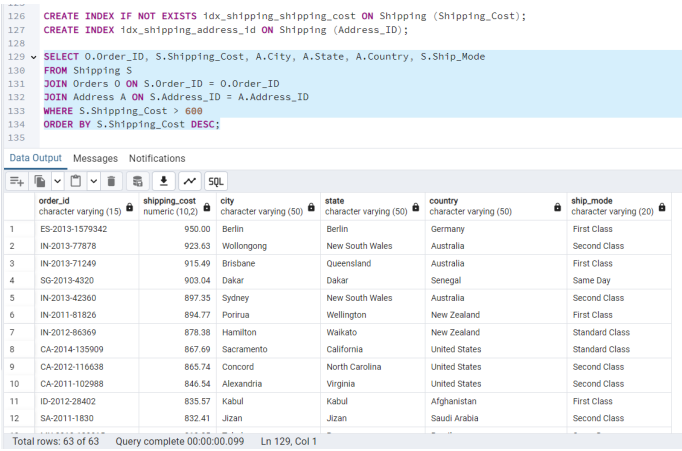
	order_id character varying (15)	shipping_cost numeric (10,2)	city character varying (50)	state character varying (50)	country character varying (50)	ship_mode character varying (20)
1	ES-2013-1579342	950.00	Berlin	Berlin	Germany	First Class
2	IN-2013-77878	923.63	Wollongong	New South Wales	Australia	Second Class
3	IN-2013-71249	915.49	Brisbane	Queensland	Australia	First Class
4	SG-2013-4320	903.04	Dakar	Dakar	Senegal	Same Day
5	IN-2013-42360	897.35	Sydney	New South Wales	Australia	Second Class
6	IN-2011-81826	894.77	Porirua	Wellington	New Zealand	First Class
7	IN-2012-86369	878.38	Hamilton	Waikato	New Zealand	Standard Class
8	CA-2014-135909	867.69	Sacramento	California	United States	Standard Class
9	CA-2012-116638	865.74	Concord	North Carolina	United States	Second Class
10	CA-2011-102988	846.54	Alexandria	Virginia	United States	Second Class
11	ID-2012-28402	835.57	Kabul	Kabul	Afghanistan	First Class
12	SA-2011-1830	832.41	Jizan	Jizan	Saudi Arabia	Second Class

Total rows: 63 of 63 Query complete 00:00:00.142 Ln 129, Col 1

Fig. 17.

- **Potential Problems:**
  - Filtering on Shipping_Cost: The WHERE clause can be slow without an index on the Shipping_Cost or Order_ID.
  - Join on Address table: The join on the Address table without indexing on Address_ID could result in a slower execution plan.
- **Optimizations:**

– Add indexes on Shipping_Cost and Address_ID to speed up the filtering and joining process [fig 18]



```sql
CREATE INDEX IF NOT EXISTS idx_shipping_shipping_cost ON Shipping (Shipping_Cost);
CREATE INDEX idx_shipping_address_id ON Shipping (Address_ID);

SELECT O.Order_ID, S.Shipping_Cost, A.City, A.State, A.Country, S.Ship_Mode
FROM Shipping S
JOIN Orders O ON S.Order_ID = O.Order_ID
JOIN Address A ON S.Address_ID = A.Address_ID
WHERE S.Shipping_Cost > 600
ORDER BY S.Shipping_Cost DESC;
```

Data Output Messages Notifications

	order_id character varying (15)	shipping_cost numeric (10,2)	city character varying (50)	state character varying (50)	country character varying (50)	ship_mode character varying (20)
1	ES-2013-1579342	950.00	Berlin	Berlin	Germany	First Class
2	IN-2013-77878	923.63	Wollongong	New South Wales	Australia	Second Class
3	IN-2013-71249	915.49	Brisbane	Queensland	Australia	First Class
4	SG-2013-4320	903.04	Dakar	Dakar	Senegal	Same Day
5	IN-2013-42360	897.35	Sydney	New South Wales	Australia	Second Class
6	IN-2011-81826	894.77	Porirua	Wellington	New Zealand	First Class
7	IN-2012-86369	878.38	Hamilton	Waikato	New Zealand	Standard Class
8	CA-2014-135909	867.69	Sacramento	California	United States	Standard Class
9	CA-2012-116638	865.74	Concord	North Carolina	United States	Second Class
10	CA-2011-102988	846.54	Alexandria	Virginia	United States	Second Class
11	ID-2012-28402	835.57	Kabul	Kabul	Afghanistan	First Class
12	SA-2011-1830	832.41	Jizan	Jizan	Saudi Arabia	Second Class

Total rows: 63 of 63 Query complete 00:00:00.099 Ln 129, Col 1

Fig. 18.

- **Benefits of Indexing:** With indexes on both Order_ID and Address_ID, the DBMS can quickly and efficiently join the Shipping, Orders, and Address tables, minimizing unnecessary row scans.
- **Improvement in Query Performance:** These indexes optimize the joins and filtering steps in the query, resulting in faster data retrieval. This dramatically reduces query execution time and enhances performance, particularly for large-scale datasets.

IX. Bonus Task - Streamlit Application

- **Installing Required Libraries:** The necessary libraries for the project include Streamlit for the web interface, psycopg2 for database connection, pandas for data manipulation, and plotly for creating visualizations.
- **Setting Up Database Connection:** To establish a connection between our web application and the PostgreSQL database, we configured the required credentials, such as the database name, user, password, and host. This allowed our app to interact with the data stored in the database.
- **Creating Streamlit Interface for Query Input:** We designed the user interface to enable users to input SQL queries. The interface includes a text input area where users can write and submit their SQL queries, enabling dynamic interaction with the database.
- **Executing Queries and Displaying Results:** Once a query is submitted, the application executes it on the database and retrieves the results. We displayed the results in a table format using Streamlit, ensuring that if no data was returned, users were appropriately notified.
- **Adding Visualizations:** To enhance user experience, we included visualizations of the query results. Using Plotly, we created interactive charts, such as bar charts, based on the data retrieved from the queries. These visualizations help users better understand the results.
- **Running the Streamlit App Locally:** We ran the app locally using Streamlit's built-in server to test its functionality. The app was accessed through a web browser, and we ensured that all features, including query execution and result display, functioned as intended.
- **Deploying the App:** Finally, we deployed the app to Streamlit Cloud (or another cloud platform), making it accessible online. We configured the necessary settings to ensure that the database was accessible from the deployed environment, enabling seamless interactions with the app.

X. Conclusion

This project successfully demonstrates the development and implementation of a relational database system tailored for optimizing data management in a global retail operation. By addressing the limitations of traditional tools like Excel, such as scalability, data integrity, and multi-user access, the proposed system ensures efficient handling of large datasets with structured relationships and constraints.

The normalized schema adheres to BCNF principles, eliminating redundancy and maintaining data consistency. Key entities like Orders, Products, Customers, and Sales are enriched with unique identifiers for better relational mapping, enabling seamless integration and analysis. Advanced query optimization techniques, such as indexing and partitioning, enhance the performance of complex operations, making the system scalable for growing business demands.

The database supports various stakeholders, including sales, marketing, logistics, and customer service teams, by providing actionable insights through SQL-based queries and reports. Real-world scenarios, such as inventory optimization, customer segmentation, and shipment tracking, showcase the database's practical utility. Additionally, the integration of a Streamlit-based interface for dynamic query execution and visualizations further extends its accessibility and usability.

Overall, the project highlights the transformative impact of a robust database system in streamlining retail operations, improving decision-making, and ensuring long-term business growth.

XI. References

- **Dataset:** https://www.kaggle.com/datasets/apoorvaappz/global-super-store-dataset/data
- **ER Tool:** https://dbdiagram.io/home