# MARKETPLACE MANAGEMENT SYSTEM

Surya Venkata Rohit Moganti - 50560088
Dept. of Computer Science and Engineering
University at Buffalo
smoganti@buffalo.edu

Srujan Reddy Brahmananda Reddy - 50560241
Dept. of Computer Science and Engineering
University at Buffalo
sbrahman@buffalo.edu

Niveditha Niveditha - 50560507
Dept. of Computer Science and Engineering
University at Buffalo
nivedit2@buffalo.edu

***Abstract*— The Marketplace Management System is a tactical framework created to help companies analyse sales patterns, manage inventories more effectively, and improve customer experiences. The system offers important insights into client preferences, regional sales trends, and product demand dynamics through the utilization of powerful algorithms and data analytics. This makes it possible for businesses to increase operational effectiveness, customize marketing campaigns, and optimize inventory levels. Real-time sales tracking, client segmentation, logistics optimization, and dynamic inventory management are some of the important aspects. In today's competitive marketplace, the Marketplace Management System is a vital tool for optimizing profits and promoting long-term expansion by implementing a systematic approach to data use and decision-making.***

## 1. INTRODUCTION

A successful marketplace in today's fast-paced business climate depends on effective inventory management, profound understanding of consumer behaviour, enhanced sales tactics, and improved logistical operations. To remain competitive, businesses need to manage complications including consumer preferences, market changes, and operational efficiency. To give actionable insights, boost customer happiness, increase sales growth, and optimize profitability, a strong marketplace management system combines data analytics, customer segmentation, and logistical optimization capabilities. By enabling companies to make well-informed decisions, adjust to changing market conditions, and provide outstanding customer experiences, this system positions them for long-term success in the contemporary marketplace.

For companies looking to stay ahead of the curve, optimizing sales methods is a continuous effort. This entails figuring out which items are doing well as well as comprehending the reasons behind their success. Through the utilization of sophisticated algorithms and data analytics, a marketplace management system may reveal previously undiscovered information on consumer preferences, market trends, and sales patterns. Equipped with this understanding, companies may make well-informed choices about product promotions, pricing policies, and inventory replacement, which will eventually lead to increased sales and profitability.

## 2. BACKGROUND AND RELATED WORK

All the data that we are going to manage can be accessed by multiple people at the organization and it is difficult for each of them to access at the same time in file system. In the case of DBMS concurrent access is possible. This ensures that all the modifications done to the database are up to date when another user accesses the database. Scalability: We are managing the dataset of a huge marketplace; we can clearly state that the data in this database will only increase but not decrease in future. This leads to scale up or system storage and with a good database management system this can easily be achieved. Complex Querying: Our main goal here is to find patters and scenarios which can be a great benefit to the organization in maximizing their profit and with DBMS we can run complex queries and use various optimization techniques for boosting the processing.

## 3. TARGET USERS

The Marketplace Management System caters to two main user groups, namely Customers and Management, who have different responsibilities and access rights.

Management:
Management: Access to extensive data analytics and monitoring capabilities within the system is available to management staff. To make wise choices about marketing tactics, inventory control, and product offers, they may monitor sales patterns, examine consumer behavior, and spot trends.

Customers:
Purchase History: Users who often buy goods from the marketplace have access to their past purchases through the system. This contains information about the items ordered, the dates of the orders, the quantities, the prices, and the payment methods. Customers can successfully plan future purchases, analyze prior orders, and keep track of their costs by having access to their purchase history.

## 4. REAL LIFE SCENARIOS

In practical applications, the implementation of a marketplace management system plays a pivotal role in optimizing workflows and augmenting effectiveness in many facets of retail administration. Inventory management is one important area in which these systems shine. Businesses are able to optimize inventory levels, minimize stockouts, and guarantee that items are accessible to meet consumer demand by offering real-time insights into stock levels, sales patterns, and demand projections. By taking a proactive stance, inventory management procedures are improved overall and expenditures related to overstocking or understocking products are reduced.

## 5. DATA PREPROCESSING

We have performed the initial data transformation and dummy data imputation using Python pandas library. In this we have created multiple new columns which contribute significantly to the final dataset. For the time being we have inserted temporary values into the columns just to avoid redundancy while loading the dataset into pg admin. We also made sure that unnecessary columns are dropped. The final pandas data frame is saved in the form of a .CSV file with UTF-8 encoding.

```
[3]: # Adding a column shipping charge and add values to that column based on the shipping mode
     Dataset['Shipping Charge'] = None

     for index , rows in Dataset.iterrows():
         if rows['Ship Mode'] == 'Standard Class':
             Dataset.at[index, 'Shipping Charge'] = 10
         elif rows['Ship Mode'] == 'Second Class':
             Dataset.at[index, 'Shipping Charge'] = 15

         elif rows['Ship Mode'] == 'First Class':
             Dataset.at[index, 'Shipping Charge'] = 20

         elif rows['Ship Mode'] == 'Same Day':
             Dataset.at[index, 'Shipping Charge'] = 30
```

Fig 5.1 Adding a column shipping charge

```
[4]: # Adding a column shipping description to the dataaset and fill in values based on shipping mode
     Dataset['Shipping Description'] = None

     for index , rows in Dataset.iterrows():
         if rows['Ship Mode'] == 'Standard Class':
             Dataset.at[index, 'Shipping Description'] = 'Shipping within 10 Days'
         elif rows['Ship Mode'] == 'Second Class':
             Dataset.at[index, 'Shipping Description'] = 'Shipping within a week'

         elif rows['Ship Mode'] == 'First Class':
             Dataset.at[index, 'Shipping Description'] = 'Shipping within 2 days'

         elif rows['Ship Mode'] == 'Same Day':
             Dataset.at[index, 'Shipping Description'] = 'Delivery on Same Day'
```

Fig 5.2 Adding a column shipping description to the datatset

```
# Dropping Unnecessary Columns
Dataset.drop('Sales', axis=1, inplace=True)
Dataset.drop('Discount', axis=1, inplace=True)
Dataset.drop('Profit', axis=1, inplace=True)
```

Fig 5.3 Dropping Unnecessary Columns

```
[12]: # Filling in temporary values to Total_Cost_Price Total_Sale_price Final_Price
      Dataset['Total_Cost_Price'] = None
      Dataset['Total_Sale_price'] = None

[13]: Dataset['Total_Cost_Price'] = (Dataset['Price_per_item'] * Dataset['Quantity'])

[15]: Dataset['Total_Sale_price'] = Dataset['Total_Cost_Price'] + (Dataset['Total_Cost_Price'] * (Dataset['Profit_percent'] / 100))

[17]: Dataset['Final_Price'] = Dataset['Total_Sale_price'] + Dataset['Shipping Charge']
```

Fig 5.4 Filling in temporary values in Total_Cost_Price ,Total_Sale_Price, Final_price

```
# Price per Item
# Adding temporary price per item into the dataset
Dataset['Price_per_item'] = None


Dataset['Price_per_item']  = Dataset['Sales'] / Dataset['Quantity']
```

Fig 5.5 Adding temporary price per item into dataset

```
[10]: # Adding new column Profit Percent and filling in values based on customer segment
      Dataset['Profit_percent'] = None

      for index , rows in Dataset.iterrows():
          if rows['Segment'] == 'Consumer':
              Dataset.at[index, 'Profit_percent'] = 20
          elif rows['Segment'] == 'Corporate':
              Dataset.at[index, 'Profit_percent'] = 10

          elif rows['Segment'] == 'Home Office':
              Dataset.at[index, 'Profit_percent'] = 5
```

Fig 5.6 Adding new column Profit Percent and filling in values based on customer segment

```
# Saving the dataset in utf-8 encoding
Dataset.to_csv('output.csv', encoding='utf-8', index=False)
```

Fig 5.7 Saving the dataset in utf-8 encoding

## 6. DATA DESCRIPTION

Shipment Table:
Ship_Mode: It explains the mode of shipping and is divided into categories.
Shipping_Charge: This attribute represents the shipping cost depending on the ship mode selected.
Shipping_Desc: This provides information on the shipment method i.e number of days taken for the shipment to complete.

Address Table:
Postal_Code: A distinct code used for location identification and addressing that is allocated to particular geographic locations.
Region: It represents the area of the country that the postal code is connected to.
City: The city that the postal code corresponds to.
State: It states the province or state that is associated with the postal code.
Country: It represents the nation that the postal code is linked to.

Segments Table:
Segment_ID: Customer segment unique identifier that is used to group consumers according to specific standards.
Segment: It explains the kind or classification of the target market.
Profit_percent: This column represents the amount of profit generated from a particular segment of customers.

Customer_details Table:
Customer_ID: Unique identifier for customers, used to track and manage customer information.
Customer_Name: Represents the name of the customer associated with the customer ID.
Segment_ID: Foreign key linking to the segments table, indicating the segment to which the customer belongs.
Postal_Code: Foreign key linking to the address table, providing location details for the customer.

Orders Table:
Order_ID: Orders have unique identifier which is used to manage and track the order details.
Product_ID: Products have unique identifier which is linked to the products table to identify the ordered item.
Quantity: Shows the quantity of specific product ordered in a transaction

Price Table:
Transaction_ID: Transactions have unique identifier which is used to link transaction details to specific orders.
Total_Cost_price: Total cost incurred for a transaction which includes product costs.
Total_Sale_price: Total revenue generated from the transaction which includes product total_cost_price and profit percentage on the customer segment
Profit: Displays the profit earned from the transaction which is calculated as the difference between total sale price and total cost price.

Order_to_Ship Table:
Order_ID: Foreign key linking to the orders table which indicates the order for which shipping details are recorded.
Order_Date: Shows the date when the order was created.
Ship_Date: Represents the date when the order was shipped.

Categories Table:
Category_ID: Product categories have unique identifier which is used to categorize the products into different groups.
Category: Represents the main category to which a product belongs to.
Sub_Category: Gives further classification within a category which specifies the sub-categories of the products.

Products Table:
Product_ID: Products have unique identifier which is used to manage and track the product information.
Category_ID: Foreign key links to the categories table which indicates the category to which the product belongs to.
Product_Name: Shows the name or description of the product.
Price_per_Item: Displays the price of each item of the product.

Transactions Table:
Transaction_ID: Transactions have unique identifier which is used to manage and track the transactional details.
Order_ID: Foreign key links to the orders table which indicates the order associated with the transaction.
Customer_ID: Foreign key links to the customers table which indicates the customer involved in the transaction.
Product_ID: Foreign key links to the products table which indicates the product involved in the transaction.
Quantity: Shows the quantity of specific product involved in the transaction.
Ship_Mode: Represents the mode of shipment for the transaction.
Final_price: Revenue generated or Total price from the transaction which includes product prices, shipping charges, etc.

## 7. DATABASE SCHEMAS

**1. Shipment Table**

```
Shipment (
        Ship_Mode varchar(20) PRIMARY KEY,
        Shipping_Charge float,
        Shipping_Desc varchar(50)
);
```

**2. Address Table**

```
Address (
        Postal_Code float PRIMARY KEY,
        Region varchar(20),
        City varchar(20),
        State varchar(20),
        Country varchar(20)
);
```

**3.Segments Table**

```
Segments (
        Segment_ID varchar(20) PRIMARY KEY,
        Segment varchar(50),
        Profit_percent integer
);
```

**4.Customer Details Table**

```
Customer_details (
        Customer_ID varchar(20) PRIMARY KEY,
         Customer_Name varchar(50),
        Segment_ID varchar(20) REFERENCES
Segments(Segment_ID) ON DELETE CASCADE ,
        Postal_Code float REFERENCES
        Address(Postal_Code) ON DELETE CASCADE
);
```

**5. Orders table**

```
Orders (
        Order_ID varchar(20),
        Product_ID varchar(20),
        Quantity integer,
        PRIMARY KEY(ORDER_ID , PRODUCT_ID)
);
```

**6. Price Table**

```
Price (
        Transaction_ID float PRIMARY KEY,
        Total_Cost_price float,
        Total_Sale_price float,
        Profit numeric GENERATED ALWAYS AS
(Total_Sale_price - Total_Cost_price) Stored
);
```

**7. Order to ship table**

```
Order_to_Ship(
        --Transaction_ID float,
        Order_ID varchar(20) Primary Key,
        Order_Date text,
        Ship_Date text
);
```

**8.Category Table**

```
Categories (
        Category_ID SERIAL PRIMARY KEY,
        Category varchar(20) ,
        Sub_Category varchar(20)
);
```

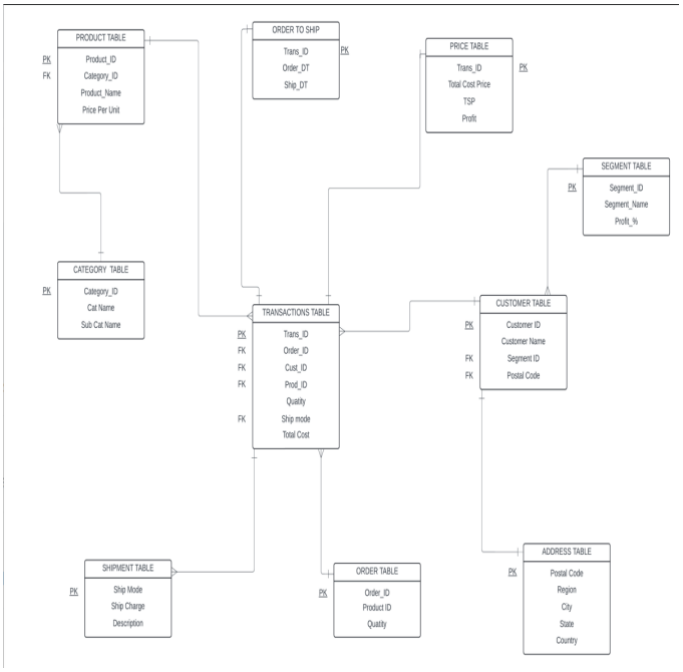
**9. Product Table**

```
Products (
         Product_ID varchar(20),
          Category_ID integer REFERENCES
        Categories(Category_ID) ON DELETE
        CASCADE,
        Product_Name varchar(200),
        Price_per_Item float
);
```

**10.Transactions Table**

```
Transactions (
        Transaction_ID float PRIMARY KEY,
        Order_ID varchar(20),
        Customer_ID varchar(20) REFERENCES
        Customer_details(Customer_ID),
        Product_ID varchar(20) REFERENCES
        Products(Product_ID),
        Quantity float,
        Ship_Mode varchar(20) REFERENCES
        Shipment(Ship_Mode),
        Final_price float
);
```

## 8. ENTITY RELATION DIAGRAM



## 9.TABLES AND KEYS

| Tables | Keys |
|---|---|
| Shipment | Primary Key: Ship_Mode<br>NOT NULL: Shipping_charges |
| Address | Primary Key: Postal_code<br>NOT NULL: Region, City, State, Country |
| Segment | Primary Key: Segment_ID<br>NOT NULL: Segment |
| Customer Details | Primary Key: Customer_ID<br>Foreign Key: REFERENCES Segments (Segment_ID), REFERENCES Address (Postal_Code)<br>NOT NULL: Customer_ID, Segment_ID, Postal_code |
| Orders | Primary Key: Order_ID, Product_ID<br>Foreign Key: REFERENCES Products (Product_ID) |
| Price | Primary Key: Transaction_ID<br>NOT NULL: Total_cost_price , Total_Sale_price |
| Order to ship | Primary Key: Order_ID<br>NOT NULL: Order_date, Ship_date |
| Category | Primary Key: Category_ID<br>NOT NULL: Category,Sub_category |
| Product | Primary Key: Ship_Mode<br>Foreign Key: REFERENCES Categories (Category_ID)<br>NOT NULL: Category_ID , Product_Name, Price_per_Item |
| Transactions | Primary Key: Transaction_ID<br>Foreign Key: REFERENCES Customer_details (Customer_ID), REFERENCES Products (Product_ID), REFERENCES Shipment (Ship_Mode)<br>NOT NULL: Customer_ID, Product_ID, Quantity , Final_price |

## 10. RELATIONS BETWEEN TABLES

- **Shipment to Transactions:** (**Many-to-One**) Shipment to Transactions: Each transaction record relates to a single shipment mode , however several transactions may use the same shipping mode.
- **Address to Customer_details:** (**One-to-one**), each customer detail relates to a single address.
- **Segments to Customer_details:** (**One-to-Many**), Several clients may be included in the same section.
- Customer details to Transactions:(**One-to-Many**) A customer may be involved in more than one transaction.
- Orders to Transactions: (**One-to-Many**) Depending on the items or dates, each order may appear in more than one transaction.
- Categories to Products: (**One-to-Many**) Many products might fall under one category.
- Products to Transactions: (**One-to-Many**) A single product may be involved in several transactions.
- Price to Transactions: (**One-to-One**) Pricing data is unique to each transaction.
- Order to Ship to Transactions: (**One-to-One**) Shipping information for each transaction is associated with a unique order ID.

## 11. FUNCTIONAL DEPENDENCIES

**Shipment Table:**
Ship_Mode -> Shipping_Charge, Shipping_Desc

**Address Table:**
Postal_Code -> Region, City, State, Country

**Segments Table:**
Segmnt_ID -> Segment, Profit_percent

**Customer_details Table:**
Customer_ID -> Customer_Name, Segment_ID, Postal_Code

**Orders Table:**
(Order_ID, Product_ID) -> Quantity

**Price Table:**
Transaction_ID -> Total_Cost_price, Total_Sale_price, Profit

**Order_to_Ship Table:**
Order_ID -> Order_Date, Ship_Date

**Categories Table:**
Category_ID -> Category, Sub_Category

**Products Table:**
Product_ID -> Category_ID, Product_Name, Price_per_Item

**Transactions Table:**
Transaction_ID -> Order_ID, Customer_ID, Product_ID, Quantity, Ship_Mode, Final_price

## 12. BCNF AND 3NF

**Shipment Table**: It is in BCNF.
It has a single candidate key: Ship_Mode.
There are no non-trivial functional dependencies other than the candidate key.

**Address Table:** It is in BCNF.
Postal_Code is the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Segments Table:** It is in BCNF.
Segment_ID is the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Customer_details Table**: It is in BCNF.
Customer_ID is the primary key.
Segment_ID and Postal_Code are foreign keys referencing other tables. There are no non-trivial functional dependencies other than the candidate key.

**Orders Table:** It is in BCNF.
Order_ID and Product_ID together form the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Price Table**: It is in BCNF.
Transaction_ID is the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Order_to_Ship Table**: It is in BCNF.
Order_ID is the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Categories Table:** It is in BCNF.
Category_ID is the primary key.
There are no non-trivial functional dependencies other than the candidate key.

**Products Table:** It is in BCNF.
Product_ID is the primary key.
Category_ID is a foreign key referencing the Categories table.There are no non-trivial functional dependencies other than the candidate key.

**Transactions Table:** It is in BCNF.
Transaction_ID is the primary key.
Order_ID, Customer_ID, Product_ID, and Ship_Mode are foreign keys.There are no non-trivial functional dependencies other than the candidate key.

Since all tables are in BCNF, they also automatically satisfy 3NF requirements. This means each table is free from transitive dependencies and redundancy, which generally ensures efficient database operations and reduces the risk of data anomalies. Therefore, it's acceptable that the tables follow 3NF.

## 13. DATBASE IMPLEMENTATION

### 13.1 CREATING TABLES

```
49 -- Create Shipment Table
50 CREATE TABLE Shipment (
51     Ship_Mode varchar(20) PRIMARY KEY,
52     Shipping_Charge float,
53     Shipping_Desc varchar(50)
54 );
```

Fig 13.1.1 Creation of Shipment Table

```
57 -- Create Address Table
58 CREATE TABLE Address (
59     Postal_Code float PRIMARY KEY,
60     Region varchar(20),
61     City varchar(20),
62     State varchar(20),
63     Country varchar(20)
64 );
```

Fig 13.1.2 Creation of Address Table

```
66 -- Create Segments Table
67 Create TABLE Segments (
68     Segment_ID varchar(20) PRIMARY KEY,
69     Segment varchar(50),
70     Profit_percent integer
71 );
```

Fig 13.1.3 Creation of Segments Table

```
73 -- Create Customer Details Table
74 CREATE TABLE Customer_details (
75
76     Customer_ID varchar(20) PRIMARY KEY,
77     Customer_Name varchar(50),
78     Segment_ID varchar(20) REFERENCES Segments(Segment_ID) ON DELETE CASCADE ,
79     Postal_Code float REFERENCES Address(Postal_Code) ON DELETE CASCADE
80 );
81
```

Fig 13.1.4 Creation of Customer Details Table.

```
82 -- Create Orders table
83 CREATE TABLE Orders (
84     Order_ID varchar(20),
85     Product_ID varchar(20),
86     Quantity integer,
87     PRIMARY KEY(ORDER_ID , PRODUCT_ID)
88 );
```

Fig 13.1.5 Creation of Orders Table

```
90 -- Create Price Table
91 CREATE TABLE Price (
92     Transaction_ID float PRIMARY KEY,
93     Total_Cost_price float,
94     Total_Sale_price float,
95     Profit numeric GENERATED ALWAYS AS (Total_Sale_price - Total_Cost_price) Stored
96 );
```

Fig 13.1.6 Creation of Price Table

**In the price table we have added a default column profit which is always total_cost_price – total_sale_price whenever a row is inserted .**

```
-- Create Order to ship table
CREATE TABLE Order_to_Ship(
    --Transaction_ID float,
    Order_ID varchar(20) Primary Key,
    Order_Date text,
    Ship_Date text

);
```

Fig 13.1.7 Creation of Order to Ship table

```
-- Create Category Table

CREATE TABLE Categories (
    Category_ID SERIAL PRIMARY KEY,
    Category varchar(20) ,
    Sub_Category varchar(20)
);
```

Fig 13.1.8 Creation of Category Table

```
-- Create Product Table

CREATE TABLE Products (

    Product_ID varchar(20),
    Category_ID integer REFERENCES Categories(Category_ID) ON DELETE CASCADE,
    Product_Name varchar(200),
    Price_per_Item float
);
```

Fig 13.1.9 Creation of Product Table

```
Create Table Transactions (
    Transaction_ID float PRIMARY KEY,
    Order_ID varchar(20),
    Customer_ID varchar(20) REFERENCES Customer_details(Customer_ID),
    Product_ID varchar(20) REFERENCES Products(Product_ID),
    Quantity float,
    Ship_Mode varchar(20) References Shipment(Ship_Mode),
    Final_price float
);
```

Fig 13.1.10 Creation of Transactions table

## 13.2 INSERTING ATTRIBUTE VALUES INTO TABLES

```
-- Inserting into Segments table

insert into Segments select distinct segment_id , segment , profit_percent from mega_table;
```

Fig 13.2.1 Inserting into Segments table

```
-- Inserting into Customer Details table
insert into Customer_details(Customer_ID) select distinct Customer_ID from mega_table;

UPDATE Customer_details AS a
SET
    Customer_Name = b.Customer_Name,
    Segment_ID = b.Segment_ID,
    Postal_Code = b.Postal_Code

FROM mega_table AS b
WHERE a.Customer_ID = b.Customer_ID;
```

Fig 13.2.2 Inserting into Customer Details table

```
-- Inserting into Category table

insert into Categories(Category , Sub_Category) select distinct Category , Sub_Category from Mega_Table;
```

Fig 13.2.3 Inserting into Category table

```
-- Inserting into Order to ship table

insert into Order_to_Ship select distinct Order_id , order_date , Ship_date from mega_table;
```

Fig 13.2.4 Inserting into Order to ship table

```
-- Inserting into Price table

insert into Price(Transaction_id , Total_Cost_price ) select transaction_id , (price_per_item * Quantity) from mega_table;

UPDATE Price AS a
SET Total_Sale_price = a.Total_Cost_price * (1 + b.profit_percent / 100)
FROM mega_table AS b
WHERE a.Transaction_ID = b.Transaction_ID;
```

Fig 13.2.5 Inserting into Price table

```
-- Inserting into Orders table

insert into orders select order_id ,product_id , quantity from mega_table;

'''
-- Checking for duplicates in order_id and product_id Column
SELECT order_id, product_id, COUNT(*)
FROM Orders
GROUP BY order_id, product_id
HAVING COUNT(*) > 1;
'''
```

Fig 13.2.6 Inserting into Orders table

```sql
-- Inserting into Customer Details table
insert into Customer_details(Customer_ID) select distinct Customer_ID from mega_table;

UPDATE Customer_details AS a
SET
    Customer_Name = b.Customer_Name,
    Segment_ID = b.Segment_ID,
    Postal_Code = b.Postal_Code

FROM mega_table AS b
WHERE a.Customer_ID = b.Customer_ID;
```

Fig 13.2.7 Inserting into Customer Details table

```sql
-- Inserting into Products table

insert into Products(product_id,category_id,product_name) select distinct Product_ID , Category_ID , Product_Name from Mega_table;
```

Fig 13.2.8 Inserting into Product table

```sql
insert into Address(postal_code) select distinct postal_code from Mega_Table;

UPDATE Address AS a
SET
    region = b.region,
    city = b.city,
    state = b.state,
    Country = b.Country
FROM mega_table AS b
WHERE a.postal_code = b.postal_code;
```

Fig 13.2.9 Insert command into Address table

```sql
-- Inserting data into transactions table
insert into Transactions(Transaction_ID,Order_ID,Customer_ID,Product_ID,Quantity,Ship_Mode)
select distinct Transaction_ID,Order_ID,Customer_ID,Product_ID,Quantity,Ship_Mode from mega_table;
```

Fig 13.2.10 Inserting data into Transactions table

## 13.3 QUERIES

```sql
-- 1 Date Difference between ORDER DATE AND SHIP DATE
SELECT
    order_ID,
    order_date,
    ship_date,
    (ship_date - order_date) AS days_between
FROM
    order_to_ship;
```

Fig 13.3.1 Query 1

The difference in days between the Ship_Date and Order_Date columns is determined using the formula DATEDIFF(ship_Date - order_Date). The columns that are selected in the output are Order_ID, Order_Date, and Ship_Date. The table from which the information is taken is called Order_to_Ship.

| | order_id [PK] character varying (20) | order_date date | ship_date date | days_between integer |
|---|---|---|---|---|
| 1 | CA-2014-138240 | 2014-10-09 | 2014-10-14 | 5 |
| 2 | CA-2016-124562 | 2016-12-08 | 2016-12-12 | 4 |
| 3 | CA-2016-108581 | 2016-06-20 | 2016-06-26 | 6 |
| 4 | CA-2014-110100 | 2014-04-25 | 2014-04-29 | 4 |
| 5 | US-2014-120175 | 2014-11-28 | 2014-12-03 | 5 |
| 6 | CA-2014-154893 | 2014-12-21 | 2014-12-27 | 6 |
| 7 | CA-2016-162187 | 2016-12-11 | 2016-12-11 | 0 |
| 8 | CA-2015-145065 | 2015-12-12 | 2015-12-15 | 3 |
| 9 | CA-2016-118934 | 2016-08-09 | 2016-08-14 | 5 |
| 10 | CA-2015-169733 | 2015-10-22 | 2015-10-26 | 4 |
| 11 | CA-2017-154816 | 2017-11-06 | 2017-11-10 | 4 |

Fig 13.3.2 Query 1 result

```sql
-- 3 Number of customers ordering from each region
SELECT
    a.Region,
    COUNT(cd.Customer_ID) AS Number_of_Customers
FROM
    Address a
JOIN
    Customer_details cd ON a.Postal_Code = cd.Postal_Code
GROUP BY
    a.Region
ORDER BY
    Number_of_Customers DESC;
```

Fig 13.3.3 Query 2

This SQL query is used to retrieve the total number of customers ordering from various regions.

| | region character varying (20) | number_of_customers bigint |
|---|---|---|
| 1 | West | 252 |
| 2 | East | 223 |
| 3 | Central | 181 |
| 4 | South | 137 |

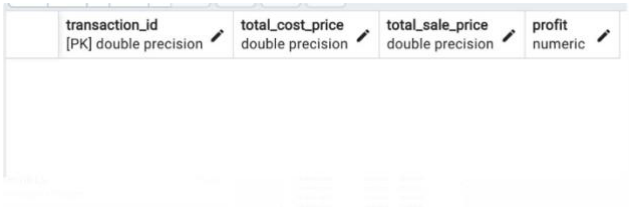
Fig 13.3.4 Query 2 result

```sql
SELECT
    Transaction_ID,
    Total_Cost_price,
    Total_Sale_price,
    Profit
FROM
    Price
WHERE
    Profit < 0;
```

Fig 13.3.5 Query 3

This SQL query uses the WHERE clause to apply a filter that only includes rows where the Profit is less than 0, indicating a negative profit or loss. It does this by selecting

particular values from the Price table (Transaction_ID, Total_Cost_price, Total_Sale_price, and Profit).



| transaction_id [PK] double precision | total_cost_price double precision | total_sale_price double precision | profit numeric |
|---|---|---|---|

Fig 13.3.6 Query 3 result

There are no transactions with a loss in the Price table.

```
-- 4 Average transaction value for each product
SELECTc.Category,AVG(t.Final_price) AS Avg_Transaction_Value
FROM
    Categories c
JOIN
    Products p ON c.Category_ID = p.Category_ID
JOIN
    Transactions t ON p.Product_ID = t.Product_ID
GROUP BY
    c.Category;
```

Fig 13.3.7 Query 4

This SQL query is used to give the total transaction value of the products sold grouped by the category of the products.

| | category character varying (20) | avg_transaction_value double precision |
|---|---|---|
| 1 | Furniture | 419.9187842910075 |
| 2 | Office Supplies | 150.01437728125612 |
| 3 | Technology | 515.8127327850248 |

Fig 13.3.8 Query 4 result

```
-- 5 Calculating the spending rank for each customer based on the total amount spent by them

WITH Customer_Sales AS (
    SELECT cd.Customer_ID, cd.Customer_Name,
        SUM(t.Final_price) AS Total_Spent
    FROM Customer_details cd
    JOIN Transactions t ON cd.Customer_ID = t.Customer_ID
    GROUP BY cd.Customer_ID, cd.Customer_Name
),
Ranked_Customers AS (
    SELECT
        Customer_Name,
        Total_Spent,
        RANK() OVER (ORDER BY Total_Spent DESC) AS Spending_Rank
    FROM Customer_Sales
)
SELECT Customer_Name, Total_Spent, Spending_Rank
FROM Ranked_Customers
WHERE Spending_Rank <= 10;
```

Fig 13.3.9 Query 5

In order to determine which customers are the highest spenders, this SQL query uses two common table expressions (CTEs) to examine customer spending trends. It first calculates the total amount spent by a customer till date and ranks them according to the amount spent in descending order.

| | customer_name character varying (50) | total_spent double precision | spending_rank bigint |
|---|---|---|---|
| 1 | Sean Miller | 26422.8961 | 1 |
| 2 | Tamara Chand | 21019.899800000007 | 2 |
| 3 | Raymond Buch | 18390.806799999995 | 3 |
| 4 | Ken Lonsdale | 17305.810799999992 | 4 |
| 5 | Adrian Barton | 17174.640399999997 | 5 |
| 6 | Sanjit Chand | 16923.888799999997 | 6 |
| 7 | Hunter Lopez | 15577.957599999996 | 7 |
| 8 | Tom Ashbrook | 15485.401000000002 | 8 |
| 9 | Sanjit Engle | 14834.937600000001 | 9 |
| 10 | Christopher Conant | 14449.574399999998 | 10 |

Fif 13.3.10 Query 5 result

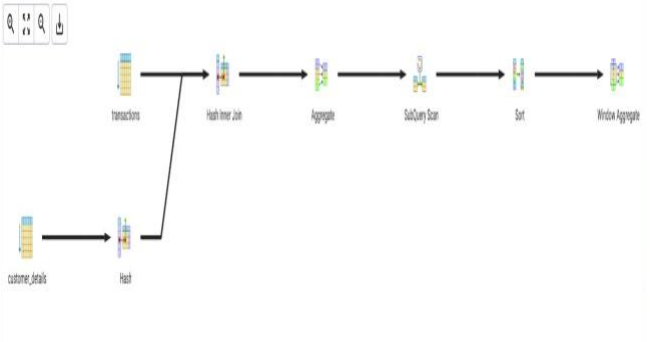



Fig 13.3.11 Query 5 explaination tool

```
-- 6 Analysis on each month total sales and then comparing the sales with the previous month sales

WITH Monthly_Sales AS (
    SELECT
        c.Category,
        EXTRACT(YEAR FROM o.Ship_Date) AS Sale_Year,
        EXTRACT(MONTH FROM o.Ship_Date) AS Sale_Month,
        SUM(t.Final_price) AS Total_Sales
    FROM Categories c
    JOIN Products p ON c.Category_ID = p.Category_ID
    JOIN Transactions t ON p.Product_ID = t.Product_ID
    JOIN Order_to_Ship o ON t.Order_ID = o.Order_ID
    GROUP BY c.Category, Sale_Year, Sale_Month
)
SELECT Category,Sale_Year,Sale_Month,Total_Sales,
    LAG(Total_Sales, 1) OVER (PARTITION BY Category ORDER BY Sale_Year, Sale_Month) AS Previous_Month_Sales
FROM Monthly_Sales
ORDER BY Category, Sale_Year, Sale_Month;
```

Fig 13.3.12 Query 6

Monthly Sales, which determines the monthly total sales for every product category. We use the LAG() window function to put the previous month sales beside the current month sales and we compare the sales.

| | category character varying (20) | sale_year numeric | sale_month numeric | total_sales double precision | previous_month_sales double precision |
|---|---|---|---|---|---|
| 1 | Furniture | 2014 | 1 | 6452.6122000000005 | [null] |
| 2 | Furniture | 2014 | 2 | 3003.7048 | 6452.6122000000005 |
| 3 | Furniture | 2014 | 3 | 13041.8107 | 3003.7048 |
| 4 | Furniture | 2014 | 4 | 11194.012300000004 | 13041.8107 |
| 5 | Furniture | 2014 | 5 | 8651.124600000001 | 11194.012300000004 |
| 6 | Furniture | 2014 | 6 | 14438.822919999999 | 8651.124600000001 |
| 7 | Furniture | 2014 | 7 | 11421.98215 | 14438.822919999999 |
| 8 | Furniture | 2014 | 8 | 11149.7562 | 11421.98215 |
| 9 | Furniture | 2014 | 9 | 22618.62889 | 11149.7562 |
| 10 | Furniture | 2014 | 10 | 15350.1717 | 22618.62889 |
| 11 | Furniture | 2014 | 11 | 24417.88177 | 15350.1717 |
| 12 | Furniture | 2014 | 12 | 31973.30290000001 | 24417.88177 |
| 13 | Furniture | 2015 | 1 | 17732.409320000002 | 31973.30290000001 |

Fig 13.3.13 Query 6 result

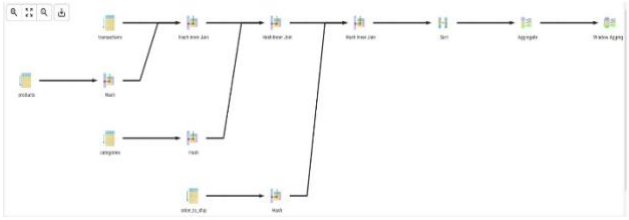


Fig 13.3.14 Query 6 explanation tool

```sql
WITH Shipping_Duration AS (
    SELECT
        s.Ship_Mode,
        AVG(s.Shipping_Charge) AS Avg_Shipping_Charge,
        AVG(o.ship_date - o.order_date) AS Average_Delivery_Days
    FROM Shipment s
    JOIN Transactions t ON s.Ship_Mode = t.Ship_Mode
    JOIN Order_to_Ship o ON t.Order_ID = o.Order_ID
    GROUP BY s.Ship_Mode
)
SELECT
    Ship_Mode,
    Avg_Shipping_Charge,
    Average_Delivery_Days,
    RANK() OVER (ORDER BY Average_Delivery_Days, Avg_Shipping_Charge) AS Efficiency_Rank
FROM Shipping_Duration
ORDER BY Efficiency_Rank;
```

Fig 13.3.15 Query 7

The average shipping charge and average number of dates took for the product to get shipped are together used to rank the efficiency of the shipping method.

| | ship_mode [PK] character varying (20) | avg_shipping_charge double precision | average_delivery_days numeric | efficiency_rank bigint |
|---|---|---|---|---|
| 1 | Same Day | 30 | 0.04457364341085271318 | 1 |
| 2 | First Class | 20 | 2.1719487525286581 | 2 |
| 3 | Second Class | 15 | 3.2465462274176408 | 3 |
| 4 | Standard Class | 10 | 5.0112671173513607 | 4 |

Fig 13.3.16 Query 7 result

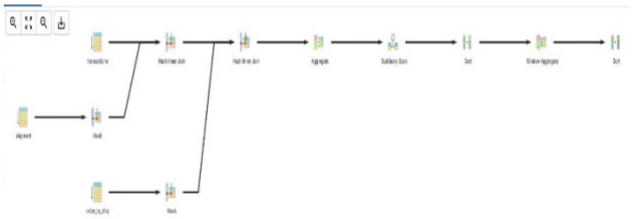


Fig 13.3.17 Query 7 explanation tool

## 13.4 DELETION AND UPDATION

We have removed the duplicate product_id's from the product table since this data is redundant and is for no use to us. We have achieved this by using the power of row_number() which flags the rows with number and using Common table expression we can store this and later use this to delete the duplicates.

```sql
-- Deleting DUplicates from products table
With duplicates AS (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY product_id) AS row_num
    FROM products
)
DELETE FROM products
WHERE (product_id) IN (SELECT product_id FROM duplicates WHERE row_num > 1);
```

Fig 13.4.1 Deleting Duplicates from Products table

Inserting data into the product table has been difficult for us. We have faced a lot of issues with the duplicate data which is coming from the mega_table. Our plan of action was to first remove the primary key constraint from the product table and then load the data with duplicates to the products tables. After loading the data we have deleted the duplicates and then later added the primary key constraint to the table.

```sql
-- Adding primary Key Constraint to product table
ALTER TABLE products
ADD CONSTRAINT product_id_pk PRIMARY KEY (product_id);
```

Fig 13.4.2 Adding primary key constraint to product table

Based on the product ID we are updated the column price per unit from the mega table

```
-- Update the mega_table with the updated price
UPDATE Mega_table AS a
SET
    price_per_item = b.price_per_item

FROM Products AS b
WHERE a.Product_id = b.Product_id;

```

Fig 13.4.3 Update the mega table with the updated price.

For the transactions table the final_price column is filled with the sum of total_sale_price column from the price table and the shipping charge based on ship mode from the shipment table.

```
-- Calculating final_price for transactions table
UPDATE transactions
SET final_price = (
    SELECT p.Total_Sale_price + s.shipping_charge
    FROM price p
    JOIN shipment s ON transactions.ship_mode = s.ship_mode
    WHERE transactions.Transaction_ID = p.Transaction_ID
);
```

Fig 13.4.4 Calculating final price for transactions table

The above SQL code is used to created a hash based index on the transaction_id column from the transactions table. This index is created by using the hash keyword.

```
CREATE INDEX trans_idx  ON transactions USING HASH (transaction_id);

SELECT * FROM pg_indexes
WHERE indexname = 'trans_idx';

```

Fig 13.4.5 Indexing on Transaction table

We can view the created index from the pg_indexes table by filtering with the index name

| | schemaname name | tablename name | indexname name | tablespace name | indexdef text |
|---|---|---|---|---|---|
| 1 | public | transactions | trans_idx | [null] | CREATE INDEX trans_idx ON public.transactions USING hash (transaction_... |

The below SQL code is used to create B treebased index on the Order_id and Product_id column from the Orders table. There is no such keyword to create b tree index since the default index type taken by postgres is B tree indexing.

```
CREATE INDEX order_product  ON Orders(Order_ID , Product_ID);

SELECT * FROM pg_indexes
WHERE indexname = 'order_product';

```

Fig 13.4.6 Indexing on Order table

We can view the created index from the pg_indexes table by filtering with the index name

| | schemaname name | tablename name | indexname name | tablespace name | indexdef text |
|---|---|---|---|---|---|
| 1 | public | orders | order_product | [null] | CREATE INDEX order_product ON public.orders USING btree (order_id, product_... |

Thought we created new indexes since the dataset is very small with just approximately 9000 rows there is not much difference in the query execution time. When the database is scaled up with lots of rows , indexing plays a vital role in reducing query runtime .

14. Graphical User Interface

In the below GUI we have few pre programmed buttons and a text area to write SQL queries.

Each button has a unique significance and executes a query which answers a specific problem .

There is a RUN QUERY button which when clicked runs the SQL query and gives us the output under the Query Results Section .


Fig 14.1 User Interface

The sample of output when Customer for each Region button is clicked .

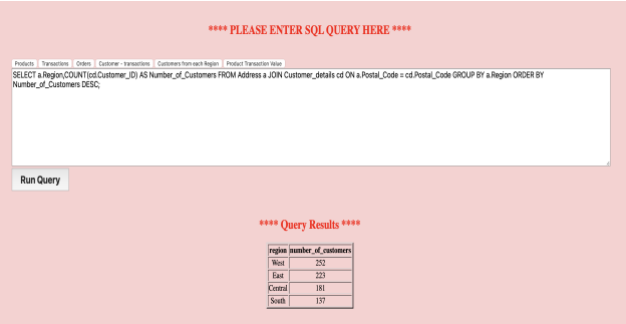



Fig 14.2 Programmed Button output

Other than the buttons we can also execute queries by typing them in the text box provided below.


Fig 14.3 Output from SQL query in text box

This Graphical user interface also has the capability to show errors when we perform anything wrong. The below picture is related to adding a duplicate value to the primary key column .

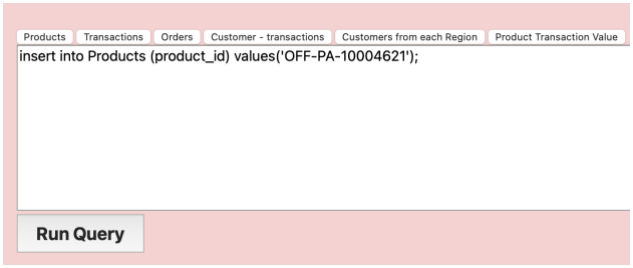



Fig 14.4 Running a Faulty Query

After hitting on the Run Query Button
we can see the following screen .

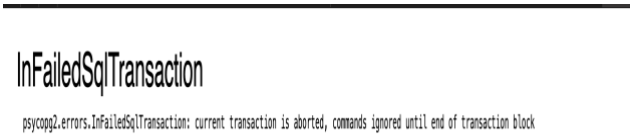



Fig 14.5 Error from the query

15. References :

Kaggle : https://www.kaggle.com

Pandas : https://pandas.pydata.org

Flask : https://flask.palletsprojects.com/en/3.0.x/installation/

Pg Admin : https://www.pgadmin.org/docs/