**Group 20 - E-commerce Order Tracking and Customer Analytics System**

1. **Brief Description of the Problem:**

In today’s fast-paced e-commerce environment, managing customer orders, monitoring inventory levels, processing payments, and evaluating customer behavior are all difficult tasks for firms.  Businesses frequently suffer from overstocking, stockouts, delayed shipping, and a lack of information on consumer buying habits when they don't have a centralized system. Poor customer service, lost sales, and an inability to maximize marketing tactics are the results of these problems.

In order to effectively manage orders, preserve real-time inventory data, and use customer analytics to enhance business outcomes, an integrated system is necessary to get over these challenges. Effective shipment and payment management also guarantees that clients' financial information is processed securely and that their products arrive on schedule with little interruptions. Putting such a system in place will improve customer happiness, expedite processes, and facilitate data-driven decision-making for future company expansion.

**b) How the Proposed Database System Will Address the Information Problem:**

The proposed database system will serve as a comprehensive e-commerce solution, addressing three core areas: **Order Tracking**, **Customer Management**, and **Inventory Monitoring**, while expanding into **Analytics** and **Review Management**.

1. **Order Tracking**:

The system will track all stages of an order, from the moment a customer adds items to their cart to payment confirmation and shipment. Orders will be linked to products through detailed order items, ensuring that businesses can track which products were sold, in what quantity, and at what price.

2. **Customer Information Management**:

It will store complete customer details, including their order history, reviews they’ve written, and purchasing patterns. This information can be used to provide personalized marketing strategies and customer support.

4. **Payment Processing and Shipment Tracking**:

Orders will be processed through secure payment gateways, and the system will track the shipment status of each order. This ensures that customers are kept informed about the status of their purchases at all times.

5. **Review Management**:

The system will store customer reviews of products, allowing businesses to assess customer satisfaction and address any issues promptly.

In summary, the system will provide an integrated solution for tracking orders, managing customer relationships, maintaining accurate inventory levels, processing payments, tracking shipments, and leveraging customer reviews and analytics to drive sales and improve the customer experience.

**c) Preliminary Entity-Relation Diagram (ERD):**

The following are the **key entities and relationships** included in the ERD, reflecting the updated structure with **8 entities**:

1. **Customer**: Stores customer details such as name, email, phone number, address, and date joined.

• **Attributes**: CustomerID (PK), Name, Email, PhoneNumber, Address, DateJoined.

2. **Order**: Represents customer orders.

• **Attributes**: OrderID (PK), CustomerID (FK), ProductID(FK), DiscountID(FK), OrderDate, Status, TotalAmount.

3. **Product**: Contains information about the products sold on the platform.

• **Attributes**: ProductID (PK), Name, Description, Price, StockQuantity, CountForSale, CountInInventory, Price.

4. **Supplier**: Stores details about the suppliers providing products.

• **Attributes**: SupplierID (PK), Name, ContactInfo, Address.

5. **Shipment**: Tracks the shipping details of each order.

• **Attributes**: ShipmentID (PK), OrderID (FK), ShipmentDate, Carrier, TrackingNumber, Status.

6. **Payment**: Stores payment details for each order.

• **Attributes**: PaymentID (PK), OrderID (FK), PaymentDate, PaymentMethod, AmountPaid.

7. **Review**: Allows customers to submit reviews for products they have purchased.

• **Attributes**: ReviewID (PK), CustomerID (FK), ProductID (FK), Rating, ReviewText, ReviewDate.

8. **Discount**: Represents discounts applied to orders.

• **Attributes**: DiscountID (PK), DiscountCode, ExpiryDate, DiscountAmount.

**Entity Relationships:**

• **Customer** can place multiple **Orders**.

• **Order** contains multiple **OrderItems**, each linked to a specific **Product**.

• **Product** is linked to **OrderItems** and also associated with **Reviews** by customers.

• **Product** inventory levels are tracked in the **Inventory** table, with suppliers providing the products.

• **Payment** and **Shipment** are tied to each **Order**.

• **Discounts** are applied to **Orders** to reduce the total amount.

A diagram of a company

Description automatically generated

A diagram of a computer

Description automatically generated

**Entity Relational Schema**

**Suppliers**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Supplier\_ID | INT NOT NULL | Primary Key |
| Name | VARCHAR(255) |  |
| Address | VARCHAR(255) |  |
| Contact\_Info | VARCHAR(255) |  |

**Products**

|  |  |  |
| --- | --- | --- |
| **Column** Name | **Data Type** | **Key** |
| Product\_ID | INT NOT NULL | Primary Key |
| Supplier\_ID | VARCHAR(100) | Foreign Key |
| Description | TEXT |  |
| Stock\_Quantity | INT |  |
| Price | DECIMAL(10, 2) |  |
| Name | VARCHAR(255) |  |
| Count\_for\_Sale | INT |  |
| Count\_in\_Inventory | INT |  |

**Customers**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Customer\_ID | **I**NT NOT NULL | Primary Key |
| Name | VARCHAR(255) |  |
| Address | VARCHAR(255) |  |
| Phone\_Number | VARCHAR(15) |  |
| Email | VARCHAR(255) |  |

**Reviews**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Review\_ID | INT NOT NULL | Primary\_Key |
| Customer\_ID | VARCHAR(100) | Foreign Key 1 |
| Product\_ID | VARCHAR(100) | Foreign Key 2 |
| Rating | INT |  |
| Review\_Date | DATE |  |
| Review\_Text | TEXT |  |

**Orders**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Order\_ID | INT NOT NULL | Primary\_Key |
| Customer\_ID | VARCHAR(100) | Foreign Key 1 |
| Product\_ID | VARCHAR(100) | Foreign Key 2 |
| Discount\_ID | VARCHAR(100) | Foreign Key 3 |
| Total\_Amount | DECIMAL(10, 2) |  |
| Order\_Date | DATE |  |
| Status | VARCHAR(100) |  |

**Discounts**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Discount\_ID | INT NOT NULL | Primary\_Key |
| Discount\_Code | VARCHAR(50) |  |
| Discount\_Amount | DECIMAL(10, 2) |  |
| Expiry\_Date | DATE |  |

**Shipments**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Shipment\_ID | INT NOT NULL | Primary\_Key |
| Order\_ID | VARCHAR(100) | Foreign\_Key 1 |
| Carrier | VARCHAR(255) |  |
| Status | VARCHAR(50) |  |
| Tracking\_Number | VARCHAR(50) |  |
| Shipment\_Date | DATE |  |

**Payments**

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Key** |
| Payment\_ID | INT NOT NULL | Primary\_Key |
| Order\_ID | VARCHAR(100) | Foreign\_Key 1 |
| Payment\_Method | VARCHAR(50) |  |
| Amount\_Paid | DECIMAL(10, 2) |  |
| Payment\_Date | DATE |  |

**Database Creation**

DROP DATABASE IF EXISTS Orderdata;

CREATE DATABASE Orderdata;

USE Orderdata;

The SQL script begins by ensuring that any existing `Orderdata` database is removed using `DROP DATABASE IF EXISTS Orderdata;`, preventing conflicts with existing databases of the same name. The `CREATE DATABASE Orderdata;` statement then creates a new, clean database named `Orderdata`. Following this, `USE Orderdata;` sets the newly created database as the active one for subsequent operations. This ensures that any `CREATE TABLE` or `INSERT` statements executed afterward will apply to `Orderdata`. These commands are foundational to setting up a fresh, conflict-free database environment for data structure creation and data population.

**Supplier Table Creation**

DROP TABLE IF EXISTS Suppliers;

-- Table: Suppliers

CREATE TABLE Suppliers (

Supplier\_ID VARCHAR(100) NOT NULL PRIMARY KEY,

Name VARCHAR(255),

Address VARCHAR(255),

Contact\_Info VARCHAR(255)

);

**Insert Statements for Suppliers Table**

INSERT INTO Suppliers (Supplier\_ID, Name, Address, Contact\_Info) VALUES ('SUPP\_1', 'Emily Smith', '444 Willow Dr., Memphis, Washington 98236', 'email: michaelbrown@example.com, phone: 121-131-8184');

INSERT INTO Suppliers (Supplier\_ID, Name, Address, Contact\_Info) VALUES ('SUPP\_2', 'Linda Jones', '482 Willow Dr., Columbus, Tennessee 47978', 'email: michaelwilliams@gmail.com, phone: 382-675-8230');

INSERT INTO Suppliers (Supplier\_ID, Name, Address, Contact\_Info) VALUES ('SUPP\_3', 'Karen Jones', '679 Maple Ave., Louisville, Colorado 27758', 'email: robertsmith@example.com, phone: 140-608-8835');

INSERT INTO Suppliers (Supplier\_ID, Name, Address, Contact\_Info) VALUES ('SUPP\_4', 'Sarah Wilson', '302 Pine St., Louisville, Pennsylvania 11926', 'email: janetaylor@example.com, phone: 183-583-5390');

INSERT INTO Suppliers (Supplier\_ID, Name, Address, Contact\_Info) VALUES ('SUPP\_5', 'Sarah Smith', '934 Main St., Los Angeles, California 37227', 'email: robertdavis@example.com, phone: 187-894-5055');

This SQL script first ensures the `Suppliers` table does not exist by using `DROP TABLE IF EXISTS Suppliers;`. The `CREATE TABLE Suppliers` statement then defines the structure of the table with four columns: `Supplier\_ID` (a unique identifier, primary key, and cannot be `NULL`), `Name` (supplier's name), `Address` (supplier's address), and `Contact\_Info` (contact details such as phone and email). Each field uses `VARCHAR` to store string data. The `INSERT INTO` statements populate the table with sample supplier data, assigning specific `Supplier\_ID`, `Name`, `Address`, and `Contact\_Info` values. The table can manage up to 300 records or more efficiently.

**Products Table Creation**

CREATE TABLE Products (

Product\_ID INT NOT NULL PRIMARY KEY,

Supplier\_ID VARCHAR(100),

Name VARCHAR(255),

Description TEXT,

Price DECIMAL(10, 2),

Stock\_Quantity INT,

Count\_for\_Sale INT,

Count\_in\_Inventory INT,

FOREIGN KEY (Supplier\_ID) REFERENCES Suppliers(Supplier\_ID)

);

**Insert Statements for Products Table**

INSERT INTO Products (Product\_ID, Supplier\_ID, Name, Description, Price, Stock\_Quantity, Count\_for\_Sale, Count\_in\_Inventory) VALUES (1, 'SUPP\_223', 'Slow Camera', 'Perfect for everyday use.', 57.576726655405764, 24, 6, 11);

INSERT INTO Products (Product\_ID, Supplier\_ID, Name, Description, Price, Stock\_Quantity, Count\_for\_Sale, Count\_in\_Inventory) VALUES (2, 'SUPP\_29', 'Small Camera', 'Stylish and fashionable.', 47.03901350429274, 4, 87, 42);

INSERT INTO Products (Product\_ID, Supplier\_ID, Name, Description, Price, Stock\_Quantity, Count\_for\_Sale, Count\_in\_Inventory) VALUES (3, 'SUPP\_185', 'Green Watch', 'A high-quality product.', 14.990449193367159, 76, 26, 34);

INSERT INTO Products (Product\_ID, Supplier\_ID, Name, Description, Price, Stock\_Quantity, Count\_for\_Sale, Count\_in\_Inventory) VALUES (4, 'SUPP\_276', 'Slow Laptop', 'Perfect for everyday use.', 46.80768897002874, 64, 43, 70);

INSERT INTO Products (Product\_ID, Supplier\_ID, Name, Description, Price, Stock\_Quantity, Count\_for\_Sale, Count\_in\_Inventory) VALUES (5, 'SUPP\_253', 'Modern Shirt', 'A high-quality product.', 84.98903238714999, 15, 91, 28);

The script starts with `DROP TABLE IF EXISTS Products;` to delete any existing `Products` table to avoid conflicts. The `CREATE TABLE Products` statement defines the table structure with columns: `Product\_ID` (unique identifier and primary key), `Supplier\_ID` (links to the `Suppliers` table), `Name` (product name), `Description` (product details), `Price` (up to 10 digits, 2 decimal places), `Stock\_Quantity`, `Count\_for\_Sale`, and `Count\_in\_Inventory` (all integers). The `FOREIGN KEY` constraint links `Supplier\_ID` to the `Suppliers` table for relational integrity. The `INSERT INTO` statements add sample products with specific IDs, supplier references, names, descriptions, prices, and inventory details, managing around 300 records efficiently.

**Customers Table Creation**

CREATE TABLE Customers (

Customer\_ID INT NOT NULL PRIMARY KEY,

Name VARCHAR(255),

Address VARCHAR(255),

Phone\_Number VARCHAR(15),

Email VARCHAR(255)

);

**Insert Statements for Customers Table**

INSERT INTO Customers (Customer\_ID, Name, Address, Phone\_Number, Email) VALUES (1, 'Karen Davis', '519 Elm St., Baltimore, Illinois 32952', '176-572-3258', 'johnbrown@hotmail.com');

INSERT INTO Customers (Customer\_ID, Name, Address, Phone\_Number, Email) VALUES (2, 'Robert Miller', '237 Pine Dr., Columbus, Maryland 91100', '334-438-5349', 'davidwilliams@hotmail.com');

INSERT INTO Customers (Customer\_ID, Name, Address, Phone\_Number, Email) VALUES (3, 'Karen Wilson', '739 Willow St., Columbus, Tennessee 50893', '787-736-4640', 'johnwilson@example.com');

INSERT INTO Customers (Customer\_ID, Name, Address, Phone\_Number, Email) VALUES (4, 'John Davis', '331 River Rd., San Antonio, Missouri 35400', '298-688-4176', 'williammoore@hotmail.com');

INSERT INTO Customers (Customer\_ID, Name, Address, Phone\_Number, Email) VALUES (5, 'Michael Brown', '885 Oak Dr., Philadelphia, Illinois 90354', '694-512-7580', 'karenanderson@hotmail.com');

The `Customers` table is created after ensuring no existing table conflicts by using `DROP TABLE IF EXISTS Customers;`. The `CREATE TABLE` statement defines five columns: `Customer\_ID` (unique primary key), `Name`, `Address`, `Phone\_Number`, and `Email`, all defined as `VARCHAR` to handle text data. The table structure supports storing detailed customer information. The `INSERT INTO` statements populate the table with sample records containing customer IDs, names, addresses, phone numbers, and email addresses. This setup enables efficient customer data management, ensuring each customer has unique identification while facilitating retrieval and querying of customer contact and address details.

**Reviews Table Creation**

CREATE TABLE Reviews (

Review\_ID INT NOT NULL PRIMARY KEY,

Customer\_ID VARCHAR(100),

Product\_ID VARCHAR(100),

Rating INT,

Review\_Date DATE,

Review\_Text TEXT,

FOREIGN KEY (Customer\_ID) REFERENCES Customers(Customer\_ID),

FOREIGN KEY (Product\_ID) REFERENCES Products(Product\_ID)

);

**Insert Statements for Reviews Table**

INSERT INTO Reviews (Review\_ID, Customer\_ID, Product\_ID, Rating, Review\_Date, Review\_Text) VALUES (1, 180, 4, 4, '2022-08-05', 'Its exactly what I was looking for.');

INSERT INTO Reviews (Review\_ID, Customer\_ID, Product\_ID, Rating, Review\_Date, Review\_Text) VALUES (2, 158, 283, 5, '2023-08-04', 'I would definitely recommend it.');

INSERT INTO Reviews (Review\_ID, Customer\_ID, Product\_ID, Rating, Review\_Date, Review\_Text) VALUES (3, 259, 152, 5, '2023-10-17', 'I have some issues with this product.');

INSERT INTO Reviews (Review\_ID, Customer\_ID, Product\_ID, Rating, Review\_Date, Review\_Text) VALUES (4, 127, 162, 5, '2023-09-13', 'I love this product!');

INSERT INTO Reviews (Review\_ID, Customer\_ID, Product\_ID, Rating, Review\_Date, Review\_Text) VALUES (5, 85, 147, 4, '2023-12-11', 'I love this product!');

The `Reviews` table is created to store customer feedback. It includes `Review\_ID` (primary key), `Customer\_ID` (links to the `Customers` table), `Product\_ID` (links to the `Products` table), `Rating` (an integer score), `Review\_Date` (date of review), and `Review\_Text` (textual feedback). The `FOREIGN KEY` constraints ensure referential integrity between `Customer\_ID` and `Product\_ID`. The `INSERT INTO` statements add sample reviews with specific IDs, customer references, product references, ratings, dates, and review comments. This structure supports capturing detailed product feedback and ensures relationships between reviews, customers, and products are maintained, enabling efficient querying and analysis of customer satisfaction.

**Orders Table Creation**

CREATE TABLE Orders (

Order\_ID INT NOT NULL PRIMARY KEY,

Customer\_ID VARCHAR(100),

Product\_ID VARCHAR(100),

Discount\_ID VARCHAR(100),

Total\_Amount DECIMAL(10, 2),

Order\_Date DATE,

Status VARCHAR(100),

FOREIGN KEY (Customer\_ID) REFERENCES Customers(Customer\_ID),

FOREIGN KEY (Product\_ID) REFERENCES Products(Product\_ID),

FOREIGN KEY (Discount\_ID) REFERENCES Discounts(Discount\_ID)

);

**Insert Statements for Orders Table**

INSERT INTO Orders (Order\_ID, Customer\_ID, Product\_ID, Discount\_ID, Total\_Amount, Order\_Date, Status) VALUES (1, 4, 231, 256, 387.7785118570393, '2023-01-05', 'Pending');

INSERT INTO Orders (Order\_ID, Customer\_ID, Product\_ID, Discount\_ID, Total\_Amount, Order\_Date, Status) VALUES (2, 232, 17, 109, 182.9654297485892, '2023-08-19', 'Pending');

INSERT INTO Orders (Order\_ID, Customer\_ID, Product\_ID, Discount\_ID, Total\_Amount, Order\_Date, Status) VALUES (3, 210, 46, 147, 38.72617908752986, '2022-06-24', 'Delivered');

INSERT INTO Orders (Order\_ID, Customer\_ID, Product\_ID, Discount\_ID, Total\_Amount, Order\_Date, Status) VALUES (4, 118, 22, 255, 348.3037038794669, '2023-10-08', 'Canceled');

INSERT INTO Orders (Order\_ID, Customer\_ID, Product\_ID, Discount\_ID, Total\_Amount, Order\_Date, Status) VALUES (5, 257, 2, 48, 312.5551806346316, '2024-02-13', 'Delivered');

The `DROP TABLE IF EXISTS Orders;` ensures any existing `Orders` table is removed to avoid conflicts. The `CREATE TABLE Orders` statement defines columns: `Order\_ID` (primary key), `Customer\_ID` (links to `Customers`), `Product\_ID` (links to `Products`), `Discount\_ID` (links to `Discounts`), `Total\_Amount` (a decimal for order totals), `Order\_Date` (date of the order), and `Status` (order status). The `FOREIGN KEY` constraints maintain referential integrity, with `Discount\_ID` allowing `NULL` if a referenced discount is deleted. The `INSERT INTO` statements add sample orders with unique IDs, customer, product, discount references, amounts, order dates, and statuses, facilitating efficient order tracking and management.

**Discounts Table Creation**

CREATE TABLE Discounts (

Discount\_ID INT NOT NULL PRIMARY KEY,

Discount\_Code VARCHAR(50),

Discount\_Amount DECIMAL(10, 2),

Expiry\_Date DATE

);

**Insert Statements for Discounts Table**

INSERT INTO Discounts (Discount\_ID, Discount\_Code, Discount\_Amount, Expiry\_Date) VALUES (1, 'DISC\_WKX05', 0.10123009228522212, '2025-01-11');

INSERT INTO Discounts (Discount\_ID, Discount\_Code, Discount\_Amount, Expiry\_Date) VALUES (2, 'DISC\_UV2LB', 0.14067379625215165, '2024-12-27');

INSERT INTO Discounts (Discount\_ID, Discount\_Code, Discount\_Amount, Expiry\_Date) VALUES (3, 'DISC\_CCKXN', 0.010041073378920224, '2025-01-26');

INSERT INTO Discounts (Discount\_ID, Discount\_Code, Discount\_Amount, Expiry\_Date) VALUES (4, 'DISC\_W1U1L', 0.26588587357832427, '2025-05-27');

INSERT INTO Discounts (Discount\_ID, Discount\_Code, Discount\_Amount, Expiry\_Date) VALUES (5, 'DISC\_KNHPW', 0.4729873987460257, '2025-03-14');

The `DROP TABLE IF EXISTS Discounts;` statement ensures any existing `Discounts` table is removed to prevent conflicts. The `CREATE TABLE Discounts` statement defines four columns: `Discount\_ID` (primary key for unique identification), `Discount\_Code` (code for the discount), `Discount\_Amount` (stored as a decimal with precision up to two decimal places), and `Expiry\_Date` (date when the discount becomes invalid). The `INSERT INTO` statements add sample records with unique IDs, discount codes, amounts, and expiry dates. This structure allows efficient tracking and management of discount offers, ensuring each discount can be easily identified, applied, and queried before its expiration date.

**Payments Table Creation**

CREATE TABLE Payments (

Payment\_ID INT NOT NULL PRIMARY KEY,

Order\_ID VARCHAR(100),

Payment\_Method VARCHAR(50),

Amount\_Paid DECIMAL(10, 2),

Payment\_Date DATE,

FOREIGN KEY (Order\_ID) REFERENCES Orders(Order\_ID)

);

**Insert Statements for Payments Table**

INSERT INTO Payments (Payment\_ID, Order\_ID, Payment\_Method, Amount\_Paid, Payment\_Date) VALUES (1, 70, 'PayPal', 167.45406469997053, '2023-01-14');

INSERT INTO Payments (Payment\_ID, Order\_ID, Payment\_Method, Amount\_Paid, Payment\_Date) VALUES (2, 121, 'Credit Card', 257.60547675154186, '2024-02-17');

INSERT INTO Payments (Payment\_ID, Order\_ID, Payment\_Method, Amount\_Paid, Payment\_Date) VALUES (3, 265, 'Debit Card', 234.08617537523926, '2024-03-21');

INSERT INTO Payments (Payment\_ID, Order\_ID, Payment\_Method, Amount\_Paid, Payment\_Date) VALUES (4, 193, 'Credit Card', 124.44704809075662, '2024-06-20');

INSERT INTO Payments (Payment\_ID, Order\_ID, Payment\_Method, Amount\_Paid, Payment\_Date) VALUES (5, 159, 'PayPal', 326.1715710353987, '2022-03-29');

The `DROP TABLE IF EXISTS Payments;` removes any existing `Payments` table to avoid conflicts. The `CREATE TABLE Payments` statement defines columns: `Payment\_ID` (primary key for unique identification), `Order\_ID` (links to the `Orders` table), `Payment\_Method` (type of payment), `Amount\_Paid` (decimal for payment amount), and `Payment\_Date` (date of payment). The `FOREIGN KEY` ensures referential integrity by linking `Order\_ID` to `Orders`. The `INSERT INTO` statements add sample payments with unique IDs, corresponding order references, payment methods (e.g., PayPal, Credit Card), amounts, and dates. This table facilitates efficient tracking and management of payments related to specific orders.

**Shipments Table Creation**

CREATE TABLE Shipments (

Shipment\_ID INT NOT NULL PRIMARY KEY,

Order\_ID VARCHAR(100),

Carrier VARCHAR(255),

Status VARCHAR(50),

Tracking\_Number VARCHAR(50),

Shipment\_Date DATE,

FOREIGN KEY (Order\_ID) REFERENCES Orders(Order\_ID)

);

**Insert Statements for Shipments Table**

INSERT INTO Shipments (Shipment\_ID, Order\_ID, Carrier, Status, Tracking\_Number, Shipment\_Date) VALUES (1, 219, 'UPS', 'Canceled', '5785KYTMFXRQDAV', '2022-07-27');

INSERT INTO Shipments (Shipment\_ID, Order\_ID, Carrier, Status, Tracking\_Number, Shipment\_Date) VALUES (2, 74, 'DHL', 'Pending', '7NXG56X0CW8LDUF', '2023-08-02');

INSERT INTO Shipments (Shipment\_ID, Order\_ID, Carrier, Status, Tracking\_Number, Shipment\_Date) VALUES (3, 16, 'FedEx', 'Delivered', '255PECFT268VDQP', '2023-10-07');

INSERT INTO Shipments (Shipment\_ID, Order\_ID, Carrier, Status, Tracking\_Number, Shipment\_Date) VALUES (4, 223, 'DHL', 'Pending', 'UBAL2N8FLGMREH2', '2023-09-06');

INSERT INTO Shipments (Shipment\_ID, Order\_ID, Carrier, Status, Tracking\_Number, Shipment\_Date) VALUES (5, 152, 'USPS', 'In Transit', 'CLCO8MVX2ERTF97', '2023-12-07');

The `DROP TABLE IF EXISTS Shipments;` statement deletes any existing `Shipments` table to avoid conflicts. The `CREATE TABLE Shipments` statement defines columns: `Shipment\_ID` (primary key for unique identification), `Order\_ID` (links to `Orders`), `Carrier` (shipping company), `Status` (shipment status), `Tracking\_Number` (unique shipment identifier), and `Shipment\_Date` (date of shipment). The `FOREIGN KEY` constraint ensures referential integrity by linking `Order\_ID` to the `Orders` table. The `INSERT INTO` statements add sample shipments with unique IDs, order references, carriers (e.g., UPS, DHL), statuses, tracking numbers, and shipment dates. This table supports efficient shipment tracking and order fulfillment management.

**SQL Queries for E-commerce Order Tracking and Customer Analytics**

**Order Tracking Queries**

1. **Find Orders That Are Pending or Cancelled**

**Purpose:**  
Retrieves orders that are pending or cancelled to help track unresolved transactions.

**SQL Query:**

SELECT

o.Order\_ID,

c.Name AS Customer\_Name,

o.Status,

o.Total\_Amount,

o.Order\_Date

FROM Orders o

JOIN Customers c ON o.Customer\_ID = c.Customer\_ID

WHERE o.Status IN ('Pending', 'Cancelled');

**Output:**

A screenshot of a computer

Description automatically generated

**Explanation:**

This query helps businesses identify orders that are not yet resolved. Managers can follow up on pending orders to ensure timely processing and investigate cancelled orders to understand why they were not fulfilled.

1. **Identify Orders with Payments and Shipments Completed**

**Purpose:**  
Identifies orders that have completed payments and shipments to ensure full order processing.

**SQL Query:**

SELECT

o.Order\_ID,

c.Name AS Customer\_Name,

pay.Payment\_Method,

s.Carrier,

s.Status AS Shipment\_Status

FROM Orders o

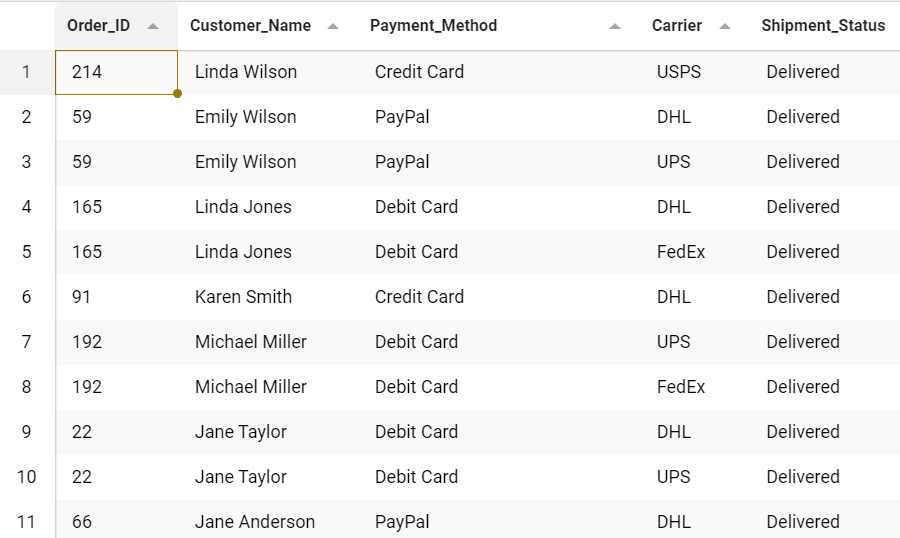
JOIN Payments pay ON o.Order\_ID = pay.Order\_ID

JOIN Shipments s ON o.Order\_ID = s.Order\_ID

JOIN Customers c ON o.Customer\_ID = c.Customer\_ID

WHERE o.Status = ‘Delivered’;

**Output:**



**Explanation:**

This query ensures that orders marked as "Delivered" have corresponding payment and shipment records. It helps verify that the full order fulfillment process has been completed successfully.

**Inventory Monitoring Queries**

1. **Find the Total Number of Products Supplied by Each Supplier**

**Purpose:**  
Counts the number of products supplied by each supplier for inventory management.

**SQL Query:**

SELECT

s.Supplier\_ID,

s.Name AS Supplier\_Name,

COUNT(p.Product\_ID) AS Total\_Products

FROM Suppliers s

JOIN Products p ON s.Supplier\_ID = p.Supplier\_ID

GROUP BY s.Supplier\_ID, s.Name;

**Output:**

A screenshot of a computer

Description automatically generated

**Explanation:**

This query helps businesses understand which suppliers are providing the most products. This insight can be used to manage supplier relationships and ensure a balanced supply chain.

1. **Identify the Most Popular Product Based on Orders**

**Purpose:**  
Determines the most frequently ordered product.

**SQL Query:**

SELECT

p.Product\_ID,

p.Name AS Product\_Name,

COUNT(o.Order\_ID) AS Total\_Orders

FROM Products p

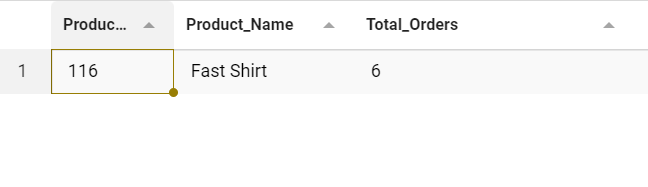
JOIN Orders o ON p.Product\_ID = o.Product\_ID

GROUP BY p.Product\_ID, p.Name

ORDER BY Total\_Orders DESC

LIMIT 1;

**Output:**

****

**Explanation:**

Identifying the most popular product helps businesses ensure they have sufficient stock to meet demand and focus their marketing efforts on high-demand items.

1. **Products Supplied by Each Supplier with Total Stock**

**Purpose:**  
Lists products supplied by each supplier along with their stock quantities.

**SQL Query:**

SELECT

s.Supplier\_ID,

s.Name AS Supplier\_Name,

p.Name AS Product\_Name,

p.Stock\_Quantity

FROM Suppliers s

JOIN Products p ON s.Supplier\_ID = p.Supplier\_ID

ORDER BY s.Supplier\_ID;

**Output:**

**A screenshot of a computer

Description automatically generated**

**Explanation:**

This query provides a detailed view of the inventory supplied by each supplier. It helps businesses manage stock levels and identify potential supply chain issues.

**Discount and Review Analysis Queries**

1. **Products with the Highest Average Rating**

**Purpose:**  
Shows the top 5 products based on average customer ratings.

**SQL Query:**

SELECT

p.Product\_ID,

p.Name AS Product\_Name,

AVG(r.Rating) AS Average\_Rating,

COUNT(r.Review\_ID) AS Total\_Reviews

FROM Products p

JOIN Reviews r ON p.Product\_ID = r.Product\_ID

GROUP BY p.Product\_ID, p.Name

ORDER BY Average\_Rating DESC

LIMIT 5;

**Output:**

**A screenshot of a graph

Description automatically generated**

**Explanation:**

This query helps businesses identify their highest-rated products based on customer reviews. It can be used to highlight these products in marketing campaigns or to understand what features customers appreciate the most.

1. **Top 5 Customers by Total Spend**

**Purpose:**  
Identifies the top 5 customers by total spending.

**SQL Query:**

SELECT

c.Customer\_ID,

c.Name AS Customer\_Name,

SUM(o.Total\_Amount) AS Total\_Spend

FROM Customers c

JOIN Orders o ON c.Customer\_ID = o.Customer\_ID

GROUP BY c.Customer\_ID, c.Name

ORDER BY Total\_Spend DESC

LIMIT 5;

**Output:**

**A screenshot of a computer

Description automatically generated**

**Explanation:**  
Identifies the highest-spending customers, helping to prioritize customer engagement and loyalty programs.

1. **Orders with the Highest Discounts and Corresponding Payment Methods**

**Purpose:**  
Displays orders with the highest discounts, along with the payment methods used.

**SQL Query:**

SELECT

o.Order\_ID,

c.Name AS Customer\_Name,

p.Name AS Product\_Name,

d.Discount\_Code,

d.Discount\_Amount,

o.Total\_Amount,

pay.Payment\_Method,

pay.Amount\_Paid

FROM Orders o

JOIN Customers c ON o.Customer\_ID = c.Customer\_ID

JOIN Products p ON o.Product\_ID = p.Product\_ID

JOIN Discounts d ON o.Discount\_ID = d.Discount\_ID

JOIN Payments pay ON o.Order\_ID = pay.Order\_ID

ORDER BY d.Discount\_Amount DESC

LIMIT 10;

**Output:**

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

**Explanation:**  
This query helps analyze the relationship between discounts and payment preferences.

**Sales and Payment Analysis Queries**

**9. Monthly Sales Analysis with Discount Impact**

**Purpose:**  
Analyzes monthly sales, including the impact of discounts.

**SQL Query:**

SELECT

DATE\_FORMAT(o.Order\_Date, '%Y-%m') AS Month,

COUNT(o.Order\_ID) AS Total\_Orders,

SUM(o.Total\_Amount) AS Total\_Sales,

SUM(IFNULL(d.Discount\_Amount, 0)) AS Total\_Discount

FROM Orders o

LEFT JOIN Discounts d ON o.Discount\_ID = d.Discount\_ID

GROUP BY Month

ORDER BY Month DESC;

**Output:**

A screenshot of a data

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

**Explanation:**  
Shows monthly sales trends and how discounts affect overall sales.

**10.Customers with the Most Expensive Orders**

**Purpose:**  
Identifies the top 10 most expensive orders and the customers who placed them.

**SQL Query:**

SELECT

c.Customer\_ID,

c.Name AS Customer\_Name,

o.Order\_ID,

o.Total\_Amount,

o.Order\_Date

FROM Orders o

JOIN Customers c ON o.Customer\_ID = c.Customer\_ID

ORDER BY o.Total\_Amount DESC

LIMIT 10;

**Output:**

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |

**Explanation:**  
Highlights high-value orders, useful for identifying premium customers.

**11.Average Order Value by Payment Method**

**Purpose:**  
Calculates the average order value for each payment method.

**SQL Query:**

SELECT

pay.Payment\_Method,

COUNT(o.Order\_ID) AS Total\_Orders,

AVG(o.Total\_Amount) AS Average\_Order\_Value

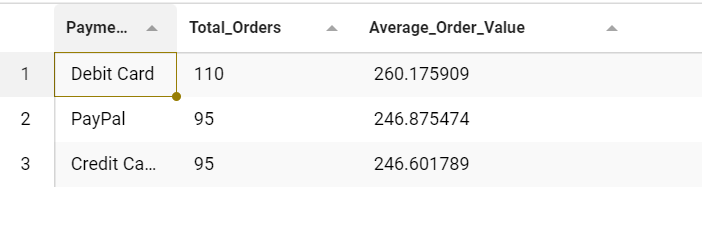
FROM Orders o

JOIN Payments pay ON o.Order\_ID = pay.Order\_ID

GROUP BY pay.Payment\_Method

ORDER BY Average\_Order\_Value DESC;

**Output:**



|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

**Explanation:**  
Helps analyze which payment methods are associated with higher-value orders.

**Comprehensive Analysis Query**

1. **Comprehensive Analysis of Orders with Discounts, Payments, Shipments, and Reviews**

**Purpose:**  
Combines details about orders, customers, products, discounts, payments, shipments, and reviews into a single table.

**SQL Query:**

DROP TABLE IF EXISTS ORDERS\_COMBINED;

CREATE TABLE ORDERS\_COMBINED AS

SELECT

o.Order\_ID,

c.Name AS Customer\_Name,

c.Email AS Customer\_Email,

p.Name AS Product\_Name,

p.Price AS Product\_Price,

IFNULL(d.Discount\_Code, 'No Discount') AS Discount\_Code,

IFNULL(d.Discount\_Amount, 0) AS Discount\_Amount,

o.Total\_Amount,

pay.Payment\_Method,

pay.Amount\_Paid,

pay.Payment\_Date,

s.Carrier,

s.Status AS Shipment\_Status,

s.Tracking\_Number,

s.Shipment\_Date,

r.Rating,

r.Review\_Text,

o.Order\_Date,

o.Status AS Order\_Status

FROM Orders o

JOIN Customers c ON o.Customer\_ID = c.Customer\_ID

JOIN Products p ON o.Product\_ID = p.Product\_ID

LEFT JOIN Discounts d ON o.Discount\_ID = d.Discount\_ID

LEFT JOIN Payments pay ON o.Order\_ID = pay.Order\_ID

LEFT JOIN Shipments s ON o.Order\_ID = s.Order\_ID

LEFT JOIN Reviews r ON o.Customer\_ID = r.Customer\_ID AND o.Product\_ID = r.Product\_ID

ORDER BY o.Order\_Date DESC;

**Output:**

A screenshot of a computer

Description automatically generated

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |

**Explanation:**  
This table provides a complete view of each order, useful for in-depth analysis of order processing, payments, shipments, and customer feedback.

**Visualizations:**

A graph of numbers and a number of people

Description automatically generated with medium confidence

A screenshot of a computer screen

Description automatically generated

A graph with blue and white bars

Description automatically generatedA pie chart with text and a red circle

Description automatically generated

A green and white chart

Description automatically generated

A graph with orange lines

Description automatically generated with medium confidence

A graph of different colored bars

Description automatically generated