DBMS Project Report: Online Retail Store (Amazon Clone)
Group 94

Vedant Gupta **2020261**

Pankaj Ramani **2020234**

Problem Statement

The objective of this project is to design and develop an end-to-end database application. The primary focus is on the design of a backend database for one of the applications that require extensive use of data entities selection, modeling of these data entities and their relationships, and the implementation of constraints. Additionally, the project will involve populating the database with fictitious data and implementing functionality for database access and data manipulation.

Project Scope

The objective of this project is to design and develop a backend database for an online retail store, similar to Amazon, that requires extensive use of data entities selection, modeling of these data entities and their relationships, and the implementation of constraints. The database will store various types of data that are essential to the functionality of the online retail store, such as customer information, product information, order information, inventory information, payment information, and customer service requests and queries. The database will be designed to support the functional requirements of the online retail store, including user registration, product browsing, shopping cart, order placement and tracking, payment processing, inventory management, and customer service. Additionally, the project aims to understand the fundamental concepts of DBMS and to experiment with ER-diagrams and their conversion into relational schemes, creating a logical database design, and the valid constraints that arise in them.

Data To Store:

- **Customers**: The database will store customer information, including UserName, E-Mail, Password, Orders, Previous Orders and Reviews/Feedback.
- **Retailers**: The database will store retailer information, including UserName, E-Mail, Password, Completed Orders, Items/Products, and Reviews/Feedback.
- **Products**: The database will store product information, including ProductID, Product Name, Retailer, and Reviews/Feedback.

The following tasks will be undertaken to achieve these objective:

- Analysis of the requirements of the online retail store and identification of the data entities and relationships that are required.
- Design of the data model for the backend database, including the implementation of constraints and relationships between data entities.
- Implementation of the database using a suitable database management system.
- Populating the database with fictitious data for testing and demonstration purposes.
- Development of database access and data manipulation functionality, including the implementation of database queries and updates.
- Testing and debugging of the developed database and database access and data manipulation functionality.
- Documentation of the design, implementation, and testing of the developed database.
- Presentation and demonstration of the developed database to stakeholders.

The stakeholders and the end-users of the online retail store will be the ultimate beneficiaries of the database design and implementation. It's worth noting that the above list is not exhaustive and may be refined or expanded as the project progresses, and it might be tailored to the specific needs of the project.

Stakeholders

Businesses, E-markets, Customers and Retailers.

Proposed Tech Stack

The following tech stack is necessary for the successful completion of this project:

- **MySQL**: It will be used to implement the backend database for the online retail store. It will be used to store and manage the data entities and relationships that are required for the online retail store, such as customer information, product information, and order

information. MySQL will also be used to implement the constraints and relationships that are required for the data model of the online retail store, such as unique keys, foreign keys, and check constraints.

- **HTML, CSS, JavaScript**: These technologies will be used to create the front-end user interface of the online retail store, including the layout and design of the website. HTML will be used to structure the content of the website, CSS will be used to control the visual presentation of the website, and JavaScript will be used to provide interactivity and dynamic functionality to the website.

- **Python**: A programming language that will be used for the server-side logic of the online retail store, including the implementation of database access and data manipulation functionality. Python will be used to interact with the MySQL database, for example, to execute SQL queries, and to implement the server-side logic for the online retail store, such as authentication, authorization, and business logic.

- **Flask/Django**: Flask or Django is a web framework that will be used to build the server-side of the online retail store. It will be used to handle HTTP requests and responses, to interact with the database, and to handle other server-side logic.

- **ReactJS**: A JavaScript library for building user interfaces that will be used for building the front-end of the online retail store. React will be used to create reusable UI components, manage the state of the application, and handle client-side logic.

A version control system like **git** will also be used to manage and track changes to the project's codebase. It's worth noting that the above list is not exhaustive and may be refined or expanded as the project progresses.

Technical Requirements

The following technical requirements are necessary for the successful completion of this project:

- User authentication and authorization using secure login credentials such as unique usernames and strong passwords.
- Implementation of different levels of access for users based on their role and responsibilities within the online retail store.
- Restriction of access to sensitive information such as customer and retailer personal information to only those users with a valid need to know.
- Control of data manipulation capabilities, such as adding, updating, and deleting products, to only certain users like retailers.
- Implementation of measures to prevent unauthorized access and protect against data breaches through the use of secure login credentials, such as unique usernames and strong passwords.

- Management of user roles and permissions using the database management system's built-in functionality.
- Capability to handle multiple concurrent user connections.
- Optimization of database performance and scalability using techniques such as indexing and partitioning.
- Testing and debugging of the system to ensure its stability and reliability.
- Enforce constraints on the data such as unique usernames for customers, unique product IDs, etc.
- Implementation of security measures to control access to the data, such as roles and permissions, access level for different users, and data encryption.
- Handling of data constraints such as an order can only be placed by a registered customer, a product can only be added by a registered retailer, etc.

Functional Requirements

The following functional requirements are necessary for the successful completion of this project:

- **User registration and login**: This feature will allow users to create an account on the online retail store by providing their personal information and a unique username and password. Once registered, users will be able to log in to their account using their username and password, which will allow them to access their account information, view order history, and place orders.

- **Product browsing and search:** This feature will allow users to browse and search for products on the online retail store. Users will be able to search for products by keywords, categories, and other relevant criteria, such as price range or brand. Additionally, the website should provide filtering options to narrow down the search results.

- **Product details:** This feature will allow users to view detailed information about a product, including its name, description, price, and images. Users will be able to view product details by clicking on a product from the search results or browsing the product categories.

- **Shopping cart:** This feature will allow users to add products to their shopping cart and view the contents of their shopping cart. Users will be able to update the quantity of the products in their cart or remove them if they wish.

- **Order placement and tracking:** This feature will allow users to place orders for products in their shopping cart and track the status of their orders. Once an order is placed, users will receive an order confirmation email and will be able to track the status of their order by logging into their account.

- **Payment processing:** This feature will allow users to make payments for their orders using various payment methods such as credit card and PayPal (a dummy one). Users will be able to select their preferred payment method during the checkout process and will receive a payment confirmation email once the payment is complete.

- **Inventory management:** This feature will allow authorized personnel to manage the inventory of the online retail store, including adding new products, updating product information, and monitoring stock levels.
- **Customer service:** This feature will allow users to contact the online retail store for assistance with queries and issues. The online retail store should provide a contact form and FAQs to help users with their concerns. Additionally, the customer service feature should also include options to track the status of their requests and get updates on their queries.
- **Administration:** This feature will allow authorized personnel to manage the website, including managing products, orders, and customers. The administration panel will provide options to add, edit or delete products, view and manage orders, and manage customer accounts. It should also provide options to generate reports on sales, inventory, and customer behavior. This feature will also be used to manage the website settings and configurations.

Looking in terms of stakeholders the following conditions should be present:

1. For Customers:
    - The ability to create and manage a customer account.
    - The ability to browse different categories of products.
    - The ability to add products to a cart and proceed to checkout.
    - The ability to place an order and make payments through various payment methods.
    - The ability to track the status of an order and view order details.
    - The ability to cancel an order.
    - The ability to write reviews and raise grievances about their orders.
2. For Retailers:
    - The ability to create and manage an admin account.
    - The ability to list their products with details such as price, estimated delivery time, description, and features.
    - The ability to add, delete, and update products.
    - The ability to view customer reviews and address grievances.
    - The ability to view their total earnings and manage their wallet.

It's worth noting that the above list is not exhaustive and may be refined or expanded as the project progresses, and it might be tailored to the specific needs of the project.

Entities Defined with Underlined Primary Key

The following entities have been defined for the database with the primary key underlined:

- **CUSTOMERS:** (CUSTOMER_ID, int, NOT NULL), (CUSTOMER_NAME, char, NOT NULL), (CUSTOMER_PHONE_NUMBER, int, NOT NULL)
- **CUSTOMER_LOCATION:** (CUSTOMER_ID, int, NOT NULL), (STREET, char), (FLAT_NO, int), (CITY/STATE, char)
- **DELIVER_TO:** (ORDER_ID, char, NOT NULL), (CUSTOMER_ID, char, NOT NULL), (SELLER_ID, char, NOT NULL)
- **SELLER_LOCATION:** (SELLER_ID, int, NOT NULL), (STREET, char), (FACTORY_NO, int), (STREET, char)
- **SELLER:** (SELLER_ID, int, NOT NULL), (SELLER_PHONE_NUMBER, int, NOT NULL), (SELLER_NAME, char, NOT NULL)
- **SELLER_INVENTORY:** (SELLER_ID, int, NOT NULL), (ITEM_ID, int, NOT NULL), (QUANTITY, int, NOT NULL)
- **BILL:** (ORDER_ID, int, NOT NULL), (SELLER_ID, int, NOT NULL), (CUSTOMER_ID, int, NOT NULL), (PRICE, int, NOT NULL), (QUANTITY, int, NOT NULL)
- **CUSTOMER_PAYMENT_MODE:** (CUSTOMER_ID, int, NOT NULL), (MODE, char, NOT NULL), (DETAILS, char)
- **IN_STOCK:** (CUSTOMER_ID, int, NOT NULL), (ITEM_ID, int, NOT NULL), (QUANTITY, int, NOT NULL)
- **NOT_IN_STOCK:** (ITEM_ID, int, NOT NULL), (EXPECTED_IN-STOCK_DATE, char)
- **ITEMS:** (ITEM_ID, char, NOT NULL), (CATEGORY, char), (PRICE, int, NOT NULL)

Established Relationships

The following entities have been defined for the database with the primary key underlined:

- **CUSTOMER - ORDER - CHECK STOCK –** This is the case of a ONE-TO-MANY relationship. Because one person can do many orders of many things.
- **CUSTOMER - LIVES IN –** This is the case of a ONE-ONE relationship because a person only lives at one location.
- **CUSTOMER - Payment Info - CUSTOMER PAYMENT MODE –** This is the case of ONE-TO-MANY as one person can have many modes of payment like COD, wallet, and card.
- **IN STOCK - ITEM PROVIDER - SELLER INVENTORY –** This is the case ONE-TO-MANY relationship. Because one item can be present in many seller's inventories.

Ternary Relationship

In the context of our project, two ternary relationships have been established:

- **ITEM PROVIDER (Bill, ITEM PROVIDER, SELLER INVENTORY)**, which was established to ensure a seamless process from the moment a customer searches for a product to the point where the product is delivered. This relationship involves three entities: the customer's Bill, the ITEM PROVIDER responsible for delivering the product, and the SELLER INVENTORY where the product is available. The reason for choosing this relationship as ternary is that the delivery process of the product to the customer can only commence after the Bill is generated and it is verified that the product is present in the SELLER INVENTORY.
- **PAYMENT AND DELIVERY (Deliver to, PAYMENT AND DELIVERY, Bill)**, which was established to ensure that the payment process and delivery process are coordinated and streamlined. This relationship involves three entities: the customer's payment information, the DELIVER TO entity responsible for delivering the product, and the Bill generated based on the customer's payment information. The reason for choosing this relationship as ternary is that both the ITEM PROVIDER and DELIVER TO entities need the customer's payment information in order to process the delivery of the product. The ITEM PROVIDER needs the payment information to inform the seller that the Bill has been generated and the delivery process can commence, while the DELIVER TO entity needs the payment information to prepare for the delivery, including handling COD payments if applicable.

Weak Entity

The following entities have been identified as weak entities in the system:

- **LIVES IN:** This entity represents a customer's location and is considered a weak entity as it cannot exist without the presence of a CUSTOMER. In absence of a CUSTOMER, the primary key attribute (CUSTOMER ID) would not be present.
- **SELLS IN:** This entity represents the location of a seller and is considered a weak entity as it cannot exist without the presence of a SELLER. Without a SELLER, the primary key attribute (SELLER ID) would not be present.
- **Payment Info:** This entity represents a customer's payment information and is considered a weak entity as it cannot exist without the presence of a CUSTOMER. Without a CUSTOMER, the primary key attribute (CUSTOMER ID) would not be present.

Relationship constraints

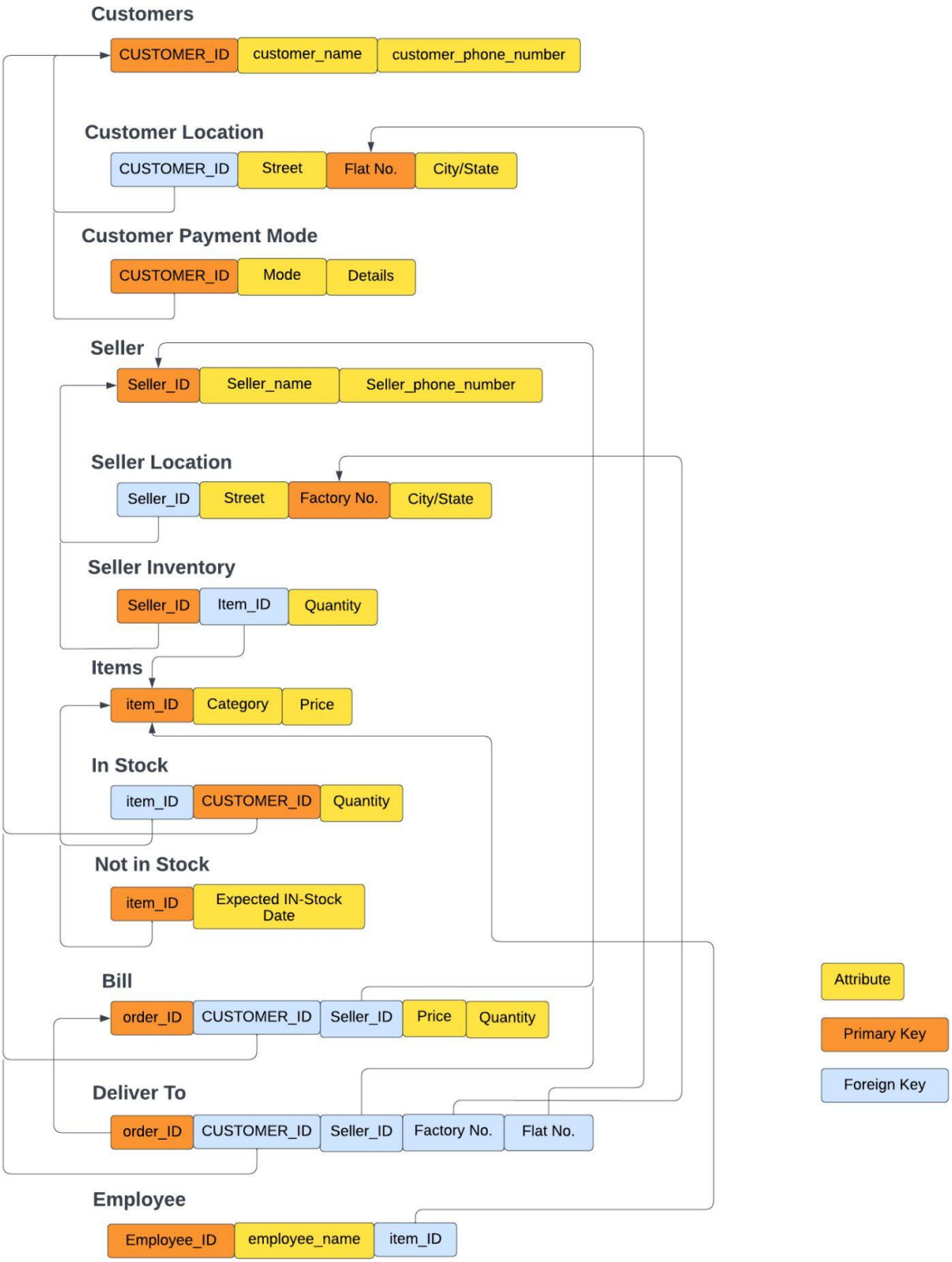
1. Each customer can have only one customer location.

2. Each seller can have only one seller location.

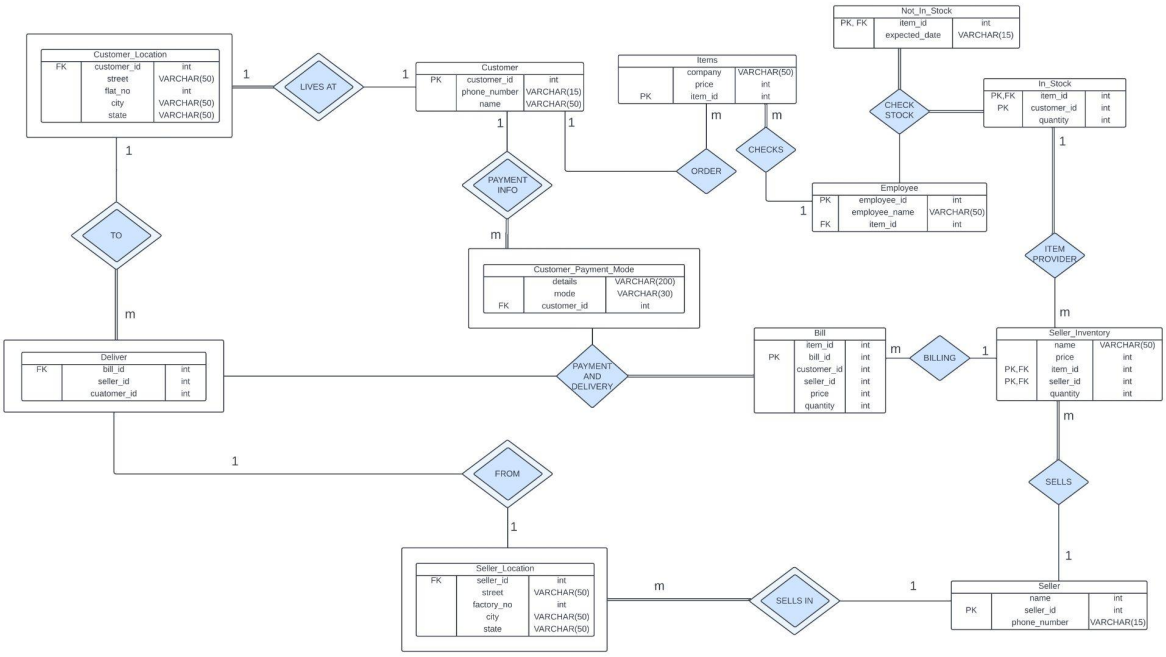
3. Items in NotStock are not present in the InStock table.

4. Delivery does not occur between every customer and seller location.

Relational Schema of the Project



ER diagram of the Project



Database Schema:

To build the database schema based on the given entities, we followed the below steps:

- Define the database name and create tables for each entity.
- Repeat the above step for all entities defined in the question.
- Create foreign keys to maintain relationships between tables.
- Repeated the above step for all relationships between tables. For example:

```
create table Customer
(
    customer_name varchar(50),
    customer_id int not NULL auto_increment,
    phone_number varchar(50) not NULL,
    PRIMARY KEY (customer_id)
);
```

Database Population:

To populate the database tables, we used the Faker library in Python, which provides a way to generate fake data that can be used to populate the database. Here's a high-level overview of the steps you would follow to populate the tables in the database schema you provided:

- Installed the Faker library: We installed the Faker library in the Python environment to get started.
- Imported the library in our script: We imported the Faker library in the Python script where we generated the fake data.
- Initialize an instance of the Faker class: We need to initialize an instance of the Faker class, which was used to generate fake data.
- Create data for each table: For each table in the database schema, we wrote code that generates fake data that can be used to populate the table. Here's an example of what this code might look like for the CUSTOMERS table:

```python
# For Customer
def generate_random_data(num_rows):
    cus = []
    for i in range(num_rows):
      # Generate random data for each column
      customer_name = fake.name()
      number = fake.random_number(digits=10, fix_len=True)
      # Store the generated data as a tuple
      row = (customer_name, number)
      cus.append(row)
    return cus

num_rows = 200
cus = generate_random_data(num_rows)

with open("Customer.txt", "w") as file:
    for row in cus:
        file.write(f"INSERT INTO Customer (customer_name, phone_number) VALUES ('{row[0]}', '{row[1]}');\n")
```

- Insert the data into the database: Once we generated the fake data, we inserted it. To do this, we use the execute method of a database cursor object and pass in an SQL INSERT statement with the values to be inserted. Here's an example of what this code might look like for the CUSTOMERS table:

```sql
INSERT INTO Customer (customer_name, phone_number) VALUES ('Anthony Williams', '3058139312');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Katie Stewart', '4543815053');
INSERT INTO Customer (customer_name, phone_number) VALUES ('William Warren', '4618407005');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Jonathan Rowe', '6815193327');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Jonathan Bailey', '7858639817');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Matthew Harrell', '7230755892');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Ian Conner', '7447587373');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Ronald King', '1478312903');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Cory Khan', '6810660220');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Anna Lara', '9841076902');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Jennifer Walker', '9831741779');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Elizabeth Kelly', '8482756932');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Roberta Hurley', '5402546865');
INSERT INTO Customer (customer_name, phone_number) VALUES ('Daniel Bentley', '6877808067');
INSERT INTO Customer (customer_name, phone_number) VALUES ('William Chavez', '9899048602');
```

- We generated this script for each entry into the database table and generated all this data in a text file. From the text file, we copied this data and pasted it into the SQL workspace.
- Repeated the above steps for each table in the database schema.

By following these steps, we were able to populate the database tables in the schema and provide fake data using the Faker library.

SQL Queries:

use Amazon_clone;

Problem Statement 1: Update the quantity of a particular item in the inventory of a particular seller.

Relational Algrebraic Equation:

Σ seller_id=5∧item_id=10 (Seller_Inventory)←γ seller_id,item_id (Seller_Inventory⋈ seller_id=5∧item_id=10 ρ seller_id,item_id,quantity→new_quantity (3))

Query:

```
UPDATE Seller_Inventory
SET quantity = 3
WHERE seller_id = 5 AND item_id = 10;
```

Problem Statement 2: Add and drop the column of DOB in the customer table.

Relational Algrebraic Equation:

Customer←ρ customer_id,customer_name (Customer)
Customer←π customer_id,customer_name (Customer)

Query:

```
ALTER TABLE Customer
DROP COLUMN DOB;
ALTER TABLE Customer
ADD DOB date;
```

Problem Statement 3: Display the total revenue generated by the seller 5.

Relational Algrebraic Equation:

Σ seller_id=5 (γ seller_id (Bill⋈Seller)⋈Items)

Query:

```
SELECT SUM(b.price * b.quantity)
FROM Bill b
WHERE b.seller_id = 5;
```

Problem Statement 4: Delete a particular customer and all their associated data from the database.

Relational Algrebraic Equation:

Customer − γ customer_id=10 (Customer)
Deliver − γ customer_id=10 (Deliver)
Bill − γ customer_id=10 (Bill)
Payment − γ customer_id=10 (Payment)

Query:

```
DELETE FROM Deliver
WHERE customer_id = 10;
DELETE FROM Bill
WHERE customer_id = 10;
DELETE FROM Payment
WHERE customer_id = 10;
DELETE FROM Customer
WHERE customer_id = 10;
```

Problem Statement 5: Display the top 5 best-selling items by quantity.

Relational Algrebraic Equation:

π item_name,total_quantity (γ item_id,total_quantity (σ item_id=b.item_id (Items⋈Bill(b)))⋈total_quantity≤ q5.total_quantity (q5 ←γ total_quantity (σ rank≤5 (γ item_id,SUM(quantity)→total_quantity (Bill)⋈item_id Items))))

Query:

```
SELECT i.item_name, SUM(b.quantity) as total_quantity
FROM Items i
JOIN Bill b ON i.item_id = b.item_id
GROUP BY i.item_name
ORDER BY total_quantity DESC
LIMIT 5;
```

Problem Statement 6: Display the average price of items in each category.

Relational Algrebraic Equation:

γ category,average_price (σ i.item_id=si.item_id ((Itemsi⋈Seller_Inventorysi)÷π category (Items))⋈category γ category,AVG(price)→average_price (Seller_Inventory))

Query:

```
SELECT i.category, AVG(si.price) as average_price
FROM Items i
JOIN Seller_Inventory si ON i.item_id = si.item_id
GROUP BY i.category;
```

Problem Statement 7: Display the number of bills generated by seller 10.

Relational Algrebraic Equation:

γ COUNT(*) (σ seller_id=10 (Bill))

Query:

```
SELECT COUNT(*)
FROM Bill b
WHERE b.seller_id = 10;
```

Problem Statement 8: Show the names of all sellers who have items in stock.

Relational Algrebraic Equation:

π seller_name (σ seller_id∈(π seller_id (σ item_id∈(π item_id (σ quantity>0 (InStock))) (Seller_Inventory))) (Seller))

Query:

```
SELECT seller_name
FROM Seller
WHERE seller_id IN (SELECT seller_id
                    FROM Seller_Inventory
                    WHERE item_id IN (SELECT item_id
                                      FROM InStock));
```

Problem Statement 9: Find the average price of items sold by the seller Michelle Ford.

Relational Algrebraic Equation:

γ AVG(price) (σ seller_id=(SELECTseller_idFROMSellerWHEREseller_name= ′MichelleFord ′) (Seller_Inventory))

Query:

```
SELECT AVG(price)
FROM Seller_Inventory
WHERE seller_id = (SELECT seller_id
                   FROM Seller
                   WHERE seller_name = 'Michelle Ford');
```

Problem Statement 10: Find all customers who have not made a purchase from a specific seller Fire.

Relational Algrebraic Equation:

π seller_id, seller_name, rating (Seller) - π seller_id (σ item_name = 'Fire' (Items) $\bowtie$ Seller_Inventory)

Query:

```
SELECT *
FROM Seller s
WHERE NOT EXISTS (SELECT *
                  FROM Seller_Inventory si
                  WHERE si.seller_id = s.seller_id
                  AND si.item_id = (SELECT item_id
                                    FROM Items
                                    WHERE item_name = 'Fire'));
```

Problem Statement 11: Retrieve a list of customers along with their payment mode and the corresponding payment amount, for all bills above 5000.

Relational Algrebraic Equation:

π customer_name, payment_mode, payment_amount (γ customer_id, payment_mode: payment_amount $\leftarrow$ SUM(price * quantity) (Bill $\bowtie$ Payment $\bowtie$ σ price * quantity > 5000 (Bill)))

Query:

```
SELECT c.customer_name, p.payment_mode, SUM(b.price * b.quantity) AS payment_amount
FROM Customer c
INNER JOIN Payment p ON c.customer_id = p.customer_id
INNER JOIN Bill b ON c.customer_id = b.customer_id
WHERE b.price * b.quantity > 5000
GROUP BY c.customer_id, p.payment_mode;
```

Problem Statement 12: Create a view that displays the item names, seller names and prices for all items with a quantity greater than 5 in the InStock table.

Relational Algrebraic Equation:

InStock_Details ← π item_name, seller_name, price (Items ⋈ Seller_Inventory ⋈ Seller ⋈ σ quantity > 5 (InStock))

Query:

```
CREATE VIEW InStock_Details AS
SELECT i.item_name, s.seller_name, si.price
FROM Items i
INNER JOIN Seller_Inventory si ON i.item_id = si.item_id
INNER JOIN Seller s ON si.seller_id = s.seller_id
INNER JOIN InStock ist ON si.item_id = ist.item_id
WHERE ist.quantity > 5;
```

Problem Statement 13: Retrieve a list of distinct item names, their categories, and the count of sellers selling those items.

Relational Algrebraic Equation:

γ item_name, category: seller_count ← COUNT(DISTINCT seller_id) (Items ⋈ Seller_Inventory)

Query:

```
SELECT i.item_name, i.category, COUNT(DISTINCT si.seller_id) AS seller_count
FROM Items i
INNER JOIN Seller_Inventory si ON i.item_id = si.item_id
GROUP BY i.item_name, i.category;
```

Problem Statement 14: Select the seller_name from the Seller table where there is no matching row in the InStock table for a particular item_id.

Relational Algrebraic Equation:

π seller_name (Seller - (Seller_Inventory ⋈ σ ¬ ∃ i.item_id(si.item_id = i.item_id) (InStock)))

Query:

```
SELECT seller_name FROM Seller s
WHERE EXISTS (SELECT * FROM Seller_Inventory si
                    WHERE si.seller_id = s.seller_id
                    AND NOT EXISTS (SELECT *
                                        FROM InStock i
                                        WHERE i.item_id = si.item_id));
```

Problem Statement 15: Get a list of all seller locations and customer locations.

Relational Algrebraic Equation:

Seller_Location' ← π seller_id: id, factory_no: location_id, street, city, state, 'Seller': type (Seller_Location)
Customer_Location' ← π customer_id: id, flat_no: location_id, street, city, state, 'Customer': type (Customer_Location)
Location ← Seller_Location' ∪ Customer_Location'

Query:

```
SELECT seller_id AS id, factory_no AS location_id, street, city, state, 'Seller' AS type
FROM Seller_Location
UNION
SELECT customer_id AS id, flat_no AS location_id, street, city, state, 'Customer' AS type
FROM Customer_Location;
```

Triggers:

Trigger 1: To prevent insertion of new items with negative quantity in the inventory.

```
DROP TRIGGER IF EXISTS check_inventory_quantity;
CREATE TRIGGER check_inventory_quantity
BEFORE INSERT ON InStock
FOR EACH ROW
BEGIN
   IF NEW.quantity < 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Inventory quantity cannot be negative';
   END IF;
End//
```

Trigger 2: To check if an item is out of stock and add it to the not_in_stock table with expected date

```
DROP TRIGGER IF EXISTS trg_item_out_of_stock;
CREATE TRIGGER trg_item_out_of_stock
AFTER UPDATE ON InStock
FOR EACH ROW
BEGIN
        IF NEW.quantity = 0 THEN
              INSERT INTO not_in_stock (item_id, expected_date) VALUES (NEW.item_id,
              DATE_ADD(NOW(), INTERVAL 7 DAY));
        END IF;
END//
```

OLAP:

Pivoting: Find the total sales of each seller for each category.

```
SELECT s.seller_name, i.category, SUM(b.quantity*b.price) AS sales
FROM Seller s
JOIN Seller_Inventory si ON s.seller_id = si.seller_id
JOIN Items i ON si.item_id = i.item_id
JOIN Bill b ON si.seller_id = b.seller_id AND si.item_id = b.item_id
GROUP BY s.seller_id, i.category;
```

Slicing: Find the total sales of items in the "Books" category.

```
SELECT i.item_name, SUM(b.quantity*b.price) AS sales
FROM Items i
JOIN Bill b ON i.item_id = b.item_id
WHERE i.category = 'Books'
GROUP BY i.item_id;
```

Roll-up: Find the total sales of each category and the overall total.

```
SELECT i.category, SUM(b.quantity*b.price) AS sales
FROM Items i
JOIN Bill b ON i.item_id = b.item_id
GROUP BY i.category WITH ROLLUP;
```

Drill-down: Count the total sales for each category and subcategory of items

```
SELECT i.category, i.item_name, SUM(b.quantity * b.price) AS total_sales
FROM Bill b
JOIN Items i ON b.item_id = i.item_id
GROUP BY i.category, i.item_name WITH ROLLUP;
```

Embedded SQL queries:

1) SELECT * FROM Customer;
2) SELECT * FROM Seller;
3) SELECT * FROM InStock;
4) SELECT * FROM not_in_stock;
5) SELECT * FROM items;
6) INSERT INTO Customer (customer_name, phone_number) VALUES ('John Doe', '+1234567890');

7) INSERT INTO InStock (item_id, quantity) VALUES (1, -10);

8) INSERT INTO InStock (item_id, quantity) VALUES (500, 1);

9) UPDATE InStock SET quantity = 0 WHERE item_id = 1;

10) SELECT s.seller_name, i.category, SUM(b.quantity*b.price) AS sales
    FROM Seller s
    JOIN Seller_Inventory si ON s.seller_id = si.seller_id
    JOIN Items i ON si.item_id = i.item_id
    JOIN Bill b ON si.seller_id = b.seller_id AND si.item_id = b.item_id
    GROUP BY s.seller_id, i.category;

11) SELECT i.item_name, SUM(b.quantity*b.price) AS sales
    FROM Items i
    JOIN Bill b ON i.item_id = b.item_id
    WHERE i.category = 'Books'
    GROUP BY i.item_id;

12) SELECT i.category, SUM(b.quantity*b.price) AS sales
    FROM Items i
    JOIN Bill b ON i.item_id = b.item_id
    GROUP BY i.category
    WITH ROLLUP;

13) SELECT i.category, i.item_name, SUM(b.quantity * b.price) AS total_sales
    FROM Bill b
    JOIN Items i ON b.item_id = i.item_id
    GROUP BY i.category, i.item_name
    WITH ROLLUP;

Conflicting queries:

The goal of conflict resolution is to ensure that all transactions are executed correctly and consistently while maintaining the integrity of the data. There are different techniques and strategies that can be used for conflict resolution, such as:

Locking: This involves acquiring locks on the data or resources being accessed by a transaction so that other transactions cannot modify them while the transaction is still executing. Locking can be either optimistic or pessimistic, depending on whether the locks are acquired before or after the transaction starts executing.

Timestamping: This involves assigning a unique timestamp to each transaction when it starts, and using these timestamps to determine the order in which transactions should be executed. This ensures that conflicting transactions are executed in a deterministic order.

Rollback and retry: This involves rolling back a transaction when a conflict is detected, and then retrying the transaction with a different strategy or parameter values. For example, a transaction failing due to a locking conflict could be retried with a longer timeout or a different set of locks.

Compensation: This involves executing compensating transactions to undo the effects of conflicting transactions. For example, if a transaction that updates a record fails due to a conflict, a compensating transaction could be executed to restore the record to its previous state.

The best approach for conflict resolution depends on the specific requirements and constraints of the database system and the application using it. A good database administrator should have a thorough understanding of the different conflict resolution techniques and be able to choose the most appropriate one for a given situation.

1) Conflicting query type: **Unrepeatable Read**

   -- A customer wants to update their phone number
   UPDATE Customer SET phone_number = '123-456-7890' WHERE customer_id = 1;

   -- A delivery person wants to retrieve the customer's phone number for delivery purposes
   SELECT phone_number FROM Customer WHERE customer_id = 1;

   Solution: In this scenario, the first query is a write query that updates the phone_number field for the customer with customer_id = 1. The second query is a read query that retrieves the phone_number field for the same customer. To handle this conflict, we can use database locks to ensure that both queries do not execute simultaneously. Specifically, we can use a table-level write lock on the Customer table for the duration of the first query, which will prevent the second query from executing until the lock is released. Once the first query completes, the lock is released and the second query can execute.

   Transaction T1: UPDATE Customer SET phone_number = '123-456-7890' WHERE customer_id = 1;
   Transaction T2: SELECT phone_number FROM Customer WHERE customer_id = 1;

   Conflict-Serializable Schedule:

T1	T2
READ(Customer WHERE customer_id = 1)	
	READ(Customer.phone_number WHERE customer_id = 1)
WRITE(Customer.phone_number = '123-456-7890' WHERE customer_id = 1)	

COMMIT	
	READ(Customer.phone_number WHERE customer_id = 1)
	COMMIT

Non-Conflict-Serializable Schedule:

T1	T2
READ(Customer WHERE customer_id = 1)	
WRITE(Customer WHERE customer_id = 1)	
COMMIT	
	READ(Customer WHERE customer_id=1)
	COMMIT

```
-- Use a table-level write lock on the Customer table for the duration of the update query
BEGIN;
   LOCK TABLES Customer WRITE;
   UPDATE Customer SET phone_number = '123-456-7890' WHERE customer_id = 1;
   UNLOCK TABLES;
COMMIT;

-- Retrieve the customer's phone number once the lock is released
SELECT phone_number FROM Customer WHERE customer_id = 1;
```

2) Conflicting query type: **Write-Write**

```
-- A seller updates the price of an item in their inventory
UPDATE Seller_Inventory SET price = 10 WHERE seller_id = 1 AND item_id = 1;

-- Another seller tries to update the same item's price at the same time
UPDATE Seller_Inventory SET price = 12 WHERE seller_id = 2 AND item_id = 1;
```

Solution: In this scenario, both queries are write queries that attempt to update the price field for the same item (item_id = 1) in the Seller_Inventory table. To handle this conflict, we can use a database transaction to ensure that both queries do not execute simultaneously. Specifically, we can use a row-level write lock on the affected rows in the Seller_Inventory table for the duration of each query, which will prevent the other query from executing until the lock is released. Once both queries are complete, the changes are committed to the database. If the rows cannot be locked due to another

transaction, one of the transactions will be rolled back.
Transaction T1: UPDATE Seller_Inventory SET price = 10 WHERE seller_id = 1 AND item_id = 1;
Transaction T2: UPDATE Seller_Inventory SET price = 12 WHERE seller_id = 2 AND item_id = 1;

Conflict-Serializable Schedule:

T1	T2
READ(Seller_Inventory WHERE seller_id=1 AND item_id=1)	
	READ(Seller_Inventory WHERE seller_id=1 AND item_id=1)
	WRITE(Seller_Inventory.price=12 WHERE seller_id=2 AND item_id=1)
	COMMIT
WRITE(Seller_Inventory.price=10 WHERE seller_id=2 AND item_id=1)	
COMMIT	

Non-Conflict-Serializable Schedule:

T1	T2
READ(Seller_Inventory WHERE seller_id=1 AND item_id=1)	
	READ(Seller_Inventory WHERE seller_id=1 AND item_id=1)
WRITE(Seller_Inventory.price=10 WHERE seller_id=2 AND item_id=1)	
COMMIT	
	WRITE(Seller_Inventory.price=12 WHERE seller_id=2 AND item_id=1)
	COMMIT

-- Use a transaction to lock the rows in the Seller_Inventory table and update the price

field
BEGIN;
-- Lock the rows affected by the first query
SELECT * FROM Seller_Inventory WHERE seller_id = 1 AND item_id = 1 FOR UPDATE;
-- Update the price for the first query
UPDATE Seller_Inventory SET price = 10 WHERE seller_id = 1 AND item_id = 1;
COMMIT;

BEGIN;
-- Lock the rows affected by the second query
SELECT * FROM Seller_Inventory WHERE seller_id = 2 AND item_id = 1 FOR UPDATE;
-- Update the price for the second query
UPDATE Seller_Inventory SET price = 12 WHERE seller_id = 2 AND item_id = 1;
COMMIT;

This solution locks the rows using the FOR UPDATE clause in the SELECT statement, ensuring that other transactions attempting to modify the same rows will have to wait until the first transaction has been committed or rolled back.

Non-Conflicting queries:

The goal of conflict resolution is to ensure that all transactions are executed correctly and consistently while maintaining the integrity of the data. There are different techniques and strategies that can be used for conflict resolution, such as:

1) A new customer is added to the database along with their address in the Customer and Customer_Location tables respectively.

START TRANSACTION;
INSERT INTO Customer (customer_name, phone_number) VALUES ('John Doe', '1234567890');
SELECT customer_id FROM Customer WHERE customer_name = 'John Doe';
INSERT INTO Customer_Location (customer_id, flat_no, street, city, state) VALUES (LAST_INSERT_ID(), 101, 'Main St', 'New York', 'NY');
COMMIT;

2) A new seller is added to the database along with their address in the Seller and Seller_Location tables respectively.

START TRANSACTION;
INSERT INTO Seller (seller_name, phone_number) VALUES ('ABC Inc.', '0987654321');
SELECT seller_id FROM Seller WHERE seller_name = 'ABC Inc.';
INSERT INTO Seller_Location (seller_id, factory_no, street, city, state) VALUES (LAST_INSERT_ID(), 201, '2nd St', 'Los Angeles', 'CA');

```
COMMIT;
```

3) A new item and its details are added to the database in the Items table.

```
START TRANSACTION;
INSERT INTO Items (item_name, category) VALUES ('T-shirt', 'Clothing');
SELECT item_id FROM Items WHERE item_name = 'T-shirt';
INSERT INTO InStock (item_id, quantity) VALUES (LAST_INSERT_ID(), 50);
COMMIT;
```

4) Delete a seller and their location

```
START TRANSACTION;
DELETE FROM Seller WHERE seller_id = 2;
DELETE FROM Seller_Location WHERE seller_id = 2;
COMMIT;
```

User Guide:

The following are necessary the features of this project:

- User authentication and authorization using secure login credentials such as unique usernames and strong passwords.
- Implementation of different levels of access for users based on their role and responsibilities within the online retail store.
- Restriction of access to sensitive information such as customer and retailer personal information to only those users with a valid need to know.
- Control of data manipulation capabilities, such as adding, updating, and deleting products, to only certain users like retailers.
- Management of user roles and permissions using the database management system's built-in functionality.
- Enforce constraints on the data such as unique usernames for customers, unique product IDs, etc.
- Implementation of security measures to control access to the data, such as roles and permissions, access level for different users, and data encryption.
- Handling of data constraints such as an order can only be placed by a registered customer, a product can only be added by a registered retailer, etc.

Functionalities:

The following functional requirements of this project:

- **User registration and login**: This feature will allow users to create an account on the online retail store by providing their personal information and a unique username and

password. Once registered, users will be able to log in to their account using their username and password, which will allow them to access their account information, view order history, and place orders.

- **Product browsing and search:** This feature will allow users to browse and search for products on the online retail store. Users will be able to search for products by keywords, categories, and other relevant criteria, such as price range or brand. Additionally, the website should provide filtering options to narrow down the search results.

- **Product details:** This feature will allow users to view detailed information about a product, including its name, description, price, and images. Users will be able to view product details by clicking on a product from the search results or browsing the product categories.

- **Shopping cart:** This feature will allow users to add products to their shopping cart and view the contents of their shopping cart. Users will be able to update the quantity of the products in their cart or remove them if they wish.

- **Administration:** This feature will allow authorized personnel to manage the website, including managing products, orders, and customers. The administration panel will provide options to add, edit or delete products, view and manage orders, and manage customer accounts. It should also provide options to generate reports on sales, inventory, and customer behavior. This feature will also be used to manage the website settings and configurations.

Looking in terms of stakeholders the following conditions should be present:

1. For Customers:
    - The ability to create and manage a customer account.
    - The ability to browse different categories of products.
    - The ability to add products to a cart and proceed to checkout.
2. For Retailers:
    - The ability to create and manage an admin account.
    - The ability to list their products with details such as price, estimated delivery time, description, and features.
    - The ability to add, delete, and update products.
    - The ability to view customer reviews and address grievances.
    - The ability to view their total earnings and manage their wallet.