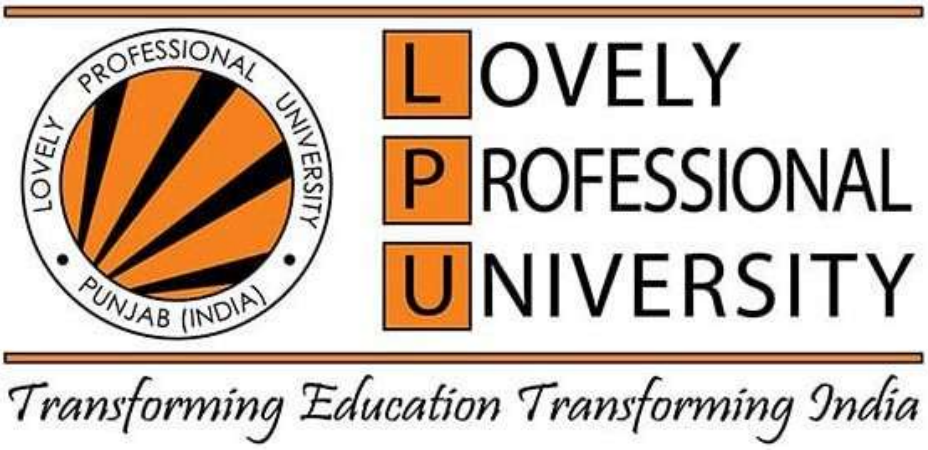


Major Project Report

on

Project Title: - Online Grocery Store

Course: Advanced Database Techniques

Course Code: CAP570

Submitted by

1. Sricharan Boggavarapu (12402606), Group: - 2
2. Ankna Chaudhary (12408388), Group: -2
3. Sanjana Sherawat (12408497), Group: -2

Submitted to

Ms. Ranjit Kaur Walia
UID: 28632
Assistant Professor, SCA, LPU

Lovely Faculty of Technology & Sciences

School of Computer Applications

Lovely Professional University

Punjab

Acknowledgment

We would like to express our sincere gratitude to **Ms. Ranjit Kaur Walia (28632), Assistant Professor, School of Computer Applications, Lovely Professional University**, for her guidance and continuous supervision, as well as for providing essential information and unwavering support throughout the project. Her consistent direction and willingness to share her extensive knowledge enabled us to gain a deep understanding of the project, which greatly assisted us in completing our tasks on time.

We would also like to extend our heartfelt thanks to all the individuals who graciously participated in the interviews and surveys conducted during the project. Their experiences and insights provided a practical understanding of how database systems are used in real-world applications, which greatly enriched the quality of this project.

Finally, we would like to thank our families and friends for their constant encouragement and support during this journey. This project would not have been possible without the contribution of everyone involved.

Thank you all.

Table of Contents

1. Introduction

1. Overview of the Database Management Project

The aim of this project is to design and implement a comprehensive **Database Management System (DBMS)** tailored for an online grocery store. The database is designed using **SQL Server Management Studio (SSMS)** and follows the principles of **3rd Normal Form (3NF)** to ensure efficiency, data integrity, and minimal redundancy. This DBMS will manage several critical operations including customer management, product categorization, order processing, inventory control, and supplier management, which are essential for the smooth operation of an e-commerce platform.

The system is structured around key entities such as Customers, Products, Orders, Payments, and Suppliers. Each entity is represented as a table in the database, with primary and foreign keys ensuring proper relationships and data integrity. The design emphasizes flexibility and scalability, enabling the database to grow alongside the business as more products are added, and the customer base expands. The focus on normalization ensures that data is stored efficiently without unnecessary duplication, which will ultimately improve the system's performance and speed.

Additionally, the DBMS offers real-time inventory tracking, secure payment processing, and user-friendly interfaces for both customers and administrators. Customers can browse through product categories, add items to their cart, and place orders with ease, while the back-end system manages stock levels, supplier information, and order fulfillment. This system not only supports day-to-day operations but also provides valuable insights through structured data, allowing for more informed decision-making related to inventory management and customer preferences.

2. Importance of Database Management Systems in Modern-Day Applications

In the digital era, **Database Management Systems (DBMS)** are fundamental to the operation of virtually all large-scale modern applications. Whether it's an e-commerce platform, a streaming service, or a social media network, managing large volumes of data efficiently is critical for maintaining system performance and providing a smooth user experience. A well-designed DBMS can manage data transactions, support concurrent user interactions, and ensure data accuracy, even as the user base grows exponentially.

Take **Netflix** as a prime example: it handles the viewing preferences, watch history, and real-time streaming demands of millions of users worldwide, delivering personalized content through the efficient management of vast amounts of data. Netflix utilizes sophisticated databases to recommend shows based on user behavior, handle subscriptions, and stream content seamlessly, even during high-demand periods. This would not be possible without a highly optimized and scalable DBMS that supports fast query processing, high availability, and fault tolerance.

Similarly, **Amazon** uses advanced DBMS systems to power its e-commerce platform, managing vast inventories, processing millions of orders daily, and ensuring real-time updates to product availability, pricing, and delivery tracking. The efficiency of Amazon's operations is highly dependent on how well its databases handle dynamic changes and deliver accurate, timely information to both customers and internal teams.

For an online grocery store, the need for an efficient DBMS is just as important. The grocery store’s DBMS needs to handle thousands of product listings, manage customer orders, and ensure that inventory levels are up-to-date at all times. Additionally, the system must be capable of securely processing payments and

storing sensitive customer information, such as addresses and payment details. A well-structured DBMS ensures that these tasks are handled efficiently, minimizing the risk of data loss, errors, or slow performance.

Just like Netflix or Amazon, a reliable DBMS in the context of this project will ensure smooth data flow between the different entities—customers, products, orders, and suppliers—thus providing a seamless shopping experience for users while optimizing operational efficiency for administrators. The system's ability to scale as the business grows makes it a powerful tool for supporting future expansion and ensuring long-term success in a competitive digital marketplace.

2. Project Overview

This project focuses on building a **Database Management System (DBMS)** for an online grocery store, designed to efficiently handle key business operations such as managing customers, processing orders, tracking inventory, and handling payments. The primary goal of the database is to streamline and automate these processes, ensuring that all the information related to customers, products, orders, and suppliers is stored in a structured and secure manner. The database will help the store maintain accurate records, manage real-time inventory, process payments securely, and provide a smooth shopping experience for customers.

Project Concept and Scope

The scope of the project includes the design and creation of a relational database that supports multiple functions:

- **Customer Management**: Store customer details like names, addresses, and contact information.
- **Product Management**: Keep track of product information, including product names, prices, stock availability, and categories.
- **Order Processing**: Record customer orders, monitor the status of each order, and ensure timely delivery.
- **Inventory Control**: Automatically update stock levels as products are purchased and reordered.
- **Payment Processing**: Safeguard transaction details and ensure smooth payment processing for both customers and the store.
- **Supplier Management**: Manage supplier details and product orders to ensure that stock is always replenished when needed.

By creating this database, the online grocery store will be able to efficiently manage its operations, reduce manual errors, and improve overall performance. The system will provide real-time updates, which is essential for a business that relies on inventory levels and fast order fulfillment.

Core Entities, Attributes, and Relationships

The database will revolve around several core entities, each representing a critical aspect of the online grocery store's operations:

1. **Customers**: The Customers table will store personal details such as name, email, address, and phone number. Each customer will have a unique ID, ensuring that orders and interactions can be linked back to the correct individual.
2. **Products**: The Products table will include attributes like product name, description, price, and quantity in stock. Each product will belong to a specific category (e.g., fruits, vegetables, dairy), and categories will be represented in the Categories table.

3. **Orders**: The Orders table will store information about customer orders, including the date, total amount, and status (e.g., pending, shipped, delivered). Each order will be linked to the respective customer and products purchased.
4. **OrderDetails**: This table acts as a bridge between Orders and Products, recording the specifics of what products were ordered, the quantity of each, and the price at the time of purchase.
5. **Cart**: The Cart table will keep track of products added by customers before placing their final order, giving them the flexibility to modify their selection.
6. **Payments**: The Payments table will store transaction details, including the payment method (credit card, UPI, etc.), transaction amount, and date. It will link each payment to the corresponding order.
7. **Suppliers**: The Suppliers table will contain details about suppliers who provide products to the grocery store, including their contact information and the products they supply.
8. **SupplierProducts**: This table will connect suppliers with the specific products they provide, ensuring that the store can easily reorder items when stock runs low.

Relationships

Each entity in the database is connected through relationships:

- **Customers** are linked to their **Orders**.
- **Orders** are connected to **OrderDetails**, which in turn are linked to **Products**.
- **Suppliers** are linked to **SupplierProducts**, which connects them with the specific **Products** they provide.

These relationships ensure that data is interlinked in a logical and efficient manner, allowing for easy retrieval and updates. For example, when a customer places an order, the system will automatically adjust the stock levels and store payment details, providing a seamless experience from the customer's side and easy management from the store's perspective.

3. Project Objectives

The primary objectives of this project are to design and implement an efficient **Database Management System (DBMS)** for an online grocery store, focusing on enhancing the store's operations and customer experience. The objectives include:

1. **Build a Structured Database**: Develop a relational database that organizes and manages key entities such as customers, products, orders, and suppliers in a systematic manner. This database will ensure all essential information is stored securely and is easily accessible.
2. **Manage Inventory in Real-Time**: Implement a system that tracks product stock levels, automatically updates inventory when purchases are made, and notifies administrators when products need to be reordered. This will help maintain product availability and streamline the inventory process.
3. **Enhance Order Management**: Create a system that handles the complete order lifecycle, from the moment customers place an order to when it is delivered. This includes tracking order status, processing payments, and ensuring customers can easily access their order history.
4. **Ensure Secure Payment Processing**: Safeguard sensitive transaction details by implementing secure payment processing that records payment methods and links them to specific orders. This objective is crucial for building customer trust and ensuring compliance with data protection standards.

5. **Improve Customer Experience**: Provide a seamless shopping experience for customers by allowing them to browse products, manage their shopping cart, and place orders efficiently. The system should offer real-time feedback on stock availability and order confirmations.
6. **Optimize Supplier Relationships**: Implement a system that manages supplier details, tracks the products each supplier provides, and allows for efficient reordering of stock. This will ensure the store maintains good relationships with suppliers and can replenish stock without delays.
7. **Ensure Scalability**: Design the database to be scalable, allowing the online grocery store to expand its operations—whether adding more products, handling a growing customer base, or managing more complex orders in the future.
8. **Minimize Data Redundancy and Ensure Data Integrity**: Follow **3rd Normal Form (3NF)** principles to minimize duplication of data and ensure that relationships between different entities (customers, products, orders, etc.) are accurate and consistent throughout the system.

4. Problem Statement

The core problem this project aims to solve is the **efficient management of an online grocery store's operations**, which involves handling multiple complex tasks simultaneously, such as managing a large inventory of products, processing customer orders, and tracking payments—all in real time. Without an optimized system in place, these tasks can become error-prone, slow, and inefficient, leading to stock discrepancies, delayed orders, and customer dissatisfaction.

Online grocery stores require a **centralized and well-structured database** that can seamlessly handle the interactions between customers, products, orders, and suppliers. As the business grows, managing the increasing volume of transactions, monitoring stock levels, and ensuring timely order fulfillment becomes more challenging without a robust database management system. The project addresses the need for a solution that reduces manual intervention, minimizes data duplication, and improves the overall efficiency of the store's operations.

This system will solve the problem by providing a **scalable, automated database** that manages everything from customer interactions and inventory tracking to payment processing and supplier coordination. Through this, the online grocery store can ensure smooth operations, maintain customer satisfaction, and support business growth.

5. Requirement Gathering

5.1 Interviews and Surveys

- **Objective:** To understand how real-world users or organizations manage such databases.
- **Method:** Since it is not possible to visit online stores so we went to offline grocery store where we can get similar type of software to explore and for some product inspection and asked some questions from the cashier and employees who are working there.

- **Proofs:**

Team Members

Unique product Id

Noting down Attributes that one product can have

Going through billing counter software

- **Key Findings:**

Questions for the Cashier:

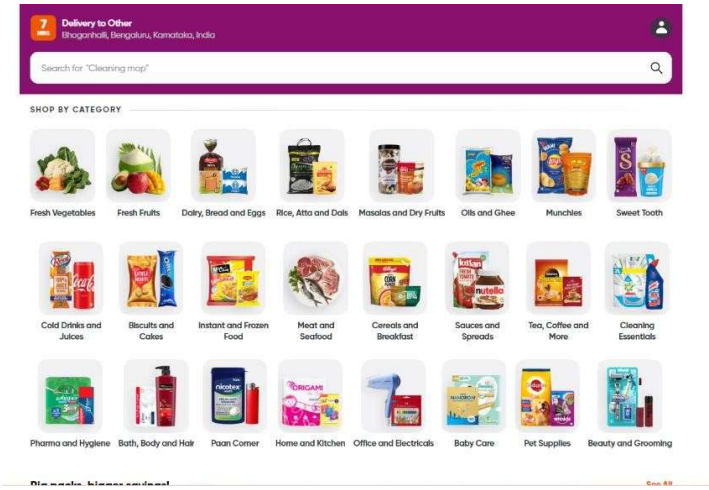
1. **How do you manage transactions during busy hours?**
   - **Answer:** I rely on the system for quick billing, but we also need manual inputs for items without barcodes or special discounts.
2. **What challenges do you face while processing payments?**
   - **Answer:** Sometimes, we face delays with card payments or network issues, but overall, the system helps speed up the process.
3. **Do you encounter any issues with tracking inventory or stock while billing?**
   - **Answer:** Yes, occasionally we run into issues where the system shows stock that's not available or vice versa.

Questions for Employees (Stockers or Managers):

1. **How do you manage product categories and supplier details?**
   - **Answer:** We categorize products based on supplier deliveries and use a system to track each category's performance.
2. **What difficulties do you face in updating stock information?**
   - **Answer:** Sometimes, stock data is not updated in real-time, which causes discrepancies in inventory.
3. **How is customer information or data handled, especially in terms of orders and feedback?**
   - **Answer:** Customer details are stored during transactions, but we mostly use them for receipts and order tracking purposes.

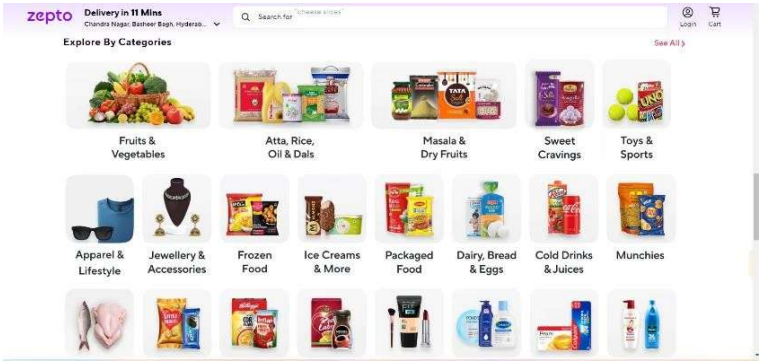
5.2 Website Exploration

- **Objective:** To explore existing websites or platforms that use similar databases.
- **Websites Explored:** SwiggyInstamart,Bigbasket,Zepto
- **Screenshots:**



Swiggy Instamart

Bigbasket



Zepto

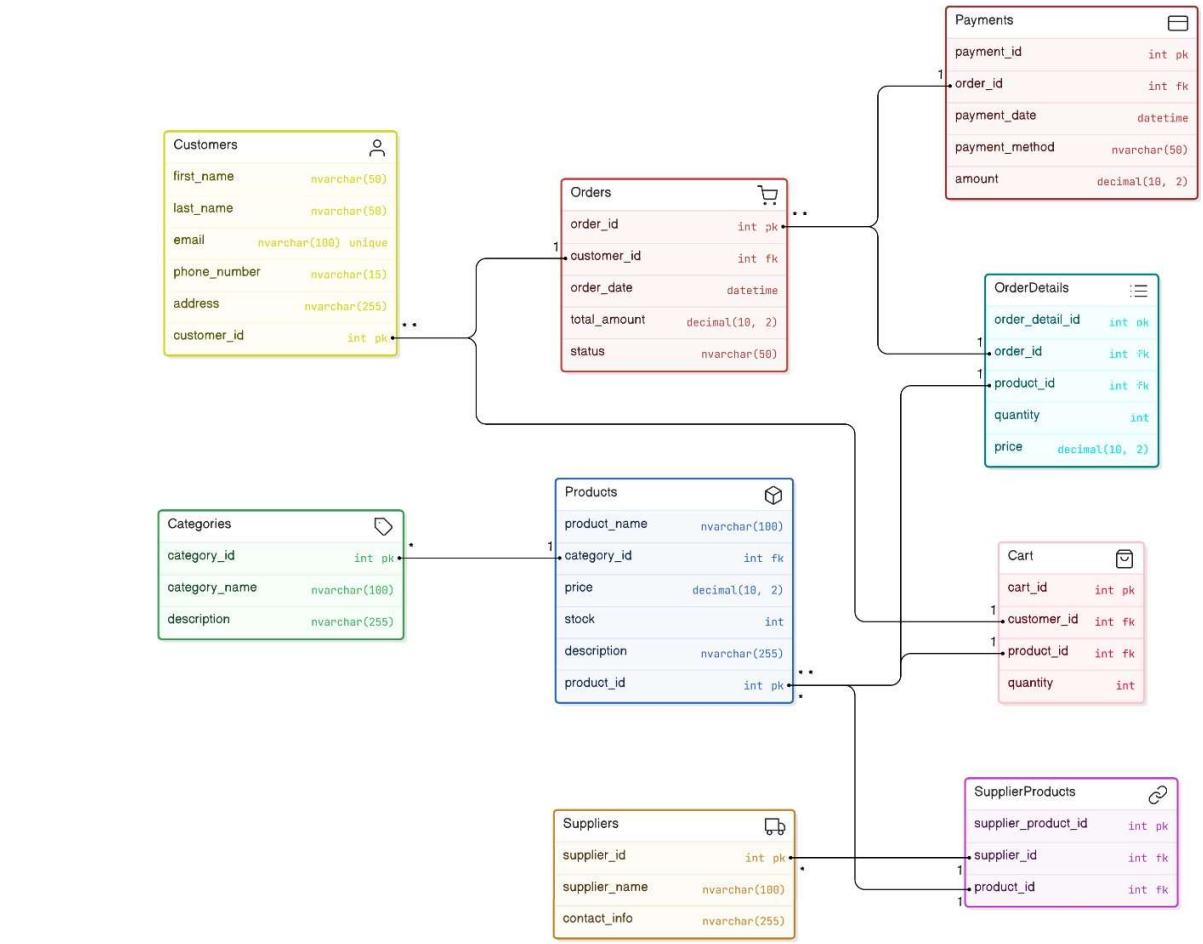
- **Observations:**

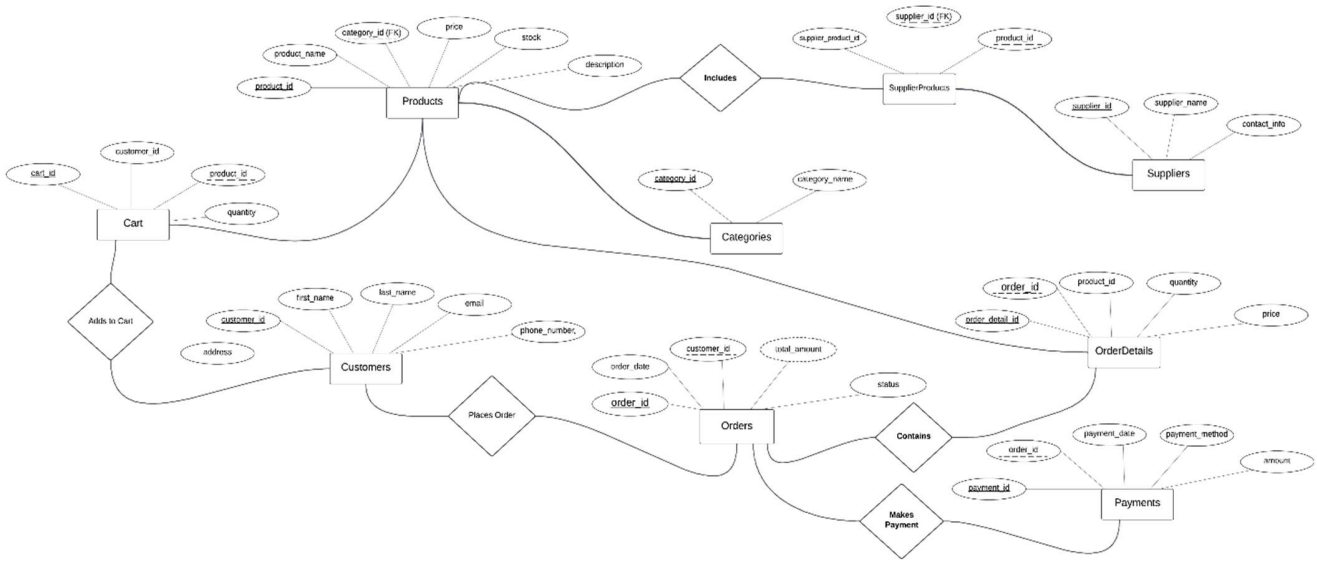
  Real-Time Inventory and Order Management: Just like Zepto, the project should implement real-time updates to ensure stock accuracy and minimize customer frustration with out-of-stock items.
  Efficient Delivery and Fulfillment Systems: Learning from Swiggy Instamart, integrating delivery services and order tracking can significantly improve customer satisfaction.
  Scalable Product Management: As seen in BigBasket, a well-structured product catalog and supplier coordination system will allow the store to handle growth and offer a wide range of products without losing operational efficiency.

6. ER Diagram and Relational Schema

- **ER Diagram:**

  Include the Entity-Relationship Diagram that represents all entities (users, content, subscriptions, etc.) and relationships between them


Payments
payment_id int pk
order_id int fk
payment_date datetime
payment_method nvarchar(50)
amount decimal(10, 2)
Customers
first_name nvarchar(50)
last_name nvarchar(50)
email nvarchar(100) unique
phone_number nvarchar(15)
address nvarchar(255)
customer_id int pk
Orders
order_id int pk
customer_id int fk
order_date datetime
total_amount decimal(10, 2)
status nvarchar(50)
OrderDetails
order_detail_id int pk
order_id int fk
product_id int fk
quantity int
price decimal(10, 2)
Categories
category_id int pk
category_name nvarchar(100)
description nvarchar(255)
Products
product_name nvarchar(100)
category_id int fk
price decimal(10, 2)
stock int
description nvarchar(255)
product_id int pk
Cart
cart_id int pk
customer_id int fk
product_id int fk
quantity int
Suppliers
supplier_id int pk
supplier_name nvarchar(100)
contact_info nvarchar(255)
SupplierProducts
supplier_product_id int pk
supplier_id int fk
product_id int fk


Products
Includes
SupplierProducts
Suppliers
Cart
Categories
Adds to Cart
Customers
Places Order
Orders
Contains
OrderDetails
Makes Payment
Payments

1. Customers Entity

- **Attributes**: customer_id (PK), first_name, last_name, email, phone_number, address.
- **Relationships**:
    - **Adds to Cart**: Customers can add products to their Cart.
    - **Places Order**: Customers place Orders.

2. Cart Entity

- **Attributes**: cart_id (PK), customer_id (FK), product_id (FK), quantity.
- **Relationships**:
    - **Adds to Cart**: Cart stores the products that customers intend to buy.
    - Connected to **Customers** via customer_id (FK).
    - Connected to **Products** via product_id (FK).

3. Products Entity

- **Attributes**: product_id (PK), product_name, category_id (FK), price, stock, description.
- **Relationships**:
    - **Includes**: Products are included in **OrderDetails**.
    - **Adds to Cart**: Products are added to the Cart by Customers.
    - Connected to **Categories** via category_id (FK).
    - Connected to **SupplierProducts** and **Suppliers**.

4. Categories Entity

- **Attributes**: category_id (PK), category_name.
- **Relationships**:
    - Connected to **Products** via category_id (FK) for classification.

5. Suppliers Entity

- **Attributes**: supplier_id (PK), supplier_name, contact_info.
- **Relationships**:
    - **Includes**: Suppliers provide products through **SupplierProducts**.
    - Connected to **SupplierProducts**.

6. SupplierProducts Entity

- **Attributes**: supplier_product_id (PK), supplier_id (FK), product_id (FK).
- **Relationships**:

- **Includes**: Connects **Suppliers** and **Products**.

7. Orders Entity

- **Attributes**: order_id (PK), customer_id (FK), order_date, total_amount, status.
- **Relationships**:
  - **Places Order**: Customers place Orders.
  - **Contains**: Orders are detailed in **OrderDetails**.
  - **Makes Payment**: Orders are paid via **Payments**.

8. OrderDetails Entity

- **Attributes**: order_detail_id (PK), order_id (FK), product_id (FK), quantity, price.
- **Relationships**:
  - **Contains**: Contains the details of each Order (product, quantity, price).

9. Payments Entity

- **Attributes**: payment_id (PK), order_id (FK), payment_date, payment_method, amount.
- **Relationships**:
  - **Makes Payment**: Payments are made for Orders.

Key Connections:

- **Customers** are connected to **Orders**, **Cart**, and **Payments**.
- **Products** are connected to **Categories**, **Cart**, and **OrderDetails**.
- **Orders** are connected to **OrderDetails** and **Payments**.
- **Suppliers** are connected to **SupplierProducts**, which links to **Products**.

This diagram provides a clear structure for the online grocery store, showing how customers interact with products, how orders are placed, and how the suppliers and products are managed.

- **Relational Schema for the Online Grocery Store Project:**

Below is the relational schema that describes the tables, their columns, and the relationships among them:

Customers

- customer_id (PK)
- first_name
- last_name
- email
- phone_number
- address

Products

- product_id (PK)
- product_name
- category_id (FK)
- price
- stock
- description

Categories

- category_id (PK)
- category_name

Suppliers

- supplier_id (PK)
- supplier_name
- contact_info

Supplier Products

- supplier_product_id (PK)
- supplier_id (FK)
- product_id (FK)

Cart

- cart_id (PK)
- customer_id (FK)
- product_id (FK)
- quantity

Orders

- order_id (PK)
- customer_id (FK)
- order_date
- total_amount
- status

Order Details

- order_detail_id (PK)
- order_id (FK)
- product_id (FK)
- quantity
- price

Payments

- payment_id (PK)
- order_id (FK)
- payment_date
- payment_method
- amount

Explanation of Relationships:

- **Customers** are related to **Orders** (via customer_id), and can have multiple **Orders**.
- **Products** are categorized by the **Categories** table (via category_id).
- **Suppliers** provide products via the **SupplierProducts** table, linking **Suppliers** and **Products** (via supplier_id and product_id).
- **Customers** can add **Products** to their **Cart**, which holds multiple products (via customer_id and product_id).
- **Orders** consist of multiple **OrderDetails**, where each **OrderDetail** holds details about the products ordered (via order_id and product_id).
- **Payments** are linked to **Orders** and store the payment information (via order_id).

7. Database Design and Normalization

7.1 Table Descriptions

- **Table 1: Customers Table**
    - Attributes: customer_id(Primary Key), first_name, last_name, email, password, phone_number, address
    - Primary Key: customer_id
    - **Description:** This table stores customer information.
- **Table 2: Products Table**
    - Attributes: product_id(Primary Key), product_name, category, price, stock
    - Primary Key: product_id
    - **Description:** A brief description of the product.
    -
- **Table 3: Categories Table**
    - Attributes: category_id(Primary Key), category_name
    - Primary Key: category_id
    - **Description:** Manages product categories to help organize products**.**

- **Table 4: Orders Table**

  - Attributes: `order_id`(Primary Key), `customer_id (Foreign Key), order_date, total_amount, status`
  - Primary Key: `order_id`
  - **Description:** This table tracks the orders placed by customers.

- **Table 5: OrderDetails Table**

  - Attributes: order_detail_id (Primary Key), order_id (Foreign Key), product_id (Foreign Key), quantity, price
  - Primary Key: order_detail_id
  - **Description:** To store the details of each item in a particular order.

- **Table 6: Cart Table**

  - Attributes: cart_id (Primary Key), customer_id (Foreign Key), product_id (Foreign Key), quantity
  - Primary Key: cart_id
  - **Description:** Temporarily stores items that a customer adds to their shopping cart before checkout.

- **Table 7: Payments Table**

  - Attributes: payment_id (Primary Key), order_id (Foreign Key), payment_date, payment_method, amount
  - Primary Key: payment_id
  - **Description:** Stores payment details related to the orders.

- **Table 8: Suppliers Table**

  - Attributes: supplier_id (Primary Key), supplier_name, contact_info
  - Primary Key: supplier_id
  - **Description:** Stores information about product suppliers.

- **Table 9: Supplier Products Table**

  - Attributes: supplier_product_id, supplier_id (Foreign Key), product_id(Foreign Key)
  - Primary Key: supplier_product_id
  - **Description:** This table links Suppliers and Products, allowing multiple suppliers to supply the same product and a single supplier to supply multiple products.

7.2 Normalization Process for the Online Grocery Store Project:

1NF (First Normal Form):

- **Objective:** Ensure that each column contains atomic values (no repeating groups or sets of values).

In the case of our tables, each column already contains atomic values, meaning no multi-valued attributes exist. Each product, customer, order, and payment detail is stored in its own unique row, making the tables compliant with **1NF**.

Example:

- In the **Products** table, the product_name, category_id, price, stock, and description columns hold atomic values for each product.

Solution for Repeating Groups:

- If a product had multiple categories or if a customer had multiple addresses, we would need to separate those out into their own tables to comply with 1NF. However, since our structure already meets these criteria, 1NF is satisfied.

2NF (Second Normal Form):

- **Objective:** Remove partial dependencies, meaning that all non-primary key attributes must be fully dependent on the entire primary key. This only applies to tables with composite primary keys.

In this case, most of our tables have simple (single-column) primary keys, so we don't need to worry about partial dependencies in those tables. However, let's consider the **OrderDetails** table, which has a composite key (order_id, product_id).

Example:

- In the **OrderDetails** table, all non-key attributes (quantity, price) are fully dependent on the combination of both order_id and product_id. There are no partial dependencies, so the table is in **2NF**.

3NF (Third Normal Form):

- **Objective:** Eliminate transitive dependencies. All non-key attributes must be directly dependent on the primary key, not through another non-key attribute.

We should check for any transitive dependencies in our database schema. For example:

- In the **Customers** table, there are no transitive dependencies because all attributes (like first_name, last_name, email) directly depend on the primary key (customer_id).
- In the **Orders** table, the customer_id directly refers to the **Customers** table, and the total_amount is an independent attribute that depends only on the order, not on any other attribute.

- In the **Payments** table, order_id is directly tied to the **Orders** table and the payment details (payment_date, payment_method, amount) are directly dependent on the **Payments** table.

Solution for Transitive Dependency:

- No transitive dependencies are detected in our schema. Therefore, all tables are in **3NF** as no attribute is dependent on anything other than the primary key.

8. SQL Queries and Database Operations

8.1 Create Tables

- Customers Table

```
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY IDENTITY(1,1),
    first_name NVARCHAR(50),
    last_name NVARCHAR(50),
    email NVARCHAR(100) UNIQUE,
    phone_number NVARCHAR(15),
    address NVARCHAR(255)
);
```

- Products Table

```
CREATE TABLE Products (
    product_id INT PRIMARY KEY IDENTITY(1,1),
    product_name NVARCHAR(100),
    category_id INT,
    price DECIMAL(10, 2),
    stock INT,
    description NVARCHAR(255),
    FOREIGN KEY (category_id) REFERENCES Categories(category_id)
);
```

- Categories Table

```
CREATE TABLE Categories (
    category_id INT PRIMARY KEY IDENTITY(1,1),
    category_name NVARCHAR(100) UNIQUE
);
```

- Orders Table

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY IDENTITY(1,1),
```

```
    customer_id INT,
    order_date DATETIME,
    total_amount DECIMAL(10, 2),
    status NVARCHAR(50),
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

- OrderDetails Table

```
CREATE TABLE OrderDetails (
    order_detail_id INT PRIMARY KEY IDENTITY(1,1),
    order_id INT,
    product_id INT,
    quantity INT,
    price DECIMAL(10, 2),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

- Cart Table

```
CREATE TABLE Cart (
    cart_id INT PRIMARY KEY IDENTITY(1,1),
    customer_id INT,
    product_id INT,
    quantity INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

- Payments Table

```
CREATE TABLE Payments (
    payment_id INT PRIMARY KEY IDENTITY(1,1),
    order_id INT,
    payment_date DATETIME,
    payment_method NVARCHAR(50),
    amount DECIMAL(10, 2),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);
```

- Suppliers Table

```
CREATE TABLE Suppliers (
    supplier_id INT PRIMARY KEY IDENTITY(1,1),
    supplier_name NVARCHAR(100),
    contact_info NVARCHAR(255)
);
```

- SupplierProducts Table

```
CREATE TABLE SupplierProducts (
    supplier_product_id INT PRIMARY KEY IDENTITY(1,1),
    supplier_id INT,
    product_id INT,
    FOREIGN KEY (supplier_id) REFERENCES Suppliers(supplier_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

8.2 Insert Data

- Customers Table

```
INSERT INTO Customers (first_name, last_name, email, phone_number, address)
VALUES
('John', 'Doe', 'john.doe@example.com', '1234567890', '123 Main St, New York, NY'),
('Jane', 'Smith', 'jane.smith@example.com', '0987654321', '456 Oak Ave, Los Angeles,
CA');
```

- Products Table

```
INSERT INTO Products (product_name, category_id, price, stock, description)
VALUES
('Apple', 1, 2.50, 100, 'Fresh red apples'),
('Banana', 1, 1.00, 200, 'Ripe yellow bananas'),
('Carrot', 2, 1.20, 150, 'Crunchy orange carrots'),
('Broccoli', 2, 1.50, 80, 'Fresh green broccoli heads'),
('Milk', 3, 3.00, 50, 'Whole milk from local farms'),
('Cheese', 3, 5.00, 30, 'Cheddar cheese block');
```

- Categories Table

```
INSERT INTO Categories (category_name)
VALUES
('Fruits'),
('Vegetables'),
('Dairy');
select * from Products
```

- Orders Table

```
INSERT INTO Orders (customer_id, order_date, total_amount, status)
VALUES
(1, GETDATE(), 15.50, 'Pending'),  -- Order for Customer 1
```

```
(2, GETDATE(), 8.00, 'Completed'),  -- Order for Customer 2
(1, GETDATE(), 25.75, 'Shipped'),    -- Another order for Customer 1
(2, GETDATE(), 12.50, 'Pending');    -- Another order for Customer 2
```

- OrderDetails Table

```
INSERT INTO OrderDetails (order_id, product_id, quantity, price)
VALUES
(1, 1, 5, 2.50),  -- Order 1 includes 5 Apples
(1, 3, 2, 1.20),  -- Order 1 includes 2 Carrots
(2, 2, 3, 1.00),  -- Order 2 includes 3 Bananas
(3, 5, 1, 3.00);  -- Order 3 includes 1 Milk
```

- Cart Table

```
INSERT INTO Cart (customer_id, product_id, quantity)
VALUES
(1, 1, 5),  -- Customer 1 adds 5 Apples to their cart
(1, 3, 2),  -- Customer 1 adds 2 Carrots to their cart
(2, 2, 3),  -- Customer 2 adds 3 Bananas to their cart
(2, 5, 1);  -- Customer 2 adds 1 Milk to their cart
```

- Payments Table

```
INSERT INTO Payments (order_id, payment_date, payment_method, amount)
VALUES
(1, GETDATE(), 'Credit Card', 15.50), -- Payment for Order 1
(2, GETDATE(), 'Debit Card', 8.00),     -- Payment for Order 2
(3, GETDATE(), 'PayPal', 25.75),        -- Payment for Order 3
(4, GETDATE(), 'Cash', 12.50);          -- Payment for Order 4
```

- Suppliers Table

```
INSERT INTO Suppliers (supplier_name, contact_info)
VALUES
('FreshFarms', 'Michael Adams, 555-1234, freshfarms@example.com'),
('DailyDairy', 'Sarah Connor, 555-9876, dailydairy@example.com'),
('GreenGrocer', 'Tom Hanks, 555-4321, greengrocer@example.com'),
('Fruitful Harvest', 'Lucy Liu, 555-6789, fruitfulharvest@example.com');
```

- SupplierProducts Table

```
INSERT INTO SupplierProducts (supplier_id, product_id)
VALUES
(1, 1),  -- FreshFarms supplies Apples
(1, 3),  -- FreshFarms supplies Carrots
(2, 5),  -- DailyDairy supplies Milk
(3, 6);  -- GreenGrocer supplies Cheese
```

8.3 Queries for Data Retrieval

```
/*1.Retrieve all orders placed by customers along with their total amount and
order status, but only for orders where the total amount exceeds 15rs.
*/

SELECT C.first_name, C.last_name, O.order_id, O.total_amount, O.status
FROM Orders O
JOIN Customers C ON O.customer_id = C.customer_id
WHERE O.total_amount > 15;
```

Results | Messages

	first_name	last_name	order_id	total_amount	status
1	John	Doe	1	15.50	Pending
2	John	Doe	3	25.75	Shipped
3	John	Doe	5	150.00	Processing

```
/*2.Find the top 3 most expensive products available in the store and display
their product name and price. */
SELECT TOP 3
    product_name, price
FROM Products
ORDER BY price DESC;
```

Results | Messages

	product_name	price
1	Organic Honey	150.00
2	Cheese	5.00
3	Milk	3.00

```
/*3. List all products from the cart of a particular customer (say customer ID = 1) along
with the quantity added to the cart. */

SELECT P.product_name, C.quantity
FROM Cart C
JOIN Products P ON C.product_id = P.product_id
WHERE C.customer_id = 1;
```

Results | Messages

	product_name	quantity
1	Apple	5
2	Carrot	2

```
/*4. Find how many products are available in each category. */

SELECT c.category_name, COUNT(P.product_id) AS total_products
FROM Products P
JOIN Categories c ON P.category_id = c.category_id
GROUP BY c.category_name;
```

Results | Messages

	category_name	total_products
1	Dairy	2
2	Fruits	2
3	Vegetables	3

```sql
/*5. Find the customers who have placed orders totaling more than 10rs in
value across all their orders combined. */
SELECT C.customer_id, C.first_name, C.last_name, SUM(O.total_amount) AS total_spent
FROM Customers C
JOIN Orders O ON C.customer_id = O.customer_id
GROUP BY C.customer_id, C.first_name, C.last_name
HAVING SUM(O.total_amount) > 10;
```

Results Messages

	customer_id	first_name	last_name	total_spent
1	1	John	Doe	191.25
2	2	Jane	Smith	20.50

```sql
/*6.Retrieve all customers who have never placed an order.
*/
SELECT C.customer_id, C.first_name, C.last_name
FROM Customers C
LEFT JOIN Orders O ON C.customer_id = O.customer_id
WHERE O.order_id IS NULL;
```

Results Messages

	customer_id	first_name	last_name
1	3	Alex	Brown
2	4	John	Doe

```sql
/*7. Get the products that have never been ordered by any customer.
*/
SELECT P.product_name
FROM Products P
LEFT JOIN OrderDetails OD ON P.product_id = OD.product_id
WHERE OD.product_id IS NULL;
```

Results Messages

	product_name
1	Broccoli
2	Cheese
3	Organic Honey

```sql
/*8. Find the total revenue generated by each product, ordered from highest
to lowest revenue.*/
SELECT P.product_name, SUM(OD.quantity * OD.price) AS total_revenue
FROM OrderDetails OD
JOIN Products P ON OD.product_id = P.product_id
GROUP BY P.product_name
ORDER BY total_revenue DESC;
```

Results Messages

	product_name	total_revenue
1	Apple	162.50
2	Banana	3.00
3	Milk	3.00
4	Carrot	2.40

```sql
/*9. Find the supplier that supplies the most number of different products.
*/
SELECT TOP 1  S.supplier_name,
    COUNT(DISTINCT SP.product_id) AS total_products
FROM Suppliers S INNER JOIN
    SupplierProducts SP ON S.supplier_id = SP.supplier_id
GROUP BY  S.supplier_name
ORDER BY  total_products DESC;
```

Results Messages

	supplier_name	total_products
1	FreshFarms	2

```sql
/*10. Retrieve all products along with their categories:
*/
SELECT P.product_name, C.category_name, P.price, P.stock
FROM Products P
JOIN Categories C ON P.category_id = C.category_id;
```

Results Messages

	product_name	category_name	price	stock
1	Apple	Fruits	2.50	95
2	Banana	Fruits	1.00	200
3	Carrot	Vegetables	1.20	150
4	Broccoli	Vegetables	1.50	80
5	Milk	Dairy	3.00	100
6	Cheese	Dairy	5.00	160
7	Organic Honey	Vegetables	150.00	100

```sql
/*11. Retrieve all orders for a specific customer:
*/
SELECT O.order_id, O.order_date, O.total_amount, O.status
FROM Orders O
WHERE O.customer_id = 1;
```

Results Messages

	order_id	order_date	total_amount	status
1	1	2024-10-23 15:10:04.173	15.50	Pending
2	3	2024-10-23 15:10:04.173	25.75	Shipped
3	5	2024-11-12 21:43:17.447	150.00	Processing

```sql
/*12. Find customers who have placed more than 1 orders:
*/
SELECT C.customer_id, C.first_name, C.last_name, COUNT(O.order_id) AS total_orders
FROM Customers C
JOIN Orders O ON C.customer_id = O.customer_id
GROUP BY C.customer_id, C.first_name, C.last_name
HAVING COUNT(O.order_id) > 1;
```

Results Messages

	customer_id	first_name	last_name	total_orders
1	1	John	Doe	3
2	2	Jane	Smith	2

```sql
/*13. Update customer email address:
*/
UPDATE Customers
SET email = 'ravivarma@gmail.com'
WHERE customer_id = 2;

-----------------------------------------------------------------------------------------------------
/*SUB QUERIES*/
/*1.Find Customers Who Placed the Most Orders*/
SELECT customer_id, first_name, last_name
FROM Customers
WHERE customer_id = (
    SELECT TOP 1 customer_id
    FROM Orders
    GROUP BY customer_id
    ORDER BY COUNT(order_id) DESC
);
```

132 %

Results | Messages

	customer_id	first_name	last_name
1	1	John	Doe

```sql
/*2. Find Products with Low Stock*/
SELECT product_id, product_name, stock
FROM Products
WHERE stock < (
    SELECT AVG(stock)
    FROM Products
);
```

Results | Messages

	product_id	product_name	stock
1	1	Apple	95
2	4	Broccoli	80
3	5	Milk	100
4	7	Organic Honey	100

```sql
/*3.Retrieve the Total Amount Spent by Each Customer*/
SELECT customer_id,
       (SELECT SUM(total_amount)
        FROM Orders
        WHERE Orders.customer_id = Customers.customer_id) AS total_spent
FROM Customers;
```

Results | Messages

	customer_id	total_spent
1	1	191.25
2	2	20.50
3	3	NULL
4	4	NULL

```sql
/*4.Find Customers Who Have Not Placed Any Orders*/
SELECT customer_id, first_name, last_name
FROM Customers
WHERE customer_id NOT IN (
    SELECT DISTINCT customer_id
    FROM Orders
);
```

Results | Messages

	customer_id	first_name	last_name
1	3	Alex	Brown
2	4	John	Doe

8.4 Transaction

```sql
/*Static Transaction for updating the stock of a product after a customer places an order. */
BEGIN TRY
    BEGIN TRANSACTION;

    -- Step 1: Insert a new order
    INSERT INTO Orders (customer_id, order_date, total_amount, status)
    VALUES (1, GETDATE(), 150.00, 'Processing');

    -- Get the last inserted order ID
    DECLARE @OrderID INT = SCOPE_IDENTITY();
    DECLARE @ProductID INT = 1;  -- Assuming product_id = 1 for this example
    DECLARE @Quantity INT = 5;   -- Assuming the customer purchased 5 units
    DECLARE @Stock INT;

    -- Step 2: Check current stock level
    SELECT @Stock = stock FROM Products WHERE product_id = @ProductID;

    -- Check if there is enough stock
    IF @Stock < @Quantity
    BEGIN
        PRINT 'Not enough stock available';
        ROLLBACK TRANSACTION;
        RETURN;
    END

    -- Step 3: Insert order details
    INSERT INTO OrderDetails (order_id, product_id, quantity, price)
    VALUES (@OrderID, @ProductID, @Quantity, 30.00);  -- Assuming price = 30.00

    -- Step 4: Update the product stock
    UPDATE Products
    SET stock = stock - @Quantity
    WHERE product_id = @ProductID;

    -- Commit the transaction if everything is successful
    COMMIT TRANSACTION;
    PRINT 'Transaction completed successfully. Product stock updated.';

END TRY
BEGIN CATCH
    -- Rollback the transaction if there is an error
    ROLLBACK TRANSACTION;
    PRINT 'Transaction failed. Changes rolled back.';
```

```
    PRINT ERROR_MESSAGE();
END CATCH;
```

8.5 Stored Procedure

```
/* Add New Customer*/
CREATE PROCEDURE AddNewCustomer
    @first_name NVARCHAR(50),
    @last_name NVARCHAR(50),
    @email NVARCHAR(100),
    @phone_number NVARCHAR(20),
    @address NVARCHAR(255)
AS
BEGIN
    INSERT INTO Customers (first_name, last_name, email, phone_number, address)
    VALUES (@first_name, @last_name, @email, @phone_number, @address);
    PRINT 'Customer added successfully';
END;
/*EXEC AddNewCustomer
    @first_name = 'Alice',
    @last_name = 'Smith',
    @email = 'alice.smith@example.com',
    @phone_number = '1234567890',
    @address = '456 Oak Street, City';*/

select * from Customers
```

```
(1 row affected)
Customer added successfully

Completion time: 2024-11-25T12:55:49.6239182+05:30
```

8.6 Function

```
/*Creating A Function */
---User-Defined Function: Calculate Total Revenue for a Product
CREATE FUNCTION CalculateTotalRevenue (@product_id INT)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @total_revenue DECIMAL(10, 2);

    -- Calculate total revenue by summing (quantity * price) for the given product
    SELECT @total_revenue = SUM(quantity * price)
    FROM OrderDetails
    WHERE product_id = @product_id;

    -- If no orders exist for the product, set revenue to 0
    IF @total_revenue IS NULL
        SET @total_revenue = 0;

    RETURN @total_revenue;
END;

---using the created function
SELECT dbo.CalculateTotalRevenue(1) AS Total_Revenue;

select * from Products
    select * from OrderDetails
```

Results | Messages

	Total_Revenue
1	162.50

8.7 Cursor

```
/*Creating an Dynamic Cursor to Restocking the products that have lower stock*/
DECLARE @product_id INT;
DECLARE @product_name NVARCHAR(100);
DECLARE @current_stock INT;
DECLARE @threshold INT = 100; -- Minimum stock level before restocking
DECLARE @restock_amount INT = 10; -- Amount to restock if stock is low

-- Declare a dynamic cursor
DECLARE LowStockCursor CURSOR FOR
    SELECT product_id, product_name, stock
    FROM Products
    WHERE stock < @threshold;

-- Open the cursor
OPEN LowStockCursor;

-- Fetch the first row into the variables
FETCH NEXT FROM LowStockCursor INTO @product_id, @product_name, @current_stock;

-- Loop through the rows
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Print the current product details
    PRINT 'Restocking product: ' + @product_name + ' (Product ID: ' + CAST(@product_id AS NVARCHAR) +
')';
    PRINT 'Current Stock: ' + CAST(@current_stock AS NVARCHAR);

    -- Update the stock for the product
    UPDATE Products
    SET stock = stock + @restock_amount
    WHERE product_id = @product_id;

    -- Print the new stock status
    PRINT 'New Stock Level: ' + CAST(@current_stock + @restock_amount AS NVARCHAR);

    -- Fetch the next row
    FETCH NEXT FROM LowStockCursor INTO @product_id, @product_name, @current_stock;
END;

-- Close and deallocate the cursor
CLOSE LowStockCursor;
    DEALLOCATE LowStockCursor;
```

```
Restocking product: Apple (Product ID: 1)
Current Stock: 95

(1 row affected)
New Stock Level: 105
Restocking product: Broccoli (Product ID: 4)
Current Stock: 80

(1 row affected)
New Stock Level: 90

Completion time: 2024-11-25T12:57:42.2521713+05:30
```

8.8 Trigger

```
  /* Trigger: Update Product Stock on Order Placement*/
-- Creating the trigger
CREATE TRIGGER trg_UpdateProductStock
ON OrderDetails
AFTER INSERT
AS
BEGIN
    SET NOCOUNT ON;

    -- Update the product stock based on the inserted order details
    UPDATE Products
    SET stock = stock - inserted.quantity
    FROM Products
    INNER JOIN inserted
    ON Products.product_id = inserted.product_id;

    -- Optionally, check for negative stock levels
    IF EXISTS (SELECT 1 FROM Products WHERE stock < 0)
    BEGIN
        PRINT 'Warning: Stock level for one or more products is negative!';
    END
END;
/*Usage:
Place an order by inserting rows into the OrderDetails table.
The Products table automatically updates its stock for the corresponding product.
This trigger helps maintain inventory accuracy without manual intervention,
    streamlining operations for your grocery store DBMS.*/
```

9. Conclusion

In conclusion, this project successfully established a robust database management system for an online grocery store, addressing the key objectives outlined at the outset. By creating a well-structured database that encompasses essential tables such as Customers, Products, Categories, Suppliers, Orders, Cart, OrderDetails, and Payments, we have developed a comprehensive framework that streamlines operations and enhances user experience.

The system effectively manages customer data, product inventory, and order processing, ensuring that customers can easily browse, select, and purchase their desired items. By implementing relationships between tables, we facilitated efficient data retrieval and integrity, allowing for smooth operations and accurate reporting.

Moreover, this database enables personalized customer interactions, such as managing shopping carts and tracking order history, which is vital for customer satisfaction and retention. The structured design also lays the foundation for potential future enhancements, such as integrating advanced analytics for personalized recommendations and inventory forecasting.

Overall, this project has not only met its initial objectives but has also positioned the grocery store for growth and adaptability in a competitive market. The successful implementation of this database management system will undoubtedly contribute to the store's operational efficiency and customer-centric approach.