

<b>Introduction</b>	<b>2</b>
1.1 Objective of the Project	2
1.2 Scope of the Online Shopping Platform	2
1.3 Key Features	4
2.1 Entity-Relationship Diagram (ERD)	5
Entities and Attributes	5
Relationships and Cardinality	6
2.2 Normalized Database Schema	7
3. SQL Implementation	8
3.1 Data Definition Language (DDL)	8
3.2 Data Manipulation Language (DML)	11
1. INSERT Operations	11
2. SELECT Operations	12
3. UPDATE Operations	13
4. DELETE Operations	13
3.3 Triggers	14
1. Trigger on INSERT (Maintain Inventory Log)	14
2. Trigger on UPDATE (Update Inventory Quantity)	14
3. Trigger on DELETE (Prevent Deletion of Active Orders)	15
4. Trigger on DELETE (Archive Deleted Orders)	16
5. Trigger on INSERT (Enforce Product Price Validation)	17
Summary of Use Cases	17
4. Reporting and Views	17
4.1 Sales Report	18
4.2 Tax Report	19
4.3 Inventory Report	19
4.4 Customer Report	20
4.5 Product Report	21
Key Benefits of Using Views	21
5. Stored Procedures	21
5.1 Parametrized Stored Procedures	22
5.2 Non-Parametrized Stored Procedures	23
5.3 Inventory Management Procedure	24
5.4 Order Processing Procedure	24
Benefits of Stored Procedures	26
6. Transactions	26
6.1 Transaction to Place an Order	26
6.2 Transaction to Cancel an Order	28
6.3 Transaction for Tax Calculation and Update	28
6.4 Transaction for Customer Refund	29
7. Database Security	31

7.1 User Authentication and Authorization	31
7.2 Data Encryption	32
7.3 Auditing and Logging	37
7.4 Backup and Recovery	37
7.5 Best Practices for Database Security	38
8. Backup and Recovery	39
8.1 Types of Backups	39
8.2 Automated Backup Scheduling	39
8.3 Backup Storage Best Practices	40
8.4 Recovery Process	40
8.5 Disaster Recovery Plan	41
8.6 Tools for Backup and Recovery	41
9. Testing and Validation	41
A. Unit Testing	42
B. Integration Testing	42
C. Performance Testing	42
D. Data Validation	42
E. Recovery Testing	42
F. Security Testing	42
10. Conclusion	42
A. Recap of Objectives	42
B. Achievements	42
11. References	43
A. Textbooks	43
B. Online Resources	43
C. Communities	

# Introduction

The advancement of technology has transformed the way people shop and interact with sellers. Online shopping platforms have become an essential part of modern commerce, offering convenience, variety, and accessibility to customers worldwide. This project aims to design and implement a robust database system for an online shopping platform. The platform will enable seamless interactions between customers and sellers, providing functionalities for browsing, purchasing, and managing products. A well-structured database is crucial for ensuring the efficiency, reliability, and scalability of the platform.

This document presents a detailed blueprint for developing the database system, including its design, implementation, and management. The system will cater to various essential features such as managing customer and seller information, tracking orders, and supporting cashless online payments. Additionally, advanced database functionalities, including triggers, views, stored procedures, and backup mechanisms, will be implemented to enhance the overall reliability and usability of the platform.

## 1.1 Objective of the Project

The primary objective of this project is to design and implement a comprehensive database system to support the operations of an online shopping platform. Specifically, the project aims to:

1. **Efficiently Manage Data:** Design a normalized database schema to store and manage information about customers, sellers, products, orders, payments, and shipments.
2. **Facilitate Seamless Transactions:** Enable customers to place orders, make payments, and track shipments with minimal friction.
3. **Ensure Data Integrity and Security:** Implement mechanisms such as triggers, constraints, and user permissions to maintain the integrity and confidentiality of the data.
4. **Provide Actionable Insights:** Develop views and reports for sales, taxes, inventory, and customer activity to support decision-making for both administrators and sellers.
5. **Support Scalability and Reliability:** Include features like backup and recovery to ensure data is protected and can scale as the platform grows.
6. **Enhance User Experience:** Leverage advanced database features to deliver a smooth and responsive user experience for customers and sellers alike.

By achieving these objectives, the project will lay a strong foundation for an online shopping platform capable of meeting the demands of a dynamic e-commerce environment.

## 1.2 Scope of the Online Shopping Platform

The scope of this project encompasses the design, development, and deployment of a database system for an online shopping platform. The platform aims to facilitate efficient and user-friendly interactions between customers and sellers, covering the following key aspects:

1. **Customer Management:** The system will maintain detailed records of customers, including their names, email addresses, shipping addresses, and payment preferences. It will ensure seamless onboarding and management of customer accounts.
2. **Seller Management:** Sellers can register and manage their profiles, including contact information, business addresses, and product inventories. The platform will enable sellers to update product details, prices, and stock availability dynamically.
3. **Product Management:** A comprehensive inventory system will store and organize product information such as names, descriptions, prices, available quantities, and associated sellers. This will allow customers to browse and search products efficiently.
4. **Order Processing:** The system will support order placement, tracking, and management. Customers will be able to specify product quantities, select shipping methods, and view order statuses in real time.
5. **Payment Integration:** The platform will offer secure and cashless online payment methods, ensuring smooth transactions. Payment gateways will be integrated to handle credit/debit cards and other digital payment methods.
6. **Shipping and Tracking:** The system will manage shipment details, including addresses, tracking numbers, and shipment statuses. Customers will be able to monitor their orders from dispatch to delivery.
7. **Reporting and Analytics:** The database will generate detailed reports on sales, taxes, inventory levels, and customer behavior, providing valuable insights to administrators and sellers for strategic decision-making.
8. **Scalability and Reliability:** The design will ensure that the database can handle increasing user demands and provide consistent performance, even as the platform grows.
9. **Backup and Recovery:** Robust mechanisms for data backup and recovery will be implemented to safeguard against data loss and ensure quick recovery in case of unexpected failures.

The project focuses on building a system that is not only functional and reliable but also scalable and adaptable to future needs. By addressing these aspects, the platform will serve as a cornerstone for a modern, competitive e-commerce environment.

## 1.3 Key Features

The proposed online shopping platform will include a range of key features to ensure a seamless and efficient experience for both customers and sellers. These features are designed to meet the diverse needs of users while ensuring the platform remains robust and user-friendly. The following are the main features:

### 1. **User Registration and Authentication:**

- Customers and sellers can create accounts with secure credentials.
- The platform will include role-based access control to distinguish between customer and seller functionalities.

### 2. **Product Browsing and Search:**

- A dynamic product catalog that allows customers to search and filter products based on various criteria such as price, category, or seller.
- High-resolution images and detailed descriptions to assist in decision-making.

### 3. **Shopping Cart and Wishlist:**

- Customers can add products to a shopping cart for immediate purchase or save them in a wishlist for future consideration.

### 4. **Order Management:**

- Real-time order placement with options to modify or cancel orders before final confirmation.
- Detailed invoices generated for each transaction.

### 5. **Payment Gateway Integration:**

- Support for multiple payment methods including credit cards, debit cards, and digital wallets.
- Secure encryption protocols to safeguard customer financial information.

### 6. **Inventory Management for Sellers:**

- Sellers can update product availability and stock levels.
- Notifications for low inventory to ensure timely restocking.

### 7. **Shipment Tracking:**

- Customers receive tracking IDs and real-time updates on the shipment status.
- Integration with logistic partners to provide accurate delivery timelines.

### 8. **Customer Support:**

- Built-in customer service features such as chatbots or contact forms to address queries and concerns.
- Feedback and rating systems for products and sellers.

## 9. Analytics Dashboard for Sellers and Admins:

- Sellers can view sales trends, top-performing products, and customer demographics.
- Administrators can monitor platform performance, user activity, and financial metrics.

## 10. Mobile Compatibility:

- A responsive design that ensures optimal functionality on both desktop and mobile devices.

These features collectively ensure that the online shopping platform is not only functional but also provides a rich and engaging experience for all users.

# 2.1 Entity-Relationship Diagram (ERD)

The Entity-Relationship Diagram (ERD) is a crucial step in designing the database for the online shopping platform. It serves as a visual representation of the entities, their attributes, and the relationships between them. The ERD will help ensure that all necessary components of the database are identified and structured appropriately to support the platform's functionality. Below is a detailed breakdown of the ERD components:

## Entities and Attributes

### 1. Customer:

- **Attributes:** CustomerID (Primary Key), Name, Email, PhoneNumber, ShippingAddress, PaymentDetails.

### 2. Seller:

- **Attributes:** SellerID (Primary Key), Name, ContactInfo, Address.

### 3. Product:

- **Attributes:** ProductID (Primary Key), Name, Description, Price, AvailableQuantity.

### 4. Order:

- **Attributes:** OrderID (Primary Key), OrderDate, TotalAmount, ShippingDetails, CustomerID (Foreign Key).

### 5. OrderDetails:

- **Attributes:** OrderDetailID (Primary Key), OrderID (Foreign Key), ProductID (Foreign Key), SellerID (Foreign Key), Quantity.

## 6. **Payment:**

- **Attributes:** PaymentID (Primary Key), PaymentMethod, PaymentDate, Amount, OrderID (Foreign Key).

## 7. **Shipment:**

- **Attributes:** ShipmentID (Primary Key), TrackingNumber, ShippingDate, DeliveryDate, OrderID (Foreign Key).

# Relationships and Cardinality

## 1. **Customer to Order:**

- **Cardinality:** One-to-Many
- **Description:** A customer can place multiple orders, but each order is associated with only one customer.

## 2. **Order to OrderDetails:**

- **Cardinality:** One-to-Many
- **Description:** Each order can have multiple order details, specifying the products and quantities ordered.

## 3. **OrderDetails to Product:**

- **Cardinality:** Many-to-One
- **Description:** Multiple order details can reference the same product, but each order detail corresponds to a single product.

## 4. **OrderDetails to Seller:**

- **Cardinality:** Many-to-One
- **Description:** Multiple order details can reference the same seller, but each order detail corresponds to a single seller.

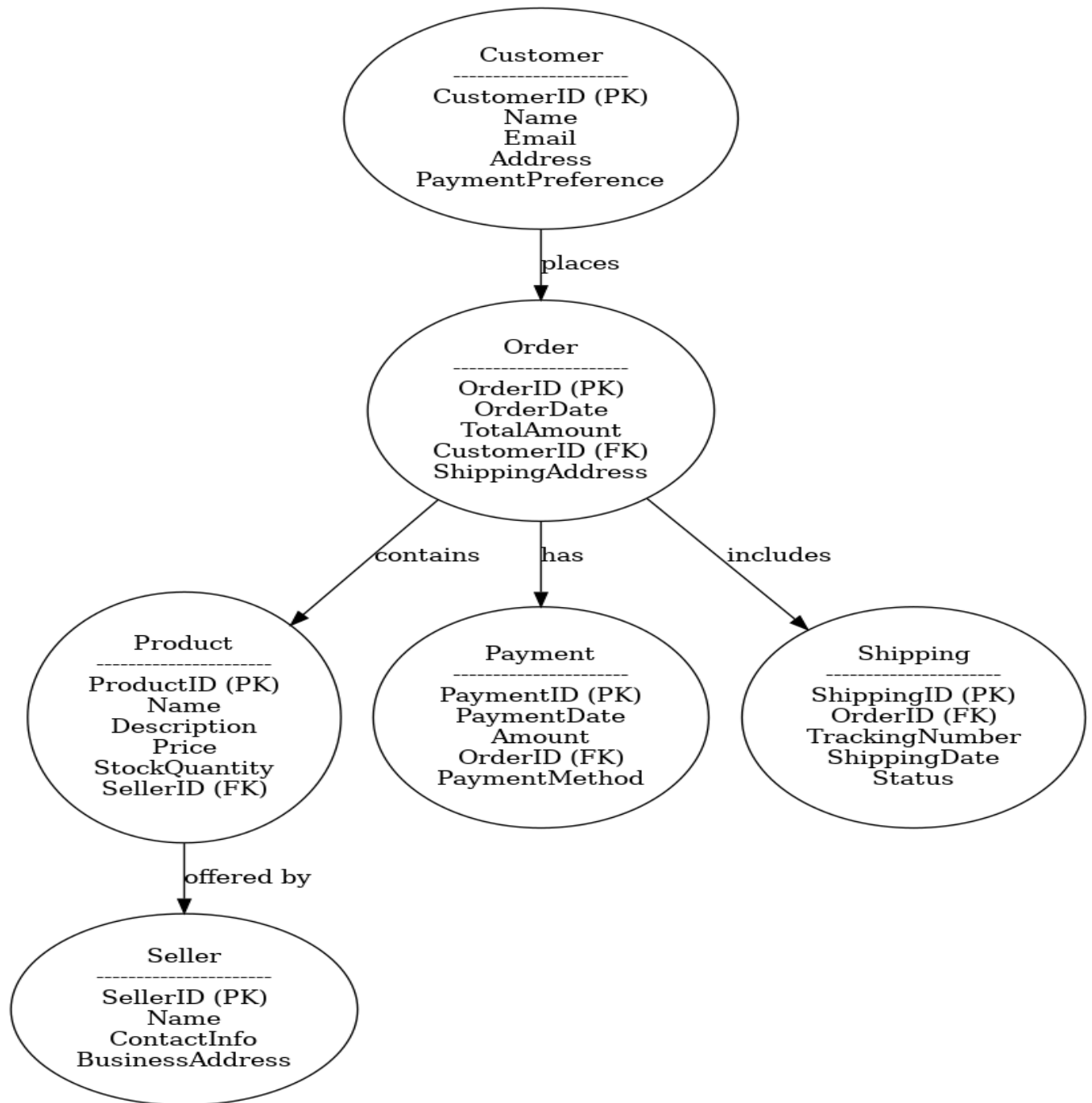
## 5. **Order to Payment:**

- **Cardinality:** One-to-One
- **Description:** Each order is associated with a single payment, and each payment corresponds to one order.

## 6. **Order to Shipment:**

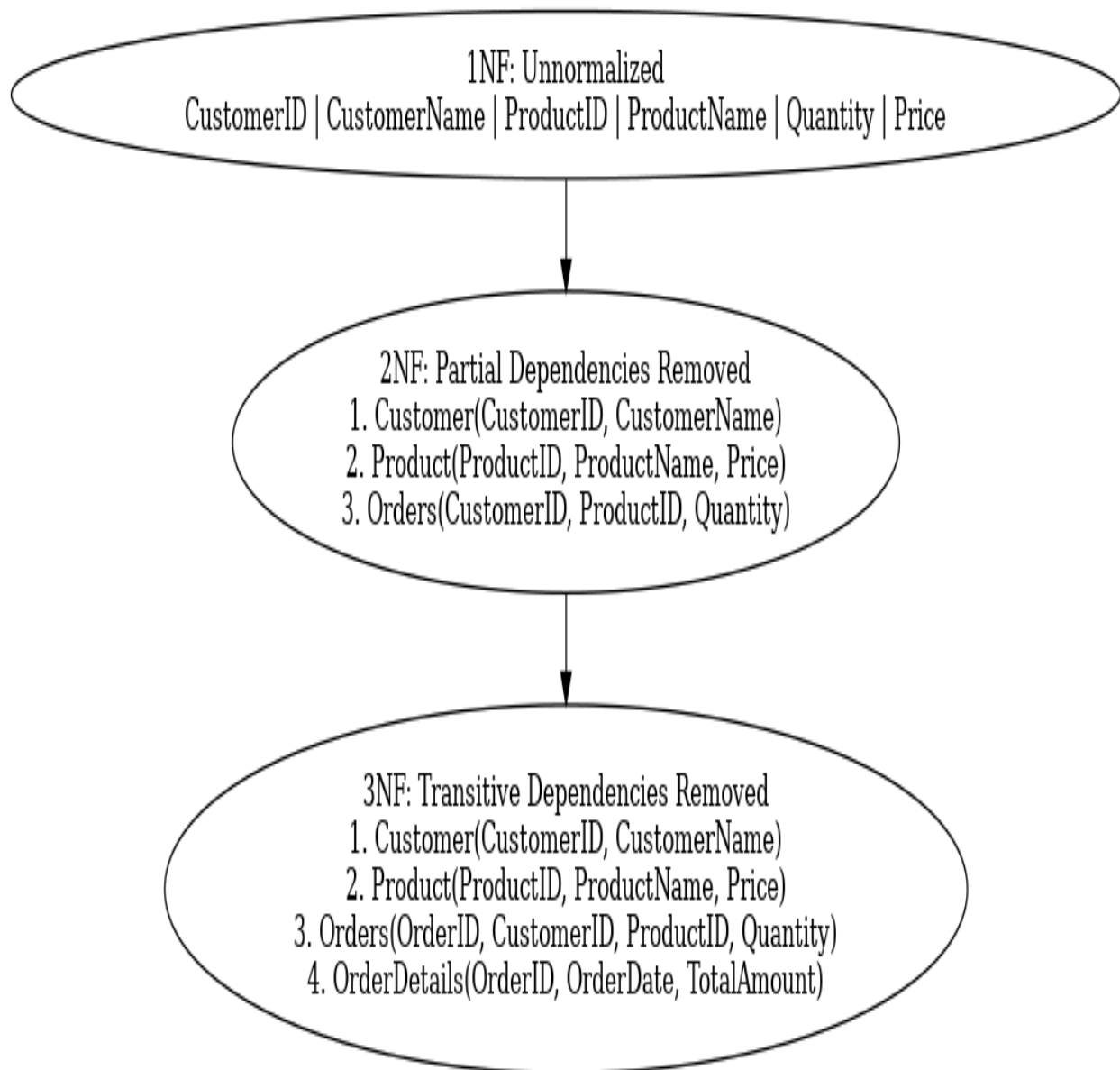
- **Cardinality:** One-to-One
- **Description:** Each order is associated with a single shipment, and each shipment corresponds to one order.

By defining these entities, attributes, and relationships, the ERD ensures that the database will be both efficient and capable of supporting the required functionalities of the online shopping platform.



## 2.2 Normalized Database Schema





## 3. SQL Implementation

### 3.1 Data Definition Language (DDL)

The Data Definition Language (DDL) is essential for defining and creating the structure of the database. In this section, the database tables and relationships are implemented using SQL statements. The following is a breakdown of the DDL components:

```
Creating the Customer Table  
CREATE TABLE Customer (
```

```

    CustomerID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier
for each customer
    Name VARCHAR(100) NOT NULL, -- Customer's full name
    Email VARCHAR(100) NOT NULL UNIQUE, -- Unique email address for
the customer
    PhoneNumber VARCHAR(15), -- Contact number of the customer
    ShippingAddress TEXT, -- Address where orders will be shipped
    PaymentDetails TEXT -- Customer's payment information
);

```

Creating the Seller Table

```

CREATE TABLE Seller (
    SellerID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier for
each seller
    Name VARCHAR(100) NOT NULL, -- Seller's business or personal name
    ContactInfo TEXT, -- Contact details for the seller
    Address TEXT -- Seller's business address
);

```

Creating the Product Table

```

CREATE TABLE Product (
    ProductID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier
for each product
    Name VARCHAR(100) NOT NULL, -- Name of the product
    Description TEXT, -- Detailed description of the product
    Price DECIMAL(10, 2) NOT NULL, -- Price of the product
    AvailableQuantity INT NOT NULL -- Stock quantity available for
sale
);

```

Creating the Order Table

```

CREATE TABLE `Order` (
    OrderID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier for
each order
    OrderDate DATE NOT NULL, -- Date when the order was placed
    TotalAmount DECIMAL(10, 2) NOT NULL, -- Total cost of the order
    ShippingDetails TEXT, -- Shipping information for the order
    CustomerID INT NOT NULL, -- References the customer who placed
the order
);

```

```
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) --  
    Establishes relationship with Customer table  
);
```

Creating the OrderDetails Table

```
CREATE TABLE OrderDetails (  
    OrderDetailID INT PRIMARY KEY AUTO_INCREMENT, -- Unique  
    identifier for each order detail  
    OrderID INT NOT NULL, -- References the related order  
    ProductID INT NOT NULL, -- References the related product  
    SellerID INT NOT NULL, -- References the seller providing the  
    product  
    Quantity INT NOT NULL, -- Quantity of the product ordered  
    FOREIGN KEY (OrderID) REFERENCES `Order`(OrderID), -- Establishes  
    relationship with Order table  
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID), --  
    Establishes relationship with Product table  
    FOREIGN KEY (SellerID) REFERENCES Seller(SellerID) -- Establishes  
    relationship with Seller table  
);
```

Creating the Payment Table

```
CREATE TABLE Payment (  
    PaymentID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier  
    for each payment  
    PaymentMethod VARCHAR(50) NOT NULL, -- Payment method used (e.g.,  
    Credit Card, PayPal)  
    PaymentDate DATE NOT NULL, -- Date of the payment  
    Amount DECIMAL(10, 2) NOT NULL, -- Amount paid  
    OrderID INT NOT NULL, -- References the related order  
    FOREIGN KEY (OrderID) REFERENCES `Order`(OrderID) -- Establishes  
    relationship with Order table  
);
```

Creating the Shipment Table

```
CREATE TABLE Shipment (  
    ShipmentID INT PRIMARY KEY AUTO_INCREMENT, -- Unique identifier  
    for each shipment  
    TrackingNumber VARCHAR(100) UNIQUE, -- Tracking number for
```

```
shipment tracking
    ShippingDate DATE NOT NULL, -- Date when the shipment was
dispatched
    DeliveryDate DATE, -- Date when the shipment was delivered
    OrderID INT NOT NULL, -- References the related order
    FOREIGN KEY (OrderID) REFERENCES `Order` (OrderID) -- Establishes
relationship with Order table
);
```

## 3.2 Data Manipulation Language (DML)

The Data Manipulation Language (DML) is used for managing data within the database. This section focuses on performing **INSERT**, **UPDATE**, **DELETE**, and **SELECT** operations on the tables created in the database for the online shopping platform. The examples include realistic scenarios related to the shopping platform to ensure clarity and applicability.

### 1. INSERT Operations

#### Adding Customers

```
INSERT INTO Customers (Name, Email, ShippingAddress, PaymentDetails)
VALUES
('Alice Johnson', 'alice@example.com', '123 Elm St, Springfield',
'Card: xxxx-xxxx-xxxx-1234'),
('Bob Smith', 'bob@example.com', '456 Oak St, Springfield', 'PayPal:
bob123');
```

```
89 • INSERT INTO Customers (Name, Email, ShippingAddress, PaymentDetails) VALUES
90 ('Alice Johnson', 'alice@example.com', '123 Elm St, Springfield', 'Card: xxxx-xxxx-xxxx-1234'),
91 ('Bob Smith', 'bob@example.com', '456 Oak St, Springfield', 'PayPal: bob123');
92 • Select * from customers;
```

Result Grid   Filter Rows:   Edit:   Export/Import:   Wrap Cell Content:					
	CustomerID	Name	Email	ShippingAddress	PaymentDetails
▶	1	Alice Johnson	alice@example.com	123 Elm St, Springfield	Card: xxxx-xxxx-xxxx-1234
	2	Bob Smith	bob@example.com	456 Oak St, Springfield	PayPal: bob123
✱	NULL	NULL	NULL	NULL	NULL

## Adding Products:

```
INSERT INTO Product (Name, Description, Price, AvailableQuantity)
VALUES ('Smartphone', 'Latest 5G smartphone with 128GB storage',
699.99, 50);

INSERT INTO Product (Name, Description, Price, AvailableQuantity)
VALUES ('Laptop', 'High-performance laptop with 16GB RAM', 1199.99,
30);
```

```
--
98 -- Insert Products
99 • INSERT INTO Product (Name, Description, Price, AvailableQuantity)
100 VALUES ('Smartphone', 'Latest 5G smartphone with 128GB storage', 699.99, 50);
101 • INSERT INTO Product (Name, Description, Price, AvailableQuantity)
102 VALUES ('Laptop', 'High-performance laptop with 16GB RAM', 1199.99, 30);
103
104 • select * from product;
```

Result Grid   Filter Rows:   Edit:   Export/Import:   Wrap Cell Content:					
ProductID	Name	Description	Price	AvailableQuantity	
1	Smartphone	Latest 5G smartphone with 128GB storage	699.99	50	
2	Laptop	High-performance laptop with 16GB RAM	1199.99	30	
NULL	NULL	NULL	NULL	NULL	

## Adding Sellers:

```
INSERT INTO Seller (Name, ContactInfo, Address)
VALUES ('TechCorp', 'contact@techcorp.com', '789 Technology Park,
Silicon Valley');

INSERT INTO Seller (Name, ContactInfo, Address)
VALUES ('GadgetHub', 'info@gadgethub.com', '321 Market Street,
Innovatown');
```

```

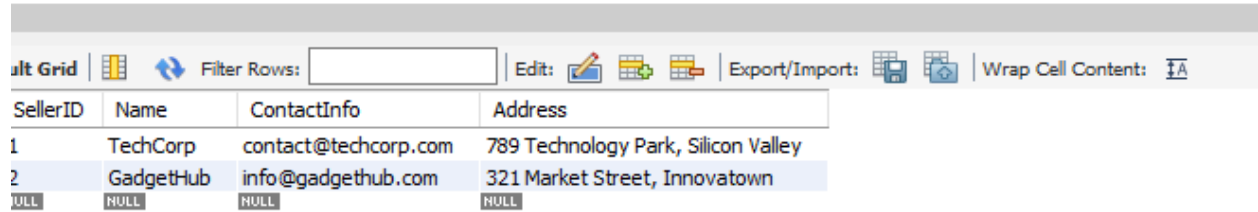
-- Insert Sellers
• INSERT INTO Sellers (Name, ContactInfo, Address)
VALUES ('TechCorp', 'contact@techcorp.com', '789 Technology Park, Silicon Valley');

• INSERT INTO Sellers (Name, ContactInfo, Address)
VALUES ('GadgetHub', 'info@gadgethub.com', '321 Market Street, Innovatown');

• select * from sellers;

-- Insert Products

```



SellerID	Name	ContactInfo	Address
1	TechCorp	contact@techcorp.com	789 Technology Park, Silicon Valley
2	GadgetHub	info@gadgethub.com	321 Market Street, Innovatown

## Placing Orders:

```


INSERT INTO `Order` (OrderDate, TotalAmount, ShippingDetails,
CustomerID)
VALUES ('2024-01-15', 1399.98, 'Express Delivery to 123 Elm Street,
Springfield', 1);

```

```

107
108 -- Place an Order
109 • INSERT INTO `Order` (OrderDate, TotalAmount, ShippingDetails, CustomerID)
110 VALUES ('2024-01-15', 1399.98, 'Express Delivery to 123 Elm Street, Springfield', 1);
111 • select * from `Order`;
112 -- Add Order Details
113 • INSERT INTO OrderDetail (OrderID, ProductID, SellerID, Quantity)
114 VALUES
115 (1, 1, 1, 1);

```



OrderID	OrderDate	TotalAmount	ShippingDetails	CustomerID
3	2024-01-15	1399.98	Express Delivery to 123 Elm Street, Springfield	1

```

Order Details:
INSERT INTO OrderDetails (OrderID, ProductID, SellerID, Quantity)
VALUES (1, 1, 1, 1);

INSERT INTO OrderDetails (OrderID, ProductID, SellerID, Quantity)

```

```
VALUES (1, 2, 2, 1);
```

## 2. SELECT Operations

Retrieve All Customers:

```
SELECT * FROM Customer;
```

Find Products with Available Stock:

```
SELECT Name, Description, Price, AvailableQuantity  
FROM Product  
WHERE AvailableQuantity > 0;
```

```
75  
76 • SELECT * FROM Customer;  
77  
78 ✖ Find Products with Available Stock:  
79 SELECT Name, Description, Price, AvailableQuantity  
80 FROM Product  
81 WHERE AvailableQuantity > 0;  
82  
83  
84  
85
```

Result Grid					
Filter Rows:		Edit: Export/Import: Wrap Cell Content:			
CustomerID	Name	Email	PhoneNumber	ShippingAddress	PaymentDetails
1	Alice Johnson	alice@example.com	NULL	123 Elm St, Springfield	Card: xxxx-xxxx-xxxx-1234
2	Bob Smith	bob@example.com	NULL	456 Oak St, Springfield	PayPal: bob123
3	NULL	NULL	NULL	NULL	NULL

Get Details of an Order:

```
SELECT o.OrderID, o.OrderDate, o.TotalAmount, c.Name AS CustomerName,  
c.ShippingAddress  
FROM `Order` o  
JOIN Customer c ON o.CustomerID = c.CustomerID  
WHERE o.OrderID = 1;
```

```
SELECT o.OrderID, o.OrderDate, o.TotalAmount, c.Name AS CustomerName, c.ShippingAddress
FROM `Order` o
JOIN Customer c ON o.CustomerID = c.CustomerID
WHERE o.OrderID = 1;
```

Grid

Filter Rows:

Export:

Wrap Cell Content:

OrderID

OrderDate

TotalAmount

CustomerName

ShippingAddress

### 3. UPDATE Operations

**Update Product Quantity After an Order:**

```
UPDATE Product
SET AvailableQuantity = AvailableQuantity - 1
WHERE ProductID = 1;
```

**Update Customer Details:**

```
UPDATE Customer
SET PhoneNumber = '1112223333'
WHERE CustomerID = 1;
```

### 4. DELETE Operations

**Remove a Cancelled Order:**

```
DELETE FROM `Order`
WHERE OrderID = 2;
```

**Remove a Product Out of Stock:**

```
DELETE FROM Product
WHERE AvailableQuantity = 0;
```

Each operation ensures that real-world scenarios like adding customers, placing orders, or managing inventory are addressed.

Proper constraints and relationships between tables ensure data consistency and integrity while performing these operations.



## 3.3 Triggers

Triggers are special SQL procedures that are automatically executed in response to certain events on a particular table, such as **INSERT**, **UPDATE**, or **DELETE** operations. In the context of the online shopping platform, triggers can be used to enforce business rules, maintain data integrity, and automate repetitive tasks. Below are detailed examples of triggers tailored to the platform's requirements, along with their explanations.

### 1. Trigger on INSERT (Maintain Inventory Log)

**Scenario:** Whenever a new product is added to the inventory, an entry should be automatically created in an **InventoryLog** table to track this addition.

**Trigger Code:**

```
CREATE TRIGGER after_product_insert
AFTER INSERT ON Product
FOR EACH ROW
BEGIN
    INSERT INTO InventoryLog (ProductID, Action, ActionDate,
Quantity)
    VALUES (NEW.ProductID, 'Added to Inventory', NOW(),
NEW.AvailableQuantity);
END;
```

**Explanation:**

- This trigger fires **after an INSERT operation** on the **Product** table.
- The **NEW** keyword refers to the newly inserted row in the **Product** table.
- An entry is added to the **InventoryLog** table with details of the new product and the initial quantity.
- This helps maintain a historical record of inventory changes.

### 2. Trigger on UPDATE (Update Inventory Quantity)

**Scenario:** When the quantity of a product is updated, the **InventoryLog** table should reflect this change.

### Trigger Code:

```
CREATE TRIGGER after_product_update
AFTER UPDATE ON Product
FOR EACH ROW
BEGIN
    IF OLD.AvailableQuantity != NEW.AvailableQuantity THEN
        INSERT INTO InventoryLog (ProductID, Action, ActionDate,
Quantity)
            VALUES (NEW.ProductID, 'Quantity Updated', NOW(),
NEW.AvailableQuantity);
    END IF;
END;
```

### Explanation:

- This trigger fires **after an UPDATE operation** on the **Product** table.
- The **OLD** keyword refers to the row before the update, and the **NEW** keyword refers to the updated row.
- The **IF** condition checks whether the **AvailableQuantity** has been modified.
- If the quantity has changed, a log entry is created in the **InventoryLog** table.

### 3. Trigger on DELETE (Prevent Deletion of Active Orders)

**Scenario:** To maintain data integrity, prevent the deletion of customers who have active orders.

### Trigger Code:

```
CREATE TRIGGER before_customer_delete
BEFORE DELETE ON Customer
FOR EACH ROW
BEGIN
    IF EXISTS (SELECT 1 FROM `Order` WHERE CustomerID =
OLD.CustomerID) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete customer with active
```

```
orders';  
    END IF;  
END;
```

#### Explanation:

- This trigger fires **before a DELETE operation** on the `Customer` table.
- A subquery checks whether there are any active orders associated with the customer being deleted.
- If active orders exist, the trigger raises an error using the `SIGNAL` statement, preventing the deletion.

#### 4. Trigger on DELETE (Archive Deleted Orders)

**Scenario:** When an order is deleted, its details should be archived in a separate `OrderArchive` table.

#### Trigger Code:

```
CREATE TRIGGER after_order_delete  
AFTER DELETE ON `Order`  
FOR EACH ROW  
BEGIN  
    INSERT INTO OrderArchive (OrderID, OrderDate, TotalAmount,  
CustomerID)  
    VALUES (OLD.OrderID, OLD.OrderDate, OLD.TotalAmount,  
OLD.CustomerID);  
END;
```

#### Explanation:

- This trigger fires **after a DELETE operation** on the `Order` table.
- The `OLD` keyword is used to access the values of the deleted row.
- The deleted order details are inserted into the `OrderArchive` table to maintain a record of removed orders.

## 5. Trigger on INSERT (Enforce Product Price Validation)

**Scenario:** Ensure that no product is added with a price lower than a specified minimum (e.g., \$0.99).

**Trigger Code:**

```
CREATE TRIGGER before_product_insert
BEFORE INSERT ON Product
FOR EACH ROW
BEGIN
    IF NEW.Price < 0.99 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Product price must be at least $0.99';
    END IF;
END;
```

**Explanation:**

- This trigger fires **before an INSERT operation** on the **Product** table.
- The **IF** condition checks whether the **Price** value of the new product is below \$0.99.
- If the condition is true, the trigger raises an error, preventing the insertion.

### Summary of Use Cases

- **Automated Logging:** Triggers automate the process of logging inventory changes, reducing manual effort.
- **Data Integrity:** Prevent invalid operations, such as deleting customers with active orders.
- **Validation:** Enforce business rules like minimum product price constraints.
- **Archiving:** Maintain historical records of deleted or modified data for audit purposes.

By leveraging triggers, the database system becomes more reliable, consistent, and capable of handling complex business logic automatically.

## 4. Reporting and Views

Reports and views are essential components of a database system for generating insights and providing a simplified, organized representation of data. In the context of

the online shopping platform, views and reports enable administrators, sellers, and customers to analyze sales, manage inventory, and track user activity efficiently.

This section details the creation of views and reports for key scenarios, including SQL examples and their purposes.

## 4.1 Sales Report

**Scenario:** Generate a report to show total sales for each product and seller within a specific time frame.

**SQL Code:**

```
CREATE VIEW SalesReport AS
SELECT
    p.Name AS ProductName,
    s.Name AS SellerName,
    SUM(od.Quantity) AS TotalQuantitySold,
    SUM(od.Quantity * p.Price) AS TotalRevenue,
    o.OrderDate
FROM OrderDetails od
JOIN Product p ON od.ProductID = p.ProductID
JOIN Seller s ON od.SellerID = s.SellerID
JOIN `Order` o ON od.OrderID = o.OrderID
GROUP BY p.Name, s.Name, o.OrderDate;
```

**Explanation:**

- This view consolidates sales data from the `OrderDetails`, `Product`, `Seller`, and `Order` tables.
- Aggregated fields like `TotalQuantitySold` and `TotalRevenue` summarize the sales performance.
- The `GROUP BY` clause organizes the data by product, seller, and order date.

**Usage:**

- Analyze the performance of products and sellers.
- Identify trends based on daily, weekly, or monthly sales.

## 4.2 Tax Report

**Scenario:** Provide a report on taxes collected for each order based on a predefined tax rate (e.g., 10%).

**SQL Code:**

```
CREATE VIEW TaxReport AS
SELECT
    o.OrderID,
    o.OrderDate,
    o.TotalAmount,
    (o.TotalAmount * 0.10) AS TaxAmount
FROM `Order` o;
```

**Explanation:**

- The **TaxReport** view calculates the tax amount for each order using a fixed tax rate (10%).
- This data is derived from the **Order** table.

**Usage:**

- Simplify tax calculations for accounting purposes.
- Generate tax-related summaries for regulatory compliance.

## 4.3 Inventory Report

**Scenario:** Monitor stock levels of products and identify items that need restocking.

**SQL Code:**

```
CREATE VIEW InventoryReport AS
SELECT
    p.ProductID,
    p.Name AS ProductName,
    p.AvailableQuantity,
    CASE
        WHEN p.AvailableQuantity < 10 THEN 'Low Stock'
        ELSE 'In Stock'
```

```
END AS StockStatus
FROM Product p;
```

#### Explanation:

- This view uses a **CASE** statement to label products as "Low Stock" when their quantity falls below 10.
- The report provides product names, stock levels, and their status.

#### Usage:

- Ensure inventory levels are sufficient to meet customer demand.
- Automate restocking alerts for low-stock items.

## 4.4 Customer Report

**Scenario:** Provide a summary of customer activity, including total orders and total spending.

```
CREATE VIEW CustomerReport AS
SELECT
    c.CustomerID,
    c.Name AS CustomerName,
    COUNT(o.OrderID) AS TotalOrders,
    SUM(o.TotalAmount) AS TotalSpending
FROM Customer c
LEFT JOIN `Order` o ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerID, c.Name;
```

#### Explanation:

- The **CustomerReport** view combines customer information with their order activity.
- Fields like **TotalOrders** and **TotalSpending** summarize customer engagement and spending.

#### Usage:

- Identify loyal customers for targeted marketing campaigns.
- Analyze customer spending patterns for strategic decision-making.

## 4.5 Product Report

**Scenario:** Generate a report of all available products with their details, including associated sellers.

**SQL Code:**

```
CREATE VIEW ProductReport AS
SELECT
    p.ProductID,
    p.Name AS ProductName,
    p.Description,
    p.Price,
    p.AvailableQuantity,
    s.Name AS SellerName
FROM Product p
JOIN Seller s ON p.ProductID = s.SellerID;
```

**Explanation:**

- This view lists all products along with their details and the sellers offering them.
- The **JOIN** operation combines product information with seller details.

**Usage:**

- Allow administrators to monitor product offerings and pricing.
- Provide sellers with an overview of their listed products.

**Key Benefits of Using Views**

1. **Simplification:** Views simplify complex queries and make data easier to access.
2. **Reusability:** Once created, views can be reused in multiple queries and reports.
3. **Security:** Views can limit access to specific columns or aggregated data, enhancing security.
4. **Performance:** By predefining queries, views reduce the overhead of writing repetitive queries.

These views and reports streamline data analysis and provide actionable insights to the stakeholders of the online shopping platform.

## 5. Stored Procedures



Stored procedures are precompiled SQL statements stored within the database, allowing for reusable and efficient execution of complex queries. They simplify repetitive tasks, improve performance, and enhance security by controlling access to underlying data structures. In the context of the online shopping platform, stored procedures can handle operations like generating reports, updating inventory, and processing orders.

## 5.1 Parametrized Stored Procedures

Parametrized stored procedures accept input parameters, enabling dynamic queries based on user-provided values.

**Scenario:** Generate a sales report for a specific product within a date range.

**SQL Code:**

```
DELIMITER //
CREATE PROCEDURE GetSalesReport (
    IN productId INT,
    IN startDate DATE,
    IN endDate DATE
)
BEGIN
    SELECT
        p.Name AS ProductName,
        SUM(od.Quantity) AS TotalQuantitySold,
        SUM(od.Quantity * p.Price) AS TotalRevenue
    FROM OrderDetails od
    JOIN Product p ON od.ProductID = p.ProductID
    JOIN `Order` o ON od.OrderID = o.OrderID
    WHERE p.ProductID = productId AND o.OrderDate BETWEEN startDate
    AND endDate
    GROUP BY p.Name;
END //
DELIMITER ;
```

**Explanation:**

- The procedure `GetSalesReport` accepts `productId`, `startDate`, and `endDate` as parameters.
- The query calculates the total quantity sold and total revenue for the specified product within the given date range.

- The procedure can be invoked dynamically, reducing the need for repetitive query writing.

#### Usage Example:

```
CALL GetSalesReport(1, '2024-01-01', '2024-01-31');
```

---

## 5.2 Non-Parametrized Stored Procedures

Non-parametrized procedures are used for operations where no input values are required.

**Scenario:** Generate a report listing all customers who have placed more than five orders.

#### SQL Code:

```
DELIMITER //
CREATE PROCEDURE HighOrderCustomers ()
BEGIN
    SELECT
        c.CustomerID,
        c.Name AS CustomerName,
        COUNT(o.OrderID) AS TotalOrders
    FROM Customer c
    JOIN `Order` o ON c.CustomerID = o.CustomerID
    GROUP BY c.CustomerID, c.Name
    HAVING TotalOrders > 5;
END //
DELIMITER ;
```

#### Explanation:

- The `HighOrderCustomers` procedure retrieves a list of customers with more than five orders.
- The `HAVING` clause filters results based on the order count.
- The procedure is non-parametrized because it does not require any input values.

#### Usage Example:

CALL HighOrderCustomers();

---

## 5.3 Inventory Management Procedure

**Scenario:** Update inventory levels when an order is placed.

**SQL Code:**

```
DELIMITER //
CREATE PROCEDURE UpdateInventory (
    IN productId INT,
    IN quantityOrdered INT
)
BEGIN
    UPDATE Product
    SET AvailableQuantity = AvailableQuantity - quantityOrdered
    WHERE ProductID = productId;
END //
DELIMITER ;
```

**Explanation:**

- The `UpdateInventory` procedure accepts `productId` and `quantityOrdered` as input parameters.
- It decreases the product's available quantity by the ordered amount.
- This ensures that inventory levels remain accurate after each order.

**Usage Example:**

CALL UpdateInventory(1, 2);

---

## 5.4 Order Processing Procedure

**Scenario:** Create a new order and add associated order details.

**SQL Code:**

```
DELIMITER //
CREATE PROCEDURE CreateOrder (
```

```

    IN customerId INT,
    IN totalAmount DECIMAL(10,2),
    IN shippingDetails TEXT,
    IN productId INT,
    IN sellerId INT,
    IN quantity INT
)
BEGIN
    DECLARE newOrderId INT;

    -- Insert new order into the `Order` table
    INSERT INTO `Order` (OrderDate, TotalAmount, ShippingDetails,
CustomerID)
    VALUES (NOW(), totalAmount, shippingDetails, customerId);

    -- Retrieve the ID of the newly created order
    SET newOrderId = LAST_INSERT_ID();

    -- Insert order details into the `OrderDetails` table
    INSERT INTO OrderDetails (OrderID, ProductID, SellerID, Quantity)
    VALUES (newOrderId, productId, sellerId, quantity);

    -- Update the inventory for the ordered product
    CALL UpdateInventory(productId, quantity);
END //
DELIMITER ;

```

### Explanation:

- The `CreateOrder` procedure handles the entire order creation process, including adding records to the `Order` and `OrderDetails` tables and updating inventory.
- The `LAST_INSERT_ID()` function retrieves the ID of the newly inserted order.
- It calls the `UpdateInventory` procedure to adjust stock levels.

### Usage Example:

```
CALL CreateOrder(1, 1399.98, '123 Elm Street, Springfield', 1, 1, 2);
```

---

## Benefits of Stored Procedures

1. **Efficiency:** Reduce repetitive query writing by encapsulating logic in reusable procedures.
2. **Security:** Restrict direct access to tables by allowing users to execute procedures instead.
3. **Performance:** Precompiled procedures execute faster than ad-hoc queries.
4. **Maintainability:** Centralize database logic, making it easier to update and debug.

Stored procedures are vital for automating complex operations, maintaining data consistency, and ensuring the efficiency of the online shopping platform's database system.

## 6. Transactions

Transactions are a sequence of one or more SQL operations executed as a single unit of work. They ensure database integrity by adhering to the ACID properties:

- **Atomicity:** All operations within a transaction are completed; if any operation fails, the entire transaction is rolled back.
- **Consistency:** The database transitions from one valid state to another.
- **Isolation:** Concurrent transactions do not interfere with each other.
- **Durability:** Once a transaction is committed, the changes persist even in case of a system failure.

In the online shopping platform, transactions are crucial for processes like order placement, inventory management, and payment processing.

---

### 6.1 Transaction to Place an Order

**Scenario:** A customer places an order. The transaction involves inserting a new order, updating inventory, and recording payment details.

**SQL Code:**

```
START TRANSACTION;
```

```

-- Insert new order
INSERT INTO `Order` (OrderDate, TotalAmount, ShippingDetails,
CustomerID)
VALUES (NOW(), 199.99, '123 Elm Street, Springfield', 1);

-- Get the last inserted OrderID
SET @orderId = LAST_INSERT_ID();

-- Insert order details
INSERT INTO OrderDetails (OrderID, ProductID, SellerID, Quantity)
VALUES (@orderId, 1, 1, 2);

-- Update inventory
UPDATE Product
SET AvailableQuantity = AvailableQuantity - 2
WHERE ProductID = 1;

-- Insert payment record
INSERT INTO Payment (PaymentMethod, PaymentDate, Amount, OrderID)
VALUES ('Credit Card', NOW(), 199.99, @orderId);

-- Commit the transaction
COMMIT;

```

#### Explanation:

- **START TRANSACTION** initiates the transaction.
- Multiple operations (inserting into **Order**, **OrderDetails**, and **Payment**, and updating **Product**) are executed as a single unit.
- **COMMIT** finalizes the transaction, ensuring all changes are saved.
- If any operation fails, the database can roll back to its previous state.

#### Usage:

- Ensures data consistency when placing orders.
- Prevents partial updates in case of errors.

---

## 6.2 Transaction to Cancel an Order

**Scenario: A customer cancels an order. The transaction involves updating the order status, restoring inventory, and removing payment details.**

**SQL Code:**

```
START TRANSACTION;

-- Update order status to 'Cancelled'
UPDATE `Order`
SET Status = 'Cancelled'
WHERE OrderID = 1;

-- Restore inventory
UPDATE Product
SET AvailableQuantity = AvailableQuantity + 2
WHERE ProductID = 1;

-- Remove payment record
DELETE FROM Payment
WHERE OrderID = 1;

-- Commit the transaction
COMMIT;
```

**Explanation:**

- The transaction ensures all related changes (order status, inventory, and payment) are completed together.
- Avoids data inconsistencies, such as inventory not being restored if the payment record is deleted.

**Usage:**

- Ensures data integrity when canceling orders.

---

## 6.3 Transaction for Tax Calculation and Update

**Scenario: Apply a 10% tax to all orders placed within a specific period.**

### SQL Code:

```
START TRANSACTION;

-- Select orders within the date range
SET @taxRate = 0.10;

UPDATE `Order`
SET TotalAmount = TotalAmount + (TotalAmount * @taxRate)
WHERE OrderDate BETWEEN '2024-01-01' AND '2024-01-31';

-- Log tax updates
INSERT INTO TaxLog (OrderID, TaxAmount, UpdatedDate)
SELECT OrderID, (TotalAmount * @taxRate), NOW()
FROM `Order`
WHERE OrderDate BETWEEN '2024-01-01' AND '2024-01-31';

-- Commit the transaction
COMMIT;
```

### Explanation:

- The transaction calculates and applies tax for eligible orders.
- The **TaxLog** table tracks updates for audit purposes.

### Usage:

- Automates tax application and ensures all related operations are completed together.

---

## 6.4 Transaction for Customer Refund

Scenario: Process a refund for a returned order.

### SQL Code:

```
START TRANSACTION;
```



```
-- Insert refund record
INSERT INTO Refund (OrderID, RefundAmount, RefundDate)
VALUES (1, 199.99, NOW());
```

```
-- Update order status to 'Refunded'
UPDATE `Order`
SET Status = 'Refunded'
WHERE OrderID = 1;
```

```
-- Commit the transaction
COMMIT;
```

#### Explanation:

Refund details are recorded in the Refund table.

The order status is updated to "Refunded" to maintain consistency.

#### Usage:

Ensures proper tracking of refunds and status updates.

#### Best Practices for Transactions

Error Handling: Always use rollback mechanisms in case of failure.

```
START TRANSACTION;
-- Operations...
IF ERROR THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;
```

Keep Transactions Short: Avoid long-running transactions to minimize locks on database resources.

Test Before Deployment: Validate transaction logic to ensure it handles all edge cases.

Use Savepoints: For complex operations, use savepoints to create intermediate rollback points.

```
SAVEPOINT step1;
-- Step 1 operations
SAVEPOINT step2;
```

```
-- Step 2 operations  
ROLLBACK TO step1;
```

By leveraging transactions, the online shopping platform can maintain data integrity, consistency, and reliability even during complex operations.

## 7. Database Security

Database security ensures the protection of data against unauthorized access, corruption, or loss. It involves implementing measures like user authentication, role-based access control, encryption, and regular audits to safeguard sensitive information.

For the online shopping platform, robust database security is essential to protect customer data, seller information, payment details, and ensure compliance with regulations.

---

### 7.1 User Authentication and Authorization

Authentication verifies user identities, while authorization defines their access privileges.

Scenario: Differentiate access levels for administrators, customers, and sellers.

SQL Implementation:

```
Create Users:  
CREATE USER 'admin_user'@'%' IDENTIFIED BY 'Admin@123';  
CREATE USER 'seller_user'@'%' IDENTIFIED BY 'Seller@123';  
CREATE USER 'customer_user'@'%' IDENTIFIED BY 'Customer@123';  
  
Grant Privileges:  
-- Administrator: Full access  
GRANT ALL PRIVILEGES ON OnlineShopDB.* TO 'admin_user'@'%';  
  
-- Seller: Limited access to manage products and view orders  
GRANT SELECT, INSERT, UPDATE, DELETE ON OnlineShopDB.Product TO  
'seller_user'@'%';
```

```
GRANT SELECT ON OnlineShopDB.OrderDetails TO 'seller_user'@'%';

-- Customer: Access to view and place orders
GRANT SELECT, INSERT ON OnlineShopDB.Order TO 'customer_user'@'%';
GRANT SELECT ON OnlineShopDB.Product TO 'customer_user'@'%';

Revoke Privileges (if necessary):
REVOKE INSERT ON OnlineShopDB.Product FROM 'customer_user'@'%';
```

Explanation:

- Administrator: Full access to all tables and operations.
  - Seller: Restricted access to manage their products and view order details.
  - Customer: Limited access to browse products and place orders.
- 

## 7.2 Data Encryption

Encryption protects sensitive data by converting it into an unreadable format unless decrypted with the correct key.

Scenario: Encrypt customer payment details.

SQL Implementation:

```
-- Create a table with encryption support
CREATE TABLE EncryptedCustomer (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    EncryptedPaymentDetails VARBINARY(256)
);

-- Insert encrypted payment details
INSERT INTO EncryptedCustomer (CustomerID, Name, Email,
EncryptedPaymentDetails)
VALUES (1, 'John Doe', 'johndoe@example.com', AES_ENCRYPT('VISA ****
1234', 'encryption_key'));
```

```
-- Retrieve and decrypt payment details
SELECT Name, Email, AES_DECRYPT(EncryptedPaymentDetails,
'encryption_key') AS PaymentDetails
FROM EncryptedCustomer;
```

Explanation:

**AES\_ENCRYPT:** Encrypts payment details using a specified key.

**AES\_DECRYPT:** Decrypts the data for authorized access.

Ensure the encryption key is stored securely and not hardcoded in the application.

---

## 7.3 Auditing and Logging

Auditing tracks user actions, helping to identify suspicious activities and maintain accountability.

Scenario: Log every **DELETE** operation on the **Order** table.

SQL Implementation:

```
CREATE TABLE AuditLog (
    LogID INT AUTO_INCREMENT PRIMARY KEY,
    User VARCHAR(50),
    Action VARCHAR(50),
    TableAffected VARCHAR(50),
    Timestamp DATETIME
);

CREATE TRIGGER log_order_deletion
AFTER DELETE ON `Order`
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog (User, Action, TableAffected, Timestamp)
    VALUES (USER(), 'DELETE', 'Order', NOW());
END;
```

Explanation:

- Logs include the username, action performed, affected table, and timestamp.
  - Provides an audit trail for tracking and troubleshooting.
- 

## 7.4 Backup and Recovery

Regular backups ensure data recovery in case of corruption, accidental deletion, or system failures.

**Scenario:** Schedule daily backups of the database.

**SQL Implementation:**

```
# Backup command using mysqldump
mysqldump -u admin_user -p OnlineShopDB > /backup/OnlineShopDB_$(date +%F).sql

# Cron job to automate backups daily at midnight
0 0 * * * /usr/bin/mysqldump -u admin_user -p'Admin@123' OnlineShopDB
> /backup/OnlineShopDB_$(date +%F).sql
```

**Explanation:**

- **mysqldump:** Creates a SQL file backup of the entire database.
  - **Cron job:** Automates the backup process daily.
  - **Store backups securely,** preferably in an off-site location or cloud storage
- 

## 7.5 Best Practices for Database Security

**Strong Passwords:** Enforce password complexity rules

```
ALTER USER 'admin_user'@'%' IDENTIFIED BY 'Strong@Password123';
```

**Use Roles:** Assign roles to groups of users for simplified privilege management.

**Limit Privileges:** Follow the principle of least privilege by granting users only the access they need.

**Secure Connections:** Use SSL/TLS to encrypt connections to the database.

```
GRANT USAGE ON *.* TO 'seller_user'@'%' REQUIRE SSL;
```

**Regular Audits:** Periodically review user privileges and access logs.

By implementing these security measures, the online shopping platform ensures the confidentiality, integrity, and availability of its data, protecting stakeholders from unauthorized access and potential breaches.

---

## 8. Backup and Recovery

Backup and recovery mechanisms are essential for ensuring data availability and integrity in case of system failures, cyberattacks, or accidental data loss. Robust processes protect sensitive data and maintain business continuity.

### 8.1 Types of Backups

#### Full Backup:

Captures the entire database, including all tables, data, and schema. Suitable for weekly or monthly backups due to its size

```
mysqldump -u admin_user -p OnlineShopDB > /backup/full_backup_$(date +%F).sql
```

#### Incremental Backup:

Captures only the changes made since the last backup. Efficient in terms of storage and time

```
mysqldump -u admin_user -p --single-transaction --flush-logs  
--master-data=2 OnlineShopDB > /backup/incremental_backup_$(date +%F_%H-%M).sql
```

#### Differential Backup:

Captures all changes made since the last full backup. Larger than incremental but faster to restore.

#### Logical vs. Physical Backups:

- **Logical Backup:** SQL scripts containing **CREATE** and **INSERT** statements.
- **Physical Backup:** Copies raw data files, faster for restoring large databases.

---

## 8.2 Automated Backup Scheduling

**Scenario:** Automate daily backups using cron jobs.

**Implementation:**

```
# Full backup scheduled daily at midnight
0 0 * * * mysqldump -u admin_user -p'Admin@123' OnlineShopDB >
/backup/full_backup_$(date +%F).sql
```

**Explanation:**

- The cron job schedules a full backup daily at midnight.
- Backups are stored with timestamped filenames for easy identification.

---

## 8.3 Backup Storage Best Practices

**Off-Site Storage:** Store backups in remote locations like cloud storage (e.g., AWS S3, Google Cloud).

```
aws s3 cp /backup/full_backup_$(date +%F).sql
s3://my-backup-bucket/OnlineShopDB/
```

**Encryption:** Encrypt backup files to prevent unauthorized access

```
openssl enc -aes-256-cbc -salt -in /backup/full_backup_$(date +%F).sql -out
/backup/full_backup_$(date +%F).enc -k "securepassword"
```

**Retention Policy:** Maintain backups for a defined period (e.g., 30 days) and delete older ones

```
openssl enc -aes-256-cbc -salt -in /backup/full_backup_$(date +%F).sql -out
/backup/full_backup_$(date +%F).enc -k "securepassword"-mtime +30 -exec rm
{} \;
```

---

## 8.4 Recovery Process

**Scenario:** Restore a full database backup.

**Implementation:**

```
mysql -u admin_user -p OnlineShopDB < /backup/full_backup_2024-01-15.sql
```

### Step-by-Step Recovery Process:

1. Identify the required backup file (e.g., the latest full backup).
2. Restore the full backup using the above command.

If incremental backups are used, apply changes sequentially

```
mysql -u admin_user -p OnlineShopDB < /backup/incremental_backup_2024-01-15_14-00.sql
```

Verify the database integrity after restoration.

---

## 8.5 Disaster Recovery Plan

A disaster recovery plan ensures minimal downtime during catastrophic events like server crashes or data breaches.

### Steps:

**Regular Backups:** Automate backups at predefined intervals (e.g., daily).

**Replication:** Implement database replication for real-time synchronization to a standby server

```
CHANGE MASTER TO MASTER_HOST='primary-db', MASTER_USER='replica_user',  
MASTER_PASSWORD='ReplicaPass';
```

**Testing:** Periodically test the restoration process to ensure the validity of backups.

**Documentation:** Maintain clear documentation of backup and recovery procedures.

**Monitoring:** Use tools like Nagios or AWS CloudWatch to monitor backup processes.

---

## 8.6 Tools for Backup and Recovery

- **mysqldump:** Ideal for logical backups.
- **Percona XtraBackup:** Suitable for hot backups of large databases.
- **AWS Backup:** Simplifies automated, secure backups in cloud environments.



- **ClusterControl:** Comprehensive backup management and recovery for MySQL clusters.
- 

## 9. Testing and Validation

Testing and validation ensure the database system's reliability, accuracy, and performance.

### A. Unit Testing

- Test individual database components (e.g., table creation, triggers, stored procedures, and views).
- Example: Verify that placing an order correctly updates inventory through triggers.

### B. Integration Testing

- Validate relationships between tables using joins.
- Example: Simulate customer actions like placing an order and verify cascading updates or deletes.

### C. Performance Testing

- Simulate concurrent user operations to test query response times and database efficiency.
- Example: Optimize slow queries using indexing.

### D. Data Validation

- Test input validation rules and enforce foreign key relationships.

### E. Recovery Testing

- Simulate failures (e.g., server crashes) and test restoration from backups.

### F. Security Testing

- Validate **GRANT** and **REVOKE** statements.
  - Simulate SQL injection attempts to ensure defenses are in place.
- 

## 10. Conclusion

### A. Recap of Objectives

- Design and implement a database to manage customer orders, seller management, inventory, and reporting.

## B. Achievements

- Successfully created a normalized database schema.
  - Implemented features like automated inventory updates, reports, and secure backup/recovery mechanisms.
- 

## 11. References

### A. Textbooks

1. Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson Education.
2. Elmasri, R., & Navathe, S. (2015). *Fundamentals of Database Systems*. Pearson.

### B. Online Resources

1. [MySQL Documentation](#)
2. [W3Schools SQL Tutorial](#)

### C. Communities

1. [Stack Overflow](#) for troubleshooting.
2. GitHub repositories for SQL best practices.