# 10. Comprehensive Database Design, Optimization, and Advanced Features

## Objective

## Design a normalized database schema for a complex business scenario (e.g., an eCommerce platform) and implement advanced SQL features to ensure performance, data integrity, and automation.

## Queries

### 1. Schema Design

* I created 4 tables (customers, products, orders and orderdetails).
* Inserted values into each table for the testing.

1. **CREATE TABLE Customers (**
2. **CustomerID INT AUTO\_INCREMENT PRIMARY KEY,**
3. **FirstName VARCHAR(50) NOT NULL,**
4. **LastName VARCHAR(50) NOT NULL,**
5. **Email VARCHAR(100) UNIQUE NOT NULL,**
6. **Phone VARCHAR(15)**
7. **);**
8. **INSERT INTO Customers (FirstName, LastName, Email, Phone)**
9. **VALUES**
10. **('Nithish', 'Kumar', 'nithish@mail.com', '9876543210'),**
11. **('Sara', 'Ali', 'sara.ali@mail.com', '9998887776'),**
12. **('Ravi', 'Sharma', 'ravi.sharma@mail.com', '8887776665');**

* In **Customers** table, I had set few columns into the table which is related to customers alone.
* Customer\_id, firstname, lastname, email and phone of the each customers.

1. **CREATE TABLE Products (**
2. **ProductID INT AUTO\_INCREMENT PRIMARY KEY,**
3. **ProductName VARCHAR(100) NOT NULL,**
4. **Category VARCHAR(50),**
5. **Price DECIMAL(10,2) NOT NULL,**
6. **Stock INT DEFAULT 0**
7. **);**
8. **INSERT INTO Products (ProductName, Category, Price, Stock)**
9. **VALUES**
10. **('Wireless Mouse', 'Electronics', 599.00, 100),**
11. **('Denim Jacket', 'Clothing', 1999.00, 50),**
12. **('Sports Shoes', 'Accessories', 2499.00, 60),**
13. **('Laptop Backpack', 'Accessories', 1299.00, 75);**

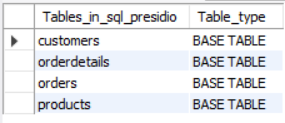
* In **Products** table, I created few columns like product\_id, name, category, unit\_price and stock of each product available in the inventory.

1. **CREATE TABLE Orders (**
2. **OrderID INT AUTO\_INCREMENT PRIMARY KEY,**
3. **CustomerID INT,**
4. **OrderDate DATETIME DEFAULT CURRENT\_TIMESTAMP,**
5. **Status VARCHAR(50) DEFAULT 'Pending',**
6. **TotalAmount DECIMAL(10,2),**
7. **FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)**
8. **);**
9. **INSERT INTO Orders (CustomerID, Status, TotalAmount)**
10. **VALUES**
11. **(1, 'Shipped', 3098.00),**
12. **(2, 'Pending', 599.00);**

* In **Orders** table, I created few columns like order\_id, customer\_id, orderdate, status and total amount of each order placed by the customer.
* Here, customer\_id is used as a foreign key to join with the **customers** table.

1. **CREATE TABLE OrderDetails (**
2. **OrderDetailID INT AUTO\_INCREMENT PRIMARY KEY,**
3. **OrderID INT,**
4. **ProductID INT,**
5. **Quantity INT NOT NULL,**
6. **UnitPrice DECIMAL(10,2) NOT NULL,**
7. **FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),**
8. **FOREIGN KEY (ProductID) REFERENCES Products(ProductID)**
9. **);**
10. **INSERT INTO OrderDetails (OrderID, ProductID, Quantity, UnitPrice)**
11. **VALUES**
12. **(1, 1, 1, 599.00),**
13. **(1, 3, 1, 2499.00),**
14. **(2, 1, 1, 599.00);**

* In **orderDetails** table, Created few columns such as orderDetail\_id, order\_id, product\_id, quantity and unit price of each orders.
* It’s more related with **orders** table where details of each order will be in this table.
* And order\_id and product\_id are used as a foreign key to join with **orders** and **products** table.



### 2. Indexing and Performance

* Indexing in SQL is used to speed up data retrieval from large tables.
* An index works like a pointer, allowing the database to quickly locate rows without scanning every record.
* It is especially useful in columns used for filtering (WHERE), joining (JOIN), and sorting (ORDER BY).
* Primary and foreign keys are automatically indexed, but custom indexes improve performance in other queries.

1. Customers table

**CREATE INDEX idx\_customers\_email ON Customers(Email);**

* + - Email is unique and frequently used in filtering.

1. Products table

**CREATE INDEX idx\_products\_name ON Products(ProductName);**

**CREATE INDEX idx\_products\_category ON Products(Category);**

* Useful for filtering and searching for products

1. Orders table

**CREATE INDEX idx\_orders\_customer ON Orders(CustomerID);**

**CREATE INDEX idx\_orders\_date ON Orders(OrderDate);**

* CustomerID is used in joins with Customers table
* OrderDate is used to order the rows by it

1. OrderDetails table

**CREATE INDEX idx\_orderdetails\_orderid ON OrderDetails(OrderID);**

**CREATE INDEX idx\_orderdetails\_productid ON OrderDetails(ProductID);**

* These foreign keys are used for JOINs with Orders and Products

### 3. Triggers

* A trigger is a database function that runs automatically in response to specific events (INSERT, UPDATE, DELETE) on a table.
* In this project, I used a trigger to maintain inventory accuracy by reducing product stock whenever an order is placed.

1. **DELIMITER $$**
2. **CREATE TRIGGER trg\_update\_stock**
3. **AFTER INSERT ON OrderDetails**
4. **FOR EACH ROW**
5. **BEGIN**
6. **UPDATE Products**
7. **SET Stock = Stock - NEW.Quantity**
8. **WHERE ProductID = NEW.ProductID;**
9. **END$$**
10. **DELIMITER ;**

* Automatically updates product stock **without writing manual queries**.
* Maintains **real-time inventory** accuracy after every order.

### 4. Transactions

* A **transaction** in SQL is a block of operations that are executed as a single unit.
* It ensures **atomicity**, meaning either **all operations succeed** or **none are applied**.
* Transactions are critical in **multi-step processes** like placing an order, where multiple tables are involved.

1. **START TRANSACTION;**
2. **--fetching data in orders table**
3. **INSERT INTO Orders (CustomerID, Status, TotalAmount)**
4. **VALUES (2, 'Pending', 3197.00);**
5. **--last\_insert\_id will give the auto increasing value of the last inserted value**
6. **SET @LastOrderID = LAST\_INSERT\_ID();**
7. **INSERT INTO OrderDetails (OrderID, ProductID, Quantity, UnitPrice)**
8. **VALUES**
9. **(@LastOrderID, 1, 1, 599.00),**
10. **(@LastOrderID, 4, 2, 1299.00);**
11. **--Commit the transaction if everything is successful**
12. **COMMIT;**
13. **--Rollback if some insertion or updating goes wrong**
14. **ROLLBACK;**

* Ensures **data consistency** - if any step fails, the order is not saved.
* Works together with the trigger to **update stock only if order is valid**.

### 5. Views

* **A view is a virtual table based on a SQL SELECT query.**
* **It helps simplify complex joins, aggregations, and makes querying more efficient.**
* **Views enhance readability, support reporting, and allow you to reuse logic easily.**
* **I created two views for this task,**
  + **Customer’s order summary**

1. **CREATE VIEW CustomerOrderSummary AS**
2. **SELECT**
3. **c.CustomerID,**
4. **CONCAT(c.FirstName, ' ', c.LastName) AS CustomerName,**
5. **COUNT(o.OrderID) AS TotalOrders,**
6. **SUM(o.TotalAmount) AS TotalSpent**
7. **FROM Customers c**
8. **LEFT JOIN Orders o ON c.CustomerID = o.CustomerID**
9. **GROUP BY c.CustomerID;** 
   * + **Top sold product**
10. **CREATE VIEW TopSellingProducts AS**
11. **SELECT**
12. **p.ProductID,**
13. **p.ProductName,**
14. **SUM(od.Quantity) AS TotalSold,**
15. **SUM(od.Quantity \* od.UnitPrice) AS TotalRevenue**
16. **FROM Products p**
17. **LEFT JOIN OrderDetails od ON p.ProductID = od.ProductID**
18. **GROUP BY p.ProductID, p.ProductName**
19. **ORDER BY TotalSold DESC;**

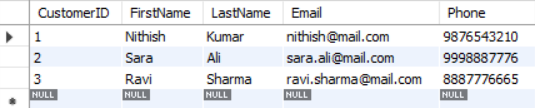
CustomerOrderSummary - Shows each customer’s total number of orders and total amount spent.

TopSellingProducts - Lists products by quantity sold and total revenue.

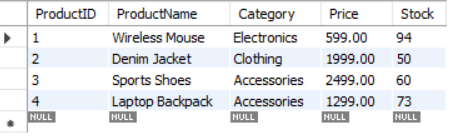
### 6. Testing

**Showing Each tables with data in it,**

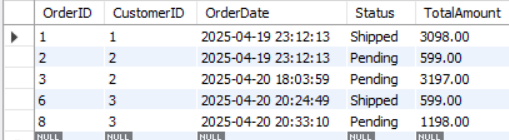
Customers table,



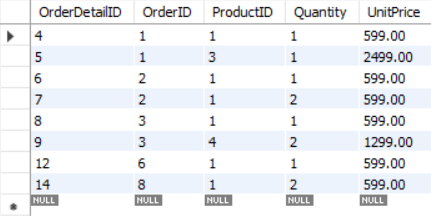
Products table,



Orders table,

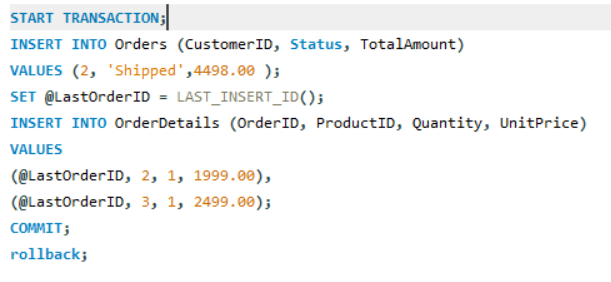


OrderDetails table,

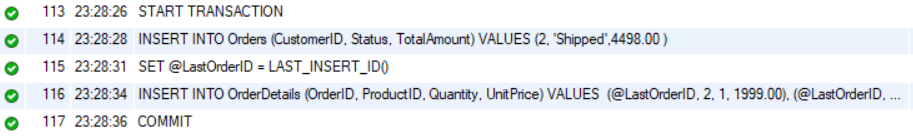


**Transaction,**

* Started the below transaction,

****

* This Transaction would insert row in **order** and **orderDetails** table.
* And update the **product** table which triggered after row inserted in the **orderDetails** table.
* The final stage was depending on the user selection of commit and rollback.

****

* The transaction committed So changes are shown below,
  + Changes in **orders** table,



* Changes in **orderDetails** table,

****

* Changes in **product** table and it is also the testing of **triggers**,

****

* The testing of transaction was successfully done.

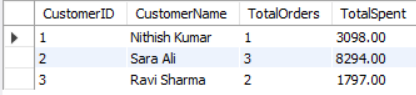
**Views,**

* There are two views available,

1. **Customer’s order summary**

**Query - > select \* from CustomerOrderSummary;**

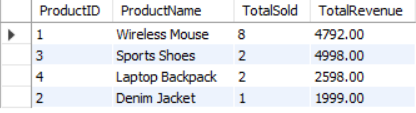
****

****

1. Top sold product

Query - > **select \* from TopSellingProducts;**

****

****