

A
PROJECT in
Database Management Systems
Based on
E-Commerce Inventory & Order
Management System



By:

Parth Arora

CSE-A

02717702723

Submitted to: Prof. Vidushi

Abstract

This project presents a database system designed for an e-commerce platform that manages product inventory, customer data, and order processing. It demonstrates the practical application of ER modeling, normalization, relational schema design, SQL queries, triggers, views, and procedures. The system efficiently handles core operations such as product inventory management, customer data storage, and order processing.

At the heart of this project is a structured relational database, designed using the **Entity-Relationship (ER) model**, and later transformed into a relational schema. The **enhanced ER (EER)** features like **specialization/generalization** and **aggregation** were applied to optimize relationships between entities such as *Products*, *Customers*, *Orders*, and *Vendors*.

The project adheres to standard normalization techniques. It moves from **1NF to 5NF** and includes **Boyce-Codd Normal Form (BCNF)** and **Domain-Key Normal Form (DKNF)** to ensure **minimal redundancy**, **data integrity**, and **lossless decomposition**. **Functional dependencies**, **multivalued dependencies**, and **join dependencies** were thoroughly analyzed.

The DBMS architecture followed a **three-tier model**, ensuring **data independence** and modularity. SQL was used extensively for database definition (DDL), manipulation (DML), and control (DCL). Various **constraints** such as **primary keys**, **foreign keys**, **NOT NULL**, and **CHECK** were implemented for data validation.

The system leverages SQL features like:

- **Joins (INNER, LEFT, RIGHT, FULL)**
- **Nested and correlated subqueries**
- **Aggregate and built-in functions**
- **GROUP BY, HAVING, and ORDER BY** clauses
- **Views and Indexes** for optimized access
- **Stored procedures and triggers** to automate key processes like order confirmation and stock updates

This project incorporates essential concepts of **transactions**, including **ACID properties**, **transaction states**, and **scheduling**. It uses **locking mechanisms** and **timestamp ordering** to ensure **serializability** and **concurrency control**, with provisions for **deadlock detection and recovery**. Commands like **COMMIT**, **ROLLBACK**, and **SAVEPOINT** are implemented for transaction control.

Database **backup** strategies and **recovery mechanisms** from **catastrophic failures** are outlined, ensuring data integrity. The system simulates a **recoverable schedule**, ensuring no data loss in case of failures.

Efficient file structures like **heap files**, **sorted files**, and **hashing** are explored. Indexing through **B-tree** and **B+ tree** is used to enhance query performance. The system supports both **single-level** and **multi-level indexing**.

The project briefly explores concepts from **Object-Oriented DBMS** and **Distributed DBMS**, providing scalability and modular data access, reflecting real-world distributed e-commerce platforms.

Problem Statement & Objectives

As online shopping becomes more popular, e-commerce businesses need to manage a large amount of data about products, customers, and orders. Using traditional methods like spreadsheets or manual tracking can cause problems, such as mistakes in data, duplicate entries, and delays in processing orders. These issues can slow down the business and lead to unhappy customers.

To fix these problems, an efficient and well-organized **Database Management System (DBMS)** is needed. A DBMS can help businesses keep their data accurate, reduce errors, and make sure everything works smoothly. This project aims to create a system that can manage e-commerce data in a reliable and efficient way.

Objectives:

- Design an Entity-Relationship (ER) model and convert it into a normalized relational schema.
- Implement all tables up to 5NF using SQL, preserving data integrity and minimizing redundancy.
- Demonstrate advanced SQL features like stored procedures, triggers, views, and indexing.
- Simulate real-world operations such as customer ordering and stock updates.
- Apply transaction concepts including commit, rollback, concurrency, and recovery mechanisms.

System Architecture

The system is designed with a modular three-tier architecture comprising the following layers:

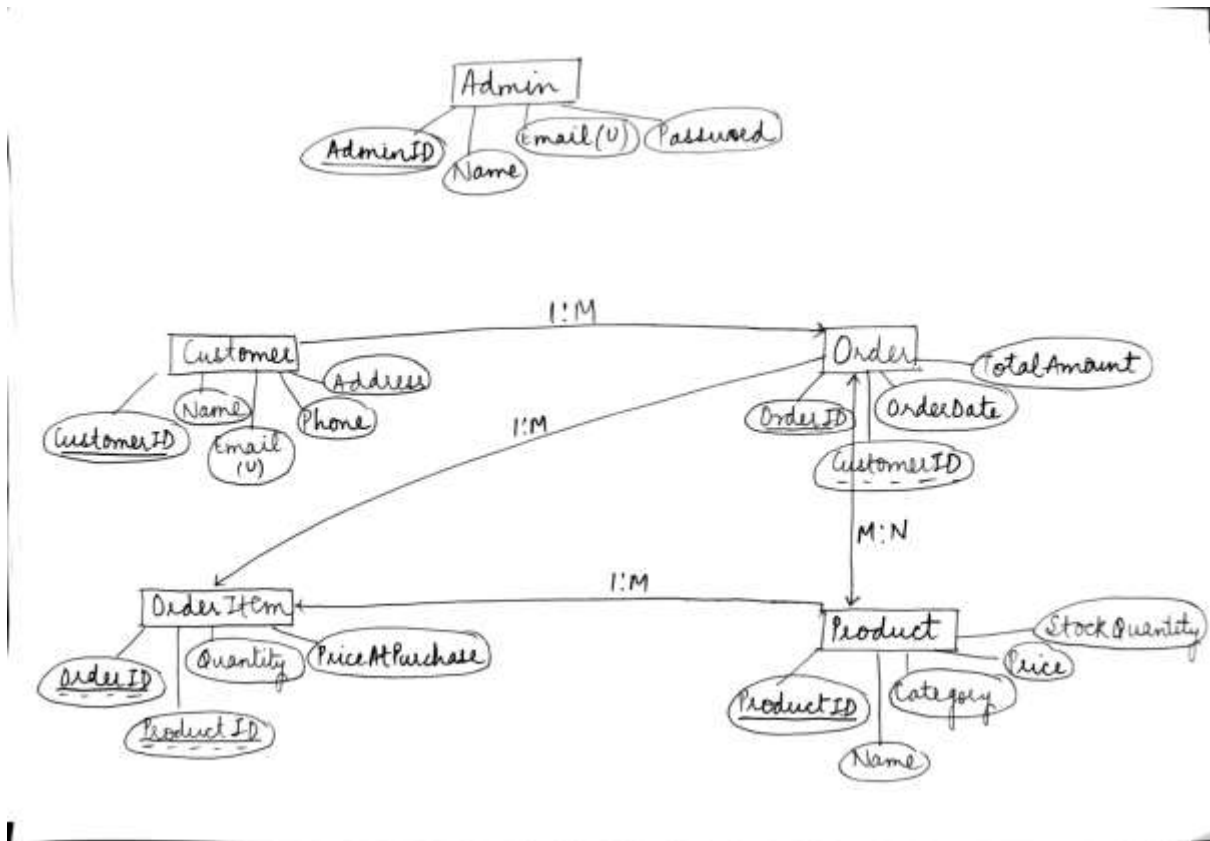
- **Presentation Layer** (SQL client interface like MySQL Workbench)
- **Logic Layer** (Stored procedures, triggers, SQL scripts)
- **Data Layer** (MySQL database with relational tables)

Modules:

- **Customer Management:** Handle customer registration and updates.
- **Product & Inventory Management:** Manage stock, product categories, and pricing.
- **Order Placement:** Customers place orders which are processed via stored procedures.
- **Order-Item Mapping:** Junction table for many-to-many mapping between orders and products.
- **Admin Dashboard:** Views and procedures for managing the system backend.

These modules interact via SQL operations and ensure database integrity and usability.

ER Diagram



Relational Schema

The relational schema for the e-commerce platform consists of five primary tables: **Customer**, **Product**, **OrderTable**, **OrderItem**, and **Admin**. Below is the structure for each table:

1. Customer Table:

Column Name	Data Type	Constraints
CustomerID	INT	PRIMARY KEY
Name	VARCHAR(100)	NOT NULL
Email	VARCHAR(100)	UNIQUE, NOT NULL
Phone	VARCHAR(15)	NOT NULL
Address	VARCHAR(255)	

2. Product Table:

Column Name	Data Type	Constraints
ProductID	INT	PRIMARY KEY
Name	VARCHAR(100)	NOT NULL
Category	VARCHAR(50)	
Price	DECIMAL(10, 2)	NOT NULL
StockQuantity	INT	NOT NULL

3. Order Table:

Column Name	Data Type	Constraints
OrderID	INT	PRIMARY KEY
CustomerID	INT	FOREIGN KEY REFERENCES Customer(CustomerID)
OrderDate	DATE	NOT NULL
TotalAmount	DECIMAL(10, 2)	NOT NULL

4. OrderItem Table (Junction Table):

Column Name	Data Type	Constraints
OrderID	INT	FOREIGN KEY REFERENCES OrderTable(OrderID)
ProductID	INT	FOREIGN KEY REFERENCES Product(ProductID)
Quantity	INT	NOT NULL
PriceAtPurchase	DECIMAL(10, 2)	NOT NULL
PRIMARY KEY	(OrderID, ProductID)	

5. Admin Table:

Column Name	Data Type	Constraints
AdminID	INT	PRIMARY KEY
Name	VARCHAR(100)	NOT NULL
Email	VARCHAR(100)	UNIQUE, NOT NULL
Password	VARCHAR(100)	NOT NULL

Relationships:

- 🔗 **Customer** places **multiple Orders** (One-to-Many Relationship).
- 🔗 **OrderTable** contains **multiple Products** through the **OrderItem** table (Many-to-Many Relationship).
- 🔗 **Admin** manages all system data (One-to-Many Relationship).

Normalization

All tables are fully normalized following these steps:

- **1NF:** Ensured atomicity of all fields (e.g., no multi-valued fields).
- **2NF:** Removed partial dependencies (e.g., breaking composite keys).
- **3NF:** Removed transitive dependencies (e.g., separating address data).
- **BCNF:** Ensured every determinant is a candidate key.
- **4NF:** Addressed multi-valued dependencies using separate tables (e.g., OrderItem).
- **5NF:** Decomposed join dependencies logically and removed redundancy.
- **DKNF:** Enforced domain and key constraints through CHECK and UNIQUE.

This normalization structure ensures data integrity, consistency, and minimal redundancy.

SQL Features Used

SQL was used to create and manipulate the database with the following features:

- ✚ **DDL:** CREATE TABLE with constraints (PRIMARY KEY, FOREIGN KEY, NOT NULL, CHECK).
- ✚ **DML:** INSERT, UPDATE, DELETE for data operations.
- ✚ **Joins:** INNER, LEFT, RIGHT JOINS for fetching combined data.
- ✚ **Subqueries:** Used in views and stored procedures.
- ✚ **Aggregate Functions:** SUM, AVG, COUNT used in reports.
- ✚ **Views:** For admin reporting on customer orders.
- ✚ **Triggers:** Automatically update stock when an order is placed.
- ✚ **Stored Procedures:** Handle order placement, price calculations, and inventory updates.
- ✚ **Indexes:** B-tree indexing applied for faster searches.

Implementation

```
-- database in use
use parth;

-- CUSTOMER TABLE
CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    Phone VARCHAR(15),
    Address TEXT
);

-- PRODUCT TABLE
CREATE TABLE Product (
    ProductID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Category VARCHAR(50),
    Price DECIMAL(10,2) CHECK (Price > 0),
    StockQuantity INT CHECK (StockQuantity >= 0)
);

-- ORDER TABLE
CREATE TABLE OrderTable (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE NOT NULL,
    TotalAmount DECIMAL(10,2),
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);

-- ORDER ITEM TABLE (Many-to-Many junction)
CREATE TABLE OrderItem (
    OrderItemID INT PRIMARY KEY,
    OrderID INT,
    ProductID INT,
    Quantity INT CHECK (Quantity > 0),
    PriceAtPurchase DECIMAL(10,2),
    FOREIGN KEY (OrderID) REFERENCES OrderTable(OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
```

```

-- ADMIN TABLE
CREATE TABLE Admin (
    AdminID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100) UNIQUE NOT NULL,
    Password VARCHAR(100) NOT NULL
);

-- CUSTOMERS
INSERT INTO Customer VALUES (1, 'Alice', 'alice@example.com', '9876543210', 'Delhi');
INSERT INTO Customer VALUES (2, 'Bob', 'bob@example.com', '9876541234', 'Mumbai');

-- PRODUCTS
INSERT INTO Product VALUES (101, 'Laptop', 'Electronics', 55000.00, 10);
INSERT INTO Product VALUES (102, 'Phone', 'Electronics', 25000.00, 20);

-- ADMIN
INSERT INTO Admin VALUES (1, 'Admin User', 'admin@example.com', 'admin123');

-- ORDER
INSERT INTO OrderTable VALUES (201, 1, '2025-05-06', 80000.00);

-- ORDER ITEMS
INSERT INTO OrderItem VALUES (301, 201, 101, 1, 55000.00);
INSERT INTO OrderItem VALUES (302, 201, 102, 1, 25000.00);

-- VIEW for Admin
CREATE VIEW CustomerOrders AS
SELECT c.Name AS CustomerName, o.OrderID, o.OrderDate, o.TotalAmount
FROM Customer c
JOIN OrderTable o ON c.CustomerID = o.CustomerID;

-- TRIGGER to update stock after order
DELIMITER //
CREATE TRIGGER trg_update_stock AFTER INSERT ON OrderItem
FOR EACH ROW
BEGIN
    UPDATE Product
    SET StockQuantity = StockQuantity - NEW.Quantity
    WHERE ProductID = NEW.ProductID;
END;
//

```

```

DELIMITER ;

-- STORED PROCEDURE to insert order
DELIMITER //
CREATE PROCEDURE AddOrder(
    IN p_orderID INT,
    IN p_customerID INT,
    IN p_orderDate DATE,
    IN p_total DECIMAL(10,2),
    IN p_productID INT,
    IN p_quantity INT,
    IN p_price DECIMAL(10,2)
)
BEGIN
    INSERT INTO OrderTable VALUES (p_orderID, p_customerID, p_orderDate, p_total);
    INSERT INTO OrderItem (OrderItemID, OrderID, ProductID, Quantity, PriceAtPurchase)
    VALUES (FLOOR(RAND()*1000), p_orderID, p_productID, p_quantity, p_price);
END;
//
DELIMITER ;

--Auto-calculate TotalAmount
DELIMITER //
CREATE TRIGGER trg_calculate_total BEFORE INSERT ON OrderItem
FOR EACH ROW
BEGIN
    DECLARE total DECIMAL(10,2);
    SET total = NEW.Quantity * NEW.PriceAtPurchase;
    UPDATE OrderTable
    SET TotalAmount = COALESCE(TotalAmount, 0) + total
    WHERE OrderID = NEW.OrderID;
END;
//
DELIMITER ;

-- Add LoyaltyPoints to Customers
ALTER TABLE Customer ADD LoyaltyPoints INT DEFAULT 0;

DELIMITER //
CREATE TRIGGER trg_loyalty_points AFTER INSERT ON OrderItem
FOR EACH ROW
BEGIN

```

```

DECLARE earnedPoints INT;
SET earnedPoints = NEW.Quantity * 10;
UPDATE Customer
SET LoyaltyPoints = LoyaltyPoints + earnedPoints
WHERE CustomerID = (SELECT CustomerID FROM OrderTable WHERE OrderID =
NEW.OrderID);
END;
//
DELIMITER ;

-- View to List Low Stock Products
CREATE VIEW LowStockProducts AS
SELECT ProductID, Name, StockQuantity
FROM Product
WHERE StockQuantity < 5;

-- Add IsDeleted column to Orders instead of deleting rows
ALTER TABLE OrderTable ADD IsDeleted BOOLEAN DEFAULT FALSE;

-- Procedure to update order status
ALTER TABLE OrderTable ADD Status VARCHAR(20) DEFAULT 'Pending';
DELIMITER //
CREATE PROCEDURE UpdateOrderStatus(IN oid INT, IN new_status VARCHAR(20))
BEGIN
    UPDATE OrderTable SET Status = new_status WHERE OrderID = oid;
END;
//
DELIMITER ;

-- Total sales per product
CREATE VIEW ProductSales AS
SELECT
    p.Name,
    SUM(oi.Quantity) AS TotalSold,
    SUM(oi.PriceAtPurchase * oi.Quantity) AS RevenueGenerated
FROM OrderItem oi
JOIN Product p ON oi.ProductID = p.ProductID
GROUP BY p.ProductID, p.Name;

-- essential commands to show output for
SELECT * FROM Customer;
SELECT * FROM Product;

```

```

SELECT * FROM OrderTable;
SELECT * FROM OrderItem;
SELECT * FROM Admin;

-- View showing customer orders
SELECT * FROM CustomerOrders;

-- View showing low stock products
SELECT * FROM LowStockProducts;

-- View showing product-wise sales
SELECT * FROM ProductSales;

-- To show the stock update, loyalty points, and order total triggers working:
-- Insert a new order (insert into OrderTable first)
INSERT INTO OrderTable VALUES (202, 2, '2025-05-07', 0.00, 'Pending', FALSE);

-- Now insert OrderItems (triggers will fire)
INSERT INTO OrderItem VALUES (303, 202, 101, 2, 55000.00); -- Laptop
INSERT INTO OrderItem VALUES (304, 202, 102, 1, 25000.00); -- Phone

-- Now check updated stock
SELECT * FROM Product;

-- Check total amount updated
SELECT * FROM OrderTable WHERE OrderID = 202;

-- Check loyalty points added
SELECT * FROM Customer WHERE CustomerID = 2;

-- Stored procedure output
CALL AddOrder(203, 1, '2025-05-07', 40000.00, 102, 2, 20000.00); -- Phone x2

-- Check the newly inserted order and order items
SELECT * FROM OrderTable WHERE OrderID = 203;
SELECT * FROM OrderItem WHERE OrderID = 203;

-- Order Status Update Procedure
CALL UpdateOrderStatus(203, 'Shipped');

-- Check updated status
SELECT OrderID, Status FROM OrderTable WHERE OrderID = 203;

```

Output

164 • `SELECT * FROM Customer;`

Result Grid						
		Filter Rows:		Edit:		Export/Import:
	CustomerID	Name	Email	Phone	Address	LoyaltyPoints
▶	1	Alice	alice@example.com	9876543210	Delhi	0
	2	Bob	bob@example.com	9876541234	Mumbai	0
•	NULL	NULL	NULL	NULL	NULL	NULL

165 • `SELECT * FROM Product;`

166 • `SELECT * FROM OrderTable;`

Result Grid					
		Filter Rows:		Edit:	
	ProductID	Name	Category	Price	StockQuantity
▶	101	Laptop	Electronics	55000.00	10
	102	Phone	Electronics	25000.00	20
•	NULL	NULL	NULL	NULL	NULL

166 • `SELECT * FROM OrderTable;`

167 • `SELECT * FROM OrderItem;`

168 • `SELECT * FROM Admin;`

169

170 -- View showing customer orders

Result Grid						
		Filter Rows:		Edit:		Export
	OrderID	CustomerID	OrderDate	TotalAmount	IsDeleted	Status
▶	201	1	2025-05-06	80000.00	0	Pending
•	NULL	NULL	NULL	NULL	NULL	NULL

167 • `SELECT * FROM OrderItem;`

168 • `SELECT * FROM Admin;`

169

170 -- View showing customer orders

Result Grid					
		Filter Rows:		Edit:	
	OrderItemID	OrderID	ProductID	Quantity	PriceAtPurchase
▶	301	201	101	1	55000.00
	302	201	102	1	25000.00
•	NULL	NULL	NULL	NULL	NULL

168 • `SELECT * FROM Admin;`

169

170 -- View showing customer orders

Result Grid			
		Filter Rows:	
		Edit:	
	AdminID	Name	Email
▶	1	Admin User	admin@example.com
•	NULL	NULL	NULL

```

171 • SELECT * FROM CustomerOrders;
172
173 -- View showing low stock products

```

Result Grid				
	CustomerName	OrderID	OrderDate	TotalAmount
▶	Alice	201	2025-05-06	80000.00

```

174 • SELECT * FROM LowStockProducts;
175

```

Result Grid		
	ProductID	Name
		StockQuantity

```

177 • SELECT * FROM ProductSales;
178

```

Result Grid			
	Name	TotalSold	RevenueGenerated
▶	Laptop	1	55000.00
	Phone	1	25000.00

```

183 -- Now insert OrderItems (triggers will fire)
184 • INSERT INTO OrderItem VALUES (303, 202, 101, 2, 55000.00); -- Laptop
185 • INSERT INTO OrderItem VALUES (304, 202, 102, 1, 25000.00); -- Phone
186
187 -- Now check updated stock

```

Result Grid			
	Name	TotalSold	RevenueGenerated
▶	Laptop	1	55000.00
	Phone	1	25000.00

ProductSales 24 x

Output

Action Output

#	Time	Action	Message
✓ 34	23:46:41	SELECT * FROM ProductSales LIMIT 0, 1000	2 row(s) returned
✗ 35	23:46:48	INSERT INTO OrderTable VALUES (202, 2, '2025-05-07', 0.00, 'Pending', FALSE)	Error Code: 1366.
✗ 36	23:50:25	INSERT INTO OrderItem VALUES (304, 202, 102, 1, 25000.00)	Error Code: 1452.

187 -- Now check updated stock

188 • `SELECT * FROM Product;`

189

190 -- Check total amount updated




191 • `SELECT * FROM OrderTable WHERE OrderID = 202;`

Result Grid					
Filter Rows: <input type="text"/>					
Edit: 					
ProductID	Name	Category	Price	StockQuantity	
101	Laptop	Electronics	55000.00	10	
102	Phone	Electronics	25000.00	20	
NULL	NULL	NULL	NULL	NULL	

191 • `SELECT * FROM OrderTable WHERE OrderID = 202;`

192

193 -- Check loyalty points added




Result Grid						
Filter Rows: <input type="text"/>						
Edit:    Export						
OrderID	CustomerID	OrderDate	TotalAmount	IsDeleted	Status	
NULL	NULL	NULL	NULL	NULL	NULL	

193 -- Check loyalty points added



194 • `SELECT * FROM Customer WHERE CustomerID = 2;`

195

196 -- Stored procedure output

Result Grid						
Filter Rows: <input type="text"/>						
Edit:    Export/Import						
CustomerID	Name	Email	Phone	Address	LoyaltyPoints	
2	Bob	bob@example.com	9876541234	Mumbai	0	
NULL	NULL	NULL	NULL	NULL	NULL	





200 • `SELECT * FROM OrderTable WHERE OrderID = 203;`

Result Grid						
Filter Rows: <input type="text"/>						
Edit:    Export						
OrderID	CustomerID	OrderDate	TotalAmount	IsDeleted	Status	
NULL	NULL	NULL	NULL	NULL	NULL	

201 • `SELECT * FROM OrderItem WHERE OrderID = 203;`

Result Grid					
Filter Rows: <input type="text"/>					
Edit:    					
OrderItemID	OrderID	ProductID	Quantity	PriceAtPurchase	
NULL	NULL	NULL	NULL	NULL	

207 • `SELECT OrderID, Status FROM OrderTable WHERE OrderID = 203;`

Result Grid		
Filter Rows: <input type="text"/>		
Edit:    Export/Import: 		
OrderID	Status	
NULL	NULL	

Conclusion

The *E-Commerce Inventory & Order Management System* project helped me apply and understand the core concepts of Database Management Systems (DBMS) in a practical and real-world scenario. It focuses on how e-commerce platforms manage product inventories, customer orders, and administrative tasks using well-structured databases.

In this project, I designed normalized tables using relational database concepts and implemented primary keys, foreign keys, and constraints to ensure data accuracy and consistency. Each table—such as *Customer*, *Product*, *OrderTable*, *OrderItem*, and *Admin*—is organized to avoid redundancy and maintain data integrity. I followed the normalization process up to 3NF to ensure the database remains efficient and easy to update.

I also implemented SQL features like **views**, **stored procedures**, and **triggers**. Views were used to simplify data access for tasks like displaying customer orders or checking low stock products. Triggers helped automate operations like updating stock after a purchase, calculating total order amounts, and assigning loyalty points to customers. The use of stored procedures made it easier to handle complex operations like placing new orders or updating the status of an existing order.

The project also touches on transaction-related concepts. For example, I used default values, constraints, and logic to ensure data consistency when multiple changes occur at once. The addition of fields like *IsDeleted* and *Status* to orders shows how soft deletion and tracking order progress can be handled efficiently.

Overall, this project gave me hands-on experience in designing, managing, and optimizing a real-world database system. It reflects how DBMS concepts like data modeling, normalization, SQL operations, and automation play a vital role in building modern systems. It also sets the foundation for expanding this system in the future with features like discount handling, multi-vendor support, or even analytics.