**Comprehensive Database Design, Optimization, and Advanced Features**

***Objective*** :

The objective of this task is to design a normalized database schema for a complex business scenario, such as an eCommerce platform, ensuring data integrity, performance, and automation. This involves creating multiple related tables with proper primary and foreign keys, unique constraints, and normalization principles.

To enhance query performance, indexing strategies should be applied to frequently accessed data. Triggers should be implemented to automate business rules, such as updating inventory upon order placement or logging changes to data. Transactions must be used to ensure consistency during multi-step operations like order processing.

Additionally, views can be created to simplify complex queries, and materialized views may be used if supported to cache expensive query results. Finally, the schema and SQL scripts should be well-documented, and test queries should be written for testing.

***Architecture and Queries* *:***

We are going to create 5 tables : **Customers, Products, Orders, Order Details** and **Inventory.**

***1) Schemas :***

**create table Customers (**

**CustomerID int primary key,**

**CustomerName varchar(50) not null,**

**Email varchar(100) unique not null,**

**PhoneNumber varchar(15),**

**Address varchar(255)**

**);**

// Schema for customers table

**create table Products (**

**ProductID int primary key,**

**ProductName varchar(255) not null,**

**Description text,**

**Price decimal(12, 2) NOT NULL,**

**Category varchar(100)**

**);**

// Schema for products table

**create table Orders (**

**OrderID int primary key,**

**CustomerID int not null,**

**OrderDate timestamp default current\_timestamp,**

**foreign key (CustomerID) references Customers(CustomerID)**

**on delete cascade**

**);**

// Schema for orders table

**create table OrderDetails (**

**OrderDetailID int primary key,**

**OrderID int not null,**

**ProductID int not null,**

**Quantity int not null,**

**Price decimal(12, 2) not null,**

**Total decimal(12, 2) as (Quantity \* Price) stored,**

**foreign key (OrderID) references Orders(OrderID)**

**on delete cascade,**

**foreign key (ProductID) references Products(ProductID)**

**on delete cascade**

**);**

// Schema for order details table

**create table Inventory (**

**ProductID int primary key,**

**StockLevel int not null,**

**LastUpdated timestamp default current\_timestamp on update current\_timestamp,**

**foreign key (ProductID) references Products(ProductID)**

**on delete cascade**

**);**

// Schema for inventory table

***2) Indexing :***

Indexing in SQL improves query performance by creating a data structure that allows faster retrieval of rows. It works like a book index, enabling quick lookups instead of scanning the entire table.

*Indexing strategy employed :*

* Indexing email column in Customers table for faster customer lookups.
* Indexing email and customer id column in Customers table for faster customer lookups.
* Indexing product\_id in order details column for product based queries.

**create index idx\_email on Customers(Email);**

**create index idx\_orders\_customer\_date on Orders(CustomerID, OrderDate);**

**create index idx\_orderdetails\_product on OrderDetails(ProductID);**

***3) Triggers :***

A trigger is a special type of stored procedure that automatically executes in response to certain events on a table. Triggers help maintain data integrity, enforce business rules, and automate database tasks. Triggers cannot be manually executed; they run automatically.

*Types of triggers :*

1. **BEFORE** - Executes before the event occurs.
2. **AFTER** - Executes after the event occurs.

**delimiter $$**

**create trigger update\_inventory\_after\_order**

**after insert on OrderDetails for each row**

**begin**

**declare current\_stock int;**

-- Fetch the current stock of the product

**select StockLevel into current\_stock from Inventory where ProductID = new.ProductID;**

-- Update stock level after the order

**update Inventory**

**set StockLevel = current\_stock - new.Quantity**

**where ProductID = new.ProductID;**

-- Check for negative stock

**if current\_stock - new.Quantity < 0 THEN**

**signal sqlstate '45000' SET message\_text = 'Insufficient stock';**

**end if;**

**end$$**

**delimiter ;**

// The trigger updates the stock level in the **Inventory** table after a new order is inserted into **OrderDetails**. It first retrieves the current stock for the ordered product, then deducts the ordered quantity. If the stock level goes negative, it raises an error using a **signal** statement with the message "Insufficient stock", preventing the order from being processed if there isn’t enough inventory.

***4) Transactions :***

A transaction ensures that all database updates in a process (e.g., creating an order) happen successfully or roll back entirely in case of errors.

**begin;**

-- Insert a new order

**insert into Orders (OrderID, CustomerID)**

**values (1, 3);**

-- Insert order details for the order

**insert into OrderDetails (OrderID, ProductID, Quantity, Price)**

**values (1, 5, 2, 350.00);**

-- Commit the transaction if everything is successful

**commit;**

In case of an error during the transaction (e.g., insufficient inventory), we use roll back:

**rollback;**

***5) Views :***

A view in SQL is a virtual table based on the result of a SQL query. It does not store data itself but provides a way to simplify complex queries, enhance security, and improve readability.

**create view OrderSummary AS**

**select o.OrderID, o.CustomerID, o.OrderDate,**

**sum(od.Quantity \* od.Price) AS TotalAmount**

**from Orders o join OrderDetails od**

**on o.OrderID = od.OrderID group by o.OrderID;**

// The above view aggregates order details by calculating the total amount for each order using a join between Orders and OrderDetails. It groups data by OrderID, displaying order details along with the total price computed as the sum of (Quantity \* Price).

***6) Testing :***

*Inserting values into Customers table*

**insert into customers values**

**(1,"John","john@gmail.com","9876543210","142 Maplewood Drive, Springfield, IL 62704, USA"),**

**(2,"Jake","jake@gmail.com","9003612012","89 Elm Street, Boston, MA 02108, USA"),**

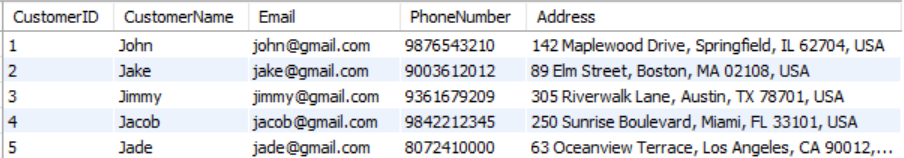
**(3,"Jimmy","jimmy@gmail.com","9361679209","305 Riverwalk Lane, Austin, TX 78701, USA"),**

**(4,"Jacob","jacob@gmail.com","9842212345","250 Sunrise Boulevard, Miami, FL 33101, USA"),**

**(5,"Jade","jade@gmail.com","8072410000","63 Oceanview Terrace, Los Angeles, CA 90012, USA");**

**select \* from Customers;**

// Displaying Customers table



*Inserting values into Products table*

**insert into Products values**

**(1,"Iphone 15","Description for Iphone 15",100000.00,"Electronics"),**

**(2,"Facewash","Description for Facewash",500.00,"Self-care"),**

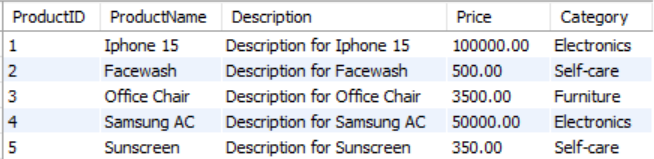
**(3,"Office Chair","Description for Office Chair",3500.00,"Furniture"),**

**(4,"Samsung AC","Description for Samsung AC",50000.00,"Electronics"),**

**(5,"Sunscreen","Description for Sunscreen",350.00,"Self-care");**

**select \* from Products;**

// Display Products table



*Inserting values into Inventory table*

**insert into Inventory (ProductID, StockLevel) values**

**(1,10),**

**(2,5),**

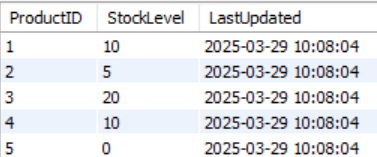
**(3,20),**

**(4,10),**

**(5,0);**

**select \* from Inventory;**

// Display Inventory table



***Transaction*** *: Placing an order*

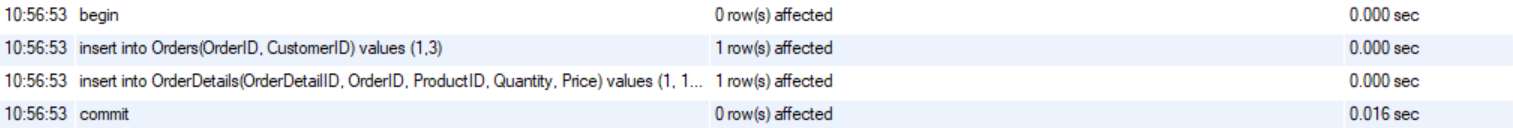
**begin;**

**insert into Orders(OrderID, CustomerID) values (1,3);**

**insert into OrderDetails(OrderDetailID, OrderID, ProductID, Quantity, Price) values (1, 1, 3, 2, 3500.00);**

**commit;**

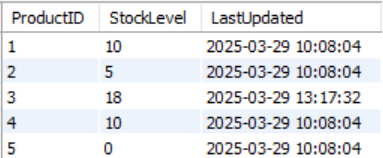
// Inserted values into Orders and OrderDetails table.



The above transaction was **successful** and we had committed it.

Check whether the trigger is executed after the value is inserted into the Orders table.

**select \* from Orders;**



Since the number of items placed is **2** for the **ProductID 3** in the order, the inventory’s **StockLevel** is updated as **18** (20 - 2) and the trigger **update\_inventory\_after\_order** is working properly.

*// Placing the same order again with the same values :*

**begin;**

**insert into Orders(OrderID, CustomerID) values (1,3);**

**rollback;**



The above transaction was a failure as we inserted a duplicate value for **OrderID** which is a **primary key** and hence we use **rollback** to the last commit state.

***Transaction :*** *Test insufficient stock*

Try placing an out of stock order. In the inventory table, ProductID 5 is out of stock. Let's check placing that order.

**begin**

**insert into Orders (OrderID, CustomerID) values (4, 2);**

**insert into OrderDetails (OrderDetailID, OrderID, ProductID, Quantity, Price)**

**values (4, 4, 5, 2, 350.00);**



Since the product is out of stock we get the error as **“Insufficient stock”**. Hence we need to delete the values inserted into the Orders table.

**delete from Orders where OrderID=4;**

**commit;**

// Deleted the order in which the product was out of stock and the **commit** statement is used.

***Views :*** *Testing views*

Let’s check the created view **OrderSummary.**

Before that let’s insert some more values into the Orders and OrderDetails table.

**insert into Orders (OrderID, CustomerID) values (2, 1);**

**insert into OrderDetails (OrderDetailID, OrderID, ProductID, Quantity, Price)**

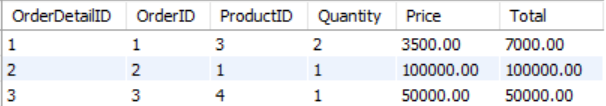
**values (2, 2, 1, 1, 100000.00);**

**insert into Orders (OrderID, CustomerID) values (3, 4);**

**insert into OrderDetails (OrderDetailID, OrderID, ProductID, Quantity, Price)**

**values (3, 3, 4, 1, 50000.00);**

**select \* from OrderSummary;**

****

The View **OrderSummary** is working fine.