**TOPICS:** Advanced Transaction Management in MySQL | **Allowed Time:** 90 Minutes
**Instructions:** | **Total Marks:** 30

1. Gossips are not allowed.
2. Teacher assistants are for your help, so be nice with them. Respect them as they are teaching you. Raise your hands if you have some problem and need help from TA. Avoid calling them by raising your voice and disturbing the environment of Lab.
3. TA may deduct your marks for any kind of ill-discipline or misconduct from your side.
4. Evaluation will be considered final and you cannot debate for the marks. So, focus on performing the tasks when the time is given to you.
5. Paste the query as well as result table screenshot as a result of each task

## Task 01: (25 Marks)

**Part 1**: **Database and Table Setup**

Define and create the following tables in your database:

### 1. Customers Table

CustomerID (Primary Key, INT)
CustomerName (VARCHAR(100))
Email (VARCHAR(100))

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(100),
    Email VARCHAR(100)
);
```

### 2. Products Table

ProductID (Primary Key, INT)
ProductName (VARCHAR(100))
Stock (INT, must be non-negative)
Price (DECIMAL(10,2))

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Stock INT NOT NULL CHECK (Stock >= 0),
    Price DECIMAL(10,2) NOT NULL
);
```

### 3. Orders Table

OrderID (Primary Key, INT, Auto Increment)
CustomerID (Foreign Key referencing Customers(CustomerID))
ProductID (Foreign Key referencing Products(ProductID))
Quantity (INT)
OrderDate (Default: Current Timestamp)

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY AUTO_INCREMENT,
    CustomerID INT,
    ProductID INT,
    Quantity INT NOT NULL,
    OrderDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

**Note:**
Ensure proper primary and foreign key constraints.

---

## Insert Sample Data

Populate each table with at least 3 rows of sample data.

**Customers:**
Alice, Bob, Charlie with appropriate email addresses.

**Products:**
Laptop (Stock: 10, Price: 1000.00), Smartphone (Stock: 15, Price: 700.00), Headphones (Stock: 50, Price: 50.00).

**Orders:**
Leave this table empty initially. Orders will be added during the transaction tasks.

```
-- Insert Customers
INSERT INTO Customers (CustomerID, CustomerName, Email) VALUES
    (1, 'Alice', 'alice@example.com'),
    (2, 'Bob', 'bob@example.com'),
    (3, 'Charlie', 'charlie@example.com');

-- Insert Products
INSERT INTO Products (ProductID, ProductName, Stock, Price) VALUES
    (1, 'Laptop', 10, 1000.00),
    (2, 'Smartphone', 15, 700.00),
    (3, 'Headphones', 50, 50.00);

-- Verify Data
SELECT * FROM Customers;
SELECT * FROM Products;
```

## Part 2: Transaction Management Tasks

**Scenario 1**: Placing an Order
Simulate placing two orders:
1. Alice orders 5 Laptops.
2. Bob orders 12 Laptops

**Steps:**
1. Start a transaction.
2. Check stock levels and decrease stock if sufficient:
    Decrease stock by the ordered quantity for valid orders.
    Rollback the transaction if the stock is insufficient.
3. Insert the order details into the Orders table.
4. Use a savepoint after the stock update but before inserting the order.
5. Commit the transaction if all operations succeed; otherwise, rollback.

```
START TRANSACTION;

-- Reduce stock
UPDATE Products SET Stock = Stock - 5 WHERE ProductID = 1;

-- Savepoint
SAVEPOINT AfterStockUpdate;
```

```
-- Insert order
INSERT INTO Orders (CustomerID, ProductID, Quantity) VALUES (1, 1, 5);

-- Commit transaction
COMMIT;

-- Verify the changes
SELECT * FROM Products;
SELECT * FROM Orders;
```

---

**Scenario 2**: Update an Order
Update Alice's order from 5 Laptops to 8 Laptops. Rollback if the total price exceeds 5000.

**Steps:**
1. Start a transaction.
2. Update the Orders table to reflect the new quantity.
3. Adjust the stock in the Products table accordingly:
Deduct or add stock based on the updated quantity.
4. Calculate the total price of the updated order.
5. Use a savepoint before updating the stock. Rollback to the savepoint if the price exceeds 5000.
6. Commit if the operations are valid; rollback otherwise.

```
START TRANSACTION;

-- Attempt to reduce stock
UPDATE Products SET Stock = Stock - 12 WHERE ProductID = 1;

-- Check and rollback
ROLLBACK;

-- Verify no changes were made
SELECT * FROM Products;
SELECT * FROM Orders;
```

---

**Scenario 3**: Cancel an Order
Cancel Alice's order for Laptops and restore the stock.

**Steps:**
1. Start a transaction.
2. Restore the product stock in the Products table.
3. Delete the order record from the Orders table.
4. Use a savepoint before deleting the order. Rollback to the savepoint if the stock update fails.
5. Commit if all operations are successful; rollback otherwise.

```
START TRANSACTION;

-- Update order quantity
UPDATE Orders SET Quantity = 8 WHERE OrderID = 1;

-- Adjust stock
UPDATE Products SET Stock = Stock - (8 - 5) WHERE ProductID = 1;

-- Savepoint before committing
SAVEPOINT BeforeCommit;

-- Calculate total price
SELECT (8 * 1000) AS TotalPrice;
```

```
-- Rollback if total price exceeds 5000
ROLLBACK TO BeforeCommit;

-- Commit changes if valid
COMMIT;

-- Verify the updates
SELECT * FROM Products;
SELECT * FROM Orders;
```

---

**Scenario 4**: Place Multiple Orders
Data for the Scenario:
Customer Charlie places the following orders:
1. 30 Headphones.
2. 10 Smartphones.

**Steps:**
1. Start a transaction.
2. Insert each order into the Orders table.
Adjust the stock in the Products table accordingly.
3. Use savepoints after each stock update to handle partial rollbacks if one order fails due to insufficient stock.
4. Commit the transaction only if all operations are valid.

```
START TRANSACTION;

-- Order 1: Headphones
INSERT INTO Orders (CustomerID, ProductID, Quantity) VALUES (3, 3, 30);
UPDATE Products SET Stock = Stock - 30 WHERE ProductID = 3;

-- Savepoint after first order
SAVEPOINT AfterFirstOrder;

-- Order 2: Smartphones
INSERT INTO Orders (CustomerID, ProductID, Quantity) VALUES (3, 2, 10);
UPDATE Products SET Stock = Stock - 10 WHERE ProductID = 2;

-- Commit the transaction
COMMIT;

-- Verify the changes
SELECT * FROM Products;
SELECT * FROM Orders;
```

---

**Part 3**: **Perform Queries on the Database**
Perform the following queries to validate and understand the state of your database:

1. Display complete details of all transactions, including:
- OrderID
- Customer Name
- Product Name
- Quantity Ordered
- Total Price

- Order Date

2. Show the updated stock levels of all products in the Products table.
3. Retrieve all orders placed by Charlie, displaying product details and total price for each order.
4. Display the total number of products ordered for each product, grouped by product name.
5. Verify that all transactions in scenarios above correctly updated or rolled back changes by comparing initial and final states of the tables.

### 1. Display Complete Details of All Transactions

```
SELECT

    Orders.OrderID,
    Customers.CustomerName,
    Products.ProductName,
    Orders.Quantity,
    Products.Price,
    (Orders.Quantity * Products.Price) AS TotalPrice,
    Orders.OrderDate
FROM Orders
JOIN Customers ON Orders.CustomerID = Customers.CustomerID
JOIN Products ON Orders.ProductID = Products.ProductID;
```

### 2. Show Updated Stock Levels

```
SELECT ProductName, Stock FROM Products;
```

### 3. Retrieve All Orders Placed by Charlie

```
SELECT
    Orders.OrderID,
    Products.ProductName,
    Orders.Quantity,
    (Orders.Quantity * Products.Price) AS TotalPrice
FROM Orders
JOIN Products ON Orders.ProductID = Products.ProductID
WHERE Orders.CustomerID = 3;
```

### 4. Display Total Quantity Ordered for Each Product

```
SELECT
    Products.ProductName,
    SUM(Orders.Quantity) AS TotalOrdered
FROM Orders
JOIN Products ON Orders.ProductID = Products.ProductID
GROUP BY Products.ProductName;
```

### 5. Verify Data Consistency After Rollbacks

Check initial state of tables:

```
SELECT * FROM Products;
SELECT * FROM Orders;
```

Check final state of tables after performing all scenarios:

```
SELECT * FROM Products;
SELECT * FROM Orders;
```

## Expected Results

1. **Scenario 1**: Alice's order is successfully placed, reducing stock for Laptops. Bob's order fails, and no changes are made.
2. **Scenario 2**: Alice's order updates only if total price does not exceed 5000. Otherwise, changes rollback.
3. **Scenario 3**: Alice's canceled order restores stock for Laptops, and the order is deleted from the Orders table.
4. **Scenario 4**: Charlie's orders for Headphones and Smartphones are processed successfully if stock allows. Partial rollback occurs if one order fails.

## Task 02: (05 Marks)

| Time | Transaction T1 | Transaction T2 | Transaction T3 |
|---|---|---|---|
| t1 | begin_transaction | | |
| t2 | read(bal_x) | begin_transaction | |
| t3 | bal_x = bal_x - 100 | read(bal_y) | begin_transaction |
| t4 | write(bal_x) | bal_y = bal_y + 50 | read(bal_z) |
| t5 | read(bal_y) | write(bal_y) | bal_z = bal_z - 200 |
| t6 | bal_y = bal_y - 20 | | write(bal_z) |
| t7 | write(bal_y) | read(bal_x) | |
| t8 | commit | bal_x = bal_x + 30 | |
| t9 | | write(bal_x) | |
| t10 | | read(bal_z) | |
| t11 | | bal_z = bal_z + 100 | |
| t12 | | write(bal_z) | |
| t13 | rollback | | |
| t14 | | bal_y = bal_y - 50 | |
| t15 | | write(bal_y) | |
| t16 | commit | | |

**SOLUTION:**

| bal_x | bal_y | bal_z |
|---|---|---|
| 1000 | 500 | 300 |
| 1000 | 500 | 300 |
| 1000 | 500 | 300 |
| 900 | 500 | 300 |
| 900 | 550 | 300 |
| 900 | 550 | 100 |
| 900 | 480 | 100 |
| 900 | 480 | 100 |
| 930 | 480 | 100 |
| 930 | 480 | 100 |
| 930 | 480 | 100 |
| 930 | 480 | 200 |
| 930 | 480 | 200 |
| 930 | 480 | 200 |
| 930 | 430 | 200 |
| 930 | 430 | 200 |