Assignment 1: Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.

Step 1: Basic SELECT Query to Retrieve All Columns
First, we'll write a basic **SELECT** query to retrieve all columns from the **customers** table.

```
SELECT * FROM customers;
```

Step 2: Modify the Query to Return Specific Columns
Next, we'll modify this query to return only the **customer_name** and **email** columns.

```
SELECT customer_name, email FROM customers;
```

Step 3: Add a Condition to Filter by City
Finally, we'll add a **WHERE** clause to filter customers based on a specific city. Let's assume we are interested in customers from the city "New York".

```
SELECT customer_name, email
FROM customers

WHERE city = 'New York';
```

```
SELECT customer_name, email

FROM customers

WHERE city = ?;
```

```
SELECT customer_name, email

FROM customers

WHERE city = 'New York';
```

Assignment 2: Craft a query using an INNER JOIN to combine 'orders' and 'customers' tables for customers in a specified region, and a LEFT JOIN to display all customers including those without orders.

Step 1: INNER JOIN to Combine 'orders' and 'customers' Tables

We'll write a query using an **INNER JOIN** to combine data from the **orders** and **customers** tables for customers in a specified region. Let's assume the region we're interested in is "West".

```
SELECT customers.*, orders.*
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.region = 'West';
```

Step 2: LEFT JOIN to Display All Customers Including Those Without Orders

Next, we'll write a query using a **LEFT JOIN** to display all customers, including those who don't have any orders. We'll include the same region filter to limit customers to a specified region.

```
SELECT customers.*, orders.*
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.region = 'West';
```

Combining both queries, we have:

Query 1: INNER JOIN for Customers in a Specified Region

```
SELECT customers.*, orders.*
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.region = 'West';
```

Query 2: LEFT JOIN to Display All Customers Including Those Without Orders in a Specified Region

```
SELECT customers.*, orders.*
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.region = 'West';
```

Assignment 3: Utilize a subquery to find customers who have placed orders above the average order value, and write a UNION query to combine two SELECT statements with the same number of columns.

Part 1: Subquery to Find Customers with Orders Above Average Order Value

1. **Step 1:** Calculate the average order value.
2. **Step 2:** Use this average to find customers who have placed orders above this value.

```
SELECT customer_id, customer_name
FROM customers
WHERE customer_id IN (
SELECT customer_id
FROM orders
WHERE order_value > (
SELECT AVG(order_value)
FROM orders
)
);
```

Part 2: UNION Query to Combine Two SELECT Statements

A **UNION** query combines the results of two **SELECT** statements with the same number of columns. The columns must have compatible data types.

Let's assume we have two different sets of criteria for selecting customers. For example:

- Customers from the "East" region.
- Customers who have placed orders in the last 30 days.

```
-- Select customers from the "East" region
SELECT customer_id, customer_name, email
FROM customers
WHERE region = 'East'
UNION
-- Select customers who have placed orders in the last 30 days
SELECT c.customer_id, c.customer_name, c.email
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

Summary

1. **Subquery to find customers who have placed orders above the average order value:**

```
SELECT customer_id, customer_name
FROM customers
WHERE customer_id IN (
  SELECT customer_id
  FROM orders
  WHERE order_value > (
  SELECT AVG(order_value)
  FROM orders
  )
);
```

UNION query to combine two SELECT statements with the same number of columns:

```
-- Select customers from the "East" region
SELECT customer_id, customer_name, email
FROM customers
WHERE region = 'East'

UNION

-- Select customers who have placed orders in the last 30 days
SELECT c.customer_id, c.customer_name, c.email
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

Assignment 4: Compose SQL statements to BEGIN a transaction, INSERT a new record into the 'orders' table, COMMIT the transaction, then UPDATE the 'products' table, and ROLLBACK the transaction.

Step 1: BEGIN a Transaction

We'll start a transaction using the **BEGIN** or **START TRANSACTION** statement.

```
BEGIN;
```

Step 2: INSERT a New Record into the 'orders' Table

Assume the **orders** table has columns **order_id**, **customer_id**, **order_date**, and **order_value**. We'll insert a new order record.

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (101, 1, '2024-05-28', 250.00);
```

Step 3: COMMIT the Transaction

We'll commit the transaction to save the changes.

```
COMMIT;
```

Step 4: BEGIN Another Transaction

We'll start another transaction to perform an update.

```
BEGIN;
```

Step 5: UPDATE the 'products' Table

Assume the **products** table has columns **product_id**, **product_name**, **stock_quantity**, and **price**. We'll update the stock quantity of a product.

```
UPDATE products
SET stock_quantity = stock_quantity - 1
WHERE product_id = 10;
```

Step 6: ROLLBACK the Transaction

We'll rollback the transaction to undo the update.

```
ROLLBACK;
```

Summary

Putting it all together, the SQL statements to manage the transactions are as follows:

```
-- Begin the first transaction
BEGIN;

-- Insert a new record into the 'orders' table
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (101, 1, '2024-05-28', 250.00);

-- Commit the first transaction
COMMIT;

-- Begin the second transaction
BEGIN;

-- Update the 'products' table
UPDATE products
SET stock_quantity = stock_quantity - 1
WHERE product_id = 10;

-- Rollback the second transaction
ROLLBACK;
```

Assignment 5: Begin a transaction, perform a series of INSERTs into 'orders', setting a SAVEPOINT after each, rollback to the second SAVEPOINT, and COMMIT the overall transaction.

Step-by-Step Process

1. **Begin a transaction.**
2. **Perform a series of INSERT operations into the orders table.**
3. **Set a SAVEPOINT after each INSERT.**
4. **Rollback to the second SAVEPOINT.**
5. **Commit the overall transaction.**

Step 1: BEGIN a Transaction:

```
BEGIN;
```

Step 2: Perform a Series of INSERT Operations:

Assume the **orders** table has columns **order_id**, **customer_id**, **order_date**, and **order_value**.

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (201, 2, '2024-05-28', 300.00);
```

Step 3: Set a SAVEPOINT After Each INSERT:

```
SAVEPOINT sp1;
```

Perform another **INSERT**:

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (202, 3, '2024-05-28', 450.00);
```

Set another **SAVEPOINT**:

```
SAVEPOINT sp2;
```

Perform one more **INSERT**:

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (203, 4, '2024-05-28', 500.00);
```

Set the final **SAVEPOINT**:

```
SAVEPOINT sp3;
```

Step 4: Rollback to the Second `SAVEPOINT`:

```
ROLLBACK TO sp2;
```

Step 5: Commit the Overall Transaction:

```
COMMIT;
```

Summary:

Putting it all together, the SQL statements for managing the transaction with **`SAVEPOINT`**s and rollback are:

```
-- Begin the transaction
BEGIN;

-- Perform the first INSERT and set the first SAVEPOINT
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (201, 2, '2024-05-28', 300.00);
SAVEPOINT sp1;

-- Perform the second INSERT and set the second SAVEPOINT
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (202, 3, '2024-05-28', 450.00);
SAVEPOINT sp2;

-- Perform the third INSERT and set the third SAVEPOINT
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (203, 4, '2024-05-28', 500.00);
SAVEPOINT sp3;

-- Rollback to the second SAVEPOINT
ROLLBACK TO sp2;

-- Commit the overall transaction
COMMIT;
```

Assignment 6: Draft a brief report on the use of transaction logs for data recovery and create a hypothetical scenario where a transaction log is instrumental in data recovery after an unexpected shutdown.

Report on the Use of Transaction Logs for Data Recovery:

Introduction:

Transaction logs are essential components in database management systems (DBMS). They record all changes made to the database, ensuring data integrity and facilitating recovery in case of failures. This report discusses the role of transaction logs in data recovery and presents a hypothetical scenario demonstrating their importance after an unexpected shutdown.

Role of Transaction Logs:

1. **Data Integrity and Consistency**:

- Transaction logs record every transaction executed on the database, including insertions, updates, deletions, and schema modifications.
- They ensure that transactions are atomic, consistent, isolated, and durable (ACID properties).
- In the event of a failure, the logs help maintain data consistency by allowing incomplete transactions to be rolled back.

2. **Data Recovery**:

- Transaction logs are vital for restoring the database to a consistent state after a crash or unexpected shutdown.
- They enable point-in-time recovery, allowing the database to be restored to a specific moment before the failure occurred.
- The logs facilitate both full and incremental backups, ensuring minimal data loss.

3. **Audit Trail**:

- Transaction logs provide a detailed audit trail of all operations performed on the database.
- This audit trail can be used for security monitoring, troubleshooting, and compliance with regulatory requirements.

Hypothetical Scenario: Data Recovery After an Unexpected Shutdown:

Scenario: A financial services company operates a database that manages transactions for its online banking system. The database handles critical operations, including customer deposits, withdrawals, and transfers. One day, the data center experiences a sudden power outage, causing an unexpected shutdown of the database server.

Impact:

- Several transactions were in progress at the time of the shutdown.
- There is a risk of data corruption or loss of the transactions that were not fully committed.
- The company's operations and customer trust are at stake due to potential discrepancies in account balances.

Data Recovery Using Transaction Logs:

1. **System Restart**:

- The database server is restarted after power is restored.
- During the startup process, the DBMS detects that the system was not shut down gracefully.

2. **Transaction Log Analysis**:

- The DBMS reads the transaction logs to identify the state of transactions at the time of the failure.
- It differentiates between committed, uncommitted, and in-progress transactions.

3. **Undo Operations**:

- For any uncommitted transactions, the DBMS performs rollback operations using the transaction log entries.
- This ensures that the database reverts to the state before these transactions began, maintaining consistency.

4. **Redo Operations**:

- The DBMS then replays the committed transactions from the transaction logs to ensure that all completed operations are reflected in the database.
- This step guarantees that no committed transactions are lost, and the database is brought up to date.

5. **Validation and Restart**:

- The database is now in a consistent state, reflecting all committed transactions and none of the uncommitted ones.
- The DBMS completes the recovery process and opens the database for normal operations.

Outcome:

- The database is successfully recovered with no data loss or corruption.
- Customers' account balances are accurate, and the financial integrity of the system is maintained.
- The company can continue its operations smoothly, ensuring customer trust and satisfaction.

Conclusion:

Transaction logs play a crucial role in ensuring data integrity, consistency, and recovery in database systems. They provide a reliable mechanism for restoring databases to a consistent state after unexpected failures. The hypothetical scenario illustrates how transaction logs can be instrumental in data recovery, enabling businesses to maintain operations and trust even in the face of unforeseen disruptions.