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.

To retrieve all columns from a 'customers' table and then modify the query to return only the customer name and email address for customers in a specific city, you can use the following SQL queries:

Retrieve All Columns from 'customers' Table:

```sql
-- Retrieve all columns from the 'customers' table
SELECT *
FROM customers;
```

Return Customer Name and Email Address for Customers in a Specific City:

Assuming the specific city is 'New York':

```sql
-- Return customer name and email address for customers in the city of 'New York'
SELECT customer_name, email
FROM customers
WHERE city = 'New York';
```

In the modified query, the `SELECT` statement specifies the columns `customer_name` and `email` to be retrieved from the 'customers' table, and the `WHERE` clause filters the results to only include customers located in the city of 'New York'. Adjust the city name as needed to retrieve data for customers in a different city.

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.

To combine the 'orders' and 'customers' tables using INNER JOIN for customers in a specified region and LEFT JOIN to display all customers, including those without orders, you can use the following SQL query:

INNER JOIN for Customers in a Specified Region:

Assuming the specified region is 'North America':

```sql
-- Combine 'orders' and 'customers' tables using INNER JOIN for customers in the 'North America' region
SELECT c.customer_id, c.customer_name, o.order_id, o.order_date
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.region = 'North America';
```

In this query:

- The `INNER JOIN` combines the 'customers' and 'orders' tables based on the 'customer_id' column.

- It retrieves the 'customer_id', 'customer_name' from the 'customers' table, and 'order_id', 'order_date' from the 'orders' table for customers in the 'North America' region.

LEFT JOIN to Display All Customers:

```sql
-- Display all customers, including those without orders
SELECT c.customer_id, c.customer_name, o.order_id, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

In this query:

- The `LEFT JOIN` retrieves all records from the 'customers' table and only matching records from the 'orders' table based on the 'customer_id' column.

- It displays the 'customer_id', 'customer_name' from the 'customers' table, and 'order_id', 'order_date' from the 'orders' table, including customers without orders.

Adjust the region or specific conditions as needed to tailor the query to your specific requirements.

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.

To find customers who have placed orders above the average order value using a subquery and to combine two SELECT statements with the same number of columns using a UNION query, you can use the following SQL queries:

Subquery to Find Customers with Orders Above Average Order Value:

```sql
-- 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
    GROUP BY customer_id
    HAVING AVG(order_value) > (
        SELECT AVG(order_value)
        FROM orders
    )
);
```

In this query:

- The subquery calculates the average order value across all orders.

- The main query retrieves customers who have placed orders with values above the calculated average order value.

UNION Query to Combine Two SELECT Statements:

Assuming you want to combine results from two similar SELECT statements:

```sql
-- Combine results from two SELECT statements using UNION
SELECT customer_id, customer_name
FROM customers
WHERE region = 'Europe'
UNION
SELECT customer_id, customer_name
FROM customers
WHERE region = 'Asia';
```

In this query:

- The first SELECT statement retrieves customers from the 'Europe' region.

- The second SELECT statement retrieves customers from the 'Asia' region.

- The UNION operator combines the results of both SELECT statements into a single result set.

By using subqueries and UNION queries, you can efficiently retrieve specific data based on conditions and combine results from multiple SELECT statements with the same number of columns.

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.

Here are the SQL statements to begin a transaction, insert a new record into the 'orders' table, commit the transaction, update the 'products' table, and then roll back the transaction:

Begin a Transaction, Insert a New Record into 'orders' Table, and Commit the Transaction:

```sql
-- Begin the transaction
BEGIN TRANSACTION;

-- Insert a new record into the 'orders' table
INSERT INTO orders (order_id, customer_id, product_id, order_date, order_value)
VALUES (1001, 101, 501, '2023-06-01', 500.00);

-- Commit the transaction
COMMIT;
```

In this part:

- The `BEGIN TRANSACTION` statement starts a new transaction.

- A new record is inserted into the 'orders' table.

- The `COMMIT` statement finalizes the transaction and makes the changes permanent.

Update the 'products' table and Roll Back the Transaction:

```sql
-- Update the 'products' table
UPDATE products
SET product_price = product_price * 1.1
WHERE product_id = 501;

-- Roll back the transaction
ROLLBACK;
```

In this part:

- The `UPDATE` statement increases the price of the product with `product_id = 501` by 10%.

- The `ROLLBACK` statement cancels the transaction and reverts any changes made during the transaction.

By using these SQL statements, you can:

1. Begin a transaction to ensure data consistency and integrity.

2. Insert a new record into the 'orders' table within the transaction.

3. Commit the transaction to make the changes permanent.

4. Update the 'products' table outside the transaction.

5. Roll back the transaction to undo any changes made during the transaction.

The `ROLLBACK` statement ensures that the update to the 'products' table is not committed and the database remains in its original state before the transaction began.

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.

To begin a transaction, perform a series of INSERTs into the 'orders' table, set a SAVEPOINT after each INSERT, rollback to the second SAVEPOINT, and finally commit the overall transaction, you can use the following SQL statements:

Begin a Transaction, Perform INSERTs, Set SAVEPOINTs, Rollback to Second SAVEPOINT, and Commit Transaction:

```sql
-- Begin the transaction
BEGIN TRANSACTION;

-- Insert the first record into the 'orders' table
INSERT INTO orders (order_id, customer_id, product_id, order_date, order_value)
VALUES (1001, 101, 501, '2023-06-01', 500.00);

-- Set the first SAVEPOINT
```

```
SAVEPOINT savepoint1;

-- Insert the second record into the 'orders' table
INSERT INTO orders (order_id, customer_id, product_id, order_date, order_value)
VALUES (1002, 102, 502, '2023-06-02', 600.00);

-- Set the second SAVEPOINT
SAVEPOINT savepoint2;

-- Insert the third record into the 'orders' table
INSERT INTO orders (order_id, customer_id, product_id, order_date, order_value)
VALUES (1003, 103, 503, '2023-06-03', 700.00);

-- Roll back to the second SAVEPOINT
ROLLBACK TO SAVEPOINT savepoint2;

-- Commit the overall transaction
COMMIT;
```

In this sequence of SQL statements:

- The `BEGIN TRANSACTION` initiates a new transaction.

- Three INSERT statements add records to the 'orders' table.

- Two SAVEPOINTs (`savepoint1` and `savepoint2`) are set after the first and second INSERTs, respectively.

- The `ROLLBACK TO SAVEPOINT` command reverts the transaction to the state at the second SAVEPOINT.

- Finally, the `COMMIT` statement finalizes the transaction, making the changes permanent up to the second SAVEPOINT.

By using SAVEPOINTs within a transaction, you can create checkpoints to roll back to specific points in the transaction's execution, providing flexibility and control over the data modifications before committing the transaction.

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 play a crucial role in ensuring data integrity and facilitating recovery in the event of system failures or unexpected shutdowns. These logs record all changes made to a database, providing a detailed history of transactions that can be used to restore data to a consistent state.

Benefits of Transaction Logs for Data Recovery:

1. **Point-in-Time Recovery:** Transaction logs allow for point-in-time recovery, enabling database administrators to restore data to a specific moment before a failure occurred.

2. **Rollback and Rollforward Operations:** Transaction logs support rollback to undo incomplete transactions and rollforward to reapply committed transactions after a failure.

3. **Minimized Data Loss:** By capturing changes as they occur, transaction logs help minimize data loss by ensuring that committed transactions can be recovered.

4. **Data Consistency:** Transaction logs maintain data consistency by preserving the order of transactions and ensuring that changes are applied in the correct sequence during recovery.

Hypothetical Scenario:

In a hypothetical scenario, consider a retail database where a sudden power outage causes the system to shut down unexpectedly during a busy sales period. Without transaction logs, the database could be left in an inconsistent state, risking data loss and potential corruption.

However, due to the presence of transaction logs, the database administrator can leverage the log files to recover the database to a consistent state:

1. **Identifying the Last Committed Transaction:** The transaction log helps identify the last committed transaction before the shutdown.

2. **Rolling Back Incomplete Transactions:** Any incomplete transactions at the time of the shutdown can be rolled back to maintain data integrity.

3. **Reapplying Committed Transactions:** The committed transactions recorded in the transaction log can be reapplied to restore the database to its state before the unexpected shutdown.

4. **Ensuring Data Consistency:** By using the transaction log, the database can be recovered with minimal data loss and ensure data consistency across the system.

Conclusion:

Transaction logs are essential for data recovery and maintaining database integrity in the face of unexpected events. They serve as a critical tool for database administrators to restore data to a consistent state, minimize data loss, and ensure business continuity in the event of system failures or

shutdowns. Investing in robust transaction logging mechanisms is vital for safeguarding data and enabling efficient recovery processes.