Assignment1: 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.

Solution:

1. Retrieve All Columns from the 'customers' Table

SQL:

```
SELECT * FROM customers;
```

2. Retrieve Only Customer Name and Email Address for Customers in a Specific City

Assuming the columns in the **customers** table include **customer_name**, **email**, and **city**:

SQL:

```
SELECT customer_name, email
FROM customers
WHERE city = 'London';
```

Assignment2: 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.

Solution:

1. Using an **INNER JOIN** to combine the **orders** and **customers** tables for customers in a specified region.
2. Using a **LEFT JOIN** to display all customers, including those without orders.

1. INNER JOIN to Combine 'orders' and 'customers' for Customers in a Specified Region

Assuming the '**customers'** table includes a column '**region'** to specify the customer's region:

SQL:

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

Example:

If the specified region is 'West', the query would be:

SQL:

```
SELECT orders.*, customers.customer_name, customers.email
```

FROM orders

INNER JOIN customers ON orders.customer_id = customers.customer_id

WHERE customers.region = 'West;

2. LEFT JOIN to Display All Customers Including Those Without Orders

SQL:

SELECT customers.customer_name, customers.email, orders.order_id, orders.order_date, orders.amount

FROM customers

LEFT JOIN orders ON customers.customer_id = orders.customer_id;

This query ensures that all customers are listed, including those who do not have any orders. The fields from the **orders** table will be **NULL** for customers without orders.

Combining both requirements, here's how you can use both queries in the same context:

Specified Region 'West' and Including All Customers:

SQL:

SELECT orders.*, customers.customer_name, customers.email

FROM orders

INNER JOIN customers ON orders.customer_id = customers.customer_id

WHERE customers.region = 'West';

Select customers.customer_name, customers.email, orders.order_id, orders.order_date, orders.amount

FROM customers

LEFT JOIN orders ON customers.customer_id = orders.customer_id;

These queries should help you achieve the desired results using **INNER JOIN** and **LEFT JOIN.**

Assignment3: 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.

Solution:

1. Using a subquery to find customers who have placed orders above the average order value.
2. Writing a **UNION** query to combine two **SELECT** statements with the same number of columns.

1. Subquery to Find Customers Who Have Placed Orders Above the Average Order Value

Assuming the **orders** table includes columns **customer_id** and **amount**:

SQL:

```
--SELECT DISTINCT customer_id
FROM orders
WHERE amount > (
 SELECT AVG(amount)
FROM orders
);
```

To get the customer details (e.g., **customer_name** and **email**) for those who placed orders above the average value, assuming **customers** table includes **customer_id**, **customer_name**, and **email**:

SQL:

```
--SELECT c.customer_id, c.customer_name, c.email
FROM customers c
WHERE c.customer_id IN (
SELECT o.customer_id
FROM orders o
WHERE o.amount > (
SELECT AVG(amount)
FROM orders
)
);
```

2. UNION Query to Combine Two SELECT Statements with the Same Number of Columns

Assuming we want to combine data from two different regions, let's say **RegionA** and **RegionB**:

SQL:

```
SELECT customer_id, customer_name, email, 'RegionA' AS region
FROM customers
WHERE region = 'RegionA'
UNION
SELECT customer_id, customer_name, email, 'RegionB' AS region
FROM customers
WHERE region = 'RegionB';
```

This **UNION** query combines the results of customers from **RegionA** and **RegionB**, with an additional column indicating the region.

Combined Example

For a complete example, including both the subquery and the **UNION** query:

SQL:

```
SELECT c.customer_id, c.customer_name, c.email
FROM customers c
WHERE c.customer_id IN (
SELECT o.customer_id
FROM orders o
WHERE o.amount > (
SELECT AVG(amount)
FROM orders
)
);
--SELECT customer_id, customer_name, email, 'RegionA' AS region
FROM customers
WHERE region = 'RegionA'
UNION
SELECT customer_id, customer_name, email, 'RegionB' AS region
FROM customers
WHERE region = 'RegionB';
```

These queries demonstrate the use of a subquery to filter data based on a condition involving an aggregate function and the use of **UNION** to combine results from multiple **SELECT** statements.

Assignment4: 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.

Solution:

1. **BEGIN a transaction**
2. **INSERT a new record into the orders table**
3. **COMMIT the transaction**
4. **UPDATE the products table**

5. **ROLLBACK the transaction**

SQL:

START TRANSACTION;

--INSERT INTO orders (order_id, customer_id, order_date, amount) VALUES (5, 105, '2024-05-19', 350.00);

-- COMMIT;

--START TRANSACTION;

--UPDATE products

SET stock = stock - 10

WHERE product_id = 1001;

--ROLLBACK;

Explanation:

1. **START TRANSACTION;** : Begins a new transaction.
2. **INSERT INTO orders...;** : Inserts a new row into the **orders** table.
3. **COMMIT;** : Commits the transaction, making the insertion into the **orders** table permanent.
4. **START TRANSACTION;** : Begins a new transaction for the subsequent update operation.
5. **UPDATE products...;** : Updates the **products** table, modifying the stock of a specified product.
6. **ROLLBACK;** : Rolls back the transaction, undoing the update operation on the **products** table.

This sequence of commands ensures that the **INSERT** operation is committed, making it permanent in the **orders** table, while the **UPDATE** operation is rolled back, meaning any changes made to the **products** table during this second transaction are discarded.

Assignment5: 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.

Solution:

SQL:

-- Begin the transaction

START TRANSACTION;

-- Perform the first INSERT and set a savepoint

INSERT INTO orders (order_id, customer_id, order_date, amount) VALUES (1, 101, '2024-05-15', 150.00); SAVEPOINT sp1;

-- Perform the second INSERT and set a savepoint

INSERT INTO orders (order_id, customer_id, order_date, amount) VALUES (2, 102, '2024-05-16', 200.00); SAVEPOINT sp2;

-- Perform the third INSERT and set a savepoint

INSERT INTO orders (order_id, customer_id, order_date, amount) VALUES (3, 103, '2024-05-17', 250.00); SAVEPOINT sp3;

-- Perform the fourth INSERT and set a savepoint

INSERT INTO orders (order_id, customer_id, order_date, amount) VALUES (4, 104, '2024-05-18', 300.00); SAVEPOINT sp4;

-- Rollback to the second savepoint

ROLLBACK TO SAVEPOINT sp2;

-- Commit the transaction

COMMIT;

Explanation:

1. **START TRANSACTION;** : Begins a new transaction.
2. **INSERT INTO orders...;** : Inserts the first row into the **orders** table.
3. **SAVEPOINT sp1;** : Sets the first savepoint named **sp1** after the first insert.
4. **INSERT INTO orders...;** : Inserts the second row into the **orders** table.
5. **SAVEPOINT sp2;** : Sets the second savepoint named **sp2** after the second insert.
6. **INSERT INTO orders...;** : Inserts the third row into the **orders** table.
7. **SAVEPOINT sp3;** : Sets the third savepoint named **sp3** after the third insert.
8. **INSERT INTO orders...;** : Inserts the fourth row into the **orders** table.
9. **SAVEPOINT sp4;** : Sets the fourth savepoint named **sp4** after the fourth insert.
10. **ROLLBACK TO SAVEPOINT sp2;** : Rolls back the transaction to the state after the second insert, effectively undoing the third and fourth inserts.
11. **COMMIT;** : Commits the transaction, making the first and second inserts permanent and discarding the third and fourth inserts due to the rollback.

Assignment6: 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.

Answer:

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 database systems. This report explores the significance of transaction logs in data recovery and presents a hypothetical scenario to illustrate their importance in mitigating the impact of unexpected shutdowns.

Importance of Transaction Logs

Transaction logs serve as a record of all modifications made to a database, capturing every transaction's details before they are applied to the database itself. This chronological log of changes enables database systems to recover from various failures, including hardware malfunctions, software crashes, and power outages. The key benefits of transaction logs include:

1. **Data Integrity**: Transaction logs maintain a consistent record of database transactions, ensuring that changes are applied in the correct order and allowing for recovery to a consistent state in case of failure.
2. **Point-in-Time Recovery**: By replaying transactions recorded in the log, database administrators can restore the database to a specific point in time, facilitating recovery from data corruption or user errors.
3. **Rollback and Roll forward Operations**: Transaction logs enable the rollback of incomplete transactions and the roll forward of committed transactions, ensuring that data changes are applied correctly during recovery.
4. **Minimized Data Loss**: With transaction logs, even in the event of a failure, only transactions that were in progress at the time of the failure may be lost, minimizing data loss and preserving the majority of committed changes.

Hypothetical Scenario: Data Recovery After an Unexpected Shutdown

Consider a multinational e-commerce platform, "E-Shop," which relies on a robust database system to manage its vast inventory and customer orders. One day, a sudden power outage occurs due to unforeseen circumstances, resulting in an unexpected shutdown of the database server during peak business hours.

Situation Before the Shutdown

- The database was actively processing numerous transactions, including customer orders, inventory updates, and payment transactions.
- Multiple transactions were in progress, while others had been committed but not yet persisted to the database files.

Recovery Process Using Transaction Logs

1. **Database Restart and Log Analysis**:
    - Upon restarting the database server, the DBMS detects an incomplete shutdown and initiates the recovery process.
    - The system analyze the transaction logs to identify the state of transactions at the time of the shutdown.
2. **Transaction Reconciliations**:

- Committed transactions recorded in the logs are reapplied to the database to ensure data consistency.
- Incomplete transactions are rolled back to maintain database integrity and prevent data corruption.

3. **Point-in-Time Recovery**:
    - Database administrators utilize the transaction logs to perform a point-in-time recovery, restoring the database to the state immediately before the unexpected shutdown.
    - By specifying a timestamp or transaction identifier, the system replays transactions up to the desired point, effectively recovering the database without loss of critical data.

Outcome: Thanks to the comprehensive transaction logging mechanism, E-Shop successfully recovers from the unexpected shutdown with minimal disruption to its operations. The transaction logs play a pivotal role in restoring data integrity, ensuring that all committed transactions are preserved, and the database remains consistent.

Conclusion:

Transaction logs are indispensable for data recovery in database systems, providing a reliable mechanism to recover from failures and maintain data integrity. The hypothetical scenario of E-Shop highlights the critical role of transaction logs in mitigating the impact of unexpected shutdowns and ensuring continuous availability of critical business data.

In conclusion, investing in robust transaction logging mechanisms is essential for organizations to safeguard their data assets and mitigate the risks associated with system failures and disasters.