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.

1. Attempt to Select from `customers` Table:
- The query attempts to select all records from the `customers` table where the city is 'YourCityName'.
- Result: An error occurs indicating no database is selected.

2. Create `people` Database:
- The command creates a new database named `people`.
- Result: Database `people` is created successfully.

3. Select `people` Database:
- The command sets the current database to `people`.
- Result: The database is changed to `people`.

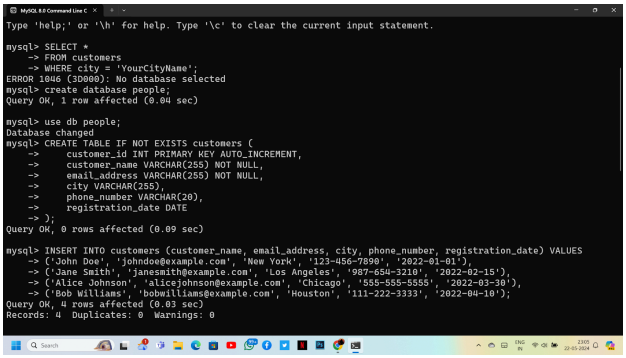
4. Create `customers` Table:
- The command creates a `customers` table with columns: `customer_id`, `customer_name`, `email_address`, `city`, `phone_number`, and `registration_date`.
- Result: The table `customers` is created successfully.

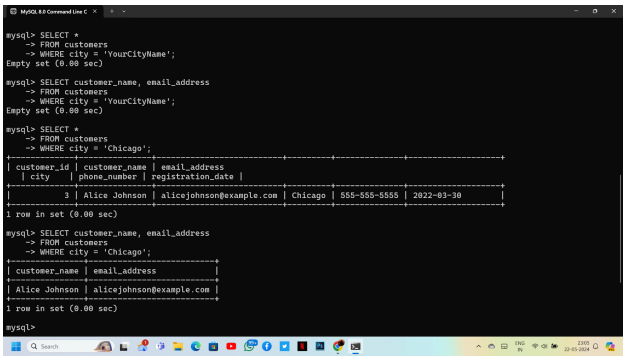
5. Insert Data into `customers` Table:
- The command inserts four records into the `customers` table with names, email addresses, cities, phone numbers, and registration dates.
- Result: Four rows are inserted successfully.

6. Select All Records from `customers` Table with Specific City:
- The query selects all records from the `customers` table where the city is 'Chicago'.
- Result: One record is found with the name Alice Johnson.

7. Select Specific Columns from `customers` Table with Specific City:
- The query selects the `customer_name` and `email_address` columns from the `customers` table where the city is 'Chicago'.
- Result: One record is found with the name Alice Johnson.

Output are as Follows us 👍





This script creates a database called 'CustomerDatabase', a 'customers' table with columns for customer ID, name, email address, city, phone number, and registration date. It then inserts four sample customer records into the table. You can modify the sample data or add more records as needed.

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.

1. INNER JOIN:

- Use an INNER JOIN to combine data from the 'orders' and 'customers' tables.
- Focus on retrieving information for customers located in a specified region.
- Only include customers who have placed orders.
- Display relevant customer and order information for those in the specified region.

2. LEFT JOIN:

- Use a LEFT JOIN to combine data from the 'orders' and 'customers' tables.
- Ensure that all customers are included in the result, regardless of whether they have placed orders.
- Show customers who have placed orders along with their order details.
- Include customers who have not placed any orders, with order-related fields showing as empty or null.

```
mysql> CREATE TABLE customers (
    ->     customer_id INT PRIMARY KEY,
    ->     customer_name VARCHAR(100),
    ->     region VARCHAR(50),
    ->     email_address VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.05 sec)

mysql> CREATE TABLE orders (
    ->     order_id INT PRIMARY KEY,
    ->     customer_id INT,
    ->     order_date DATE,
    ->     total_amount DECIMAL(10, 2),
    ->     FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
    -> );
Query OK, 0 rows affected (0.08 sec)

mysql>
```

```
mysql> INSERT INTO customers (customer_id, customer_name, region, email_address)
    -> VALUES
    -> (1, 'John Doe', 'North America', 'john.doe@example.com'),
    -> (2, 'Jane Smith', 'Europe', 'jane.smith@example.com'),
    -> (3, 'Alice Johnson', 'Asia', 'alice.johnson@example.com'),
    -> (4, 'Bob Johnson', 'North America', 'bob.johnson@example.com'),
    -> (5, 'Emily Brown', 'Europe', 'emily.brown@example.com');
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> INSERT INTO orders (order_id, customer_id, order_date, total_amount)
    -> VALUES
    -> (101, 1, '2024-05-25', 100.00),
    -> (102, 1, '2024-05-26', 150.00),
    -> (103, 2, '2024-05-27', 200.00),
    -> (104, 4, '2024-05-27', 250.00),
    -> (105, 5, '2024-05-28', 300.00);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT c.customer_id, c.customer_name, c.region, c.email_address, o.order_id, o.order_date, o.total_amount
    -> FROM customers c
    -> LEFT JOIN orders o ON c.customer_id = o.customer_id
    -> WHERE c.region = 'North America' AND o.customer_id IS NULL;
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
| customer_id | customer_name | region        | email_address         | order_id | order_date | total_amount |
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
|          10 | Michael Johnson | North America | michael.johnson@example.com |     NULL | NULL       |         NULL |
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT c.customer_id, c.customer_name, c.region, c.email_address, o.order_id, o.order_date, o.total_amount
    -> FROM customers c
    -> INNER JOIN orders o ON c.customer_id = o.customer_id AND c.region = 'North America'
    -> ;
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
| customer_id | customer_name | region        | email_address         | order_id | order_date | total_amount |
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
|           1 | John Doe      | North America | john.doe@example.com  |      101 | 2024-05-25 |       100.00 |
|           1 | John Doe      | North America | john.doe@example.com  |      102 | 2024-05-26 |       150.00 |
|           4 | Bob Johnson   | North America | bob.johnson@example.com |      104 | 2024-05-27 |       250.00 |
+-------------+---------------+---------------+-----------------------+----------+------------+--------------+
3 rows in set (0.00 sec)
```

```
mysql> INSERT INTO customers (customer_id, customer_name, region, email_address)
    -> VALUES
    -> (10, 'Michael Johnson', 'North America', 'michael.johnson@example.com');
Query OK, 1 row affected (0.02 sec)
```

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.

1. Subquery for Orders Above Average Value:

- Create a subquery to calculate the average order value.
- Use this subquery to filter and find customers who have placed orders with values above the calculated average.
- Ensure that the subquery is correctly integrated to compare each order value against the average.

2. UNION Query:

- Write two SELECT statements that retrieve data with the same number of columns.
- The first SELECT statement can, for instance, select data from one table or a specific condition.
- The second SELECT statement should also select data with the same columns but from a different table or under a different condition.
- Use the UNION keyword to combine the results of these two SELECT statements into a single result set.
- Ensure that both SELECT statements have matching column data types and order.

Subquery to find customers who have placed orders above the average order value

```
mysql> SELECT *
    -> FROM customers
    -> WHERE customer_id IN (
    ->     SELECT customer_id
    ->     FROM orders
    ->     GROUP BY customer_id
    ->     HAVING AVG(total_amount) > (SELECT AVG(total_amount) FROM orders)
    -> );
+-------------+---------------+---------------+-------------------------+
| customer_id | customer_name | region        | email_address           |
+-------------+---------------+---------------+-------------------------+
|           4 | Bob Johnson   | North America | bob.johnson@example.com |
|           5 | Emily Brown   | Europe        | emily.brown@example.com |
+-------------+---------------+---------------+-------------------------+
2 rows in set (0.01 sec)
```

UNION query to combine two SELECT statements

```
mysql> select order_id,total_amount from orders UNION select customer_name,region from customers;
+-----------------+--------------+
| order_id        | total_amount |
+-----------------+--------------+
| 101             | 100.00       |
| 102             | 150.00       |
| 103             | 200.00       |
| 104             | 250.00       |
| 105             | 300.00       |
| John Doe        | North America |
| Jane Smith      | Europe       |
| Alice Johnson   | Asia         |
| Bob Johnson     | North America |
| Emily Brown     | Europe       |
| Michael Johnson | North America |
+-----------------+--------------+
11 rows in set (0.00 sec)
```

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.

1. BEGIN a Transaction:

- Start a new transaction to ensure that a series of operations can be executed as a single unit.
- This helps in maintaining data integrity and consistency.

2. INSERT a New Record into the 'Orders' Table:

- Add a new entry to the 'orders' table with the necessary details.
- This action is part of the current transaction.

3. COMMIT the Transaction:

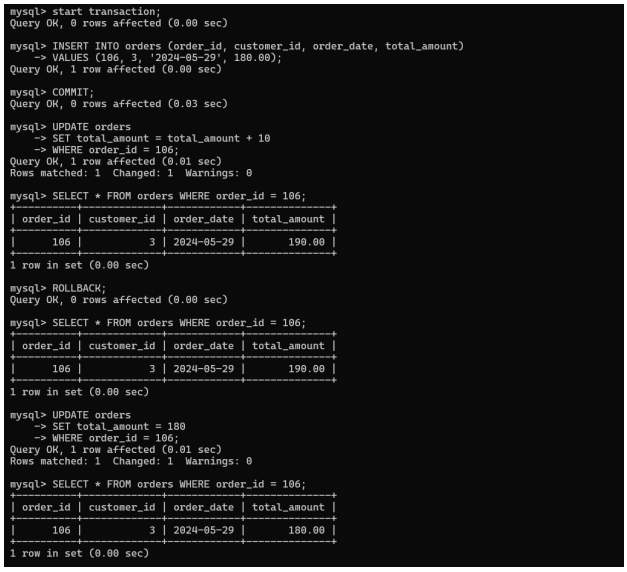
- Finalize the transaction after the insertion.
- This ensures that the new record is permanently saved in the 'orders' table.

4. UPDATE the 'Products' Table:

- Make changes to the 'products' table, such as updating the stock quantity or price.
- This action is part of a new transaction following the previous commit.

5. ROLLBACK the Transaction:

- Undo the changes made in the 'products' table update.
- This reverts the 'products' table to its state before the update was initiated, in case of an error or issue.



```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO orders (order_id, customer_id, order_date, total_amount)
    -> VALUES (106, 3, '2024-05-29', 180.00);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)

mysql> UPDATE orders
    -> SET total_amount = total_amount + 10
    -> WHERE order_id = 106;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM orders WHERE order_id = 106;
+----------+-------------+------------+--------------+
| order_id | customer_id | order_date | total_amount |
+----------+-------------+------------+--------------+
|      106 |           3 | 2024-05-29 |       190.00 |
+----------+-------------+------------+--------------+
1 row in set (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM orders WHERE order_id = 106;
+----------+-------------+------------+--------------+
| order_id | customer_id | order_date | total_amount |
+----------+-------------+------------+--------------+
|      106 |           3 | 2024-05-29 |       190.00 |
+----------+-------------+------------+--------------+
1 row in set (0.00 sec)

mysql> UPDATE orders
    -> SET total_amount = 180
    -> WHERE order_id = 106;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM orders WHERE order_id = 106;
+----------+-------------+------------+--------------+
| order_id | customer_id | order_date | total_amount |
+----------+-------------+------------+--------------+
|      106 |           3 | 2024-05-29 |       180.00 |
+----------+-------------+------------+--------------+
1 row in set (0.00 sec)
```

Involves managing transactions in SQL. First, begin a transaction to ensure a series of operations are executed as a single unit. Insert a new record into the 'orders' table and commit the transaction to save the changes permanently. Next, start another transaction to update the 'products' table. Finally, perform a rollback to undo the update if needed, ensuring the 'products' table reverts to its previous state before the update. This process helps maintain data integrity and allows for controlled, reversible changes to the database.

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.

1. Begin a Transaction:
- Start a new transaction to group multiple operations together.

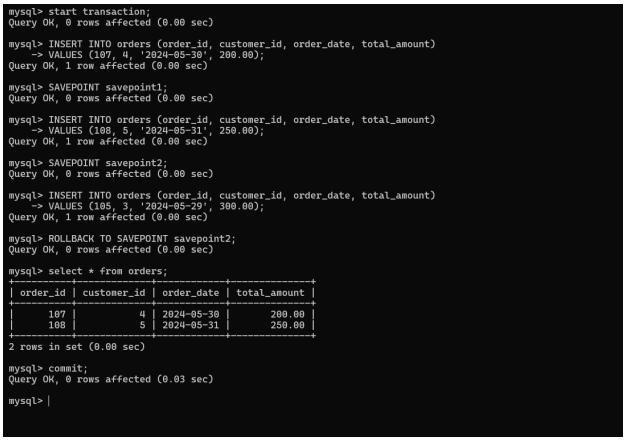
2. Perform a Series of INSERTs into 'Orders':
- Insert several new records into the 'orders' table.
- These operations are part of the ongoing transaction.

3. Set a SAVEPOINT After Each INSERT:
- Create a savepoint after each insert to mark specific points within the transaction.
- These savepoints allow for partial rollbacks to specific stages.

4. Rollback to the Second SAVEPOINT:
- Revert the transaction to the state it was in after the second savepoint.
- This undoes all changes made after the second savepoint.

5. Commit the Overall Transaction:
- Finalize the transaction by saving all changes up to the second savepoint.
- This ensures that the changes up to that point are permanently applied.

Here Output as follows 👍



```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO orders (order_id, customer_id, order_date, total_amount)
    -> VALUES (107, 4, '2024-05-30', 200.00);
Query OK, 1 row affected (0.00 sec)

mysql> SAVEPOINT savepoint1;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO orders (order_id, customer_id, order_date, total_amount)
    -> VALUES (108, 5, '2024-05-31', 250.00);
Query OK, 1 row affected (0.00 sec)

mysql> SAVEPOINT savepoint2;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO orders (order_id, customer_id, order_date, total_amount)
    -> VALUES (105, 3, '2024-05-29', 300.00);
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK TO SAVEPOINT savepoint2;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from orders;
+----------+-------------+------------+--------------+
| order_id | customer_id | order_date | total_amount |
+----------+-------------+------------+--------------+
|      107 |           4 | 2024-05-30 |       200.00 |
|      108 |           5 | 2024-05-31 |       250.00 |
+----------+-------------+------------+--------------+
2 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.03 sec)

mysql>
```

Involves managing a transaction with savepoints. Start a transaction and perform several inserts into the 'orders' table, setting a savepoint after each insert. Rollback to the second savepoint to undo changes made after it. Finally, commit the transaction to save all changes up to the second savepoint, ensuring partial changes are applied while allowing controlled reversibility.

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 crucial components of database management systems (DBMS) that record all changes made to the database. These logs provide a reliable means of data recovery in the event of system failures, ensuring the integrity and consistency of data.

Importance of Transaction Logs

1. Data Integrity: Transaction logs capture every change, maintaining a sequential record.

2. Recovery from Failures: In the event of an unexpected shutdown or system crash, transaction logs

3. Auditing and Monitoring: Logs provide a detailed account of all operations, useful for auditing and identifying the source of issues.

4. Replication and Backup: Transaction logs are essential for database replication and creating consistent backups.

Hypothetical Scenario: Data Recovery Using Transaction Logs

Scenario: An e-commerce company uses a relational database to manage customer orders. The database experiences an unexpected shutdown due to a power failure

Sequence of Events:

1. Order Processing:

- Several new orders are being processed and inserted into the 'orders' table.
- Each transaction is recorded in the transaction log

.

2. Unexpected Shutdown:

- A power failure causes the server to shut down abruptly.
- The database is left in an inconsistent state, with some transactions partially completed.

3. System Restart:

- Upon restarting, the DBMS detects the unclean shutdown.
- The transaction log is used to identify incomplete transactions.

4. Recovery Process:

- Redo Operations: Transactions that were committed but not written to the database are redone, ensuring all committed changes are applied.
- Undo Operations: Transactions that were in progress but not committed are rolled back, discarding any partial changes.

5. Restoration:

- The database is restored to the state it was in just before the shutdown, with all committed transactions intact and no partial transactions affecting the data integrity.

- **Minimal Data Loss**: Thanks to the transaction log, the database can recover without losing any committed transactions.
- **Data Consistency:** The database is returned to a consistent state, maintaining the integrity of customer orders and other critical data.