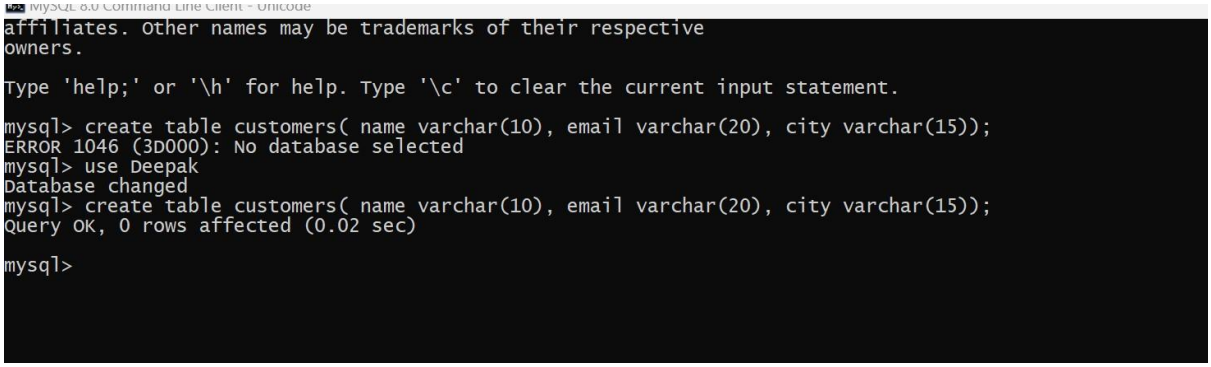
K DEEPAK **ASSIGNMENT –(DAY-10)**

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.

Customers Table Creation :

- Creating an customer table by using entities.



Inserting the values to the tables:

- Insert the values to the tables entites using INSERT commands.

Getting a customer details using select name,email and city :

```
mysql> select name,email from customers where city="chittoor";
+--------+------------------+
| name   | email            |
+--------+------------------+
| Deepak | Deepak@gmail.com |
+--------+------------------+
1 row in set (0.00 sec)

mysql>
```

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.

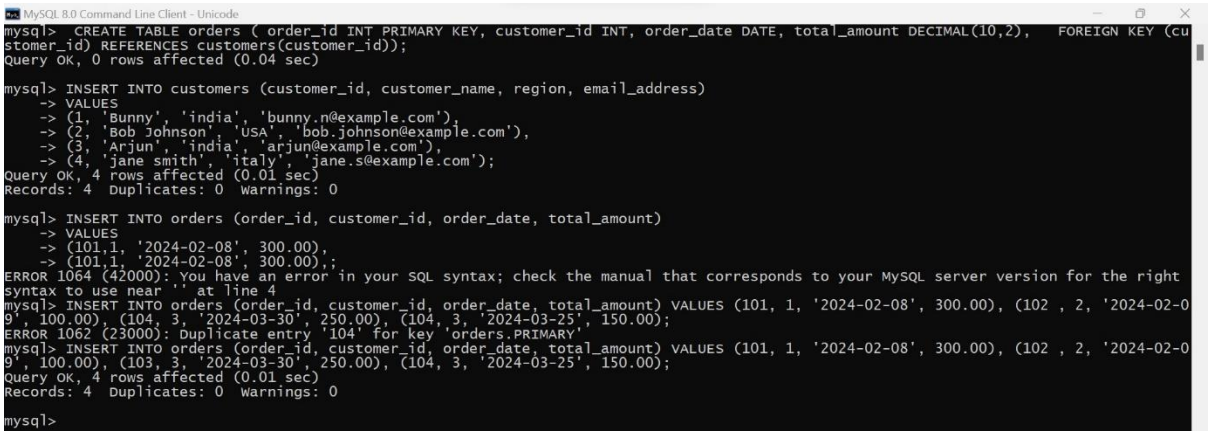
Creating the Tables of ORDERS and CUSTOMERS:

- Creating a table named customers and orders.

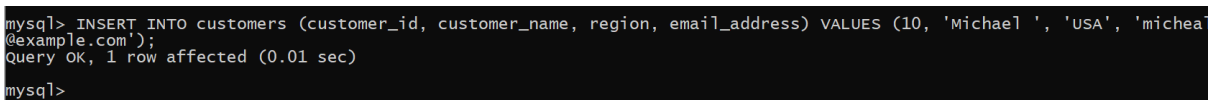
```
mysql>  CREATE TABLE Customers ( customer_id INT PRIMARY KEY, customer_name VARCHAR(255), region VARCHAR(50), email_address VARCHAR(100
));
Query OK, 0 rows affected (0.02 sec)

mysql>  CREATE TABLE orders ( order_id INT PRIMARY KEY, customer_id INT, order_date DATE, total_amount DECIMAL(10,2),   FOREIGN KEY (cu
stomer_id) REFERENCES customers(customer_id));
Query OK, 0 rows affected (0.04 sec)

mysql>
```

Instering records:

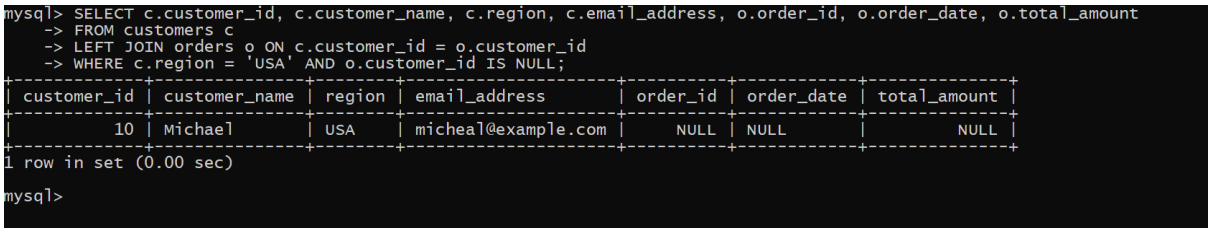
- Here we insert the values to the both tables.



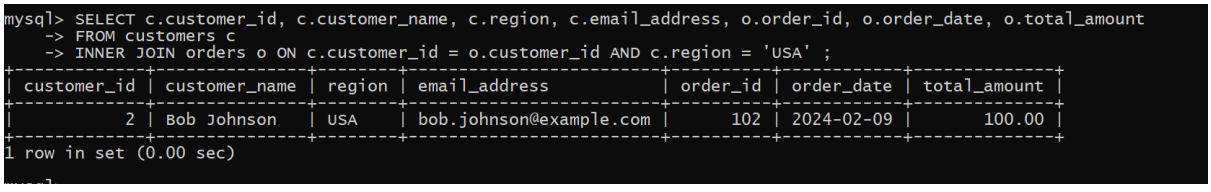
Inserting without order :



Display all the customers including those not in orders:



Query using an INNER JOIN to combine 'orders' and 'customers' tables for customers in a specified region:



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.

Finding customers who have placed orders above the average order value using subquery:

- In the outer query we select all columns "*" from the customers table and WHERE Clause matches in the "customer_id"
- We write the inner query or subquery SELECT customer_id FROM orders table
- Taking GROUP BY clause for results using "customer_id"
- Filtering the groups using "HAVING AVG(total_amount) having an average "total_amount" greater than allover average are selected
- The Nested subquery SELECT AVG(total_amount) FROM orders this calculates the overall average "total_amount" of the orders

```
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       |
+-------------+---------------+--------+---------------------+
|           1 | Bunny         | india  | bunny.n@example.com |
+-------------+---------------+--------+---------------------+
1 row in set (0.01 sec)

mysql>
```

Two combine SELECT statement using UNION:

- SELECT statement selects the "order_id" and "total_amount" columns from the "orders" table.

- The "UNION" operator combines the results of two "select" statements into a single result set. By default, it removes duplicate rows.

- The Second SELECT statement selects customer_name and region columns from the "customers" table

```
mysql> select order_id, total_amount from orders UNION select customer_name, region from customers;
+-------------+--------------+
| order_id    | total_amount |
+-------------+--------------+
| 101         | 300.00       |
| 102         | 100.00       |
| 103         | 250.00       |
| 104         | 150.00       |
| Bunny       | india        |
| Bob Johnson | USA          |
| Arjun       | india        |
| jane smith  | italy        |
| Michael     | USA          |
+-------------+--------------+
9 rows in set (0.00 sec)

mysql>
```

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 ROLLBACKthe transaction.

- Starting the new Transaction and insert a new row into the “orders” table

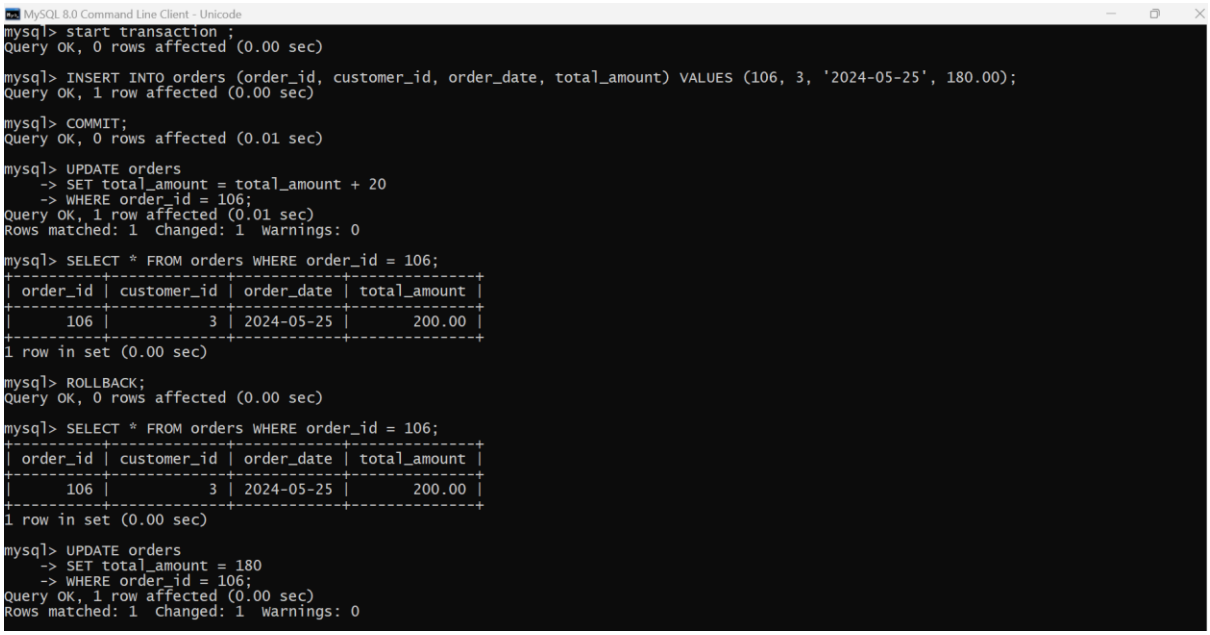
- Commit transaction saves the changes made during the transaction and the next step update the “total_amount” for “order_id” 106 by adding 20 to it.

- The select operation displays the updated row with “total_amount” now with 200.00.

- The rollback transaction undoes the changes made after the last commit. The update was committed before so the rollback doesn’t revert it.

- The SELECT statements displays the same row as the update was committed and rollback has no effect and

- The update resets“total_amount” for “order_id” 106 to 180 and restoring the initial value.



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.

- Starting the new Transaction and insert a new row into the “orders” table
- Creating SAVEPOINT savepoint1 and inserting an new row into the table
- Creating SAVEPOINT savepoint2 and same repeating the last step as inserting a new row.
- ROLLBACK transaction TO SAVEPOINT savepoint in the table and using SELECT and using * to show data on the orders table and commit the table.
- Below attaching the executing commands.

```
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)
```

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.

SIGNIFICANCE OF TRANSACTION LOGS:

Transaction logs are vital for maintaining data integrity and enabling recovery in database management systems. They keep a detailed record of all changes made to the database, which is essential for restoring the database to a consistent state after system failures or data corruption

CORE FUNCTION OF TRANSACTION LOGS:

1.Logging Transcations: Every SQL Server database has a transaction log that logs all transactions and modifications.

2.Maintaining Consistency:

In the event of system failures, the transaction log is crucial for bringing the database back to a consistent state.

OPERATIONS ENAVLED BY THE TRANSACTION LOG:

Recovery of Individual Transactions:

If a ROLLBACK statement is issued or an error occurs (e.g., communication loss with a client), the log records are used to undo incomplete transaction modifications.

Recovery During SQL Server Startup:

After a server failure, SQL Server performs recovery for each database during startup. Changes recorded in the log but not yet written to data files are rolled forward. Incomplete transactions are rolled back to maintain database integrity.

Restoring to the Point of Failure:

After a hardware or disk failure, the database is restored to the most recent state. This process involves restoring the last full database backup, the latest differential backup, and subsequent transaction log backups. The Database Engine reapplies changes from the log to roll forward transactions to the point of failure.

Enhancing high Availability and Disaster Recovery:

Transaction logs are critical for high availability solutions such as Always On availability groups, database mirroring, and log shipping.

Example Scenario : Library Management System

Consider a library system where users borrow and return books. An unexpected shutdown occurs during a busy day:

Prior to Shutdown:

Users borrow books, and these transactions are recorded in the database. The transaction log captures these changes.

During ShutDown:

The server crashes due to a power outage. Uncommitted changes remain in memory, and some transactions are incomplete.

Recovery Steps:

Upon startup, SQL Server uses the transaction log to roll forward committed changes. It identifies and rolls back incomplete transactions. The database is restored to a consistent state.

After Recovery:

Users can continue borrowing and returning books without any data loss.