Name:Vaishali Ramesh Kale

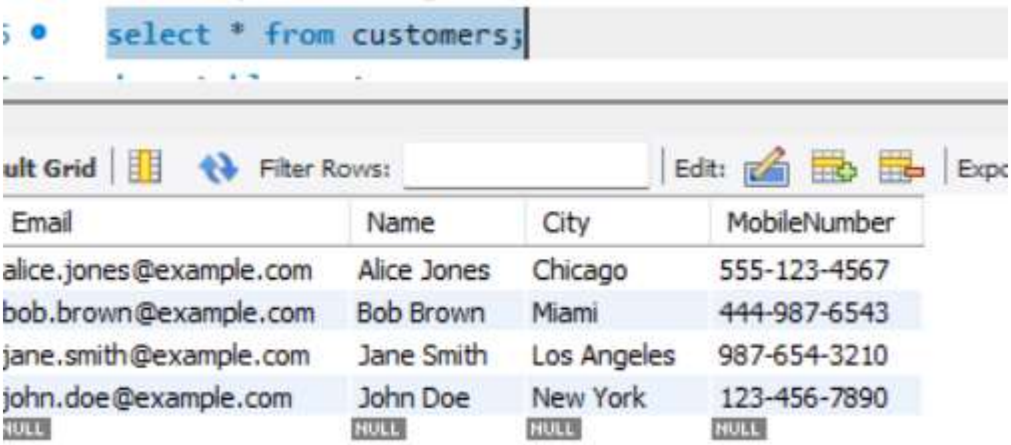
Email id: kalevaishalir16@gmail.com

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: Retrieve All Columns from the customers Table

```
SELECT * FROM customers;
```

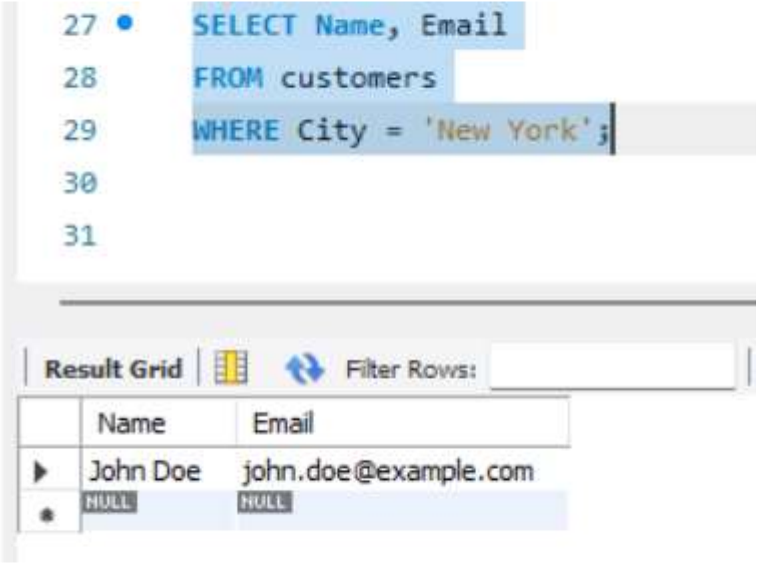


```
select * from customers;
```

Email	Name	City	MobileNumber
alice.jones@example.com	Alice Jones	Chicago	555-123-4567
bob.brown@example.com	Bob Brown	Miami	444-987-6543
jane.smith@example.com	Jane Smith	Los Angeles	987-654-3210
john.doe@example.com	John Doe	New York	123-456-7890
NULL	NULL	NULL	NULL

Step 2: Retrieve Customer Name and Email Address for Customers in a Specific City

```
SELECT Name, Email
FROM customers
WHERE City = 'New York';
```



```
SELECT Name, Email
FROM customers
WHERE City = 'New York';
```

Name	Email
John Doe	john.doe@example.com
NULL	NULL

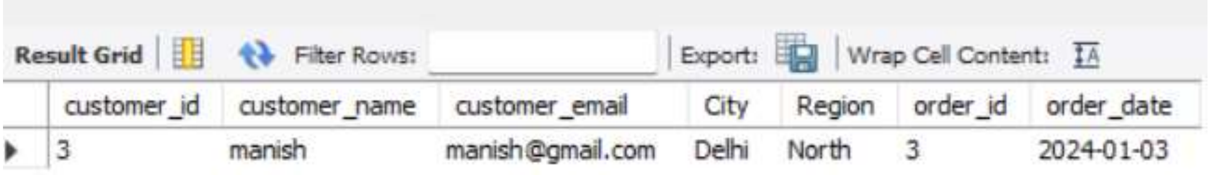
Assignment 2:

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.

Query 1: INNER JOIN to Combine orders and customers for Customers in a Specified Region

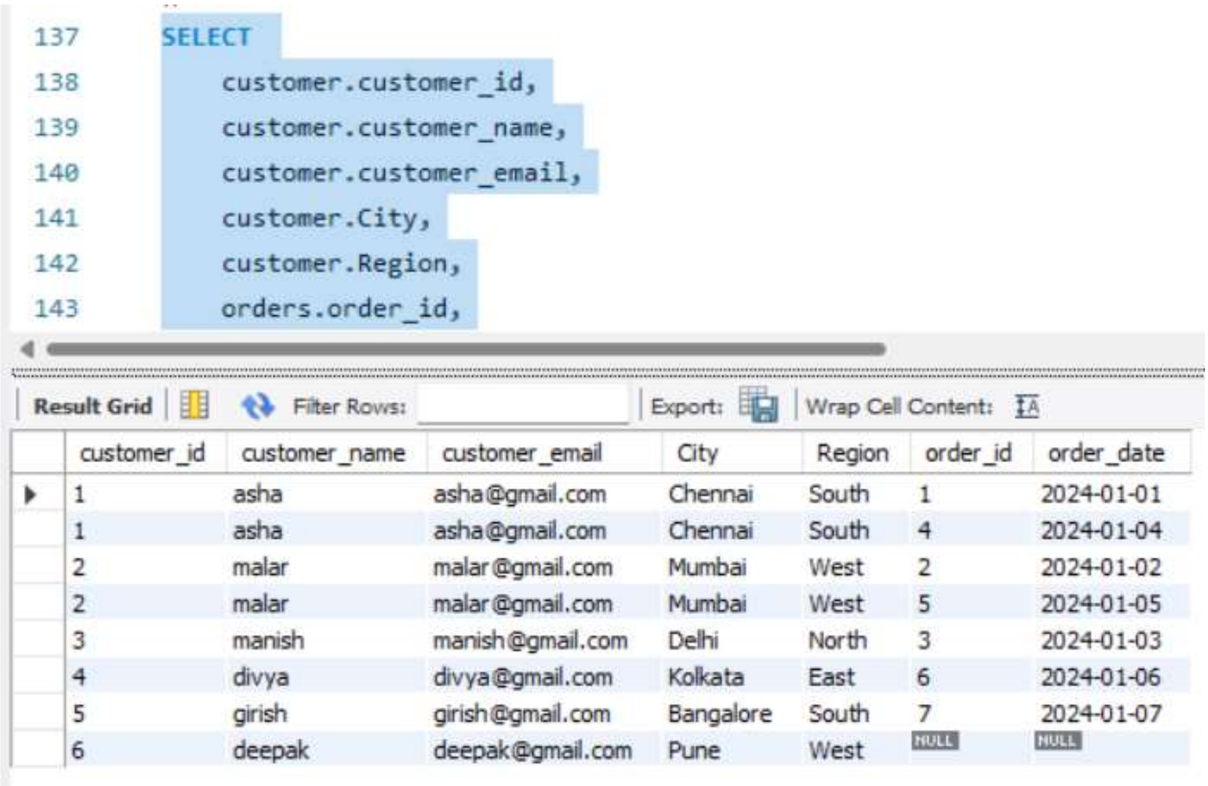
```
SELECT
    customer.customer_id,
    customer.customer_name,
    customer.customer_email ,
    customer.City,
    customer.Region,
    orders.order_id,
    orders.order_date
FROM
    customer
INNER JOIN
    orders ON customer.customer_id = orders.customer_id
WHERE
```

customer.Region = 'North';



customer_id	customer_name	customer_email	City	Region	order_id	order_date
3	manish	manish@gmail.com	Delhi	North	3	2024-01-03

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



```
SELECT
    customer.customer_id,
    customer.customer_name,
    customer.customer_email,
    customer.City,
    customer.Region,
    orders.order_id,
```

customer_id	customer_name	customer_email	City	Region	order_id	order_date
1	asha	asha@gmail.com	Chennai	South	1	2024-01-01
1	asha	asha@gmail.com	Chennai	South	4	2024-01-04
2	malar	malar@gmail.com	Mumbai	West	2	2024-01-02
2	malar	malar@gmail.com	Mumbai	West	5	2024-01-05
3	manish	manish@gmail.com	Delhi	North	3	2024-01-03
4	divya	divya@gmail.com	Kolkata	East	6	2024-01-06
5	girish	girish@gmail.com	Bangalore	South	7	2024-01-07
6	deepak	deepak@gmail.com	Pune	West	NULL	NULL

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 Who Have Placed Orders Above the Average Order Value

Step 1: Alter the orders Table to Add order_value Column

```
ALTER TABLE orders

ADD COLUMN order_value DECIMAL(10, 2);
```

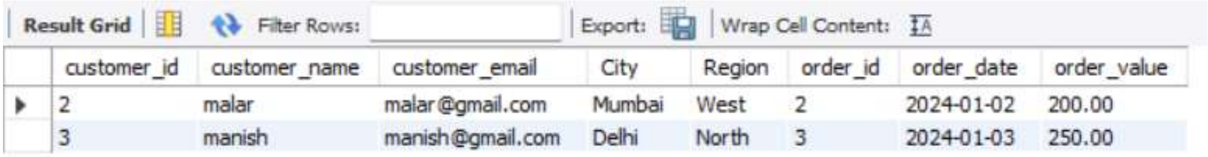
Step 2: Insert Example Data for order_value

```
UPDATE orders

SET order_value = CASE

    WHEN order_id = 1 THEN 150.00

    WHEN order_id = 2 THEN 200.00

    WHEN order_id = 3 THEN 250.00

    WHEN order_id = 4 THEN 100.00

END;
```

Step 3: Subquery to Find Customers Who Have Placed Orders Above the Average Order Value

```
SELECT

    customer.customer_id,

    customer.customer_name,

    customer.customer_email,

    customer.City,

    customer.Region,

    orders.order_id,

    orders.order_date,

    orders.order_value

FROM
```

```
    customer
INNER JOIN
    orders ON customer.customer_id = orders.customer_id
WHERE
    orders.order_value > (SELECT AVG(order_value) FROM orders);
```

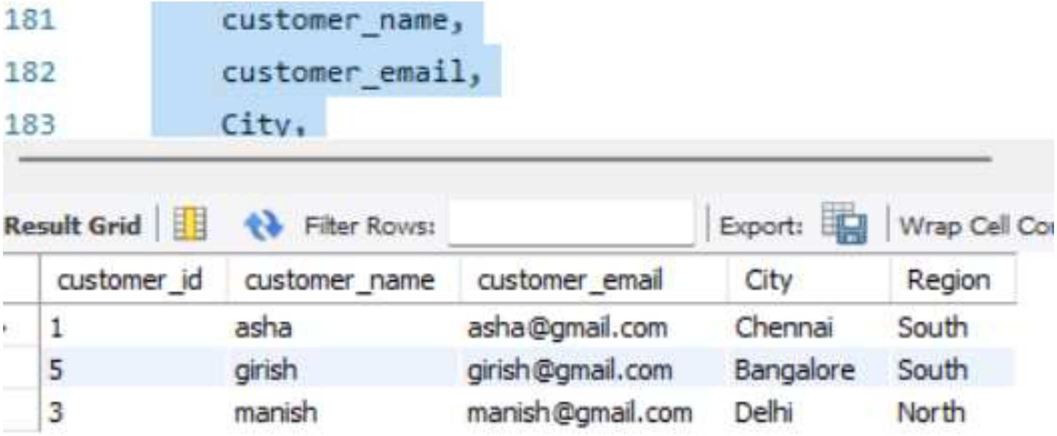


customer_id	customer_name	customer_email	City	Region	order_id	order_date	order_value
2	malar	malar@gmail.com	Mumbai	West	2	2024-01-02	200.00
3	manish	manish@gmail.com	Delhi	North	3	2024-01-03	250.00

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

Step 4: UNION Query

```
SELECT  customer_id, customer_name,   customer_email,
    City,Region
FROM  customer
WHERE   Region = 'South'
UNION SELECT  customer_id,  customer_name,  customer_email,
    City, Region
FROM customer
WHERE Region = 'North';
```



```
        customer_name,
        customer_email,
        City,
```

customer_id	customer_name	customer_email	City	Region
1	asha	asha@gmail.com	Chennai	South
5	girish	girish@gmail.com	Bangalore	South
3	manish	manish@gmail.com	Delhi	North

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. **Transaction to Insert a New Record into the orders Table and Commit**

-- Start the transaction

BEGIN TRANSACTION;

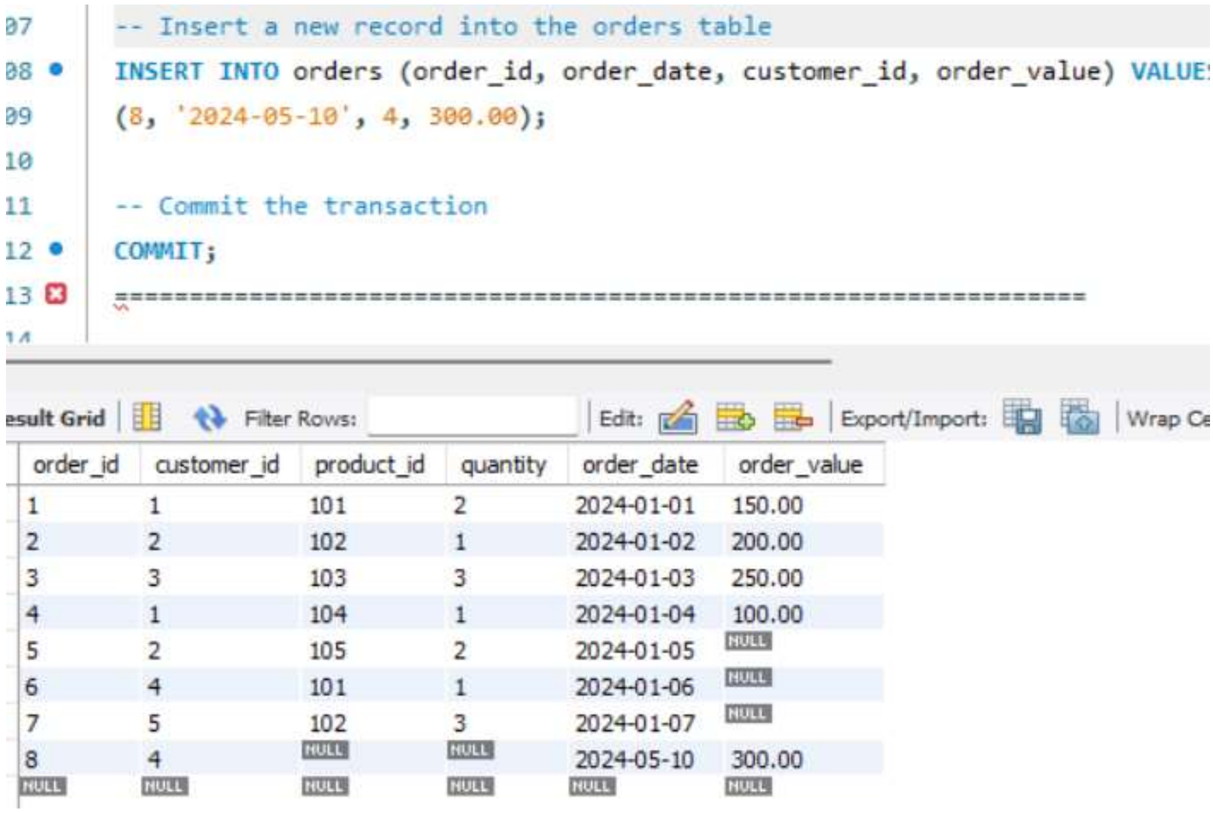
-- Insert a new record into the orders table

INSERT INTO orders (order_id, order_date, customer_id, order_value) VALUES

(5, '2024-05-10', 4, 300.00);

-- Commit the transaction

COMMIT;



```
-- Insert a new record into the orders table
INSERT INTO orders (order_id, order_date, customer_id, order_value) VALUE
(8, '2024-05-10', 4, 300.00);

-- Commit the transaction
COMMIT;
==================================================================
```

order_id	customer_id	product_id	quantity	order_date	order_value
1	1	101	2	2024-01-01	150.00
2	2	102	1	2024-01-02	200.00
3	3	103	3	2024-01-03	250.00
4	1	104	1	2024-01-04	100.00
5	2	105	2	2024-01-05	NULL
6	4	101	1	2024-01-06	NULL
7	5	102	3	2024-01-07	NULL
8	4	NULL	NULL	2024-05-10	300.00
NULL	NULL	NULL	NULL	NULL	NULL

Step 4: Transaction to Update the products Table and Rollback

-- Start the transaction

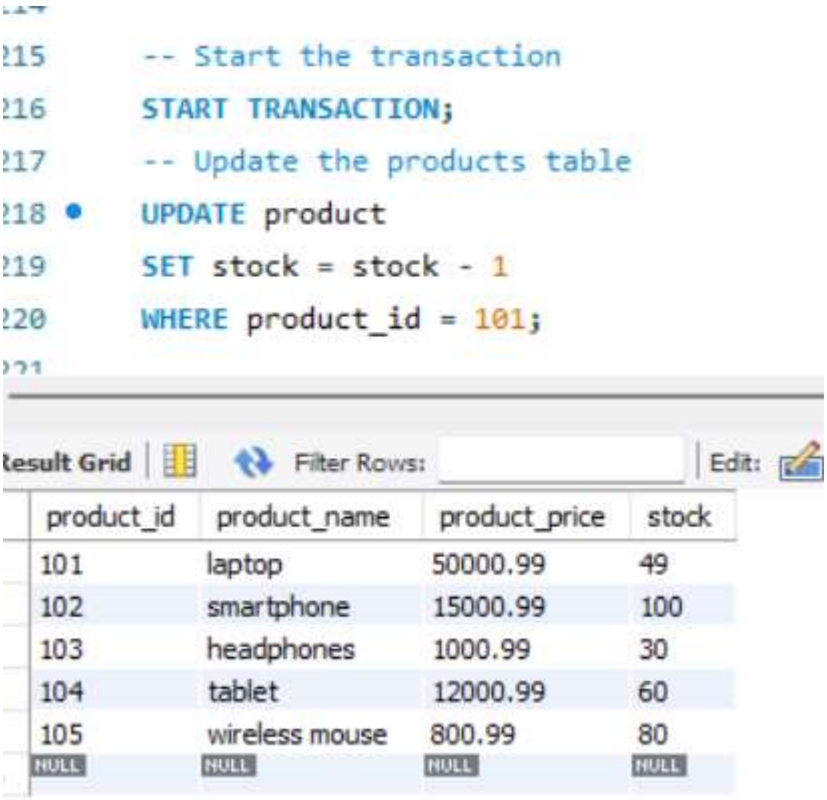
BEGIN TRANSACTION;

-- Update the products table

UPDATE products

SET stock = stock - 1

WHERE product_id = 1;



```
-- Start the transaction
START TRANSACTION;
-- Update the products table
UPDATE product
SET stock = stock - 1
WHERE product_id = 101;
```

product_id	product_name	product_price	stock
101	laptop	50000.99	49
102	smartphone	15000.99	100
103	headphones	1000.99	30
104	tablet	12000.99	60
105	wireless mouse	800.99	80
NULL	NULL	NULL	NULL

-- Rollback the transaction

ROLLBACK;

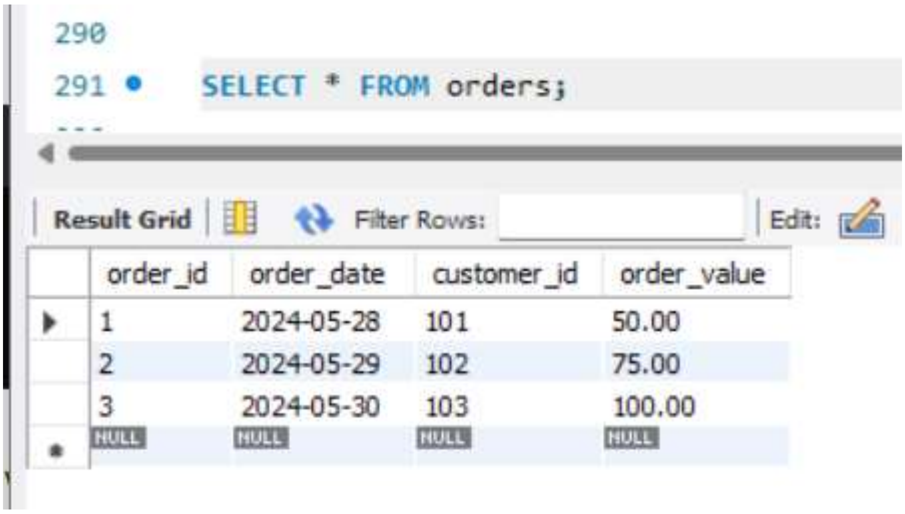
--

--

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.

Solution:→



```
SELECT * FROM orders;
```

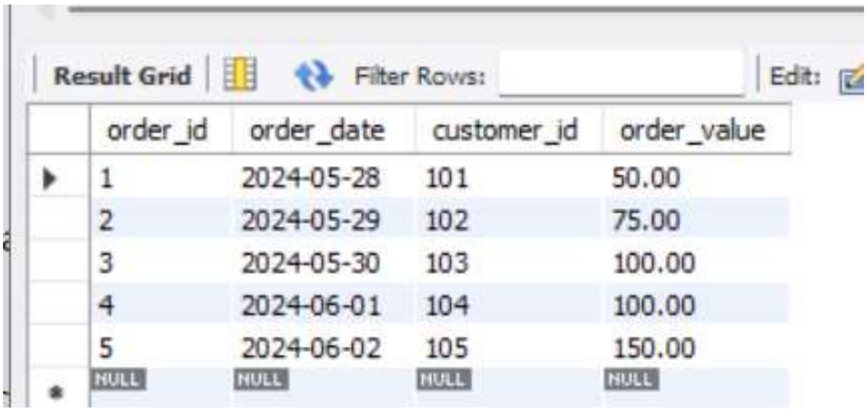
order_id	order_date	customer_id	order_value
1	2024-05-28	101	50.00
2	2024-05-29	102	75.00
3	2024-05-30	103	100.00
NULL	NULL	NULL	NULL

PART 1: Begin a transaction, perform a series of INSERTs into 'orders', setting a SAVEPOINT after each, rollback to the second SAVEPOINT

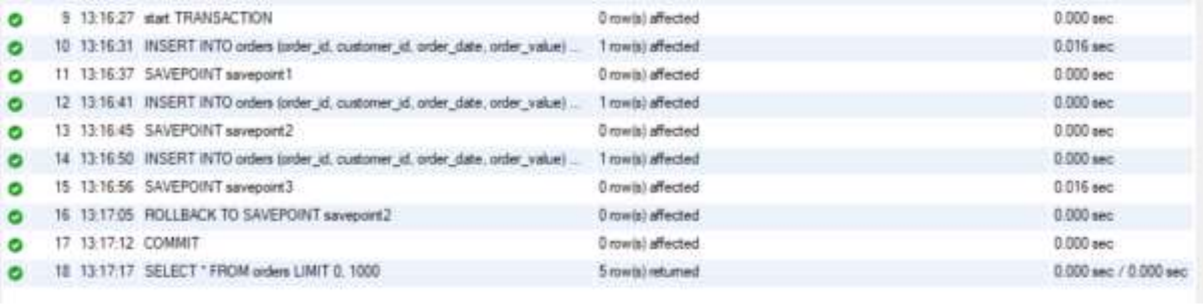
```
start TRANSACTION;

SELECT * FROM orders;

-- Insert into 'orders' and set the first savepoint

INSERT INTO orders (order_id, customer_id, order_date, order_value) VALUES (4, 104, '2024-06-01', 100.00);

SAVEPOINT savepoint1;


-- Insert into 'orders' and set the second savepoint

INSERT INTO orders (order_id, customer_id, order_date, order_value) VALUES (5, 105, '2024-06-02', 150.00);

SAVEPOINT savepoint2;


-- Insert into 'orders' and set the third savepoint

INSERT INTO orders (order_id, customer_id, order_date, order_value) VALUES (6, 106, '2024-06-03', 200.00);

SAVEPOINT savepoint3;


-- Rollback to the second savepoint

ROLLBACK TO SAVEPOINT savepoint2;
```

-- Commit the transaction

COMMIT;



	order_id	order_date	customer_id	order_value
▶	1	2024-05-28	101	50.00
	2	2024-05-29	102	75.00
	3	2024-05-30	103	100.00
	4	2024-06-01	104	100.00
	5	2024-06-02	105	150.00
*	NULL	NULL	NULL	NULL



```
9  13:16:27 start TRANSACTION                                                         0 row(s) affected   0.000 sec
10 13:16:31 INSERT INTO orders (order_id, customer_id, order_date, order_value) ...   1 row(s) affected   0.016 sec
11 13:16:37 SAVEPOINT savepoint1                                                      0 row(s) affected   0.000 sec
12 13:16:41 INSERT INTO orders (order_id, customer_id, order_date, order_value) ...   1 row(s) affected   0.000 sec
13 13:16:45 SAVEPOINT savepoint2                                                      0 row(s) affected   0.000 sec
14 13:16:50 INSERT INTO orders (order_id, customer_id, order_date, order_value) ...   1 row(s) affected   0.000 sec
15 13:16:56 SAVEPOINT savepoint3                                                      0 row(s) affected   0.016 sec
16 13:17:05 ROLLBACK TO SAVEPOINT savepoint2                                          0 row(s) affected   0.000 sec
17 13:17:12 COMMIT                                                                    0 row(s) affected   0.000 sec
18 13:17:17 SELECT * FROM orders LIMIT 0, 1000                                        5 row(s) returned   0.000 sec / 0.000 sec
```

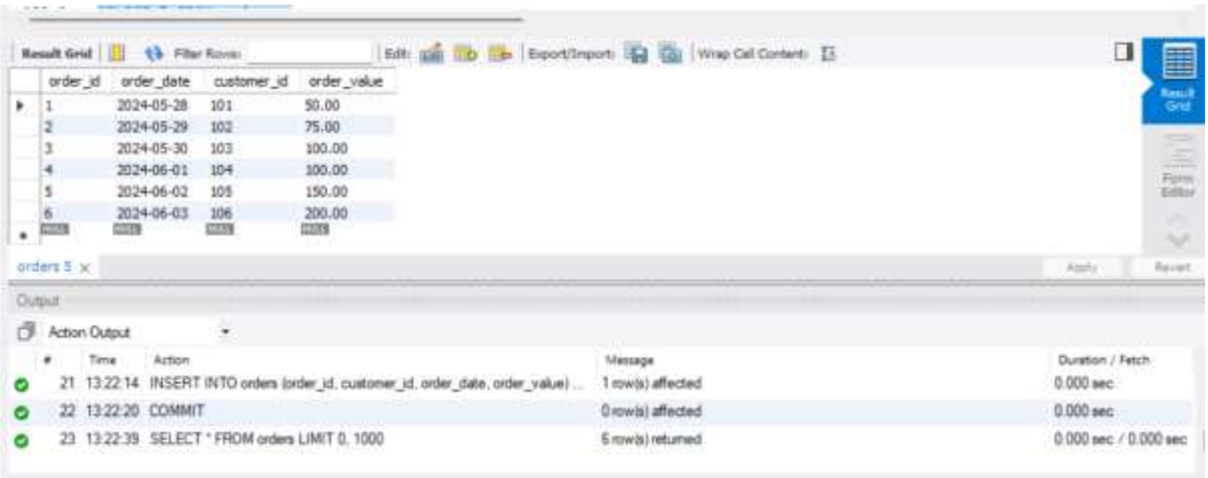
Part 2: COMMITED the overall transaction

INSERT INTO orders (order_id, customer_id, order_date, order_value) VALUES (6, 106, '2024-06-03', 200.00);

-- Commit the transaction

COMMIT;

SELECT * FROM ORDERS;



order_id	order_date	customer_id	order_value
1	2024-05-28	101	50.00
2	2024-05-29	102	75.00
3	2024-05-30	103	100.00
4	2024-06-01	104	100.00
5	2024-06-02	105	150.00
6	2024-06-03	106	200.00
NULL	NULL	NULL	NULL

#	Time	Action	Message	Duration / Fetch
21	13:22:14	INSERT INTO orders (order_id, customer_id, order_date, order_value) ...	1 row(s) affected	0.000 sec
22	13:22:20	COMMIT	0 row(s) affected	0.000 sec
23	13:22:39	SELECT * FROM orders LIMIT 0, 1000	6 row(s) returned	0.000 sec / 0.000 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

Report on the Use of Transaction Logs for Data Recovery

Introduction

Transaction logs are a fundamental component of database management systems (DBMS). They are used to ensure data integrity and to recover data in the event of a system failure. A transaction log records all the changes made to the database, allowing for recovery to a consistent state after unexpected events like system crashes or power failures.

How Transaction Logs Work

A transaction log records each transaction executed by the DBMS and the state of the database before and after the transaction. It typically includes:

Begin Transaction: Marks the start of a transaction.

Transaction Data: Logs the actual operations performed by the transaction, such as insertions, updates, deletions, and the before and after states of the data.

Commit Transaction: Marks the successful completion of a transaction.

Rollback Transaction: Marks the rollback of a transaction, undoing its changes.

Transaction logs ensure that even if a failure occurs, the DBMS can use the logs to redo committed transactions and undo uncommitted ones, ensuring the database remains consistent.

Data Recovery Using Transaction Logs

Transaction logs are instrumental in various recovery scenarios:

System Crash Recovery: If the system crashes, the transaction log can be used to restore the database to the last consistent state. The DBMS will replay committed transactions and undo uncommitted transactions based on the logs.

Point-in-Time Recovery: Transaction logs can be used to restore the database to a specific point in time, which is particularly useful in cases of accidental data deletion or corruption.

Media Failure Recovery: In the event of media failure, such as a disk crash, transaction logs can be used in conjunction with backups to restore the database.

Hypothetical Scenario: Data Recovery After Unexpected Shutdown

Scenario Description

Imagine a retail company, XYZ Retail, uses a DBMS to manage its sales transactions. One busy shopping day, the power suddenly goes out, causing an unexpected system shutdown. At the time of the outage, several transactions were being processed.

Using Transaction Logs for Recovery

Transaction Log Contents at the Time of Shutdown:

Transaction 1001: Inserted a new sales order, but the transaction was not committed.

Transaction 1002: Updated the stock levels for a product and committed successfully.

Transaction 1003: Deleted an old sales record, but the transaction was in progress and not committed.

Steps for Data Recovery:

Step 1: Analyse the Transaction Log: After the system is restored, the DBMS will analyze the transaction log to determine the state of each transaction at the time of the crash.

Step 2: Rollback Uncommitted Transactions: The DBMS will identify Transaction 1001 and Transaction 1003 as uncommitted and will rollback these transactions to ensure the database is not left in an inconsistent state.

Step 3: Redo Committed Transactions: The DBMS will identify Transaction 1002 as committed. It will ensure that all changes made by this transaction are applied to the database, ensuring the transaction's effects are preserved.

Outcome:

The sales order from Transaction 1001 is rolled back, ensuring no partial data is left in the database.

The stock update from Transaction 1002 remains in place, as it was committed successfully before the crash.

The deletion from Transaction 1003 is rolled back, as it was in progress and not committed at the time of the shutdown.