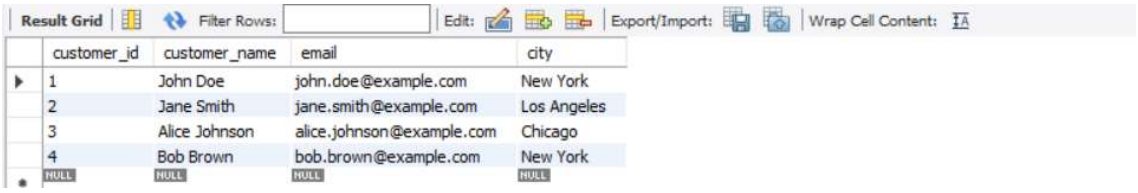
Assignment 1: Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.

Answer: first of all , I have created a Customer table and inserted the value to the table .

SELECT query to retrieve all columns from a 'customers' table

SELECT * FROM customers;



customer_id	customer_name	email	city
1	John Doe	john.doe@example.com	New York
2	Jane Smith	jane.smith@example.com	Los Angeles
3	Alice Johnson	alice.johnson@example.com	Chicago
4	Bob Brown	bob.brown@example.com	New York
NULL	NULL	NULL	NULL

modify it to return only the customer name and email address for customers in a specific city

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

customer_name	email
John Doe	john.doe@example.com
Bob Brown	bob.brown@example.com

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 order.

Answer:

Step 1: Create the ‘order’ table.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

Step 2: Insert data into ‘Order’ table

```
INSERT INTO orders (customer_id, order_date, amount)
VALUES
```

(1, '2024-01-10', 250.00),
(2, '2024-02-15', 300.00),
(1, '2024-03-20', 150.00),
(3, '2024-04-05', 400.00);

Step 3: INNER JOIN to combine 'orders' and 'customer' table for a specified region

```
SELECT c.customer_name, c.email, o.order_id, o.order_date, o.amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.city = 'New York';
```

customer_name	email	order_id	order_date	amount
John Doe	john.doe@example.com	1	2024-01-10	250.00
John Doe	john.doe@example.com	3	2024-03-20	150.00

Step 4: LEFT JOIN to Display All Customers Including Those Without an Order

This query will display all customers, including those who have not placed any orders:

```
SELECT c.customer_name, c.email, o.order_id, o.order_date, o.amount
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customer_name	email	order_id	order_date	amount
John Doe	john.doe@example.com	1	2024-01-10	250.00
John Doe	john.doe@example.com	3	2024-03-20	150.00
Jane Smith	jane.smith@example.com	2	2024-02-15	300.00
Alice Johnson	alice.johnson@example.com	4	2024-04-05	400.00
Bob Brown	bob.brown@example.com	NULL	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.

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

Step 1: Calculate the Average Order Value

First, we'll calculate the average order value from the orders table.

```
SELECT AVG(amount) AS average_order_value FROM orders;
```

Step 2: Use the Subquery to Find Orders Above the Average Order Value

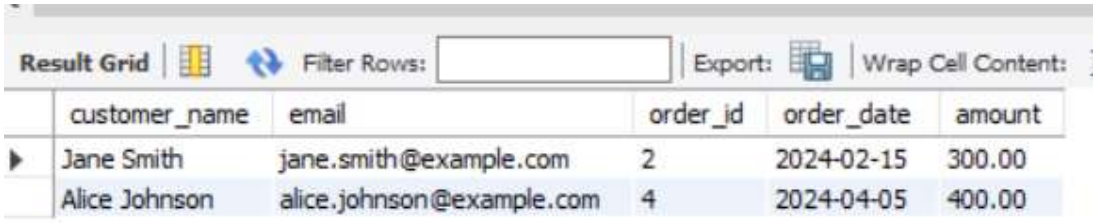
Next, we will use this average value in a subquery to find customers who have placed orders above this average.

SELECT c.customer_name, c.email, o.order_id, o.order_date, o.amount

FROM customers c

INNER JOIN orders o ON c.customer_id = o.customer_id

WHERE o.amount > (SELECT AVG(amount) FROM orders);



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

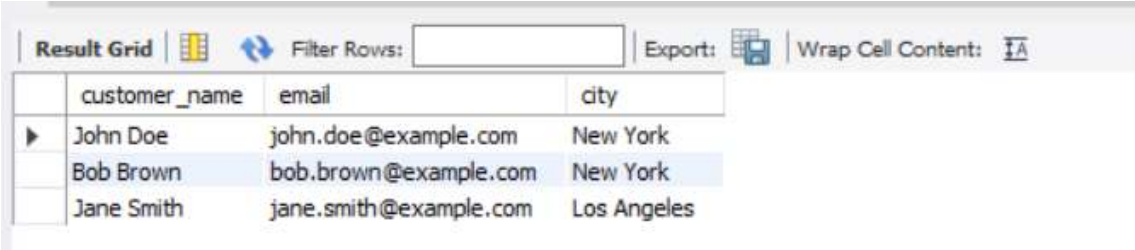
For this part, we'll create two simple SELECT statements that return the same number of columns and combine them using a UNION.

Example: Combine Customer Names and Emails from Two Different Cities

SELECT customer_name, email, city FROM customers WHERE city = 'New York'

UNION

SELECT customer_name, email, city FROM customers WHERE city = 'Los Angeles';



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.

Answer: -- Begin the first transaction

```
START TRANSACTION;
```

-- Insert a new record into the orders table

```
INSERT INTO orders (customer_id, order_date, amount)
VALUES (4, '2024-05-25', 150.00);
```

-- Commit the transaction

```
COMMIT;
```

-- Begin the second transaction

```
START TRANSACTION;
```

-- Update the products table

```
UPDATE products
SET price = price * 1.10
WHERE product_id = 5;
```

-- Rollback the transaction

```
ROLLBACK;
```

The first transaction starts, inserts a new record into the orders table, and commits the changes.

The second transaction starts, attempts to update the products table, but then rolls back, discarding the update and leaving the products table unchanged.

These operations demonstrate how transactions can be used to ensure atomicity in database operations, ensuring that each transaction is completed fully or not at all.

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.

Answer: We will use SQL transaction control commands to achieve this:

- Begin a transaction.
- Perform a series of INSERTs into the orders table.
- Set a SAVEPOINT after each INSERT.
- Rollback to the second SAVEPOINT.
- Commit the overall transaction.

Script:

```
START TRANSACTION;


-- Insert into the orders table and set the first SAVEPOINT

INSERT INTO orders (customer_id, order_date, amount) VALUES (1, '2024-05-25', 100.00);

SAVEPOINT savepoint1;


-- Insert into the orders table and set the second SAVEPOINT

INSERT INTO orders (customer_id, order_date, amount) VALUES (2, '2024-05-26', 200.00);

SAVEPOINT savepoint2;


-- Insert into the orders table and set the third SAVEPOINT
```

```
INSERT INTO orders (customer_id, order_date, amount) VALUES (3, '2024-05-27', 300.00);

SAVEPOINT savepoint3;


-- Rollback to the second SAVEPOINT

ROLLBACK TO savepoint2;


-- Commit the overall transaction

COMMIT;
```

After executing this script:

The order with customer_id = 1 and amount = 100.00 will be committed.

The order with customer_id = 2 and amount = 200.00 will be committed.

The order with customer_id = 3 and amount = 300.00 will be rolled back and not committed.

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.

Transaction logs are crucial components of modern database management systems (DBMS). They record all transactions and changes made to the database, providing a reliable way to recover data in the event of system failures, crashes, or other unforeseen issues. This report explores the role of transaction logs in data recovery and presents a hypothetical scenario to illustrate their importance.

Transaction Logs Overview

A transaction log is a sequential record of all transactions that update data in a database. It includes details such as:

- Transaction ID
- Type of operation (INSERT, UPDATE, DELETE)

- Before and after values of the modified data
- Timestamps
- Commit and rollback information

Importance of Transaction Logs

1. Data Integrity: Ensures that all committed transactions are preserved, maintaining the integrity of the database.
2. Crash Recovery: In the event of a system crash, transaction logs can be used to redo committed transactions and undo uncommitted transactions, restoring the database to a consistent state.
3. Point-in-Time Recovery: Allows for the restoration of the database to a specific point in time by replaying the transaction log up to a desired moment.
4. Audit Trail: Provides a historical record of all changes made to the database, useful for auditing and compliance purposes.

Hypothetical Scenario: Transaction Log for Data Recovery

Scenario Description

Imagine a financial services company, FinSecure, that handles transactions for thousands of customers. FinSecure uses a database to manage account balances, transactions, and customer data. On a busy Friday afternoon, the database server unexpectedly shuts down due to a hardware failure. When the server is restarted, the database is found to be in an inconsistent state.

Role of Transaction Logs

1. Detection of Inconsistent State: Upon restart, the DBMS detects that the database is in an inconsistent state by checking the transaction log. It identifies that some transactions were in progress and not committed at the time of the crash.
2. Undo Uncommitted Transactions: The transaction log shows several transactions that were not completed. The DBMS uses the log to roll back these transactions, ensuring that no partial updates are left in the database.
3. Redo Committed Transactions: The transaction log also contains records of transactions that were committed but not yet written to the main database files at the time of the crash. The DBMS replays these transactions to ensure that all committed transactions are reflected in the database.
4. Point-in-Time Recovery: If FinSecure decides to restore the database to a state just before the failure, the transaction log can be used to replay

transactions up to that point, ensuring that all valid transactions are included and no data is lost.