Name: Swarnali Ghosh

Java Pre-Skilling Training Session

Assignment -4.2

Module-4

Mail-id: swarnalighosh666@gmail.com

ASSIGNMENT-1

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

SOLUTION:

1.Firstly I have a created a database name 'wipro_swarnali', where I will perform all the functions-

```
CREATE DATABASE wipro_swarnali;
use wipro_swarnali;
```

Output

Action Output

#	Time	Action
5	18:27:39	CREATE DATABASE wipro_swarnali
6	18:28:20	use wipro_swarnali

2. Created a table named "customers" →

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    email_address VARCHAR(100),
    city VARCHAR(100),
    phone_number VARCHAR(20)
);
```

3. Through select query, checking our creation:

```sql
select * from customers;
```

Result Grid | Filter Rows: | Edit: | Export/Import

customer_id	customer_name	email_address	city	phone_number
NULL	NULL	NULL	NULL	NULL

4. Inserting value into this: -

```sql
INSERT INTO customers (customer_id,customer_name, email_address, city, phone_number)
VALUES
(1,'Priya', 'priya123@gmail.com', 'Kolkata', '9987654566'),
(2,'Sweety', 'sweetyghosh123@gmail.com', 'Kolkata', '9876543210'),
(3,'Tina', 'alice@yahoo.com', 'Siliguri', '9765434265'),
(4,'Sunny', 'sunny@example.com', 'Durgapur', '9076456749');

select * from customers;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

customer_id	customer_name	email_address	city	phone_number
1	Priya	priya123@gmail.com	Kolkata	9987654566
2	Sweety	sweetyghosh123@gmail.com	Kolkata	9876543210
3	Tina	alice@yahoo.com	Siliguri	9765434265
4	Sunny	sunny@example.com	Durgapur	9076456749
NULL	NULL	NULL	NULL	NULL

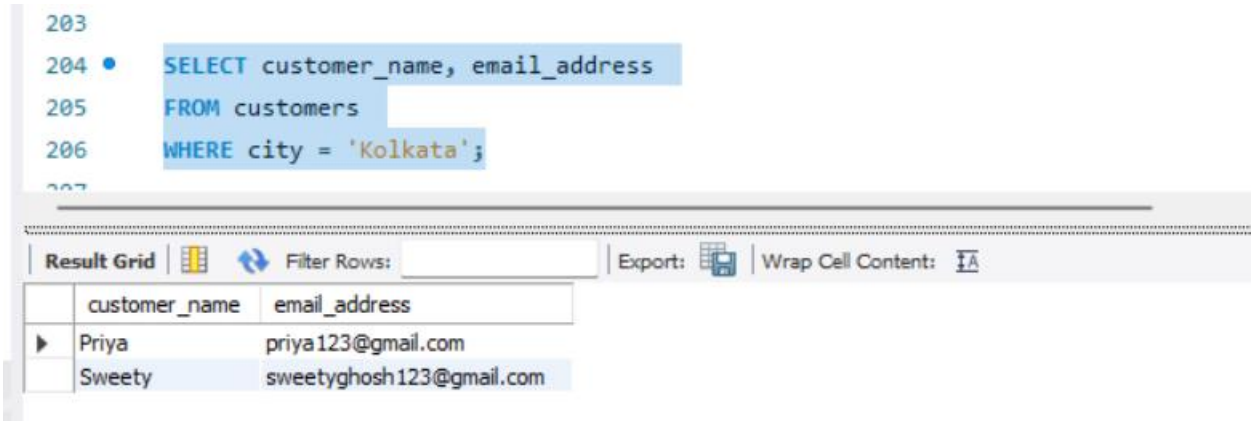
5. We can modify it to return only the customer's name and email address for customers in a specific city (let's say the city is 'Kolkata'):

→

SELECT customer_name, email_address

FROM customers

WHERE city = 'Kolkata';



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.

SOLUTION:

Created another table named “orders”->

```sql
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
INSERT INTO orders (order_id, customer_id, order_date, amount)
VALUES
(101, 1, '2024-05-01', 150.00),
(201, 2, '2024-05-15', 200.00),
(301, 3, '2024-05-10', 300.00),
(401, 4, '2024-05-20', 250.00);

select * from orders;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

order_id	customer_id	order_date	amount
101	1	2024-05-01	150.00
201	2	2024-05-15	200.00
301	3	2024-05-10	300.00
401	4	2024-05-20	250.00
NULL	NULL	NULL	NULL

Inserted one more row in customers table-

```sql
select * from customers;

INSERT INTO customers (customer_id,customer_name, email_address, city, phone_number)
VALUES
(5,'Ram', 'ram@yahoo.in', 'Durgapur', '8976056749');
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

customer_id	customer_name	email_address	city	phone_number
1	Priya	priya123@gmail.com	Kolkata	9987654566
2	Sweety	sweetyghosh123@gmail.com	Kolkata	9876543210
3	Tina	alice@yahoo.com	Siliguri	9765434265
4	Sunny	sunny@example.com	Durgapur	9076456749
5	Ram	ram@yahoo.in	Durgapur	8976056749
NULL	NULL	NULL	NULL	NULL

INNER JOIN Query

To combine the orders and customers tables for customers in a specified region using an INNER JOIN, we will match customer_id and filter by a specific region (e.g., cities in West Bengal like 'Kolkata', 'Siliguri', 'Durgapur'):

SELECT customers.customer_id, customers.customer_name, customers.email_address, customers.city, customers.phone_number, orders.order_id, orders.order_date, orders.amount

FROM customers

INNER JOIN orders ON customers.customer_id = orders.customer_id

WHERE customers.city IN ('Kolkata', 'Siliguri', 'Durgapur');

```
SELECT customers.customer_id, customers.customer_name,
customers.email_address, customers.city, customers.phone_number,
orders.order_id, orders.order_date, orders.amount
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.city IN ('Kolkata', 'Siliguri', 'Durgapur');
```

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

customer_id	customer_name	email_address	city	phone_number	order_id	order_date	amount
1	Priya	priya123@gmail.com	Kolkata	9987654566	101	2024-05-01	150.00
2	Sweety	sweetyghosh123@gmail.com	Kolkata	9876543210	201	2024-05-15	200.00
3	Tina	alice@yahoo.com	Siliguri	9765434265	301	2024-05-10	300.00
4	Sunny	sunny@example.com	Durgapur	9076456749	401	2024-05-20	250.00

Note: Ram will not appear in the INNER JOIN results since he does not have any orders.

LEFT JOIN to display all customers including those without orders.

We will join the tables on `customer_id` and include all customers, regardless of whether they have matching rows in the `orders` table:

SELECT customers.customer_id, customers.customer_name, customers.email_address, customers.city, customers.phone_number, orders.order_id, orders.order_date, orders.amount

FROM customers

LEFT JOIN orders ON customers.customer_id = orders.customer_id

WHERE customers.city IN ('Kolkata', 'Siliguri', 'Durgapur');

```sql
SELECT customers.customer_id, customers.customer_name,
customers.email_address, customers.city, customers.phone_number,
orders.order_id, orders.order_date, orders.amount
FROM customers
LEFT JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.city IN ('Kolkata', 'Siliguri', 'Durgapur');
```

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

customer_id	customer_name	email_address	city	phone_number	order_id	order_date	amount
1	Priya	priya123@gmail.com	Kolkata	9987654566	101	2024-05-01	150.00
2	Sweety	sweetyghosh123@gmail.com	Kolkata	9876543210	201	2024-05-15	200.00
3	Tina	alice@yahoo.com	Siliguri	9765434265	301	2024-05-10	300.00
4	Sunny	sunny@example.com	Durgapur	9076456749	401	2024-05-20	250.00
5	Ram	ram@yahoo.in	Durgapur	8976056749	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.

SOLUTION:

Subquery to Find Customers with Orders Above the Average Order Value:

First, we need to find the average order value. Then we will use a subquery to find customers who have placed orders above this average.

1. To Calculate the Average Order Value→

➔ SELECT AVG(amount) AS average_order_value FROM orders;

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

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

average_order_value
225.000000

2. <u>To Find Customers with Orders Above the Average Order Value→</u>

→SELECT customers.customer_id, customers.customer_name, customers.email_address, customers.city, customers.phone_number, orders.amount

FROM customers

INNER JOIN orders ON customers.customer_id = orders.customer_id

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

<u>Subquery Result:</u>

The customers who have placed orders above the average order value will be listed.

The average order value is 225, and here is the list:

```
SELECT customers.customer_id, customers.customer_name,
customers.email_address, customers.city, customers.phone_number, orders.amount
FROM customers
INNER JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.amount > (SELECT AVG(amount) FROM orders);
```

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

customer_id	customer_name	email_address	city	phone_number	amount
3	Tina	alice@yahoo.com	Siliguri	9765434265	300.00
4	Sunny	sunny@example.com	Durgapur	9076456749	250.00

Now, according to the question:

UNION Query to Combine Two SELECT Statements

Let's assume we want to combine two SELECT statements, one that retrieves all customers from 'Kolkata' and another that retrieves all customers from 'Durgapur'. Both SELECT statements will have the same number of columns.

→SELECT customer_id, customer_name, email_address, city, phone_number

FROM customers

WHERE city = 'Kolkata'

UNION

SELECT customer_id, customer_name, email_address, city, phone_number

FROM customers

WHERE city = 'Durgapur';

→All customers from 'Kolkata' and 'Durgapur' will be listed:

```
SELECT customer_id, customer_name, email_address, city, phone_number
FROM customers
WHERE city = 'Kolkata'
UNION
SELECT customer_id, customer_name, email_address, city, phone_number
FROM customers
WHERE city = 'Durgapur';
```

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

customer_id	customer_name	email_address	city	phone_number
1	Priya	priya123@gmail.com	Kolkata	9987654566
2	Sweety	sweetyghosh123@gmail.com	Kolkata	9876543210
4	Sunny	sunny@example.com	Durgapur	9076456749
5	Ram	ram@yahoo.in	Durgapur	8976056749

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.

SOLUTION:

-- Creating the products table

```
CREATE TABLE products (
   product_id INT PRIMARY KEY,
   product_name VARCHAR(100),
   price DECIMAL(10, 2),
   stock_quantity INT
);
```

-- Inserting values into the products table

```
INSERT INTO products (product_id, product_name, price, stock_quantity) VALUES
(101, 'Product A', 50.00, 100),
(102, 'Product B', 75.00, 150),
(103, 'Product C', 100.00, 200),
(104, 'Product D', 120.00, 75);
```

Result Grid | Filter Rows: | Edit: | Expo

product_id	product_name	price	stock_quantity
101	Product A	50.00	100
102	Product B	75.00	150
103	Product C	100.00	200
104	Product D	120.00	75
NULL	NULL	NULL	NULL

```
-- BEGIN a transaction
BEGIN;

-- INSERT a new record into the 'orders' table
INSERT INTO orders (order_id, customer_id, order_date, amount)
VALUES (6, 4, '2024-06-05', 180.00);

-- COMMIT the transaction
COMMIT;

-- UPDATE the 'products' table
UPDATE products
SET stock_quantity = stock_quantity- 1
WHERE product_id = 104;

-- ROLLBACK the transaction
ROLLBACK;
```

Result Grid | Filter Rows: | Edit: | Export

product_id	product_name	price	stock_quantity
101	Product A	50.00	100
102	Product B	75.00	150
103	Product C	100.00	200
104	Product D	120.00	74
NULL	NULL	NULL	NULL

These SQL statements perform the following actions:

1. Begins a transaction.

2. Inserts a new record into the 'orders' table.
3. Commits the transaction, thereby making the changes permanent.
4. Updates the 'products' table, reducing the stock quantity of a specific product.
5. Rolls back the transaction, reverting any changes made since the transaction began, ensuring data integrity if any errors occur during the update process.

ASSIGNMENT-5

Begin a transaction, perform a series of INSERT' s into 'orders', setting a SAVEPOINT after each, rollback to the second SAVEPOINT, and COMMIT the overall transaction.

SOLUTION:

```
-- Begin the transaction

BEGIN;


-- Perform the first INSERT into the 'orders' table

INSERT INTO orders (order_id, customer_id, order_date, amount)

VALUES (7, 1, '2024-06-01', 220.00);

-- Set the first SAVEPOINT

SAVEPOINT savepoint1;


-- Perform the second INSERT into the 'orders' table

INSERT INTO orders (order_id, customer_id, order_date, amount)

VALUES (8, 2, '2024-06-02', 180.00);

-- Set the second SAVEPOINT

SAVEPOINT savepoint2;
```

```
-- Perform the third INSERT into the 'orders' table
INSERT INTO orders (order_id, customer_id, order_date, amount)
VALUES (9, 3, '2024-06-03', 150.00);
-- Set the third SAVEPOINT
SAVEPOINT savepoint3;

-- Perform the fourth INSERT into the 'orders' table
INSERT INTO orders (order_id, customer_id, order_date, amount)
VALUES (10, 4, '2024-06-04', 300.00);
-- Set the fourth SAVEPOINT
SAVEPOINT savepoint4;
```

Result Grid | Filter Rows: | Edit: | Export/Import:

order_id	customer_id	order_date	amount
6	4	2024-06-05	180.00
7	1	2024-06-01	220.00
8	2	2024-06-02	180.00
9	3	2024-06-03	150.00
10	4	2024-06-04	300.00
101	1	2024-05-01	150.00
201	2	2024-05-15	200.00
301	3	2024-05-10	300.00
401	4	2024-05-20	250.00
NULL	NULL	NULL	NULL

```
-- Rollback to the second SAVEPOINT
ROLLBACK TO SAVEPOINT savepoint2;

-- Commit the overall transaction
COMMIT;
```

```sql
-- Rollback to the second SAVEPOINT
ROLLBACK TO SAVEPOINT savepoint2;

-- Commit the overall transaction
COMMIT;

select * from orders;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Conter

order_id	customer_id	order_date	amount
6	4	2024-06-05	180.00
7	1	2024-06-01	220.00
8	2	2024-06-02	180.00
101	1	2024-05-01	150.00
201	2	2024-05-15	200.00
301	3	2024-05-10	300.00
401	4	2024-05-20	250.00
NULL	NULL	NULL	NULL

Explanation of the SQL Statements:

1. **BEGIN;**
   - Begins a new transaction.
2. **First INSERT and SAVEPOINT**
   - Inserts a new record into the orders table and sets SAVEPOINT savepoint1.
3. **Second INSERT and SAVEPOINT**
   - Inserts a new record into the orders table and sets SAVEPOINT savepoint2.
4. **Third INSERT and SAVEPOINT**
   - Inserts a new record into the orders table and sets SAVEPOINT savepoint3.
5. **Fourth INSERT and SAVEPOINT**
   - Inserts a new record into the orders table and sets SAVEPOINT savepoint4.
6. **ROLLBACK TO SAVEPOINT savepoint2;**
   - Rolls back the transaction to the state after SAVEPOINT savepoint2, effectively undoing the third and fourth INSERT statements.
7. **COMMIT;**
   - Commits the transaction, making the first two INSERT statements permanent and discarding the changes after the second SAVEPOINT.

The third and fourth INSERTs are not committed due to the rollback to `SAVEPOINT savepoint2`.

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.

SOLUTION:

Brief Report on the Use of Transaction Logs for Data Recovery: --

Introduction

Transaction logs are essential for ensuring data integrity and facilitating recovery in database systems. They record every transaction that modifies the database, enabling restoration to a consistent state after failures such as crashes or power outages.

Mechanism of Transaction Logs

1. **Recording Changes**: Logs capture all database modifications, including inserts, updates, and deletes.
2. **Write-Ahead Logging (WAL)**: Changes are first written to the log before being applied to the database.
3. **Commit and Rollback**: Committed transactions are marked in the log; uncommitted transactions can be rolled back in case of failure.

Advantages

- **Data Integrity**: Ensures consistent data states.
- **Point-in-Time Recovery**: Allows restoration to specific points in time.
- **Crash Recovery**: Redo committed and undo uncommitted transactions after a failure.
- **Audit Trail**: Provides a history of all changes for security and compliance.

Hypothetical Scenario: Data Recovery After an Unexpected Shutdown

Scenario: FinServ Ltd. experiences a power outage during several customer transactions.

Impact: Transactions in progress include:

- A $500 transfer from Customer A to Customer B.
- A $200 deposit by Customer C.
- A $100 withdrawal by Customer D.

Recovery Process:

1. **Initiating Recovery**: Upon restart, the DBMS uses the transaction log to identify in-progress transactions.
2. **Rolling Back**: Uncommitted transactions (the $500 transfer and $100 withdrawal) are rolled back.
3. **Replaying**: Committed transactions (the $200 deposit) are replayed.

Final State: The database is restored to a consistent state, accurately reflecting only the completed transactions.

Conclusion: -

Transaction logs are crucial for data recovery, ensuring databases can be restored to a consistent state after unexpected failures.

In the scenario of FinServ Ltd., transaction logs enabled accurate recovery and ensured customer transactions were correctly processed despite the power outage.