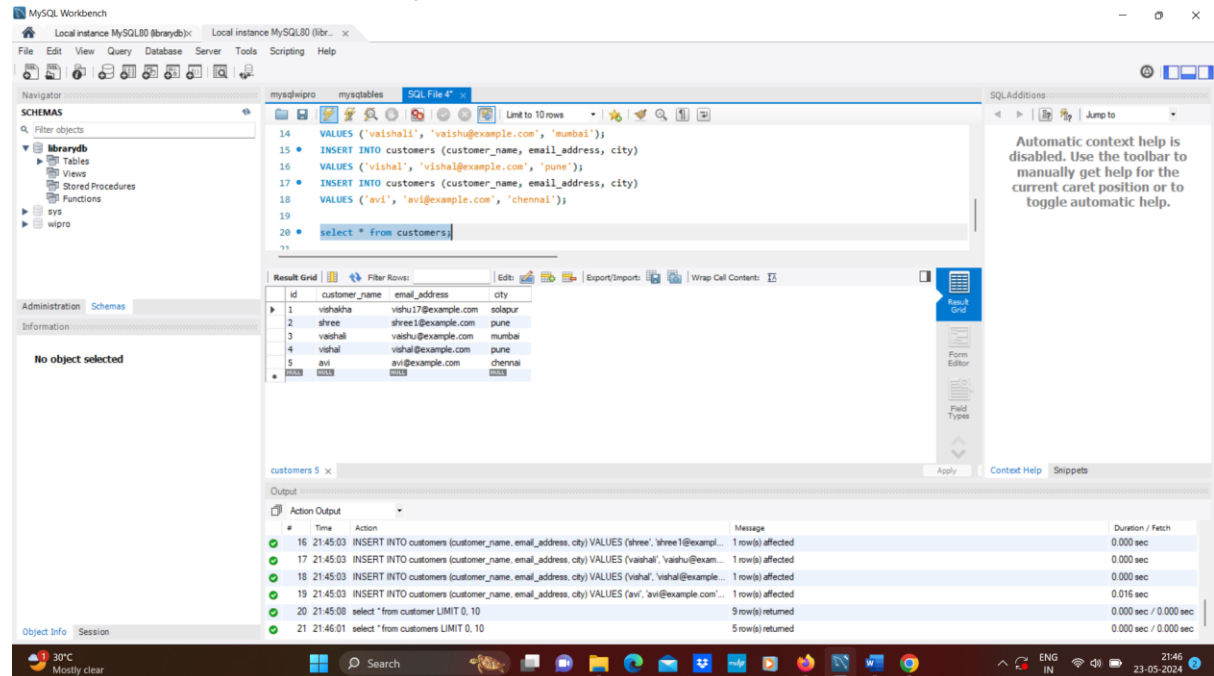


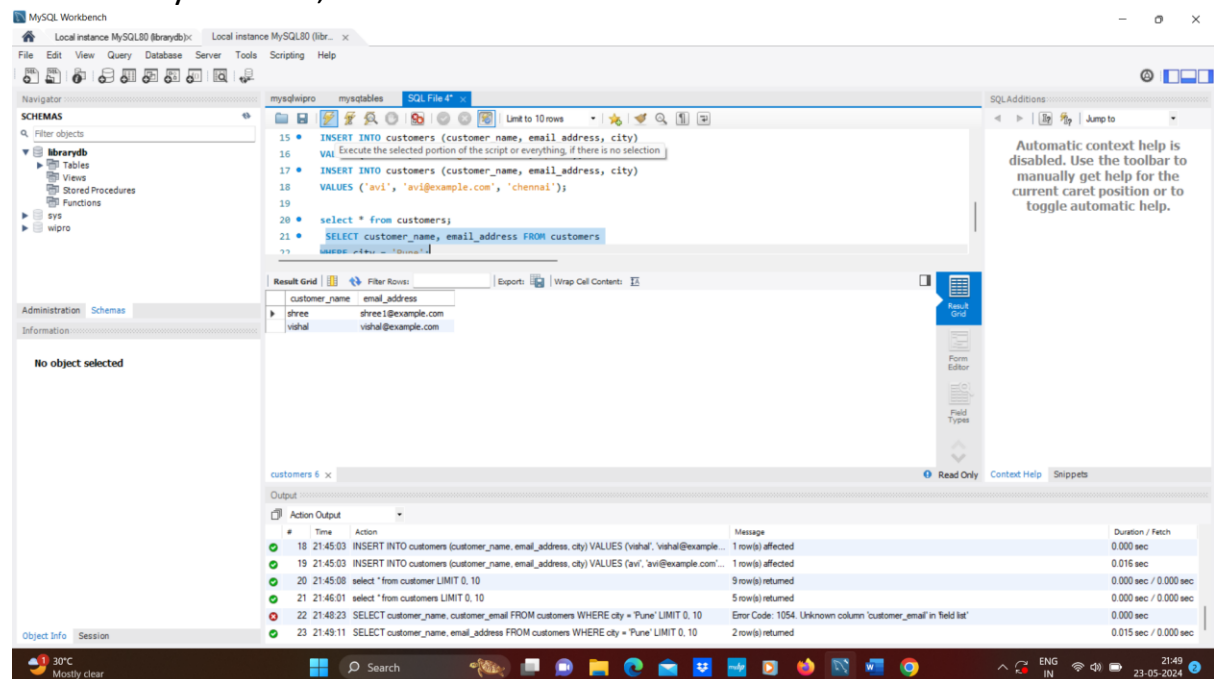
Assignment 1: Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.

1. SELECT query to retrieve all columns from a 'customers'
SELECT * FROM customers;



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

SELECT customer_name, email_address FROM customers
WHERE city = 'Pune';



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.

SELECT

```
customers.customer_id,  
customers.customer_name,  
customers.email_address,  
customers.city,  
customers.region,  
orders.order_id,  
orders.order_date,  
orders.amount
```

FROM

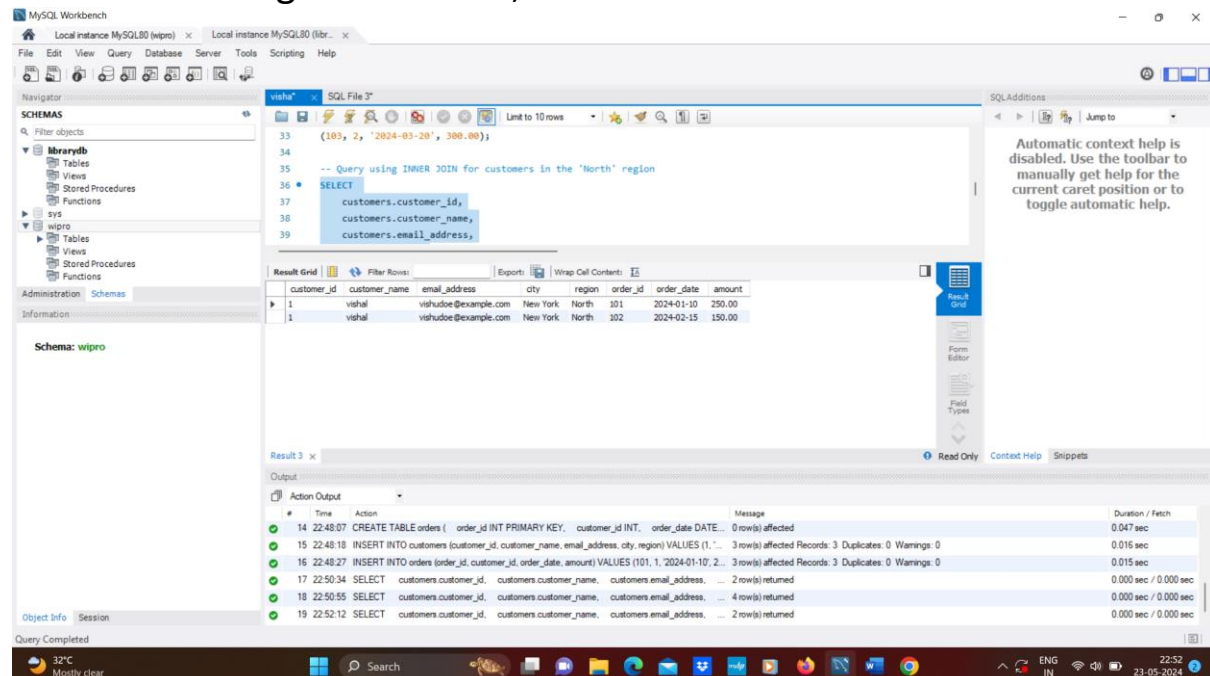
```
customers
```

INNER JOIN

```
orders ON customers.customer_id = orders.customer_id
```

WHERE

```
customers.region = 'North';
```



a LEFT JOIN to display all customers including those without orders

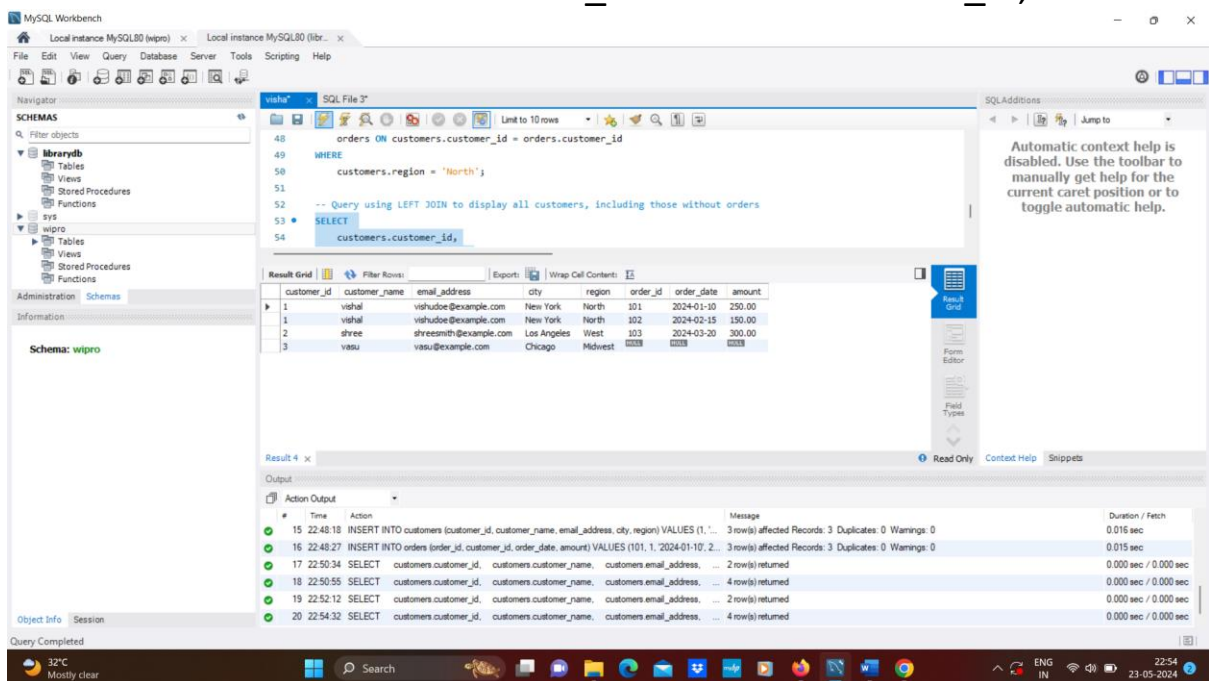
SELECT

```
customers.customer_id,
```

```

customers.customer_name,
customers.email_address,
customers.city,
customers.region,
orders.order_id,
orders.order_date,
orders.amount
FROM
    customers
LEFT JOIN
    orders ON customers.customer_id = orders.customer_id;

```



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

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

```

SELECT DISTINCT c.customer_id, c.customer_name, c.customer_email, c.city
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN product p ON o.product_id = p.product_id
WHERE (o.quantity * p.product_price) > (
    SELECT AVG(o2.quantity * p2.product_price)
    FROM orders o2

```

JOIN product p2 ON o2.product_id = p2.product_id
);

The screenshot shows a SQL Developer window with a query editor and a results grid. The query is as follows:

```

421
422
423 -- assignment
424 * SELECT DISTINCT c.customer_id, c.customer_name, c.customer_email, c.city
425 FROM customers c
426 JOIN orders o ON c.customer_id = o.customer_id
427 JOIN product p ON o.product_id = p.product_id
428 WHERE (o.quantity * p.product_price) > (

```

The results grid shows the following data:

customer_id	customer_name	customer_email	city
1	asha	asha@gmail.com	pune
4	divya	divya@gmail.com	thane
5	grish	grish@gmail.com	pune

The output pane shows the execution of the query, with the following messages:

```

58 21:38:38 SELECT * FROM customers LIMIT 0, 1000 6 row(s) returned 0.000 sec / 0.000 sec
59 21:41:00 SELECT customer_name, customer_email FROM customers WHERE city = 'Pune' LIMIT 0, 1000 2 row(s) returned 0.000 sec / 0.000 sec
60 21:41:17 SELECT customer_name, customer_email, city FROM customers WHERE city = 'Pune' LIMIT 0, 1000 2 row(s) returned 0.000 sec / 0.000 sec
61 21:45:22 SELECT c.customer_id, c.customer_name, c.customer_email, c.city, o.order_id, o.product_id, ... 2 row(s) returned 0.000 sec / 0.000 sec
62 21:48:21 SELECT c.customer_id, c.customer_name, c.customer_email, c.city, o.order_id, o.product_id, ... 8 row(s) returned 0.000 sec / 0.000 sec
63 21:54:08 SELECT DISTINCT c.customer_id, c.customer_name, c.customer_email, c.city FROM customers c JOIN order... 3 row(s) returned 0.016 sec / 0.000 sec

```

UNION query to combine two SELECT statements with the same number of columns.

SELECT customer_id, customer_name, city FROM customers WHERE city = 'Mumbai'

UNION

SELECT customer_id, customer_name, city FROM customers WHERE city = 'Pune';

The screenshot shows a SQL Developer window with a query editor and a results grid. The query is as follows:

```

429
430 SELECT AVG(o2.quantity * p2.product_price)
431 FROM orders o2
432 JOIN product p2 ON o2.product_id = p2.product_id
433 );
434 * SELECT customer_id, customer_name, city FROM customers WHERE city = 'Mumbai'
435 UNION
436 SELECT customer_id, customer_name, city FROM customers WHERE city = 'Pune';

```

The results grid shows the following data:

customer_id	customer_name	city
2	malar	mumbai
6	deepak	mumbai
1	asha	pune
5	grish	pune

The output pane shows the execution of the query, with the following messages:

```

59 21:41:00 SELECT customer_name, customer_email FROM customers WHERE city = 'Pune' LIMIT 0, 1000 2 row(s) returned 0.000 sec / 0.000 sec
60 21:41:17 SELECT customer_name, customer_email, city FROM customers WHERE city = 'Pune' LIMIT 0, 1000 2 row(s) returned 0.000 sec / 0.000 sec
61 21:45:22 SELECT c.customer_id, c.customer_name, c.customer_email, c.city, o.order_id, o.product_id, ... 2 row(s) returned 0.000 sec / 0.000 sec
62 21:48:21 SELECT c.customer_id, c.customer_name, c.customer_email, c.city, o.order_id, o.product_id, ... 8 row(s) returned 0.000 sec / 0.000 sec
63 21:54:08 SELECT DISTINCT c.customer_id, c.customer_name, c.customer_email, c.city FROM customers c JOIN order... 3 row(s) returned 0.016 sec / 0.000 sec
64 21:56:45 SELECT customer_id, customer_name, city FROM customers WHERE city = 'Mumbai' UNION SELECT custo... 4 row(s) returned 0.000 sec / 0.000 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.

BEGIN;

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)

VALUES (8, 1, 101, 2, '2024-02-15');

COMMIT;

BEGIN;

UPDATE product

SET product_price = product_price * 1.1 -- Increasing price by 10%

WHERE product_id = 101;

ROLLBACK;

```

> BEGIN;
INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES (8, 1, 101, 2, '2024-02-15');
COMMIT;

> BEGIN;
UPDATE product
SET product_price = product_price * 1.1 -- Increasing price by 10%
WHERE product_id = 101;
select * from orders;

ROLLBACK;

-- 1. retrieve all the orders with customer and product details
select o.order_id, c.customer_id, p.product_id, o.quantity, o.order_date from orders o

```

Time	Action	Message	Duration / Fetch
22:04:42	COMMIT	0 row(s) affected	0.015 sec
22:05:42	BEGIN	0 row(s) affected	0.000 sec
22:05:42	UPDATE product SET product_price = product_price * 1.1 -- Increasing price by 10% WHERE product_id = 1...	1 row(s) affected, 1 warning(s): 1255 Data truncated for column 'product_price' at row 1 Rows matched: 1 Cha...	0.015 sec
22:06:28	select * from product LIMIT 0, 1000	5 row(s) returned	0.000 sec / 0.000 sec
22:07:09	select * from orders LIMIT 0, 1000	8 row(s) returned	0.000 sec / 0.000 sec
22:07:43	ROLLBACK	0 row(s) affected	0.000 sec

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.

Begin;

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)

VALUES (10, 1, 101, 1, '2024-02-16');

SAVEPOINT SP1;

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)

VALUES (11, 2, 102, 3, '2024-02-17');

SAVEPOINT SP2;

```

INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
VALUES (12, 3, 103, 2, '2024-02-18');
SAVEPOINT SP3;
ROLLBACK TO SAVEPOINT SP2;
COMMIT;

```

The screenshot shows a SQL IDE with a script editor and an output window. The script in the editor is as follows:

```

-- assignment 5
-- Insert the first order and set a savepoint
448 begin;
449
450 INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
451 VALUES (10, 1, 101, 1, '2024-02-16');
452
453 SAVEPOINT SP1;
454
455 -- Insert the second order and set a savepoint
456 INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
457 VALUES (11, 2, 102, 3, '2024-02-17');
458
459 SAVEPOINT SP2 ;
460
461 -- Insert the third order and set a savepoint
462 INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date)
463 VALUES (12, 3, 103, 2, '2024-02-18');
464
465 SAVEPOINT SP3;
466
467 ROLLBACK TO SAVEPOINT SP2;
468
469 COMMIT;
470
471

```

The output window shows the following results:

#	Time	Action	Message	Duration / Fetch
76	22:15:51	INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date) VALUES (11, 2, 102, 3, '2024-02-17')	1 row(s) affected	0.000 sec
77	22:15:56	SAVEPOINT SP2	0 row(s) affected	0.000 sec
78	22:16:03	INSERT INTO orders (order_id, customer_id, product_id, quantity, order_date) VALUES (12, 3, 103, 2, '2024-02-18')	1 row(s) affected	0.000 sec
79	22:16:11	SAVEPOINT SP3	0 row(s) affected	0.000 sec
80	22:17:02	ROLLBACK TO SAVEPOINT SP2	0 row(s) affected	0.000 sec
81	22:17:34	COMMIT	0 row(s) affected	0.016 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.

Transaction logs are crucial components of database management systems (DBMS) designed to maintain a continuous record of all changes made to the data within a database. Every transaction that modifies, adds, or deletes data is sequentially logged with complete details regarding the type of change, the data affected, and the time of the transaction. This meticulous recording allows databases not only to maintain data integrity but also to facilitate recovery operations in case of failures.

Key Functions of Transaction Logs

1. **Data Recovery:** Transaction logs play a vital role in data recovery processes. They ensure that any changes made during incomplete transactions at the time of a system failure can either be rolled back or completed during

system recovery.

2. Audit and Compliance: Transaction logs provide a traceable history of all data interactions, which is critical for auditing and compliance purposes.

3. Replication: In distributed databases, transaction logs are used to replicate data changes across different database systems, ensuring consistency across geographically dispersed infrastructure.

Hypothetical Scenario: Recovery After an Unexpected Shutdown

Scenario Description

Consider a financial services company, "FinCorp," which manages an online trading platform. The database holds critical information such as user profiles, transaction records, and trading histories. One day, due to an unforeseen power outage, the main data center experiences an abrupt shutdown. This incident occurs during a high-volume trading period, and there are multiple transactions related to stock trades in process.

Role of Transaction Logs in Recovery

Upon restoration of power, the DBMS initiates a recovery process. Here's how transaction logs are used:

1. Analysis of Logs: The system starts by analyzing the transaction logs, identifying the last transaction checkpoint where the database was in a consistent state.

2. Redo Operations: Transactions that had been committed prior to the shutdown and were logged after the last checkpoint are replayed. These redo operations ensure that all committed transactions are restored to the database even if they were not fully written to disk before the shutdown.

3. Undo Operations: Transactions that were in progress and not committed at the time of the shutdown are identified. The logs provide the necessary information to reverse (undo) these

transactions, ensuring that the database remains in a consistent state without partial or corrupt data entries.