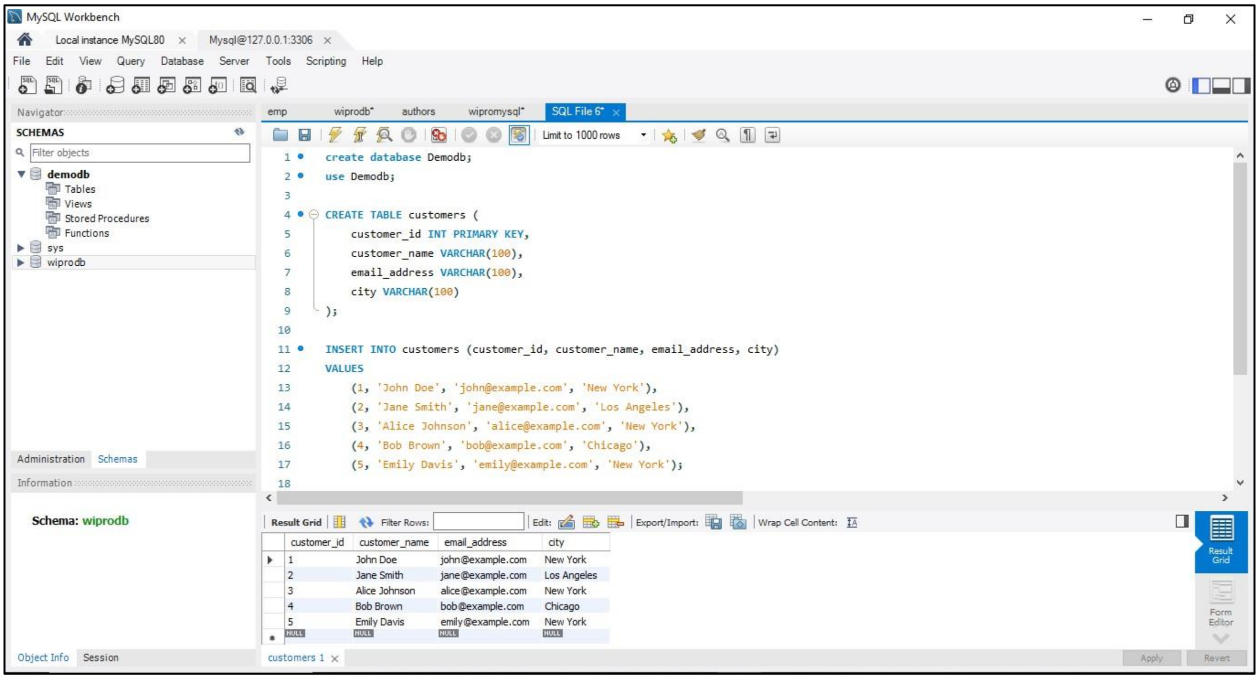
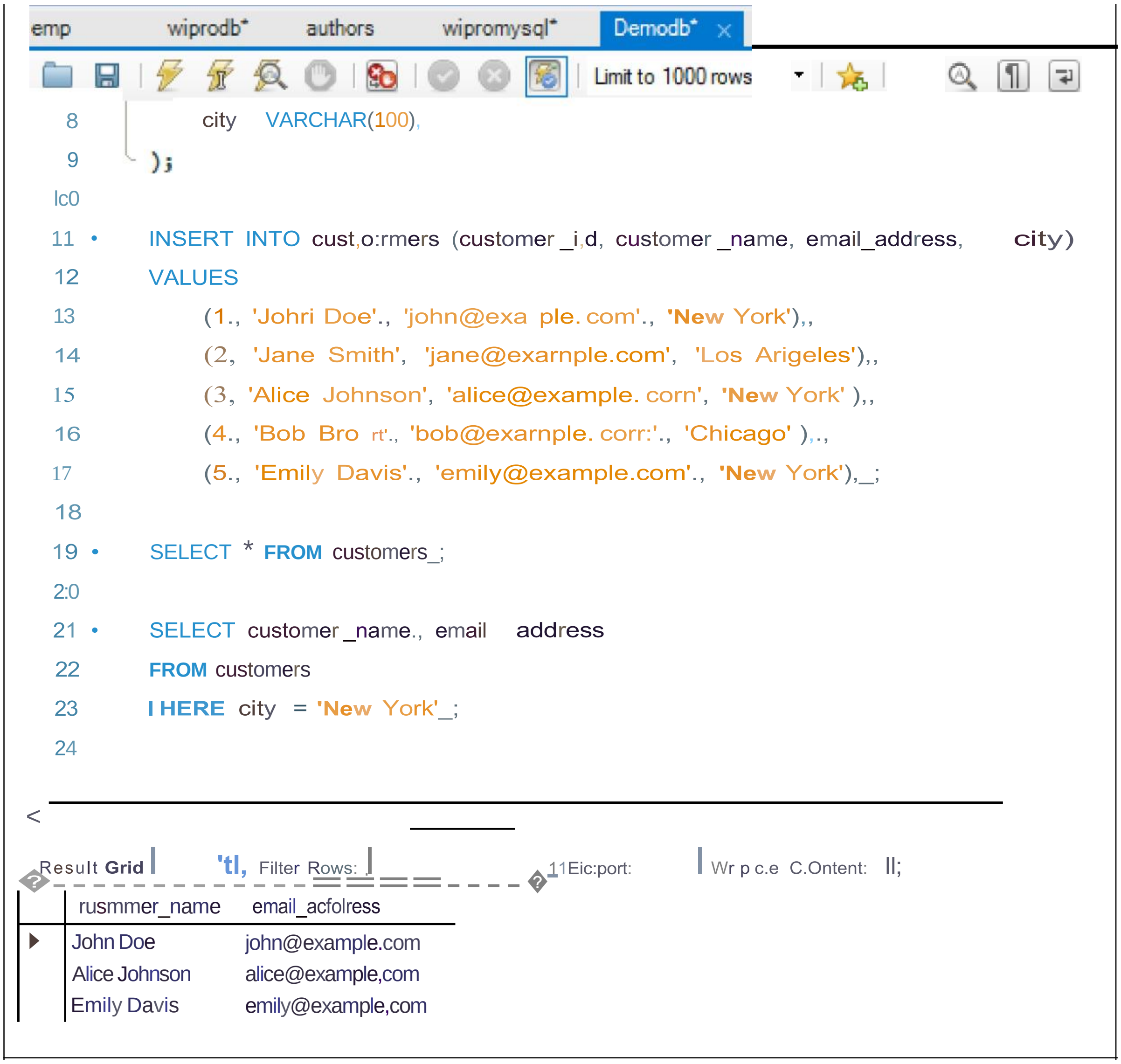
Assignment 1: Write a SELECT query to retrieve all columns from a 'customers' table, and modify it to return only the customer name and email address for customers in a specific city.



```sql
create database Demodb;
use Demodb;

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

INSERT INTO customers (customer_id, customer_name, email_address, city)
VALUES
    (1, 'John Doe', 'john@example.com', 'New York'),
    (2, 'Jane Smith', 'jane@example.com', 'Los Angeles'),
    (3, 'Alice Johnson', 'alice@example.com', 'New York'),
    (4, 'Bob Brown', 'bob@example.com', 'Chicago'),
    (5, 'Emily Davis', 'emily@example.com', 'New York');

```

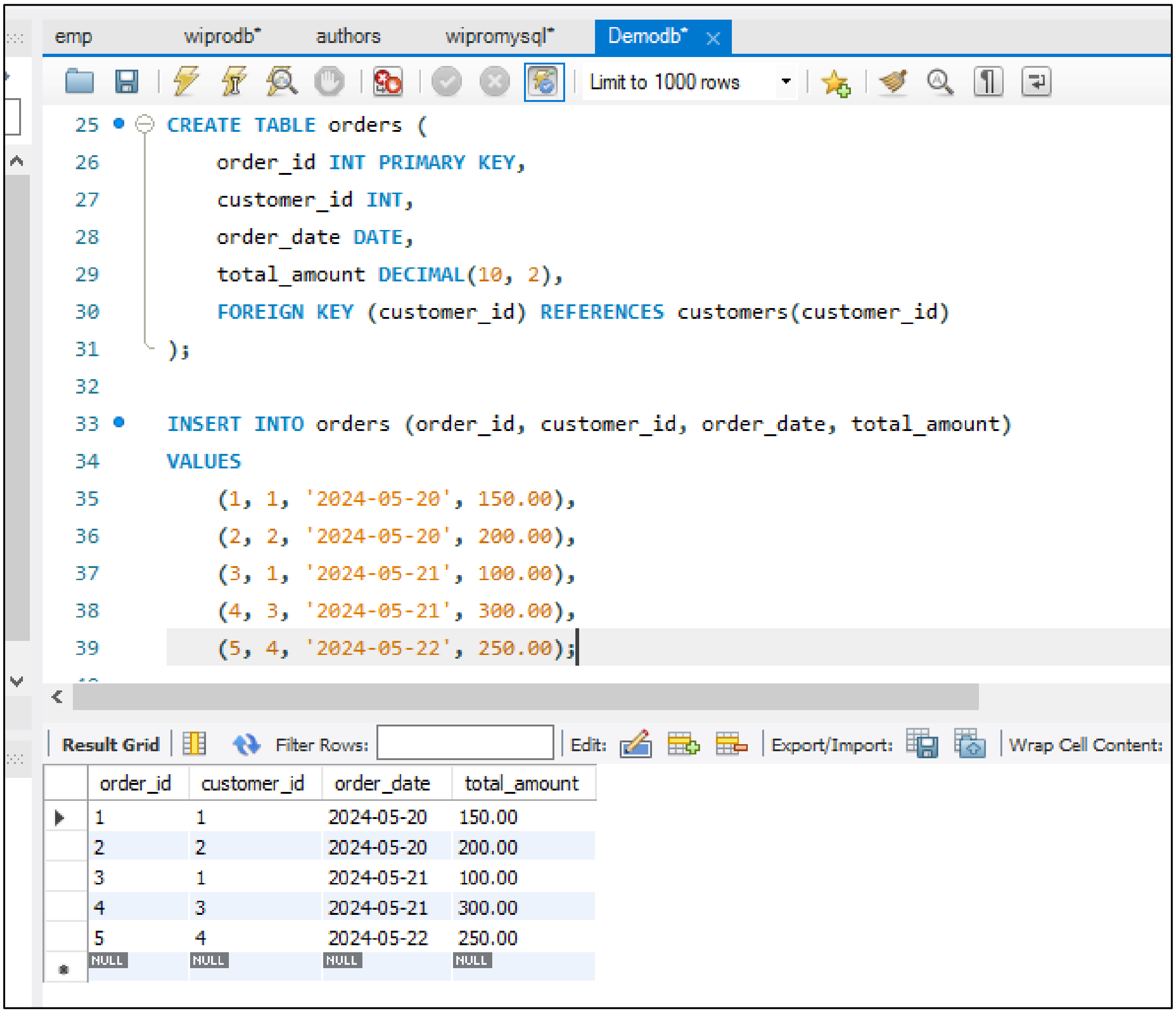
customer_id	customer_name	email_address	city
1	John Doe	john@example.com	New York
2	Jane Smith	jane@example.com	Los Angeles
3	Alice Johnson	alice@example.com	New York
4	Bob Brown	bob@example.com	Chicago
5	Emily Davis	emily@example.com	New York
NULL	NULL	NULL	NULL



```
        city    VARCHAR(100),
);

INSERT  INTO  cust,o:rmers  (customer _i,d,  customer _name,  email_address,      city)
VALUES
        (1., 'Johri Doe'., 'john@exa ple. com'., 'New York'),,
        (2,  'Jane  Smith',  'jane@exarnple.com',  'Los  Arigeles'),,
        (3,  'Alice  Johnson',  'alice@example. corn',  'New York' ),,
        (4.,  'Bob  Bro  rt'.,  'bob@exarnple. corr:'.,  'Chicago' ),.,
        (5.,  'Emily  Davis'.,  'emily@example.com'.,  'New  York'),_;

SELECT * FROM customers_;

SELECT  customer _name.,  email    address
FROM customers
I HERE  city  =  'New  York'_;
```

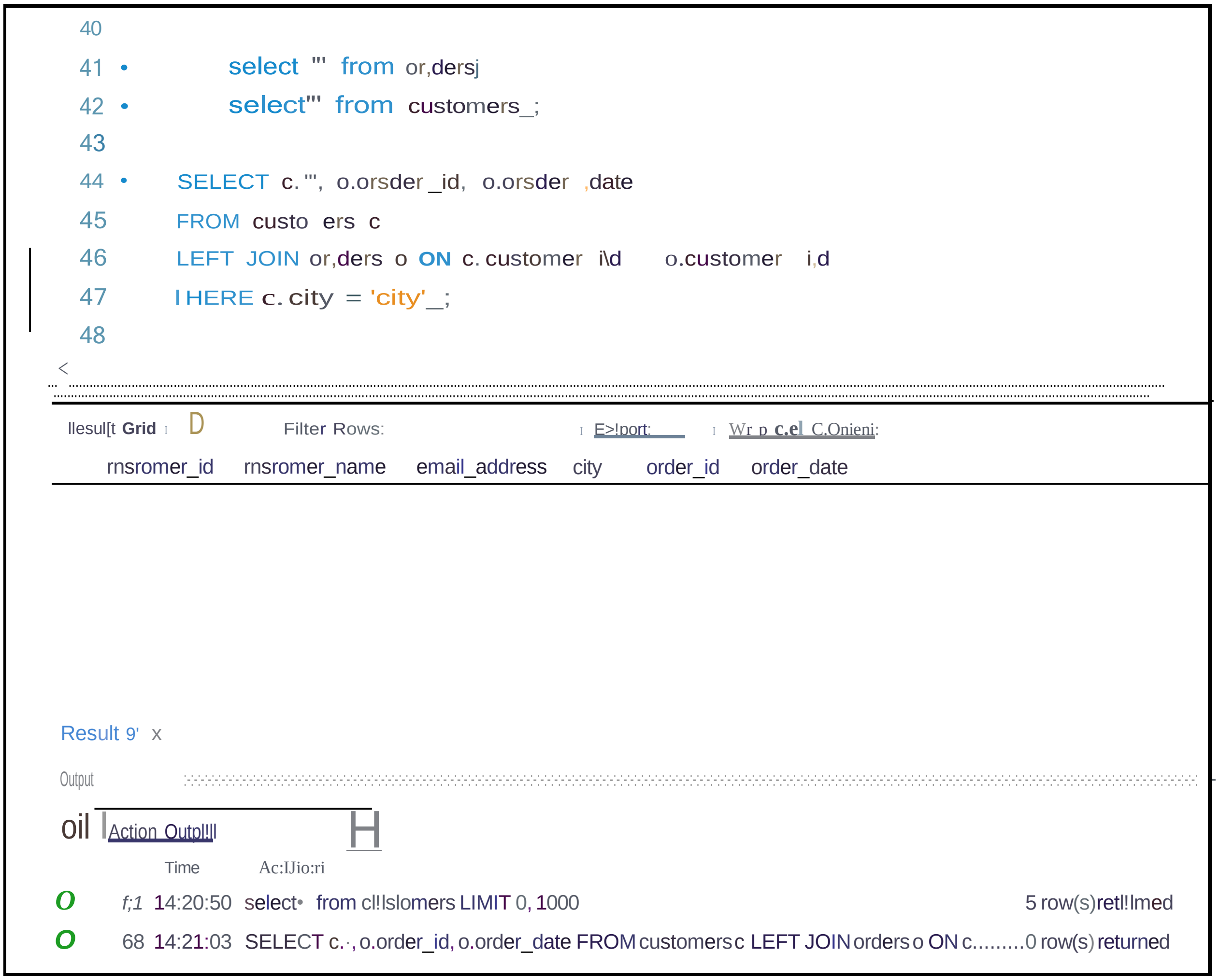
rusmmer_name	email_acfolress
John Doe	john@example.com
Alice Johnson	alice@example,com
Emily Davis	emily@example,com

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 orders.



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

INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES
    (1, 1, '2024-05-20', 150.00),
    (2, 2, '2024-05-20', 200.00),
    (3, 1, '2024-05-21', 100.00),
    (4, 3, '2024-05-21', 300.00),
    (5, 4, '2024-05-22', 250.00);
```

order_id	customer_id	order_date	total_amount
1	1	2024-05-20	150.00
2	2	2024-05-20	200.00
3	1	2024-05-21	100.00
4	3	2024-05-21	300.00
5	4	2024-05-22	250.00
NULL	NULL	NULL	NULL

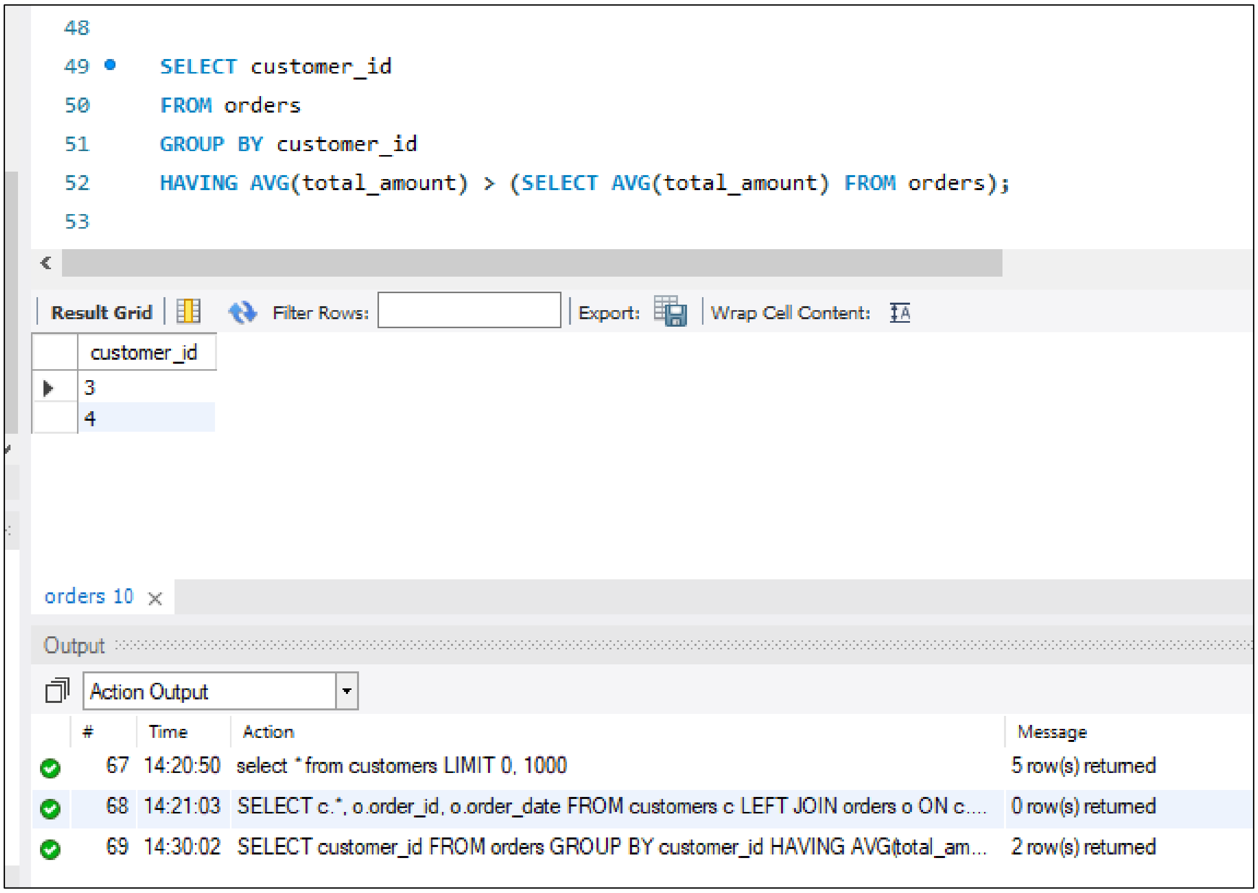


```
        select "' from or,dersj
        select"' from customers_;

    SELECT c.'", o.orsder _id, o.orsder ,date
    FROM custo ers c
    LEFT JOIN or,ders o ON c. customer i\d o.customer i,d
    l HERE c. city = 'city'_;
```

```
rnsromer_id   rnsromer_name   email_address   city   order_id   order_date
```

```
        Time       Ac:lJio:ri
O  f;1  14:20:50  select• from cl!lslomers LIMIT 0, 1000                                              5 row(s)retl!lmed
O  68   14:21:03  SELECT c.·, o.order_id, o.order_date FROM customers c LEFT JOIN orders o ON c.........0 row(s) returned
```

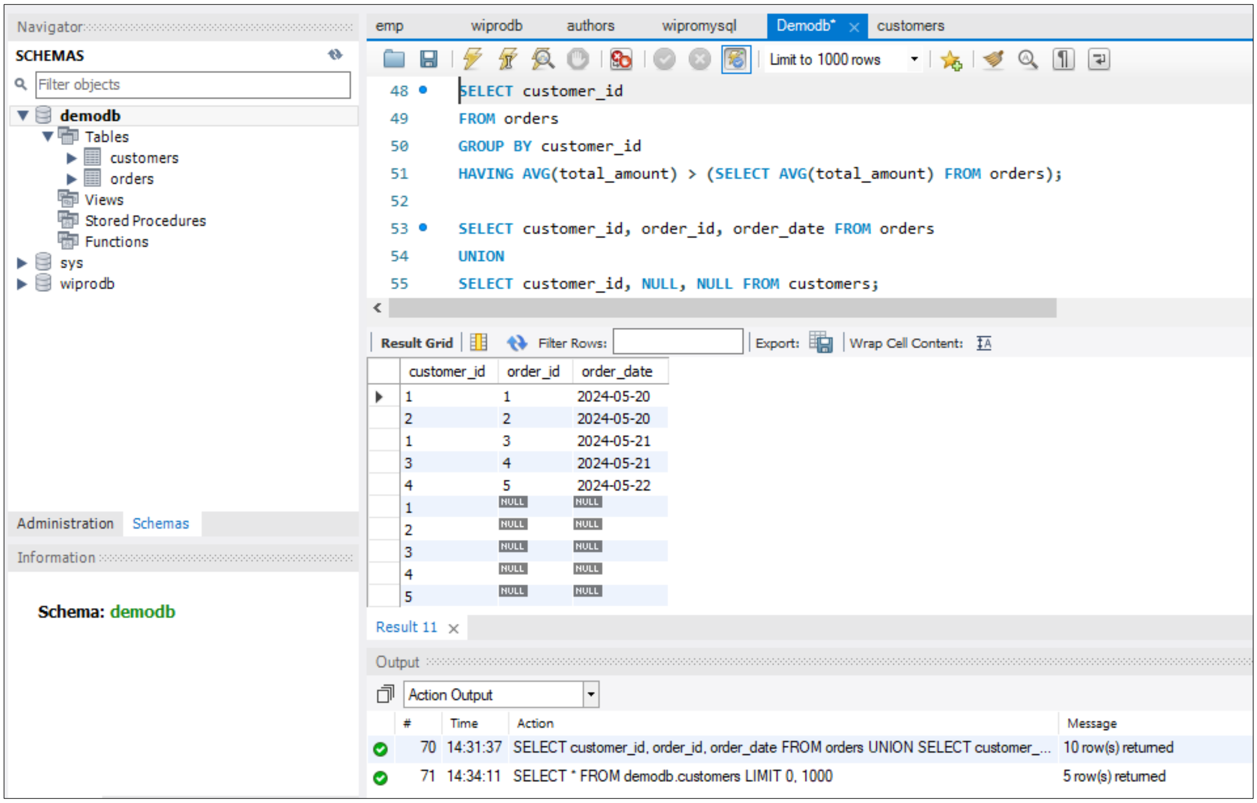
Assignment 3: Utilize a subquery to find customers who have placed orders above the average order value, and write a UNION query to combine two SELECT statements with the same number of columns.



```sql

SELECT customer_id
FROM orders
GROUP BY customer_id
HAVING AVG(total_amount) > (SELECT AVG(total_amount) FROM orders);

```

customer_id
3
4

```
   #   Time      Action                                                                Message
   67  14:20:50  select * from customers LIMIT 0, 1000                                 5 row(s) returned
   68  14:21:03  SELECT c.*, o.order_id, o.order_date FROM customers c LEFT JOIN orders o ON c....  0 row(s) returned
   69  14:30:02  SELECT customer_id FROM orders GROUP BY customer_id HAVING AVG(total_am...    2 row(s) returned
```

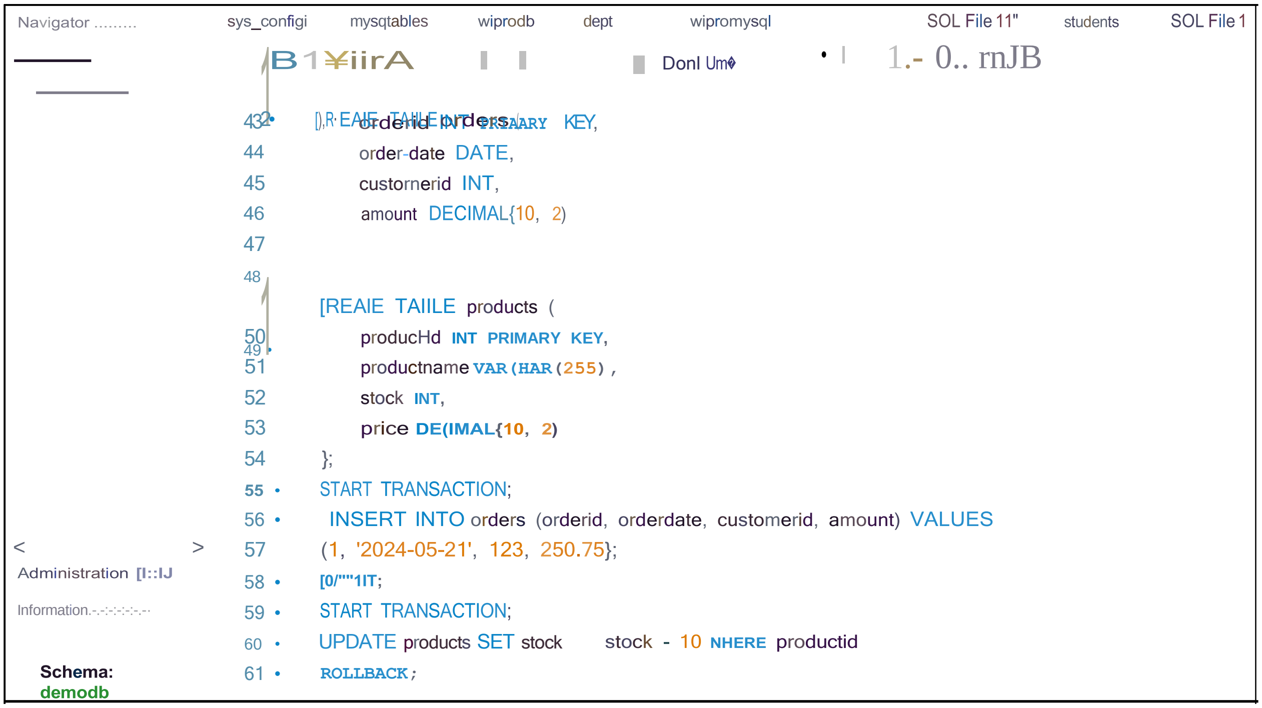


```sql
SELECT customer_id
FROM orders
GROUP BY customer_id
HAVING AVG(total_amount) > (SELECT AVG(total_amount) FROM orders);

SELECT customer_id, order_id, order_date FROM orders
UNION
SELECT customer_id, NULL, NULL FROM customers;
```

customer_id	order_id	order_date
1	1	2024-05-20
2	2	2024-05-20
1	3	2024-05-21
3	4	2024-05-21
4	5	2024-05-22
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	NULL	NULL
5	NULL	NULL

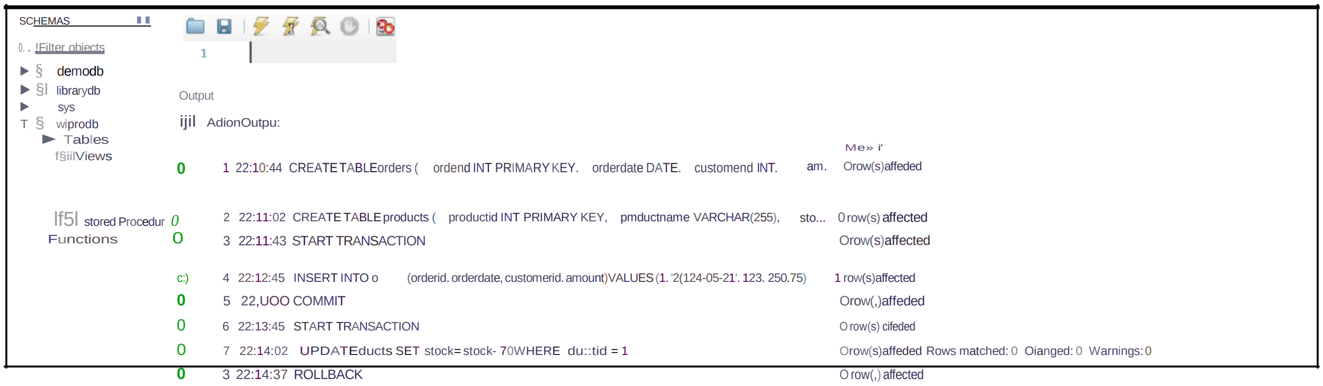
#	Time	Action	Message
70	14:31:37	SELECT customer_id, order_id, order_date FROM orders UNION SELECT customer_...	10 row(s) returned
71	14:34:11	SELECT * FROM demodb.customers LIMIT 0, 1000	5 row(s) returned

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.

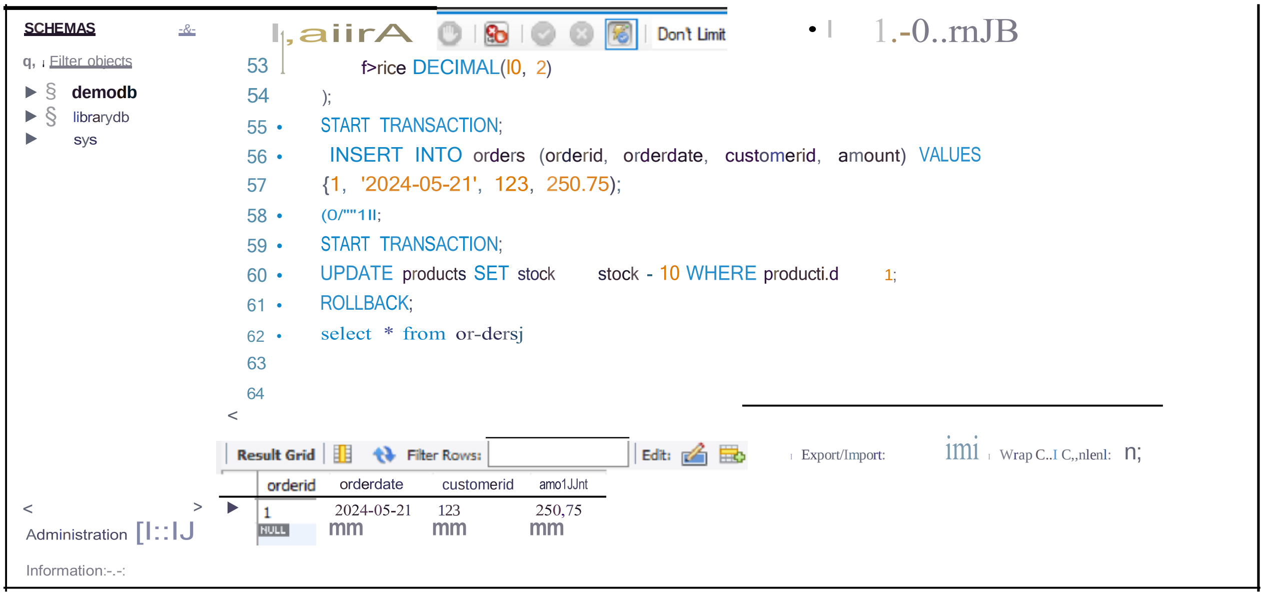


```
[),R·EAIE TAILE orders (
    orderid INT PRIMARY KEY,
    order-date DATE,
    custornerid INT,
    amount DECIMAL{10, 2)

[REAIE TAIILE products (
    producHd INT PRIMARY KEY,
    productname VAR(HAR(255),
    stock INT,
    price DE(IMAL{10, 2)
};
START TRANSACTION;
 INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(1, '2024-05-21', 123, 250.75};
[0/""1IT;
START TRANSACTION;
UPDATE products SET stock      stock - 10 NHERE productid
ROLLBACK;
```

Schema: demodb



```
0  1 22:10:44 CREATE TABLEorders (   ordend INT PRIMARY KEY.  orderdate DATE.  customend INT.   am.   Orow(s)affeded
0  2 22:11:02 CREATE TABLE products (  productid INT PRIMARY KEY,  pmductname VARCHAR(255),  sto...  0 row(s) affected
0  3 22:11:43 START TRANSACTION                                                                    Orow(s)affected
c:) 4 22:12:45 INSERT INTO o    (orderid. orderdate, customerid. amount)VALUES (1. '2(124-05-21'. 123. 250.75)  1 row(s)affected
0  5 22,UOO COMMIT                                                                                 Orow(,)affeded
0  6 22:13:45 START TRANSACTION                                                                    O row(s) cifeded
0  7 22:14:02 UPDATEducts SET stock=stock- 70WHERE du::tid = 1     Orow(s)affeded Rows matched: 0 Oianged: 0 Warnings: 0
0  3 22:14:37 ROLLBACK                                                                             O row(,) affected
```

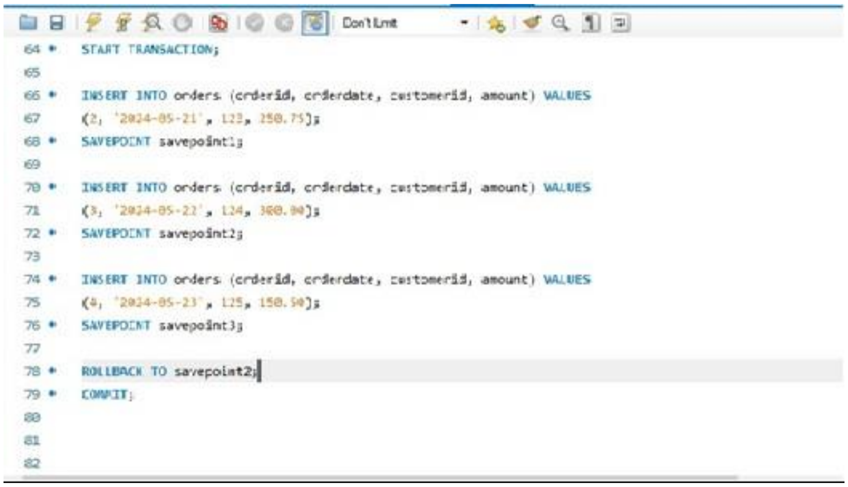


```
    f>rice DECIMAL(I0, 2)
);
START TRANSACTION;
 INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
{1, '2024-05-21', 123, 250.75);
(O/""1II;
START TRANSACTION;
UPDATE products SET stock      stock - 10 WHERE producti.d      1;
ROLLBACK;
select * from or-dersj
```

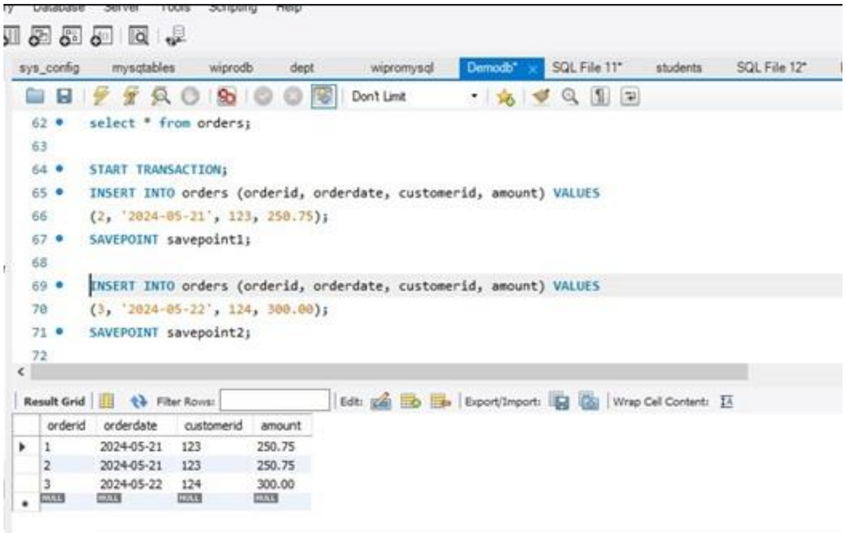
orderid	orderdate	customerid	amo1JJnt
1	2024-05-21	123	250,75
NULL	mm	mm	mm

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.

Ans:

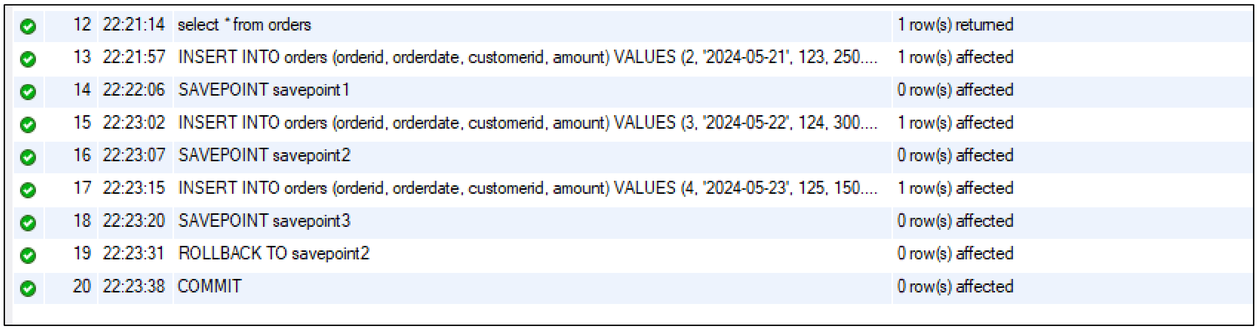


```
START TRANSACTION;

INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(2, '2024-05-21', 123, 250.75);
SAVEPOINT savepoint1;

INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(3, '2024-05-22', 124, 300.00);
SAVEPOINT savepoint2;

INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(4, '2024-05-23', 125, 150.50);
SAVEPOINT savepoint3;

ROLLBACK TO savepoint2;
COMMIT;
```



```
select * from orders;

START TRANSACTION;
INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(2, '2024-05-21', 123, 250.75);
SAVEPOINT savepoint1;

INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES
(3, '2024-05-22', 124, 300.00);
SAVEPOINT savepoint2;
```

orderid	orderdate	customerid	amount
1	2024-05-21	123	250.75
2	2024-05-21	123	250.75
3	2024-05-22	124	300.00
NULL	NULL	NULL	NULL



```
✓  12  22:21:14  select * from orders                                                                    1 row(s) returned
✓  13  22:21:57  INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES (2, '2024-05-21', 123, 250....   1 row(s) affected
✓  14  22:22:06  SAVEPOINT savepoint1                                                                    0 row(s) affected
✓  15  22:23:02  INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES (3, '2024-05-22', 124, 300....   1 row(s) affected
✓  16  22:23:07  SAVEPOINT savepoint2                                                                    0 row(s) affected
✓  17  22:23:15  INSERT INTO orders (orderid, orderdate, customerid, amount) VALUES (4, '2024-05-23', 125, 150....   1 row(s) affected
✓  18  22:23:20  SAVEPOINT savepoint3                                                                    0 row(s) affected
✓  19  22:23:31  ROLLBACK TO savepoint2                                                                  0 row(s) affected
✓  20  22:23:38  COMMIT                                                                                  0 row(s) affected
```

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.

Ans:

Report: Leveraging Transaction Logs for Data Recovery in MySQL

Introduction: Transaction logs are vital components of MySQL database management, playing a crucial role in ensuring data integrity and facilitating recovery in the event of system failures or unexpected shutdowns. These logs record every change made to a database, providing a detailed trail of transactions. This report explores the significance of transaction logs for data recovery in MySQL and illustrates their importance through a hypothetical scenario.

The Importance of Transaction Logs for Data Recovery: Transaction logs, specifically the binary logs in MySQL, serve as reliable sources of information for recovering data after a system failure. They maintain a chronological record of all events that modify the database, including inserts, updates, and deletes. By capturing changes before they are permanently written to the database, transaction logs enable the reconstruction of the database to a consistent state, even in the face of unforeseen disruptions.

Key Functions of Transaction Logs in MySQL:

1. **Redo Logging:** MySQL's binary logs capture the changes made to the database, allowing for the replay of transactions that were committed but not yet written to disk at the time of the failure. This process, known as redo logging, ensures that committed transactions are not lost during recovery.

2. **Undo Logging:** InnoDB, the default storage engine for MySQL, uses undo logs to store information necessary to reverse or undo transactions that were in progress but not yet committed at the time of the failure. This capability enables the restoration of the database to its pre-transaction state, maintaining data consistency.

3. **Point-in-Time Recovery:** MySQL's binary logs enable point-in-time recovery, allowing database administrators to restore the database to a specific moment before the failure occurred. By replaying transactions up to the desired timestamp, organizations can minimize data loss and maintain business continuity.

Hypothetical Scenario: Consider a scenario where an e-commerce platform experiences an unexpected shutdown of its MySQL database server due to a hardware failure. As a result, critical customer order data becomes inaccessible, posing a significant operational risk. However, due to the diligent use of MySQL binary logs, the organization can recover the data swiftly and minimize the impact on its operations.

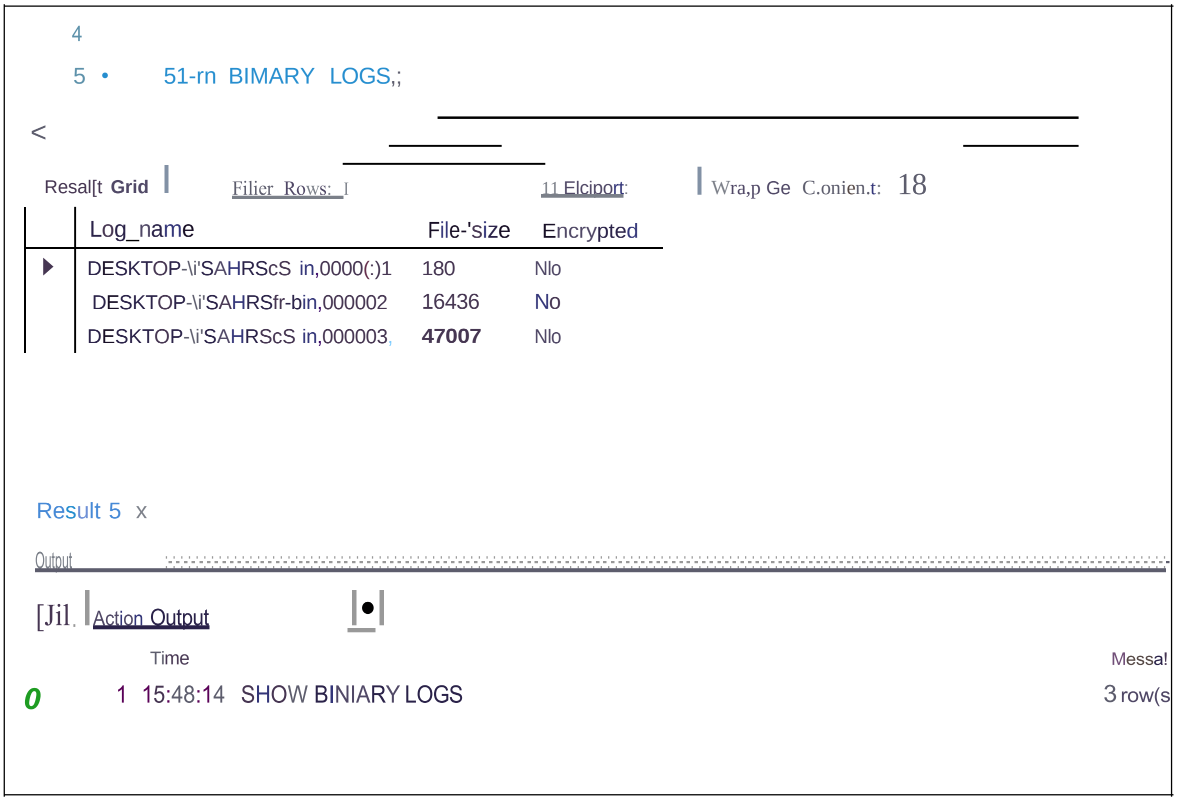
- **Event:** The database server abruptly shuts down, leading to the loss of unsaved changes and potentially jeopardizing the integrity of customer order records.

- **Response:** Upon restarting the database server, database administrators immediately initiate the recovery process using MySQL binary logs. By applying the binary logs, the system reconstructs the database to a consistent state, ensuring that all committed transactions are preserved.

Outcome: Despite the unexpected shutdown, the e-commerce platform successfully restores access to critical customer order data with minimal data loss. The MySQL binary logs prove instrumental in facilitating rapid recovery, demonstrating their indispensable role in ensuring data resilience and business continuity.

Step-by-Step Recovery Process:

1. **Identify the Binary Log Files:** Locate the binary log files generated by MySQL before the crash.



```
51-rn  BIMARY  LOGS,;
```

Log_name	File-'size	Encrypted
DESKTOP-\i'SAHRScS in,0000(:)1	180	Nlo
DESKTOP-\i'SAHRSfr-bin,000002	16436	No
DESKTOP-\i'SAHRScS in,000003	**47007**	Nlo

```
          Time                                              Messa!
0     1   15:48:14   SHOW BINIARY LOGS                      3 row(s
```