Prajwalhonashetti143@gmail.com

10th day Assignment

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.

Let's say we have a customers table with the following columns: **customer_id, customer_name, email_address, city, and phone_number.**

customer_id	customer_name	email_address	city	phone_number
1	John Doe	john@example.com	New York	123-456-7890
2	Jane Smith	jane@example.com	Los Angeles	987-654-3210
3	Mike Johnson	mike@example.com	New York	555-555-5555
4	Emily Davis	emily@example.com	Chicago	444-444-4444

- **Query to retrieve all columns from the customers table:**

```
SELECT * FROM customers;   //* will select all column from table
```

Output:

customer_id	customer_name	email_address	city	phone_number
1	John Doe	john@example.com	New York	123-456-7890
2	Jane Smith	jane@example.com	Los Angeles	987-654-3210
3	Mike Johnson	mike@example.com	New York	555-555-5555
4	Emily Davis	emily@example.com	Chicago	444-444-4444

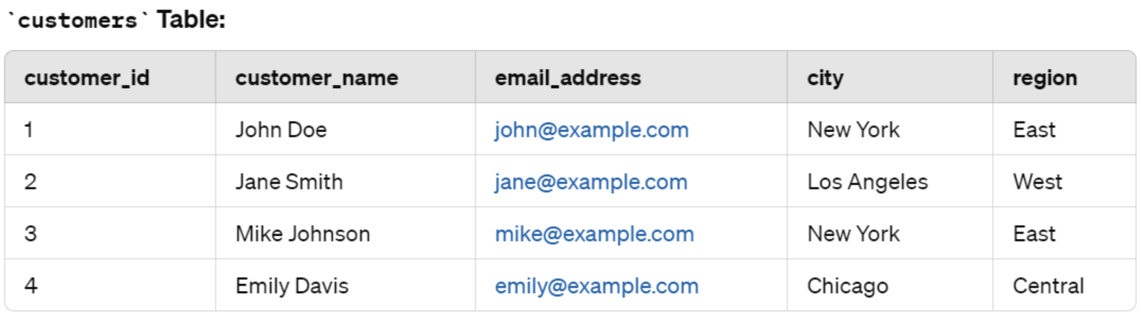
- **Query to retrieve only the customer name and email address for customers in 'New York':**

```
SELECT customer_name, email_address

FROM customers

WHERE city = 'New York';
```

Output:

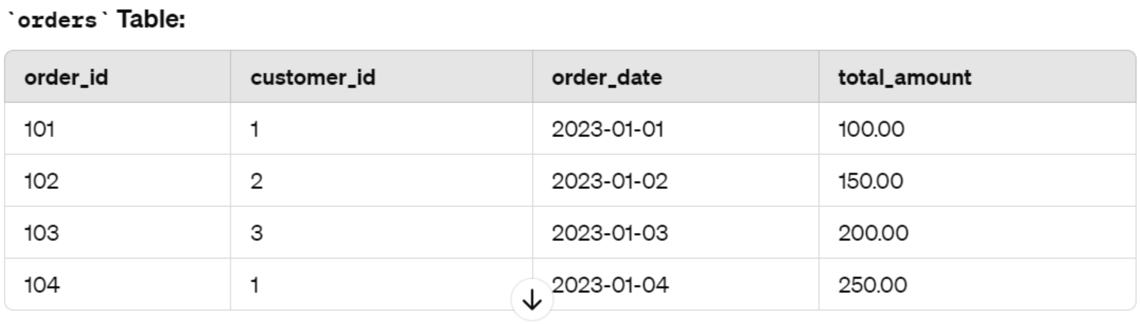
customer_name	email_address
John Doe	john@example.com
Mike Johnson	mike@example.com

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.



`customers` Table:

customer_id	customer_name	email_address	city	region
1	John Doe	john@example.com	New York	East
2	Jane Smith	jane@example.com	Los Angeles	West
3	Mike Johnson	mike@example.com	New York	East
4	Emily Davis	emily@example.com	Chicago	Central



`orders` Table:

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00
103	3	2023-01-03	200.00
104	1	2023-01-04	250.00

INNER JOIN

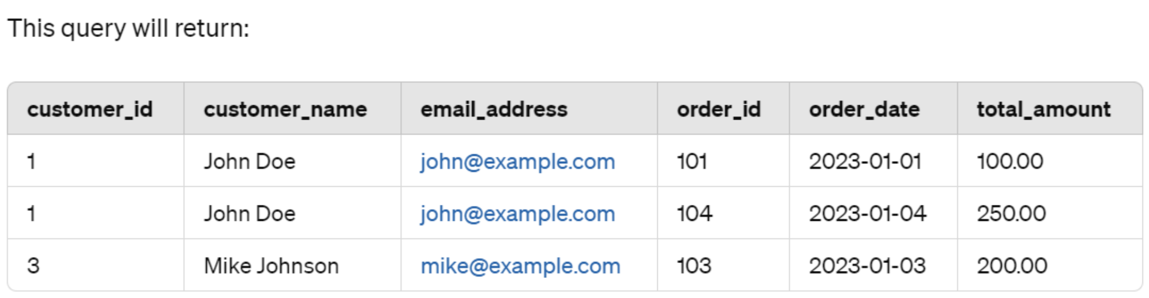
An **INNER JOIN** returns only the rows where there is a match between the two tables based on the specified condition. In our case, we are joining the **customers** table with the **orders** table using the **customer_id** column, and then filtering the results to only include customers from a specified region (e.g., 'East').

SELECT c.customer_id, c.customer_name, c.email_address, o.order_id, o.order_date, o.total_amount

FROM customers c

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

WHERE c.region = 'East';



This query will return:

customer_id	customer_name	email_address	order_id	order_date	total_amount
1	John Doe	john@example.com	101	2023-01-01	100.00
1	John Doe	john@example.com	104	2023-01-04	250.00
3	Mike Johnson	mike@example.com	103	2023-01-03	200.00

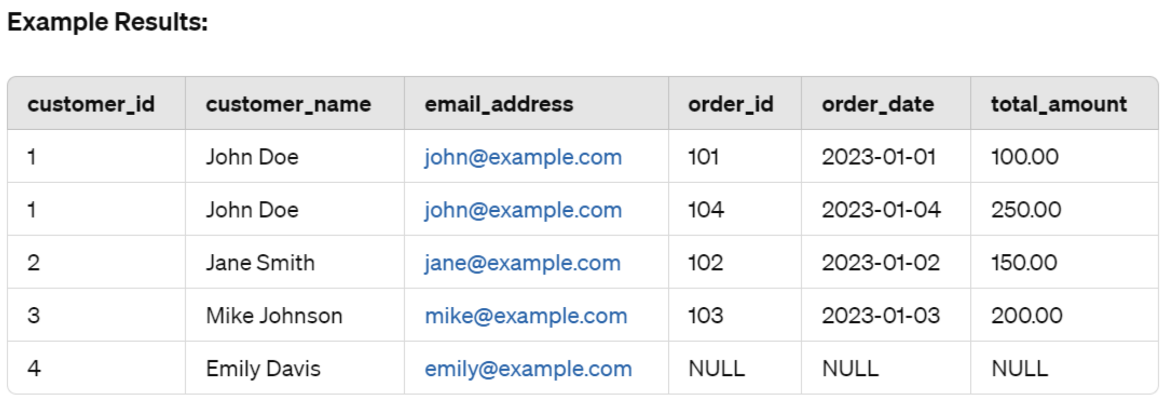
LEFT JOIN

A LEFT JOIN returns all rows from the left table (customers), and the matched rows from the right table (orders). If there is no match, the result is NULL on the side of the right table.

SELECT c.customer_id, c.customer_name, c.email_address, o.order_id, o.order_date, o.total_amount

FROM customers c

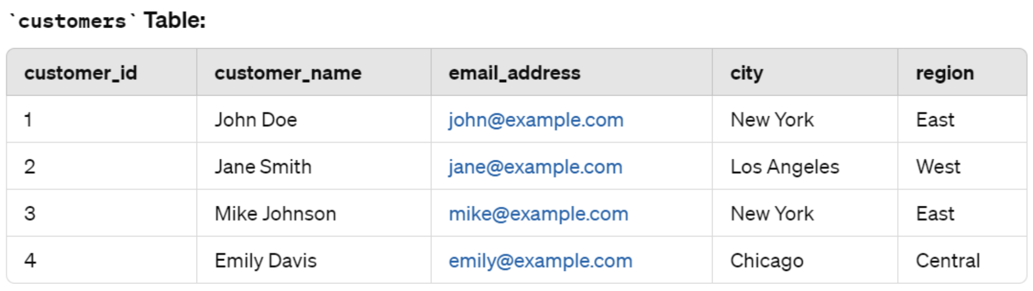
LEFT JOIN orders o ON c.customer_id = o.customer_id;



Example Results:

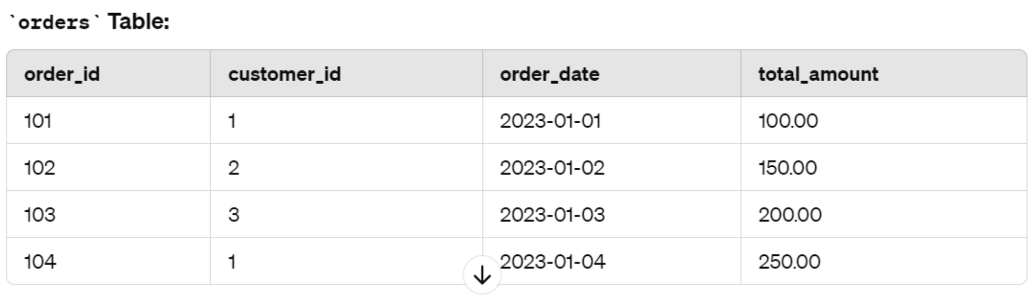
customer_id	customer_name	email_address	order_id	order_date	total_amount
1	John Doe	john@example.com	101	2023-01-01	100.00
1	John Doe	john@example.com	104	2023-01-04	250.00
2	Jane Smith	jane@example.com	102	2023-01-02	150.00
3	Mike Johnson	mike@example.com	103	2023-01-03	200.00
4	Emily Davis	emily@example.com	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



`` `customers` `` **Table:**

customer_id	customer_name	email_address	city	region
1	John Doe	john@example.com	New York	East
2	Jane Smith	jane@example.com	Los Angeles	West
3	Mike Johnson	mike@example.com	New York	East
4	Emily Davis	emily@example.com	Chicago	Central

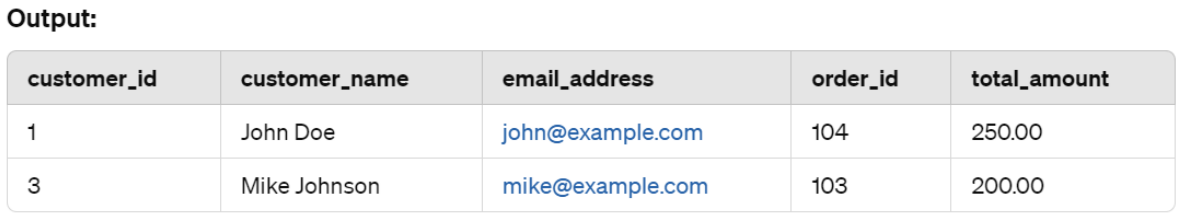


`` `orders` `` **Table:**

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00
103	3	2023-01-03	200.00
104	1	2023-01-04	250.00

- **Part 1: Subquery to Find Customers Who Have Placed Orders Above the Average Order Value**

```
SELECT c.customer_id, c.customer_name, c.email_address, o.order_id,
o.total_amount

FROM customers c

WHERE o.total_amount > (SELECT AVG(total_amount) FROM orders);
```



Output:

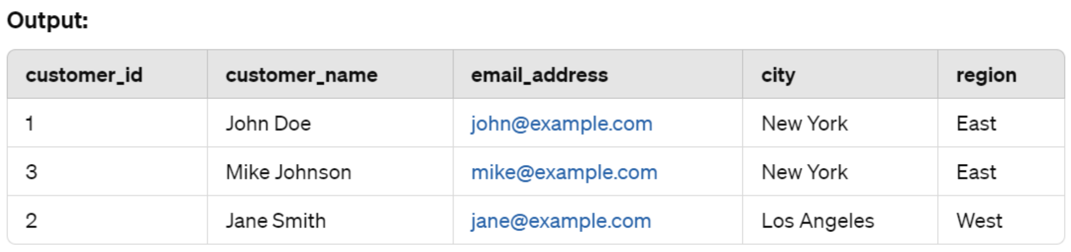
customer_id	customer_name	email_address	order_id	total_amount
1	John Doe	john@example.com	104	250.00
3	Mike Johnson	mike@example.com	103	200.00

- The subquery (SELECT AVG(total_amount) FROM orders) calculates the average order value from the orders table.
- The main query joins the customers and orders tables using INNER JOIN on the customer_id column.
- The WHERE clause filters the results to include only those orders where the total_amount is greater than the average order value.

Part 2: UNION Query to Combine Two SELECT Statements

To combine two SELECT statements using UNION, we ensure both SELECT statements have the same number of columns and compatible data types. In this example, we combine customers from 'New York' and 'Los Angeles'.

```
SELECT customer_id, customer_name, email_address, city, region

FROM customers

WHERE city = 'New York'

UNION

SELECT customer_id, customer_name, email_address, city, region

FROM customers

WHERE city = 'Los Angeles';
```



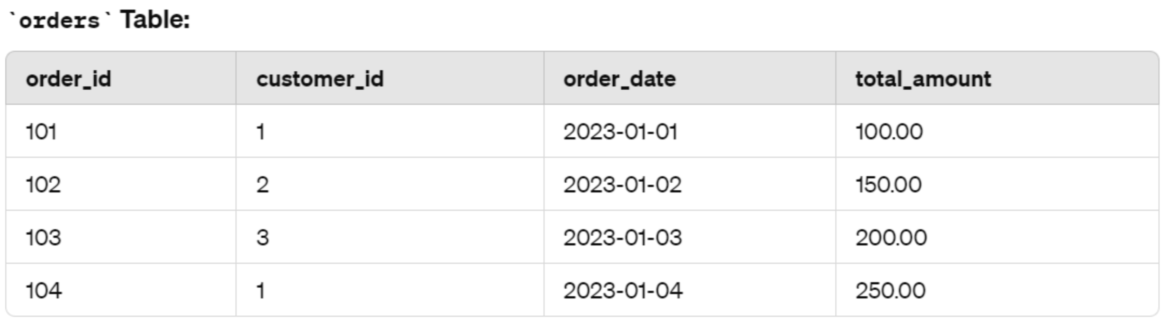
Output:

customer_id	customer_name	email_address	city	region
1	John Doe	john@example.com	New York	East
3	Mike Johnson	mike@example.com	New York	East
2	Jane Smith	jane@example.com	Los Angeles	West

- The first SELECT statement retrieves customers from 'New York'.
- The second SELECT statement retrieves customers from 'Los Angeles'.
- The UNION operator combines the results of both queries into a single result set, excluding duplicates. If you want to include duplicates, you can use UNION ALL instead of UNION.

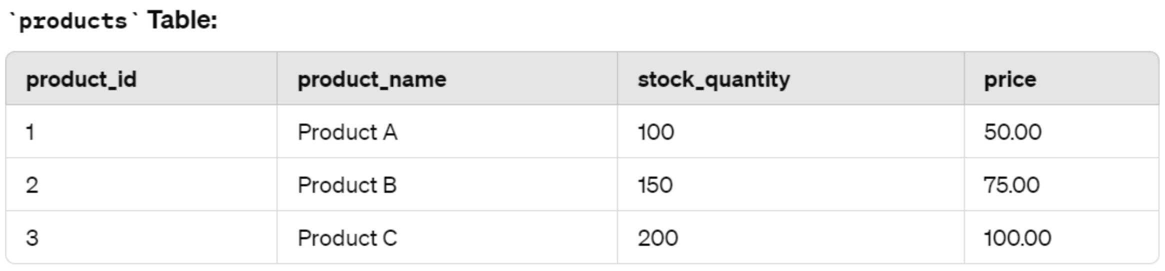
Assignment 4: Compose SQL statements to BEGIN a transaction, INSERT a new record into the 'orders' table, COMMIT the transaction, then UPDATE the 'products' table, and ROLLBACK the transaction.

Let's assume the initial state of the tables:



`orders` Table:

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00
103	3	2023-01-03	200.00
104	1	2023-01-04	250.00



`products` Table:

product_id	product_name	stock_quantity	price
1	Product A	100	50.00
2	Product B	150	75.00
3	Product C	200	100.00

- **To start a transaction, you use the BEGIN TRANSACTION statement.**

  BEGIN TRANSACTION;

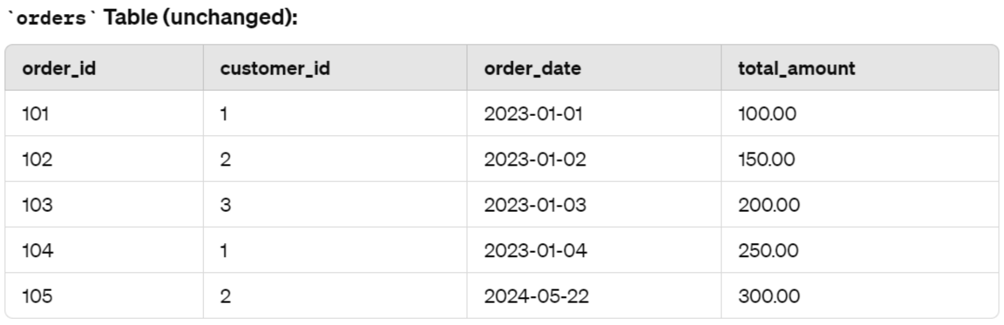
- **INSERT a New Record into the 'orders' Table**
  INSERT INTO orders (order_id, customer_id, order_date, total_amount)
  VALUES (105, 2, '2024-05-22', 300.00);

- **To save the changes made in the transaction, you use the COMMIT statement.**
  COMMIT;

- **UPDATE the 'products' Table**
  Assume we want to update the stock_quantity of a product with product_id 1, reducing the stock by 10 units.

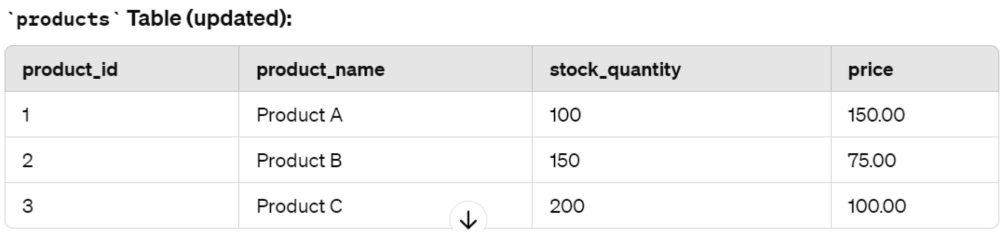
```
BEGIN TRANSACTION;

UPDATE products
SET price = 150.00
WHERE product_id = 1;
```

- **ROLLBACK the Transaction**
  To undo the changes made in the current transaction, you use the ROLLBACK statement.
  ROLLBACK;



`orders` Table (unchanged):

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00
103	3	2023-01-03	200.00
104	1	2023-01-04	250.00
105	2	2024-05-22	300.00



`products` Table (updated):

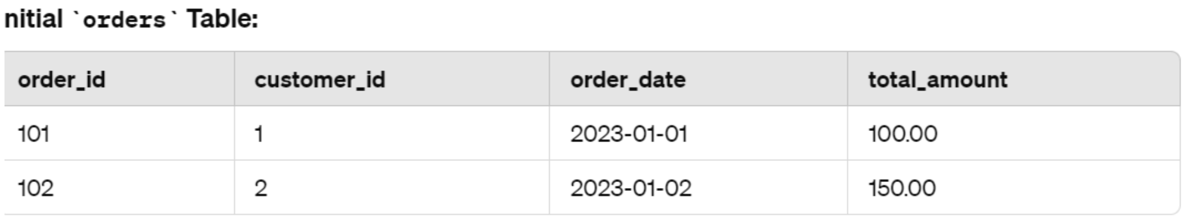
product_id	product_name	stock_quantity	price
1	Product A	100	150.00
2	Product B	150	75.00
3	Product C	200	100.00

Explanation:

- **BEGIN TRANSACTION;:** Starts a new transaction. All subsequent SQL statements will be part of this transaction until it is committed or rolled back.
- **UPDATE** products ...;: Updates the **price** of the product with **product_id 1 to 150.00.**
- **COMMIT;:** Commits the current transaction, making all changes permanent.
- **SELECT * FROM products;:** Selects all rows from the products table to show the updated data.

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.

Assume we have the following initial state of the orders table:



nitial `orders` Table:

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00

- **BEGIN TRANSACTION;**

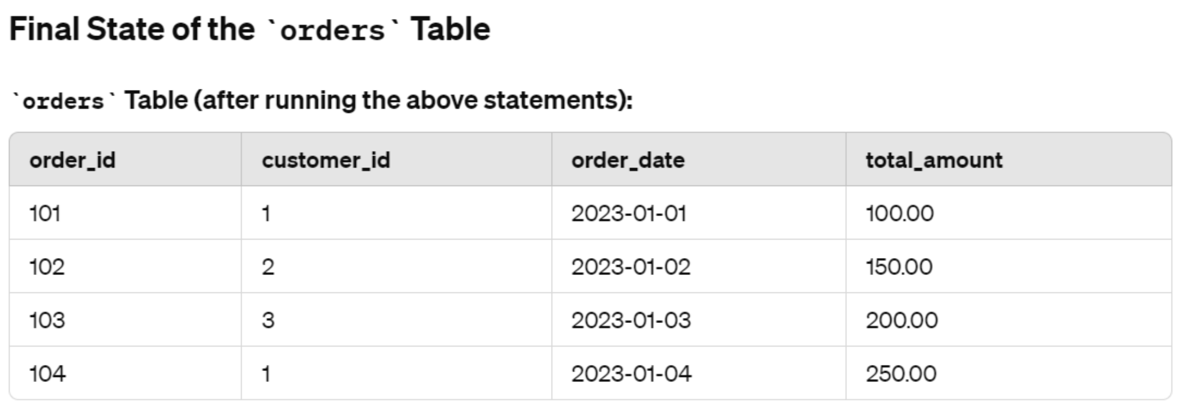
- **Insert the first order into the 'orders' table and set a SAVEPOINT:**
  INSERT INTO orders (order_id, customer_id, order_date, total_amount)
  VALUES (103, 3, '2023-01-03', 200.00);

  SAVEPOINT savepoint1;

- **Insert the second order into the 'orders' table and set a SAVEPOINT:**
  INSERT INTO orders (order_id, customer_id, order_date, total_amount)
  VALUES (104, 1, '2023-01-04', 250.00);
  SAVEPOINT savepoint2;

- **Insert the third order into the 'orders' table (no SAVEPOINT needed):**
  INSERT INTO orders (order_id, customer_id, order_date, total_amount)
  VALUES (105, 2, '2023-01-05', 300.00);

- **ROLLBACK TO SAVEPOINT savepoint2;**
- **COMMIT;**



Final State of the `orders` Table

`orders` Table (after running the above statements):

order_id	customer_id	order_date	total_amount
101	1	2023-01-01	100.00
102	2	2023-01-02	150.00
103	3	2023-01-03	200.00
104	1	2023-01-04	250.00

Explanation:

- **BEGIN TRANSACTION;:** Starts a new transaction. All subsequent SQL statements will be part of this transaction until it is committed or rolled back.
- **INSERT INTO orders ...;:** Inserts rows into the **orders** table with specified values.
- **SAVEPOINT savepoint1;:** Sets a SAVEPOINT named **savepoint1** after the first INSERT.
- **SAVEPOINT savepoint2;:** Sets a SAVEPOINT named **savepoint2** after the second INSERT.
- **ROLLBACK TO SAVEPOINT** savepoint2;: Rolls back the transaction to the state at savepoint2, undoing the third INSERT.
- **COMMIT;:** Commits the current transaction, making all changes permanent up to the second SAVEPOINT.

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.

Introduction

Transaction logs are critical components in database management systems (DBMS) that serve to ensure data integrity, enable point-in-time recovery, and facilitate disaster recovery processes. This report explores the importance of transaction logs in data recovery and presents a hypothetical scenario demonstrating their utility after an unexpected shutdown.

Importance of Transaction Logs

Transaction logs, also known as redo logs or write-ahead logs (WAL), record all changes made to a database as transactions are executed. They provide the following benefits:

1. **Redundancy and Durability:** Transaction logs redundantly store data changes, ensuring that even if the main database files are corrupted or lost, the data can be recovered.
2. **Point-in-Time Recovery:** By capturing every transaction, transaction logs allow DBAs to restore databases to a specific point in time before an error or failure occurred.
3. **Rollback and Rollforward Operations:** Transaction logs enable the rollback of transactions that were not committed, as well as the rollforward of committed transactions after a recovery operation.
4. **Disaster Recovery**: In the event of a system crash or failure, transaction logs facilitate the recovery of the database to its consistent state.

Hypothetical Scenario

Context: A medium-sized e-commerce company operates an online store that processes thousands of transactions daily. The company uses a robust DBMS with transaction logging enabled to maintain data integrity and ensure business continuity.

Scenario:

1. **Initial State:** The e-commerce platform is running smoothly, processing customer orders and updating the product inventory in real-time. All transactions are logged in the transaction log.

2. **Unexpected Shutdown:** Due to a power outage or hardware failure, the database server abruptly shuts down, resulting in an inconsistent state of the database. Transactions that were in progress at the time of the shutdown were not committed.

3. **Recovery Process:**Identifying the Last Checkpoint: The recovery process begins by identifying the last checkpoint recorded in the transaction log. This checkpoint represents the last consistent state of the database before the shutdown.

- **Applying Transaction Logs:** DBAs start applying the transaction logs from the point of the last checkpoint forward. This process involves:
    1. **Rolling Forward**: Applying committed transactions to restore the database to its most recent state.
    2. **Rolling Back**: Identifying and rolling back any incomplete transactions that were not committed before the shutdown.

- **Ensuring Consistency:** The transaction logs are carefully reviewed to ensure that all changes made to the database are correctly applied and that no data is lost.

4. **Database Recovery**: Once all transaction logs have been applied and verified, the database is recovered to its consistent state before the unexpected shutdown.

5. **Business Continuity:** Thanks to the transaction logs, the e-commerce platform is quickly restored to full functionality. Customers can resume placing orders, and inventory management operations can proceed without interruption.

Conclusion

Transaction logs are indispensable tools for data recovery and maintaining database integrity. They enable DBAs to recover data to a consistent state after unexpected shutdowns or failures, ensuring minimal downtime and business continuity. Companies that prioritize the use of transaction logs can mitigate risks associated with data loss and system failures, thereby enhancing their overall operational resilience.

In conclusion, the hypothetical scenario outlined demonstrates the critical role of transaction logs in database management and data recovery, highlighting their importance in ensuring business continuity and maintaining customer satisfaction.