ASSIGNMENT –9

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.**

Customer_id	1	2	3
Customer_name	siva	sai	pravallika
Email	siva@wipro.com	sai@wipro.com	pravalli@wipro.com
City	New York	Mumbai	New York

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

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.**

INNER JOIN for Customers with Orders in a Specific Region
SELECT c.customer_name, c.region, o.order_id, o.order_date
FROM customers c INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.region = 'YourSpecificRegion';

INNER JOIN: This combines rows from both tables only where there's a match based on the customer_id column. This ensures we only get customers who have placed orders.
WHERE c.region = 'YourSpecificRegion': This filters the results to include only customers from the region you specify (replace 'YourSpecificRegion' with the actual region name).
LEFT JOIN to Display All Customers, Even Those Without Orders
SELECT c.customer_name, c.region, o.order_id, o.order_date
FROM customers c LEFT JOIN orders o ON c.customer_id = o.customer_id;
LEFT JOIN: This returns all rows from the left table (customers), even if there's no match in the right table (orders). If there's no matching order, the order columns (order_id, order_date) will be NULL for those
customersExample Table

customer_name	region	order_id	order_date
Alice Johnson	North	101	2024-05-15
Bob Smith	South	NULL	NULL

Carol Williams	North	102	2024-05-20
David Lee	East	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.**

Finding Customers with Orders Above Average Order Value (Using Subquery)

```
SELECT customer_name, order_id, order_total FROM orders
WHERE order_total > ( SELECT AVG(order_total) FROM orders );
```

Subquery: The inner SELECT statement calculates the average order total from the orders table.
Outer Query: The main query selects the customer name, order ID, and order total for orders where the order_total is greater than the average calculated in the subquery.
Combining Two SELECT Statements with UNION

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

UNION: This combines the result sets of two or more SELECT statements. It removes duplicate rows.

Matching Columns: Both SELECT statements must have the same number of columns, and the corresponding columns must have compatible data types (e.g., both strings or both numbers).

UNION ALL: If you want to keep duplicate rows, use UNION ALL instead of UNION.Example Output (UNION)

customer_name	**email**
pp	pp937@gamil.com
sai	sai@wipro
Naveen	naveen@wipro.com

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 the transaction
BEGIN TRANSACTION;

-- INSERT a new record into the 'orders' table
```

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (101, 123, '2024-05-21', 500.00);

-- COMMIT the transaction
COMMIT;

-- UPDATE the 'products' table
UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';

-- ROLLBACK the transaction
ROLLBACK;
```

Explanation

BEGIN TRANSACTION: This statement marks the start of a transaction. Any changes made after this point won't be permanent until you explicitly commit them.

INSERT INTO orders (...): This inserts a new row into your orders table with the provided values.

COMMIT: This commits the transaction, making the insert into the orders table permanent. If an error occurred before this point, nothing would have been saved to the database.

BEGIN TRANSACTION (Again): We start a new transaction for the update operation. This is good practice to isolate changes and make them easier to manage.

UPDATE products (...): This updates the quantity_in_stock column in the products table for the product with ID 54321. We assume you're decrementing the quantity because this update is related to the new order.ROLLBACK: Let's say an error occurs during the update (maybe the product doesn't exist). This statement rolls back the transaction, undoing the update and leaving the products table unchanged.

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 TRANSACTION;
-- INSERT first record into 'orders' and set SAVEPOINT SAVEPOINT savepoint1;

INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (101, 123, '2024-05-21', 500.00);

-- INSERT second record into 'orders' and set SAVEPOINT savepoint2;
```

```
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (102, 124, '2024-05-22', 700.00);

-- INSERT third record into 'orders' and set SAVEPOINT savepoint3;

INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (103, 125, '2024-05-23', 900.00);

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

-- COMMIT the overall transaction
COMMIT;
```

Explanation:
BEGIN TRANSACTION: Starts the transaction, grouping the subsequent SQL statements together.

First INSERT: Inserts the first order into the orders table.
SAVEPOINT after_first_insert: Creates a savepoint named "after_first_insert" to mark the state of the database after the first insert.

Second INSERT: Inserts the second order.
SAVEPOINT after_second_insert: Creates another savepoint named "after_second_insert" to mark the state after the second insert.

Third INSERT: Inserts the third order (this one is meant to be rolled back).ROLLBACK TO SAVEPOINT after_second_insert: Undoes the third insert by rolling back the database to the state saved at the "after_second_insert" savepoint. The first and second inserts remain.

COMMIT: Commits the transaction, permanently saving the first and second inserts to the database.

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.**

Report: Transaction Logs: A Lifeline for Data Recovery
Transaction logs are an essential component of database management systems (DBMS). They serve as a chronological record of every transaction performed on the database, capturing details like:

Transaction Type: INSERT, UPDATE, DELETE, etc.
Data Modifications: The old and new values of data changed by the transaction.
Time: When the transaction occurred.
Status: Whether the transaction was committed or not.
Role in Data Recovery
The primary purpose of transaction logs is to ensure data integrity and facilitate recovery in the event of failures or errors. This includes scenarios like:
Unexpected Shutdowns: Power outages, hardware failures, or system crashes.
Software Errors: Bugs or glitches in the database software.
Human Error: Accidental deletion or modification of data.
When a failure occurs, the DBMS can use the transaction log to determine the state of the database at the time of the failure. It can then perform two critical operations:
Undo (Rollback): Unfinished transactions are reversed, ensuring the database is not left in an inconsistent state.
Redo (Rollforward): Completed transactions are reapplied, guaranteeing that all committed changes are preserved.
This process allows the DBMS to restore the database to a consistent and reliable state, minimizing data loss and ensuring the accuracy of information.
Hypothetical Scenario: Server Crash and Transaction Log Recovery
Scenario:A company's database server experiences a sudden power outage, leading to an abrupt shutdown. The database was actively processing customer orders when the crash occurred.
Recovery with Transaction Logs:
Server Restart: The IT team restores power and restarts the server.
Database Recovery: The DBMS automatically initiates recovery, examining the transaction log.
Rollback: The log reveals incomplete transactions related to customer orders. The DBMS undoes these
unfinished operations, ensuring no partial or invalid orders are recorded.
Rollforward: The log also identifies committed transactions that were successfully processed before the crash but not yet written to the main database files. These transactions are re-executed, restoring all completed orders.
Outcome:
Thanks to the transaction log, the database is recovered to its last consistent state before the crash. All completed customer orders are preserved, and any partially processed orders are correctly rolled back. The company avoids potential financial losses and maintains customer trust.
Conclusion:
Transaction logs act as a safety net for databases, ensuring data integrity and enabling recovery from unexpected events. They are an indispensable tool for maintaining the reliability and availability of critical business information.