SQL

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.

Solution:

#Retrieve all columns from the 'customers' table:

```
SELECT  *FROM  customers;
```

#Retrieve customer name and email address for customers in a specific city:

```
SELECT customer_name, email_address
FROM customers
WHERE city = 'HYD';
```

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:

#INNER JOIN for customers with orders in a specific region:

```
SELECT c.customer_name, o.order_id, o.order_date FROM  customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.region = 'southeast';
```

#LEFT JOIN to display all customers and their orders:

```
SELECT c.customer_name, o.order_id, o.order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.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.

Solution:

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

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

#UNION query to combine two SELECT statements with the same number of columns
SELECT customer_id, order_id  FROM  orders
WHERE customer_id IN (SELECT customer_id  FROM orders     GROUP BY customer_id
HAVING AVG(order_value) > (SELECT AVG(order_value) FROM orders))
UNION
SELECT customer_id, NULL FROM customers
WHERE customer_id NOT IN (SELECT customer_id FROM orders  GROUP BY customer_id
HAVING AVG(order_value) > (SELECT AVG(order_value) FROM orders));
```

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:

```
#INSERT a new record into the 'orders' table
INSERT INTO orders (order_id, customer_id, order_date, order_value)
VALUES (6, 107, 23/may/2024, 300);

#COMMIT the transaction
COMMIT;

#UPDATE the 'products' table
UPDATE products
SET stock_quantity = stock_quantity - ORDERED_QUANTITY
WHERE product_id = 108;

#ROLLBACK the transaction
ROLLBACK;
```

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.

Solution:

```
#Begin the transaction

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

VALUES (7, 108, 12/June/2024, 400);

SAVEPOINT savepoint1;

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

VALUES (9, 110, 13/June/2024, 345);

SAVEPOINT savepoint2;


#Rollback to the second SAVEPOINT

ROLLBACK TO savepoint2;


#Commit the overall transaction

COMMIT;
```

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:

Transaction Logs:

Transaction logs are the unsung heroes of data management. These chronological records of all database modifications play a crucial role in ensuring data integrity and enabling recovery from unexpected events.

How Transaction Logs Work:

Each database transaction, whether an insert, update, or delete, is meticulously documented in the transaction log. This log captures details like:

Transaction ID: A unique identifier for the operation.

Start and End Time: Timestamps marking the beginning and completion of the transaction.

Data Changes: Details of the modifications made to specific records or tables.

Data Recovery with Transaction Logs:

There are two primary ways transaction logs facilitate data recovery:

Rollback: If a transaction fails midway due to errors or system crashes, the log allows the database to "undo" the changes, restoring the data to its pre-transaction state.

Point-in-Time Recovery: In case of a system outage or hardware failure, the database can use the transaction log to replay all successful transactions that occurred up to a specific point in time. This allows recovery to a consistent state closest to the last known good point.

Hypothetical Scenario: Recovering lost orders after a power outage

Imagine a busy online store experiencing a sudden power outage during peak order processing. The database server shuts down abruptly, leaving a question mark on the fate of recent orders. Here's how transaction logs come to the rescue:

Power Outage: The server abruptly shuts down, potentially leaving some orders incomplete in the database.

Recovery with Transaction Logs: The database administrator initiates recovery using the transaction log.

Identifying Incomplete Transactions: The log helps identify which orders were undergoing processing when the outage occurred.

Rollback or Redo: For incomplete transactions, the log facilitates rolling back any partially applied changes, ensuring data integrity.

Recovery to Consistent State: The remaining transaction log entries can be replayed, reapplying successful order inserts to the database.

Restored Orders: This process recovers all confirmed orders placed before the outage, minimizing data loss and ensuring customer satisfaction.