**SQL Assignment 2**

1. For an online purchasing database, create entity relationship diagrams. Create a database object from your entity diagram.

* Creating an entity-relationship diagram (ERD) for an online purchasing database involves identifying the main entities, their attributes, and their relationships. Here's a simplified ERD for an online purchasing system:

Entities:

1. Customer
2. Product
3. Order
4. Payment

Attributes:

1. Customer:
   * Customer\_ID (Primary Key)
   * Name
   * Email
   * Address
2. Product:
   * Product\_ID (Primary Key)
   * Name
   * Price
   * Description
3. Order:
   * Order\_ID (Primary Key)
   * Order\_Date
   * Customer\_ID (Foreign Key referencing Customer table)
4. Payment:
   * Payment\_ID (Primary Key)
   * Amount
   * Payment\_Date
   * Order\_ID (Foreign Key referencing Order table)

Relationships:

* Each customer can place multiple orders (One-to-Many relationship between Customer and Order).
* Each order can contain multiple products, and each product can be in multiple orders (Many-to-Many relationship between Order and Product). This relationship is implemented using an intermediary table, commonly known as the OrderItem table, which stores the quantity of each product in each order.
* Each order can have one payment (One-to-One relationship between Order and Payment).

Now, let's create a simple SQL script to create the database objects (tables) based on the entity-relationship diagram:

-- Create the Customer table

CREATE TABLE Customer (

Customer\_ID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100),

Address VARCHAR(200) );

-- Create the Product table

CREATE TABLE Product (

Product\_ID INT PRIMARY KEY,

Name VARCHAR(100),

Price DECIMAL(10, 2),

Description VARCHAR(500) );

-- Create the Order table

CREATE TABLE Order (

Order\_ID INT PRIMARY KEY,

Order\_Date DATE,

Customer\_ID INT,

FOREIGN KEY (Customer\_ID) REFERENCES Customer(Customer\_ID) )

-- Create the OrderItem table for the many-to-many relationship between Order and Product

CREATE TABLE OrderItem (

OrderItem\_ID INT PRIMARY KEY,

Order\_ID INT,

Product\_ID INT,

Quantity INT,

FOREIGN KEY (Order\_ID) REFERENCES Order(Order\_ID),

FOREIGN KEY (Product\_ID) REFERENCES Product(Product\_ID) );

-- Create the Payment table

CREATE TABLE Payment (

Payment\_ID INT PRIMARY KEY, Amount DECIMAL(10, 2),

Payment\_Date DATE,

Order\_ID INT UNIQUE,

FOREIGN KEY (Order\_ID) REFERENCES Order(Order\_ID) );

1. Create a SQL store process to register the use of the database, complete it with proper validation and transaction rollback and commit.

CREATE PROCEDURE RegisterDatabaseUse

@UserID INT,

@Action VARCHAR(100)

AS

BEGIN

SET NOCOUNT ON;

BEGIN TRY

BEGIN TRANSACTION;

-- Validate UserID (if required)

-- Add your validation logic here, for example:

IF NOT EXISTS (SELECT 1 FROM Users WHERE UserID = @UserID)

BEGIN

RAISERROR('Invalid UserID. User not found.', 16, 1);

ROLLBACK TRANSACTION;

RETURN;

END

-- Log the database use

INSERT INTO DatabaseLog (UserID, Action, Timestamp)

VALUES (@UserID, @Action, GETDATE());

COMMIT TRANSACTION;

PRINT 'Database use successfully registered.';

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

PRINT ERROR\_MESSAGE();

END CATCH;

END;

1. List the SQL aggregate function and demonstrate how to utilize it.

* SQL provides several aggregate functions that allow you to perform calculations on groups of rows and return a single result. These functions operate on a set of values and return a single value as the result. Here are some commonly used SQL aggregate functions:

1. COUNT(): Returns the number of rows in a specified column or the total number of rows in a result set.
2. SUM(): Calculates the sum of numeric values in a specified column.
3. AVG(): Computes the average of numeric values in a specified column.
4. MAX(): Finds the maximum value in a specified column.
5. MIN(): Retrieves the minimum value in a specified column.

Now, let's demonstrate how to utilize these aggregate functions with an example. Consider a table named "Sales" with the following columns:

* SaleID (Primary Key)
* ProductID (Foreign Key referencing the Products table)
* SaleDate
* Quantity
* Price

1. COUNT():

To find the total number of sales in the "Sales" table, you can use the COUNT() function as follows:

SELECT COUNT(\*) AS TotalSales FROM Sales;

1. SUM():

To calculate the total revenue generated from all sales, you can use the SUM() function along with the multiplication of "Quantity" and "Price":

SELECT SUM(Quantity \* Price) AS TotalRevenue FROM Sales;

1. AVG():

To find the average quantity of products sold per transaction, you can use the AVG() function:

SELECT AVG(Quantity) AS AvgQuantity FROM Sales;

1. MAX():

To retrieve the highest sale price from all transactions, you can use the MAX() function:

SELECT MAX(Price) AS HighestSalePrice FROM Sales;

1. MIN():

To find the lowest sale price from all transactions, you can use the MIN() function:

SELECT MIN(Price) AS LowestSalePrice FROM Sales;

These are just basic examples of how to use SQL aggregate functions. You can combine aggregate functions with other clauses like GROUP BY to calculate aggregates on grouped data. Aggregates are useful when you need to summarize or analyze data in a database and obtain meaningful insights from large datasets.

1. In SQL, create a pivot query.

* In SQL, a pivot query allows you to transform rows into columns, effectively rotating the data to provide a more structured and summarized view. The PIVOT function is used to achieve this transformation. The PIVOT function is available in some database management systems like Microsoft SQL Server, Oracle, and PostgreSQL.
* Let's assume you have a table named "Sales" with the following columns:

ProductID

SaleDate

Quantity

You want to create a pivot query to display the total quantity of each product sold on different dates as separate columns. Here's how you can achieve this with a PIVOT query:

-- Sample data for the "Sales" table

CREATE TABLE Sales (

ProductID INT,

SaleDate DATE,

Quantity INT

);

INSERT INTO Sales (ProductID, SaleDate, Quantity)

VALUES

(1, '2023-07-01', 10),

(1, '2023-07-02', 5),

(2, '2023-07-01', 15),

(2, '2023-07-02', 20);

-- Pivot query

SELECT \*

FROM (

SELECT ProductID, SaleDate, Quantity

FROM Sales

) AS SourceTable

PIVOT (

SUM(Quantity)

FOR SaleDate IN ([2023-07-01], [2023-07-02])

) AS PivotTable;

1. With an example, describe how to join in SQL.

1. How to locate the 4th highest value in a column in a row. Create your table.

To locate the 4th highest value in a column, you can use the ORDER BY clause in SQL to sort the column in descending order and then use the LIMIT or TOP clause (depending on the SQL database management system you are using) to retrieve the fourth row. Here's how you can achieve this:

Let's create a sample table called "Numbers" with a single column "Value":

CREATE TABLE Numbers (

Value INT

);

-- Insert sample data

INSERT INTO Numbers (Value) VALUES (10);

INSERT INTO Numbers (Value) VALUES (20);

INSERT INTO Numbers (Value) VALUES (30);

INSERT INTO Numbers (Value) VALUES (40);

INSERT INTO Numbers (Value) VALUES (50);

INSERT INTO Numbers (Value) VALUES (60);

Now, to find the 4th highest value in the "Value" column:

For MySQL or PostgreSQL, you can use the LIMIT clause:

SELECT Value

FROM Numbers

ORDER BY Value DESC

LIMIT 1 OFFSET 3;

For SQL Server, you can use the TOP clause:

SELECT TOP 1 Value

FROM (

SELECT TOP 4 Value

FROM Numbers

ORDER BY Value DESC

) AS T

ORDER BY Value;

In both cases, the query sorts the "Value" column in descending order and retrieves the fourth row, which corresponds to the 4th highest value in the column. In our example, the output would be:

Value

-----

40