**SQL Assignment 2**

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

+------------+ +-------------+

| Order | | Customer |

+------------+ +-------------+

| OrderID |<>--------+ | CustomerID |

| OrderDate | | | FirstName |

| TotalPrice | | | LastName |

| CustomerID |+--------<>|---| Email |

+------------+ | | Phone |

| +-------------+

|

|

| +------------+

| | Product |

| +------------+

+-->| ProductID |

| Name |

| Description|

| Price |

+------------+

In this ERD, there are three entities: Order, Customer, and Product.

An Order has an OrderID, OrderDate, TotalPrice, and a CustomerID which links to the Customer entity.

A Customer has a CustomerID, FirstName, LastName, Email, and Phone.

A Product has a ProductID, Name, Description, and Price.

CREATE TABLE Customer (

CustomerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Email VARCHAR(100),

Phone VARCHAR(20)

);

CREATE TABLE Product (

ProductID INT PRIMARY KEY,

Name VARCHAR(50),

Description TEXT,

Price DECIMAL(10, 2)

);

CREATE TABLE Order (

OrderID INT PRIMARY KEY,

OrderDate DATE,

TotalPrice DECIMAL(10, 2),

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)

);

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

(

@CustomerID INT,

@ProductID INT,

@Quantity INT

)

AS

BEGIN

DECLARE @OrderID INT;

DECLARE @OrderTotal DECIMAL(10, 2);

DECLARE @ProductName VARCHAR(50);

DECLARE @ProductPrice DECIMAL(10, 2);

DECLARE @CustomerBalance DECIMAL(10, 2);

DECLARE @NewBalance DECIMAL(10, 2);

BEGIN TRY

BEGIN TRANSACTION;

-- Get product details

SELECT @ProductName = Name, @ProductPrice = Price

FROM Product

WHERE ProductID = @ProductID;

-- Calculate order total

SET @OrderTotal = @ProductPrice \* @Quantity;

-- Check customer balance

SELECT @CustomerBalance = Balance

FROM Customer

WHERE CustomerID = @CustomerID;

IF @CustomerBalance < @OrderTotal

BEGIN

RAISERROR('Customer balance is insufficient', 16, 1);

ROLLBACK TRANSACTION;

RETURN;

END

-- Insert order record

INSERT INTO [Order] (OrderDate, TotalPrice, CustomerID)

VALUES (GETDATE(), @OrderTotal, @CustomerID);

SET @OrderID = SCOPE\_IDENTITY();

-- Insert order item record

INSERT INTO OrderItem (OrderID, ProductID, Quantity)

VALUES (@OrderID, @ProductID, @Quantity);

-- Update customer balance

SET @NewBalance = @CustomerBalance - @OrderTotal;

UPDATE Customer

SET Balance = @NewBalance

WHERE CustomerID = @CustomerID;

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION;

PRINT 'Error: ' + ERROR\_MESSAGE();

END CATCH

END

This stored procedure takes three parameters: @CustomerID, @ProductID, and @Quantity. It then performs several tasks to register the use of the database, including:

Retrieving the name and price of the product.

Calculating the total price of the order.

Checking the customer's balance to ensure it is sufficient for the order.

Inserting a new order record into the Order table.

Inserting a new order item record into the OrderItem table.

Updating the customer's balance.

The stored procedure also includes proper validation and transaction handling. If any error occurs during the execution of the stored procedure, the transaction is rolled back and an error message is printed. If the transaction is successful, it is committed.

Note that this is just an example, and the actual stored procedure for your database may differ based on the specific requirements and schema of your system.

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

SQL aggregate functions are used to perform calculations on a set of values and return a single value as a result. Here are some of the most common aggregate functions in SQL:

1. **COUNT**: Counts the number of rows in a table or the number of non-null values in a column.
2. **SUM**: Calculates the sum of all values in a column.
3. **AVG**: Calculates the average of all values in a column.
4. **MIN**: Returns the minimum value in a column.
5. **MAX**: Returns the maximum value in a column.

Here's an example of how to use the **SUM** function in SQL:

Suppose we have a table called **Sales** with the following columns:

* **SaleID** (INT)
* **SaleDate** (DATE)
* **Product** (VARCHAR)
* **Price** (DECIMAL)

To calculate the total sales for a specific product, we can use the **SUM** function like this:

sqlCopy code

SELECT SUM(Price) AS TotalSales FROM Sales WHERE Product = 'Widget';

This query will return the total sales for the 'Widget' product. The **SUM** function will add up all the values in the **Price** column for the selected rows and return the result as a single value.

Here's another example that demonstrates the use of the **AVG** function:

sqlCopy code

SELECT AVG(Price) AS AvgPrice FROM Sales WHERE SaleDate BETWEEN '2022-01-01' AND '2022-12-31';

This query will return the average price of all sales made during the year 2022. The **AVG** function will calculate the average value of all the values in the **Price** column for the selected rows and return the result as a single value.

Note that when using aggregate functions, it is common to use the **GROUP BY** clause to group the results by a specific column. This allows you to perform aggregate calculations on subsets of data within the table.

Top of Form

SQL aggregate functions are used to perform calculations on a set of values and return a single value as a result. Here are some of the most common aggregate functions in SQL:

COUNT: Counts the number of rows in a table or the number of non-null values in a column.

SUM: Calculates the sum of all values in a column.

AVG: Calculates the average of all values in a column.

MIN: Returns the minimum value in a column.

MAX: Returns the maximum value in a column.

Here's an example of how to use the SUM function in SQL:

Suppose we have a table called Sales with the following columns:

SaleID (INT)

SaleDate (DATE)

Product (VARCHAR)

Price (DECIMAL)

To calculate the total sales for a specific product, we can use the SUM function like this:

sql

SELECT SUM(Price) AS TotalSales

FROM Sales

WHERE Product = 'Widget';

This query will return the total sales for the 'Widget' product. The SUM function will add up all the values in the Price column for the selected rows and return the result as a single value.

Here's another example that demonstrates the use of the AVG function:

sql

SELECT AVG(Price) AS AvgPrice

FROM Sales

WHERE SaleDate BETWEEN '2022-01-01' AND '2022-12-31';

This query will return the average price of all sales made during the year 2022. The AVG function will calculate the average value of all the values in the Price column for the selected rows and return the result as a single value.

Note that when using aggregate functions, it is common to use the GROUP BY clause to group the results by a specific column. This allows you to perform aggregate calculations on subsets of data within the table.

1. In SQL, create a pivot query.

Sure, here's an example of how to create a pivot query in SQL:

Let's say we have a table named "sales" that contains the following data:

Product Month Sales

A Jan 100

B Jan 150

C Jan 75

A Feb 200

B Feb 50

C Feb 125

A Mar 175

B Mar 225

C Mar 100

We want to create a pivot table that shows the total sales for each product by month. We can do this using the following SQL query:

sql

Copy code

SELECT

Product,

SUM(CASE WHEN Month = 'Jan' THEN Sales ELSE 0 END) AS Jan\_Sales,

SUM(CASE WHEN Month = 'Feb' THEN Sales ELSE 0 END) AS Feb\_Sales,

SUM(CASE WHEN Month = 'Mar' THEN Sales ELSE 0 END) AS Mar\_Sales

FROM

sales

GROUP BY

Product

This query uses the SUM function and the CASE statement to pivot the data. The CASE statement checks the value of the "Month" column for each row, and if it matches a particular month (e.g. 'Jan'), it includes the corresponding "Sales" value in the calculation. The GROUP BY clause groups the results by product, so we get one row per product with the total sales for each month shown in separate columns.

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

In SQL, JOIN is used to combine rows from two or more tables based on a related column between them.

For example, suppose we have two tables, "employees" and "departments", with the following data:

employees table:

emp\_id emp\_name dept\_id

1 John 1

2 Sarah 1

3 Tom 2

4 Jane 2

5 Bob 3

departments table:

dept\_id dept\_name

1 Sales

2 Marketing

3 Finance

4 IT

We want to join the two tables to get a list of all employees with their respective departments. We can use the following SQL query:

vbnet

Copy code

SELECT

emp\_name,

dept\_name

FROM

employees

JOIN

departments

ON

employees.dept\_id = departments.dept\_id

This query uses the JOIN keyword to combine the "employees" and "departments" tables based on their common "dept\_id" column. The ON clause specifies the join condition, which is that the "dept\_id" values must match between the two tables. The SELECT statement specifies the columns we want to include in the output, which in this case are the employee name and department name.

The result of this query would be:

emp\_name dept\_name

John Sales

Sarah Sales

Tom Marketing

Jane Marketing

Bob Finance

This result set shows all employees along with their respective departments based on the join condition.

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 in a row, we can use the ORDER BY clause and LIMIT keyword in SQL. Here's an example of how to do it with a sample table:

Let's create a table named "scores" with the following data:

id name score

1 Alice 90

2 Bob 85

3 Carl 95

4 Dave 80

5 Eve 88

We want to find the 4th highest score in the "score" column. We can use the following SQL query:

sql

SELECT

score

FROM

scores

ORDER BY

score DESC

LIMIT

1 OFFSET 3

This query orders the rows in the "scores" table by the "score" column in descending order using the ORDER BY clause. Then, it uses the LIMIT keyword to return only one row, and the OFFSET keyword to skip the first three rows (i.e. the three highest scores) and return the fourth highest score.

The result of this query would be:

score

85

This means that the 4th highest score in the "scores" table is 85.