**SQL Assignment 2**

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

A.I can guide you through creating an entity-relationship diagram (ERD) for an online purchasing database and then creating a corresponding database object.

Entity-Relationship Diagram (ERD):

Here are the main entities and their relationships in an online purchasing database:

Customer: Represents the individuals who make purchases.

Attributes: CustomerID (Primary Key), Name, Email, Address

Product: Represents the items available for purchase.

Attributes: ProductID (Primary Key), Name, Description, Price

Order: Represents a purchase made by a customer.

Attributes: OrderID (Primary Key), CustomerID (Foreign Key), OrderDate

OrderItem: Represents individual items within an order.

Attributes: OrderItemID (Primary Key), OrderID (Foreign Key), ProductID (Foreign Key), Quantity

Relationships:

A customer can place many orders, but each order is placed by exactly one customer. This is a one-to-many relationship between Customer and Order.

Each order can contain multiple items (OrderItem), and each item can belong to only one order. This is a one-to-many relationship between Order and OrderItem.

Each OrderItem relates to one product, but one product can be part of multiple orders. This is a one-to-many relationship between Product and OrderItem.

Database Object Creation:

For the database object, you'll need to use a database management system (DBMS) like MySQL, PostgreSQL, or SQLite. Here's an example of creating a simple version of this database in SQL:

sqlCopy code

-- Create Customer table

CREATE TABLE Customer (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100),

Address VARCHAR(255)

);

-- Create Product table

CREATE TABLE Product (

ProductID INT PRIMARY KEY,

Name VARCHAR(100),

Description TEXT,

Price DECIMAL(10, 2)

);

-- Create Order table

CREATE TABLE Order (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)

);

-- Create OrderItem table

CREATE TABLE OrderItem (

OrderItemID INT PRIMARY KEY,

OrderID INT,

ProductID INT,

Quantity INT,

FOREIGN KEY (OrderID) REFERENCES Order(OrderID),

FOREIGN KEY (ProductID) REFERENCES Product(ProductID)

);

This script creates tables for Customer, Product, Order, and OrderItem with appropriate columns and relationships.

You can execute this SQL script in your preferred DBMS to create the database object. Make sure to adjust data types and constraints according to your specific requirements and the capabilities of your chosen DBMS.

1. Create a SQL store process to register the use of the database, complete it with proper validation and transaction rollback and commit.
2. An example of a SQL stored procedure to register the use of the database, including validation and transaction handling:

CREATE PROCEDURE RegisterDatabaseUse

@UserID INT,

@Action VARCHAR(50),

@TableName VARCHAR(100)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @ErrorMessage NVARCHAR(4000);

DECLARE @ErrorSeverity INT;

-- Validate inputs

IF @UserID IS NULL OR @UserID <= 0

BEGIN

SET @ErrorMessage = 'Invalid UserID';

SET @ErrorSeverity = 16; -- Severity level 16: General error

RAISEERROR(@ErrorMessage, @ErrorSeverity, 1);

RETURN;

END

IF @Action IS NULL OR @Action = ''

BEGIN

SET @ErrorMessage = 'Action cannot be empty';

SET @ErrorSeverity = 16;

RAISEERROR(@ErrorMessage, @ErrorSeverity, 1);

RETURN;

END

IF @TableName IS NULL OR @TableName = ''

BEGIN

SET @ErrorMessage = 'TableName cannot be empty';

SET @ErrorSeverity = 16;

RAISEERROR(@ErrorMessage, @ErrorSeverity, 1);

RETURN;

END

BEGIN TRY

BEGIN TRANSACTION;

-- Insert into audit table

INSERT INTO AuditTable (UserID, Action, TableName, TimeStamp)

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

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Error occurred, rollback transaction

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

-- Log error details

SET @ErrorMessage = ERROR\_MESSAGE();

SET @ErrorSeverity = ERROR\_SEVERITY();

RAISEERROR(@ErrorMessage, @ErrorSeverity, 1);

END CATCH;

END;

In this stored procedure:

@UserID, @Action, and @TableName are parameters representing the user ID, action performed, and table name involved in the database operation.

It includes input validation to ensure that required parameters are not null or empty.

It uses a TRY...CATCH block to handle potential errors. If an error occurs, it rolls back the transaction and raises an error message.

The actual database operation (in this case, inserting into an audit table) is wrapped in a transaction to ensure atomicity.

You would need to replace AuditTable with the actual name of your audit table. Additionally, you may want to adjust the error handling and logging according to your specific requirements and environment.

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

A.

Sure! SQL aggregate functions are used to perform calculations on sets of values and return a single value. Here are some common SQL aggregate functions:

COUNT: Returns the number of rows in a specified table or the number of rows that meet a specified condition.

SUM: Returns the sum of values in a specified column.

AVG: Returns the average value of a specified column.

MIN: Returns the minimum value in a specified column.

MAX: Returns the maximum value in a specified column.

Here's how you can utilize each of these functions in SQL:

COUNT:

-- Count the number of rows in a table

SELECT COUNT(\*) AS total\_rows FROM my\_table;

-- Count the number of rows meeting a condition

SELECT COUNT(\*) AS active\_users FROM users WHERE is\_active = 1;

**SUM**:

-- Calculate the total sales amount

SELECT SUM(amount) AS total\_sales FROM sales;

-- Calculate the total quantity of items sold

SELECT SUM(quantity) AS total\_quantity FROM order\_details;

**AVG:**

-- Calculate the average age of users

SELECT AVG(age) AS average\_age FROM users;

-- Calculate the average salary of employees

SELECT AVG(salary) AS average\_salary FROM employees;

MIN:

-- Find the minimum value in a column

SELECT MIN(price) AS min\_price FROM products;

-- Find the earliest joining date of employees

SELECT MIN(join\_date) AS earliest\_join\_date FROM employees;

MAX:

-- Find the maximum value in a column

SELECT MAX(price) AS max\_price FROM products;

-- Find the latest joining date of employees

SELECT MAX(join\_date) AS latest\_join\_date FROM employees;

1. In SQL, create a pivot query.

A. -- Sample data creation

CREATE TABLE Sales (

Product VARCHAR(50),

Month VARCHAR(50),

Revenue DECIMAL(10, 2)

);

INSERT INTO Sales (Product, Month, Revenue) VALUES

('Product\_A', 'January', 1000),

('Product\_A', 'February', 1500),

('Product\_A', 'March', 2000),

('Product\_B', 'January', 1200),

('Product\_B', 'February', 1300),

('Product\_B', 'March', 1800);

-- Pivot query

SELECT \*

FROM (

SELECT Product, Month, Revenue

FROM Sales

) AS SourceTable

PIVOT (

SUM(Revenue)

FOR Month IN ([January], [February], [March])

) AS PivotTable;

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

A. In SQL, joining is a fundamental operation used to combine rows from two or more tables based on a related column between them. There are several types of joins, including INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN. Let me illustrate with an example:

Consider two tables: Employees and Departments.

**Employees** table:

| EmployeeID | Name | DepartmentID |

|------------|----------|--------------|

| 1 | John | 101 |

| 2 | Sarah | 102 |

| 3 | Michael | 101 |

| 4 | Emily | 103 |

**Departments** table:

| DepartmentID | DepartmentName |

|--------------|----------------|

| 101 | HR |

| 102 | IT |

| 103 | Sales |

retrieve a list of employees with their respective department names. We can achieve this using an INNER JOIN:

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;

| Name | DepartmentName |

|----------|----------------|

| John | HR |

| Sarah | IT |

| Michael | HR |

| Emily | Sales |

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

A. | A | B | C | D | E |

|-----|-----|-----|-----|-----|

| 10 | 25 | 18 | 30 | 15 |

| 22 | 14 | 27 | 20 | 12 |

| 17 | 21 | 19 | 23 | 16 |

| 28 | 24 | 29 | 26 | 13 |

Now, let's say we want to locate the 4th highest value in row 1 (which is the first row in the table). Here's how you can do it using Excel formulas:

Sort the row in descending order: You can use the SORT function in Excel to sort the row in descending order. Assuming your data starts from cell A1, the formula would be:

=SORT(A1:E1, -1)

his will rearrange the values in the row from largest to smallest.

Retrieve the 4th highest value: After sorting the row, you can use array indexing to get the 4th highest value. Assuming you've sorted the row in cells A1 to E1, you can use the formula:

=INDEX(A1:E1, 4)

This formula will return the 4th highest value in the row.

So, using the example table above, after sorting row 1 in descending order, you'll get:

| D | B | C | A | E |

|-----|-----|-----|-----|-----|

| 30 | 25 | 18 | 15 | 10 |

nd the 4th highest value in this row is 15.