**SQL Assignment 4**

**Explain different types of views. Demonstrate with suitable examples.**

**ans -**View: A view in SQL is like a virtual table that shows the result of a saved query. It doesn't store data by itself but displays information from one or more tables.

**Why use Views?**

Views make complex things simple. They help organize data and control what information people can see. It's like creating a customized way to look at your data.

**What's a Simple View?**

A simple view is like looking at specific columns from one table. It's handy when you only want to see certain information, not everything.

CREATE VIEW EmployeeView AS

SELECT EmployeeID, FirstName, LastName

FROM Employees

WHERE Department = 'IT';

**What's a Complex View?**

A complex view involves more than one table or includes calculations. It's like combining information from different places to get a complete picture.

CREATE VIEW SalesSummary AS

SELECT ProductID, ProductName, SUM(Quantity) AS TotalSold

FROM Sales

JOIN Products ON Sales.ProductID = Products.ProductID

GROUP BY ProductID, ProductName;

**What's an Indexed View?**

An indexed view is like a fast-access version of a complex view. It stores the result on the disk for quicker lookups. It's useful when you want to find things fast.

CREATE VIEW IndexedSalesSummary

WITH SCHEMABINDING

AS

SELECT ProductID, ProductName, SUM(Quantity) AS TotalSold

FROM dbo.Sales

JOIN dbo.Products ON Sales.ProductID = Products.ProductID

GROUP BY ProductID, ProductName;

CREATE UNIQUE CLUSTERED INDEX IX\_IndexedSalesSummary\_ProductID

ON IndexedSalesSummary(ProductID);

**What's an Updatable View?**

An updatable view lets you change data through the view. It's like making edits without directly messing with the original data.

CREATE VIEW UpdatableEmployeeView AS

SELECT EmployeeID, FirstName, LastName

FROM Employees

WHERE Department = 'IT'

WITH CHECK OPTION;

**What's a Partitioned View?**

A partitioned view divides data into parts across different tables. It's like splitting your information to keep things organized.

CREATE VIEW MonthlySales

AS

SELECT \* FROM SalesJan

UNION ALL

SELECT \* FROM SalesFeb

UNION ALL

SELECT \* FROM SalesMar;

**What's a Materialized View?**

A materialized view is like taking a snapshot of your data. It's stored physically and updated regularly. It's useful when you want to keep a summary of information without recalculating everything every time.

CREATE MATERIALIZED VIEW MonthlySalesSummary

AS

SELECT Month, SUM(SalesAmount) AS TotalSales

FROM Sales

GROUP BY Month;

Remember, views are tools that help you manage and view your data in a way that makes sense for what you need. Don't hesitate to experiment with simple queries and views to get a hands-on understanding.

**What is the difference between function and stored procedure? Write syntax for creating functions and stored procedures.**

Ans- **Function**: A function is like a little worker that takes some input, does a specific task, and gives you back a result. It's like a mini-program that performs a calculation or operation and sends the result back to you.

**Syntax for Creating Functions (in SQL):**

CREATE FUNCTION function\_name (parameters)

RETURNS return\_data\_type

AS

BEGIN

-- SQL statements to perform the task

RETURN result;

END;

**Example**

CREATE FUNCTION MultiplyNumbers (@num1 INT, @num2 INT)

RETURNS INT

AS

BEGIN

RETURN @num1 \* @num2;

END;

**Stored Procedure:**

A stored procedure is more like a set of instructions that can do a bunch of things. It's a collection of tasks grouped together, and you can ask the stored procedure to do those tasks whenever you want. It's like a to-do list for your database.

**Syntax for Creating Stored Procedures (in SQL):**

CREATE PROCEDURE procedure\_name (parameters)

AS

BEGIN

-- SQL statements to perform tasks

END;

Example:

CREATE PROCEDURE InsertEmployee (@empName VARCHAR(50), @empSalary INT)

AS

BEGIN

INSERT INTO Employees (EmployeeName, Salary) VALUES (@empName, @empSalary);

END;

**In the examples:**

The function MultiplyNumbers takes two numbers, multiplies them, and returns the result.

The stored procedure InsertEmployee inserts a new employee with a given name and salary into a hypothetical "Employees" table.

Functions are like specific calculators that give you answers, and stored procedures are like sets of instructions to get things done in your database

**Return Type:**

A function must return a value, and the return type is specified during its creation.

A stored procedure may or may not return a value. If it does, the return type is not explicitly declared.

**Usage in Queries:**

A function can be used in a SELECT statement or as part of an expression.

A stored procedure is called using the EXECUTE statement and is often used for performing actions rather than returning values directly.

**Transaction Control:**

Functions do not allow explicit transaction control (e.g., BEGIN TRANSACTION, COMMIT, ROLLBACK).

Stored procedures can include explicit transaction control statements.

**Error Handling:**

Functions have limited error-handling capabilities and generally return an error value.

Stored procedures allow more robust error handling through TRY...CATCH blocks.

**Scope:**

Functions are typically used for computations and return a single value.

Stored procedures are used for performing a sequence of actions or operations.

**Calling:**

Functions can be called directly within a SELECT statement or as part of an expression.

Stored procedures are called using the EXECUTE statement.

**What is an index in SQL? What are the different types of indexes in SQL?**   
In SQL, an index is like the index in a book – it helps you find things faster. Imagine you have a big book with lots of information. If you want to find a specific topic, you could go through every page one by one, but that would take a long time. Instead, you use the index at the back of the book to quickly locate the pages related to your topic.

Similarly, in a database, an index is a structure that makes searching for data faster. It's like a roadmap that points directly to where the data is stored, saving time and making queries more efficient.

**Different Types of Indexes in SQL:**

1. **Single-Column Index:**

This is like an index for a single word in a book's index. It helps you quickly find rows based on the values in one column.

CREATE INDEX index\_name ON table\_name(column\_name);

1. **Composite Index:**

**i**magine an index in a book that lists two words together. A composite index involves more than one column and is helpful when searching for specific combinations of values.

1. **Unique Index:**

Similar to a regular index but ensures that all values in the indexed column(s) are unique. It's like an index that lists only unique words in a book's index.

CREATE UNIQUE INDEX index\_name ON table\_name(column\_name);

1. **Clustered Index:**

This physically sorts the data on the disk based on the indexed column. It's like arranging the pages in a book in a specific order.

CREATE CLUSTERED INDEX index\_name ON table\_name(column\_name);

1. **Non-Clustered Index:**

This creates a separate structure for the index, keeping the data order unchanged. It's like having a separate index at the end of the book.

CREATE NONCLUSTERED INDEX index\_name ON table\_name(column\_name);

These indexes help databases quickly locate and retrieve data, making your queries run faster. Just like an index in a book makes finding information a breeze, SQL indexes make searching for data a much quicker process!

**Showcase an example of exception handling in SQL stored procedure.**

Certainly! In SQL, you can use the `TRY...CATCH` block to handle exceptions in stored procedures. Here's a simple example demonstrating exception handling in a stored procedure:

CREATE PROCEDURE DivideNumbers

@dividend INT,

@divisor INT

AS

BEGIN

-- Start of TRY block

BEGIN TRY

DECLARE @result FLOAT;

-- Check if the divisor is zero

IF @divisor = 0

BEGIN

-- Raise a custom error if the divisor is zero

RAISEERROR('Cannot divide by zero.', 16, 1);

END

-- Perform the division

SET @result = @dividend / @divisor;

-- Display the result

PRINT 'Result of division: ' + CAST(@result AS VARCHAR(20));

END TRY

-- Start of CATCH block

BEGIN CATCH

-- Handle the exception

PRINT 'Error Number: ' + CAST(ERROR\_NUMBER() AS VARCHAR(10));

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

END CATCH

END;

Explanation:

- The `TRY` block contains the code that might cause an exception.

- If an error occurs within the `TRY` block, the control is transferred to the `CATCH` block.

- In this example, the stored procedure attempts to perform a division and checks if the divisor is zero. If the divisor is zero, a custom error is raised using the `RAISEERROR` statement.

- The `CATCH` block handles the exception by printing information about the error, such as the error number and message.

You can execute this stored procedure by providing values for the `@dividend` and `@divisor` parameters. Here's an example:

-- Execute the stored procedure with a non-zero divisor

EXEC DivideNumbers @dividend = 10, @divisor = 2;

-- Execute the stored procedure with a divisor of zero (to trigger the exception)

EXEC DivideNumbers @dividend = 10, @divisor = 0;

In the second execution, where the divisor is zero, the `CATCH` block will handle the exception, and the error information will be printed.

**Create a SQL function to split strings into rows on a given character?**

Input String: Stephen;peter;berry;Olivier;caroline;

|  |
| --- |
| Stephen |
| Peter |
| Berry |
| Oliver |
| Caroline |

Certainly! Let's create a simple SQL function to split a string into rows based on a given character. In this example, we'll use the semicolon (`;`) as the delimiter.

CREATE FUNCTION dbo.SplitStringIntoRows

(

@inputString VARCHAR(MAX),

@delimiter CHAR(1)

)

RETURNS @ResultTable TABLE (SplitValue VARCHAR(MAX))

AS

BEGIN

DECLARE @startIndex INT = 1;

DECLARE @endIndex INT;

WHILE CHARINDEX(@delimiter, @inputString, @startIndex) > 0

BEGIN

SET @endIndex = CHARINDEX(@delimiter, @inputString, @startIndex) - 1;

INSERT INTO @ResultTable (SplitValue)

VALUES (SUBSTRING(@inputString, @startIndex, @endIndex));

SET @startIndex = CHARINDEX(@delimiter, @inputString, @startIndex) + 1;

END

-- Insert the last part of the string

INSERT INTO @ResultTable (SplitValue)

VALUES (SUBSTRING(@inputString, @startIndex, LEN(@inputString)));

RETURN;

END;

**Explanation**:

- The function takes two parameters: `@inputString` (the string to be split) and `@delimiter` (the character to use as a delimiter).

- Inside the function, we use a `WHILE` loop to iterate through the string and find occurrences of the delimiter.

- We extract substrings between delimiters and insert them into a table variable `@ResultTable`.

- The function continues until there are no more delimiters in the string.

- Finally, the table variable containing the split values is returned as the result.

**You can use this function like this:**

-- Example Usage

DECLARE @inputString VARCHAR(MAX) = 'Stephen;peter;berry;Olivier;caroline;';

SELECT SplitValue FROM dbo.SplitStringIntoRows(@inputString, ';');

This will return a result set with each name on a separate row:

SplitValue

----------

Stephen

peter

berry

Olivier

caroline

Feel free to adjust the delimiter or modify the function according to your specific requirements.

**What is a temporary and a variable table? Write suitable syntax to create temporary tables and variable tables.**

In SQL, temporary tables and table variables are used to store and manipulate data temporarily within a session or a specific scope. They serve different purposes and have different characteristics.

**\*\*Temporary Table:\*\***

A temporary table is a physical table stored in the tempdb database. It can be global (accessible by all sessions) or local (accessible only within the session that created it). Temporary tables persist for the duration of a user session or a specific scope and are typically used for complex query intermediate results or when a significant amount of data needs to be stored temporarily.

**\*\*Syntax to Create a Temporary Table:\*\***

-- Local Temporary Table

CREATE TABLE #TempTable (

Column1 INT,

Column2 VARCHAR(50)

);

-- Global Temporary Table

CREATE TABLE ##GlobalTempTable (

Column1 INT,

Column2 VARCHAR(50)

);

**\*\*Variable Table (Table Variable):\*\***

A table variable is a variable with a table data type. It's declared and used like any other variable, but it behaves more like a regular table within a query. Unlike temporary tables, table variables are stored in memory rather than the tempdb database. They have a limited scope within the batch, stored procedure, or function where they are declared.

**\*\*Syntax to Declare a Table Variable:\*\***

DECLARE @TableVariable TABLE (

Column1 INT,

Column2 VARCHAR(50)

);

These examples demonstrate how to create a temporary table and declare a table variable with columns named `Column1` and `Column2`. You can adjust the column data types and names based on your specific requirements.

**\*\*Example Usage:\*\***

-- Inserting data into a temporary table

INSERT INTO #TempTable (Column1, Column2) VALUES (1, 'Value1'), (2, 'Value2');

-- Inserting data into a table variable

INSERT INTO @TableVariable (Column1, Column2) VALUES (3, 'Value3'), (4, 'Value4');

-- Querying data from the temporary table

SELECT \* FROM #TempTable;

-- Querying data from the table variable

SELECT \* FROM @TableVariable;

Remember to drop temporary tables (`DROP TABLE #TempTable`) when they are no longer needed to free up resources. Table variables are automatically deallocated when they go out of scope.