**Functions**

In SQL Server, functions are a set of SQL statements that can be invoked by a user or other functions to perform a specific task. They can take parameters, perform calculations, and return a single value or a table result. SQL Server supports two main types of functions: user-defined functions (UDFs) and system functions.

User-Defined Functions (UDFs): User-defined functions are created by users to encapsulate frequently used business logic or calculations. They can be written in Transact-SQL (T-SQL) and stored in the database for reuse.

Types of UDFs:

a. Scalar Functions: Scalar functions are functions that return a single value. They can accept input parameters and perform calculations or operations on those parameters, returning a single scalar value.

b. Inline Table-Valued Functions (Inline TVFs): Inline TVFs return a table data type and are used to encapsulate a set of T-SQL statements that return a table result. They are similar to views but can accept parameters.

c. Multi-Statement Table-Valued Functions (Multi-Statement TVFs): Multi-Statement TVFs are similar to inline TVFs but can include multiple T-SQL statements. They are useful when more complex logic is required to generate the table result.

System Functions: System functions are built-in functions provided by SQL Server to perform various tasks such as string manipulation, date and time operations, mathematical calculations, and more.

Common types System Functions:

a. Scalar Functions: These functions operate on a single value and return a single value. They include functions like GETDATE(), LEN(), UPPER(), LOWER(), etc.

b. Aggregate Functions: Aggregate functions operate on a set of values and return a single value summarizing that set. Examples include SUM(), AVG(), COUNT(), MIN(), MAX(), etc.

c. Ranking Functions: Ranking functions assign a rank to each row within a partition of a result set. Examples include ROW\_NUMBER(), RANK(), DENSE\_RANK(), NTILE(), etc.

d. Window Functions: Window functions perform calculations across a set of rows related to the current row. They differ from aggregate functions in that they do not collapse the result set into a single value but instead operate on a "window" of rows. Examples include LAG(), LEAD(), FIRST\_VALUE(), LAST\_VALUE(), etc.

Examples of UDF in SQL Server along with their syntax:

* Scalar functions return a single value. Here's the syntax:

CREATE FUNCTION [SchemaName].[FunctionName] (@Parameter1 datatype, @Parameter2 datatype, ...)

RETURNS datatype

AS

BEGIN

-- Function logic

DECLARE @Result datatype;

-- Perform calculations or operations

-- Set @Result value

RETURN @Result;

END;

Example:

CREATE FUNCTION dbo.CalculateCircleArea (@Radius FLOAT)

RETURNS FLOAT

AS

BEGIN

DECLARE @Area FLOAT;

SET @Area = PI() \* @Radius \* @Radius;

RETURN @Area;

END;

Calling the function:

SELECT dbo.CalculateCircleArea(5) AS CircleArea;

-- Output: CircleArea -- 78.53981633974483

* Inline Table-Valued Function (Inline TVF): Inline TVFs return a table result. Here's the syntax:

CREATE FUNCTION [SchemaName].[FunctionName] (@Parameter1 datatype, @Parameter2 datatype, ...)

RETURNS TABLE

AS

RETURN (

-- Table definition and query

SELECT Column1, Column2, ...

FROM TableName

WHERE Condition

);

Example: create an inline TVF called GetEmployeesByDepartment that returns employees within a specified department.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(100),

DepartmentID INT,

Salary MONEY

);

INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary)

VALUES

(1, 'John Doe', 1, 60000),

(2, 'Jane Smith', 2, 55000),

(3, 'Michael Johnson', 1, 62000),

(4, 'Emily Davis', 3, 58000),

(5, 'Chris Brown', 2, 54000);

CREATE FUNCTION dbo.GetEmployeesByDepartment (@DeptID INT)

RETURNS TABLE

AS

RETURN (

SELECT EmployeeID, EmployeeName, DepartmentID

FROM Employees

WHERE DepartmentID = @DeptID

);

Calling the function:

SELECT \* FROM dbo.GetEmployeesByDepartment(1);

* Multi-Statement Table-Valued Function (Multi-Statement TVF): Multi-Statement TVFs also return a table result and can have multiple T-SQL statements. Here's the syntax:

CREATE FUNCTION [SchemaName].[FunctionName] (@Parameter1 datatype, @Parameter2 datatype, ...)

RETURNS @TableVariable TABLE (Column1 datatype, Column2 datatype, ...)

AS

BEGIN

-- Populate @TableVariable

INSERT INTO @TableVariable (Column1, Column2, ...)

SELECT Column1, Column2, ...

FROM TableName

WHERE Condition;

-- Additional statements if needed

RETURN;

END;

Example: create a multi-statement TVF called GetHighSalaryEmployees that returns employees with salaries above a specified threshold.

CREATE FUNCTION dbo.GetHighSalaryEmployees (@Threshold MONEY)

RETURNS @HighSalaryEmployees TABLE (EmployeeID INT, EmployeeName VARCHAR(100), Salary MONEY)

AS

BEGIN

INSERT INTO @HighSalaryEmployees (EmployeeID, EmployeeName, Salary)

SELECT EmployeeID, EmployeeName, Salary

FROM Employees

WHERE Salary > @Threshold;

RETURN;

END;

Calling the function:

SELECT \* FROM dbo.GetHighSalaryEmployees(50000);

Another example:

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

INSERT INTO Departments (DepartmentID, DepartmentName)

VALUES

(1, 'Engineering'),

(2, 'Marketing'),

(3, 'Sales');

CREATE FUNCTION GetEmployeesBySalaryRange

(

@MinSalary DECIMAL(10, 2),

@MaxSalary DECIMAL(10, 2)

)

RETURNS @Employees TABLE

(

EmployeeID INT,

EmployeeName VARCHAR(100),

DepartmentName VARCHAR(100), -- Added DepartmentName column

Salary DECIMAL(10, 2)

)

AS

BEGIN

-- Declare variables

DECLARE @DepartmentName VARCHAR(100);

-- Retrieve department name based on department ID

SELECT @DepartmentName = DepartmentName

FROM Departments

WHERE DepartmentID = 1; -- Example: DepartmentID is hardcoded for demonstration

-- Insert employees with salaries in the specified range into the result table

INSERT INTO @Employees (EmployeeID, EmployeeName, DepartmentName, Salary)

SELECT e.EmployeeID, e.EmployeeName, @DepartmentName, e.Salary

FROM Employees e

WHERE e.Salary BETWEEN @MinSalary AND @MaxSalary

AND e.DepartmentID = 1; -- Example: DepartmentID is hardcoded for demonstration

-- Additional T-SQL statements can be added here

RETURN;

END;

-- Call the function to get employees with salaries between $40,000 and $60,000

SELECT \* FROM GetEmployeesBySalaryRange(40000.00, 60000.00);

In this example:

1. We declare a variable @DepartmentName to store the name of the department.
2. We use a SELECT statement to retrieve the department name based on a hardcoded department ID.
3. We include an INSERT INTO statement to populate the @Employees table variable with employee data whose salaries fall within the specified range and belong to the department identified by the department ID.
4. Additional T-SQL statements can be added between the BEGIN and END blocks to perform further processing or logic as needed.

This example now includes multiple T-SQL statements within the function body.

Note: Internally, SQL Server treats an inline table valued function much like it is a view and treats a multi-statement table valued function similar to how it would a stored procedure. It is found that where possible inline table valued functions should be used in preference to multi-statement ones due to potential performance issues.

Examples of System Functions:

Scalar Functions: Scalar functions operate on a single value and return a single value:

select LEN(EmployeeName) from Employees where EmployeeID = 3;

Aggregate Functions: Aggregate functions operate on a set of values and return a single value summarizing that set:

-- Calculate total salary

SELECT SUM(Salary) AS TotalSalary FROM Employees;

Ranking Functions: Ranking functions assign a rank to each row within a partition of a result set:

-- Rank employees based on salary

SELECT EmployeeID, EmployeeName, Salary,

RANK() OVER (ORDER BY Salary DESC) AS SalaryRank

FROM Employees;

Window Functions: Window functions perform calculations across a set of rows related to the current row:

-- Calculate running total salary

SELECT EmployeeID, EmployeeName, Salary,

SUM(Salary) OVER (ORDER BY EmployeeID) AS RunningTotalSalary

FROM Employees;

Or:

SELECT DISTINCT

DepartmentID,

FIRST\_VALUE(EmployeeName) OVER(PARTITION BY DepartmentID ORDER BY EmployeeID) AS EarliestHiredEmployee,

FIRST\_VALUE(EmployeeID) OVER(PARTITION BY DepartmentID ORDER BY EmployeeID) AS EarliestHiredEmployeeID

FROM Employees;

Here:

* We partition the data by DepartmentID using the PARTITION BY clause.
* We order the data within each partition by EmployeeID using the ORDER BY clause.
* We apply the FIRST\_VALUE() function to the EmployeeName and EmployeeID columns within each partition to retrieve the earliest hired employee's name and employee ID for each department, based on the assumption that the employee with the lowest EmployeeID is the earliest hired employee.

When you execute this query, it will return a result set with columns DepartmentID, EarliestHiredEmployee, and EarliestHiredEmployeeID, where EarliestHiredEmployee represents the earliest hired employee's name, and EarliestHiredEmployeeID represents the earliest hired employee's ID for each department.

In SQL, PARTITION BY is a clause used in conjunction with window functions to divide the result set into partitions to which the window function is applied separately. It allows you to perform calculations or retrieve specific data within each partition separately, rather than across the entire result set.

Here's a breakdown of how PARTITION BY works:

1. Partitioning the Result Set: The PARTITION BY clause divides the result set into partitions based on the specified column or expression. Each partition forms a separate group within the result set.
2. Applying Window Functions: After partitioning the result set, window functions are applied independently to each partition. This means that calculations or operations performed by window functions are scoped to the data within each partition.
3. Separate Results for Each Partition: As a result, window functions return separate results for each partition. This allows you to analyze or manipulate data within each partition independently of other partitions.

list of some commonly used system functions used in the SQL Server:

* String Functions (LEN, SUBSTRING, REPLACE, CONCAT, TRIM, LTRIM, UPPER, LOWER)
* Date and Time Functions (datetime, datetime2, smalldatetime)
* Aggregate Functions (COUNT, MAX, MIN, SUM, AVG)
* Mathematical Functions (ABS, POWER, PI, RAND, ROUND, EXP, CONVERT, LOG)

Important aspects to consider for functions:

* Performance Considerations:

Functions can impact query performance, especially scalar functions used in the SELECT clause or WHERE clause. They may inhibit the query optimizer's ability to generate efficient execution plans. Scalar functions used in queries can prevent parallelism, resulting in decreased performance for large datasets. User-defined functions, especially multi-statement table-valued functions, may lead to performance issues if they involve complex logic or operations on large datasets.

* Deterministic and Non-deterministic Functions:

Functions in SQL Server are classified as either deterministic or non-deterministic. Deterministic functions always return the same result for the same input parameters and can be used in indexes and indexed views. Non-deterministic functions may return different results for the same input parameters (e.g., GETDATE()), and their usage can affect query optimization.

* Security Considerations:

User-defined functions execute with the permissions of the user who created them, which may lead to security vulnerabilities if they are not properly secured. Avoid using functions to perform security-sensitive operations unless necessary and ensure that they are protected from SQL injection attacks.

* Best Practices:

Use functions judiciously and avoid unnecessary function calls, especially within loops or large result sets. Prefer inline table-valued functions over multi-statement table-valued functions for better performance. Avoid using scalar functions in computed columns or WHERE clauses if possible, as they may hinder query performance.

* Compatibility and Portability:

Consider the portability of functions when developing applications that may need to run on different database platforms. SQL Server functions may have different syntax or behavior compared to other database systems.

* Documentation and Maintenance:

Document the purpose and usage of user-defined functions to aid in understanding and maintenance. Regularly review and optimize functions for improved performance, especially in high-traffic or critical applications.

Example 1:

-- Create Departments table

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

-- Insert sample data into Departments table

INSERT INTO Departments (DepartmentID, DepartmentName)

VALUES

(1, 'Engineering'),

(2, 'Marketing'),

(3, 'Sales');

-- Create Employees table

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(100),

DepartmentID INT,

Salary DECIMAL(10, 2)

);

-- Insert sample data into Employees table

INSERT INTO Employees (EmployeeID, EmployeeName, DepartmentID, Salary)

VALUES

(1, 'John Doe', 1, 60000.00),

(2, 'Jane Smith', 2, 55000.00),

(3, 'Michael Johnson', 1, 62000.00),

(4, 'Emily Davis', 3, 58000.00),

(5, 'Chris Brown', 2, 54000.00);

-- Scalar function example: Concatenate first and last names

CREATE FUNCTION ConcatenateNames (@FirstName VARCHAR(50), @LastName VARCHAR(50))

RETURNS VARCHAR(100)

AS

BEGIN

RETURN @FirstName + ' ' + @LastName;

END;

Run the function

SELECT dbo.ConcatenateNames('John', 'Doe') AS FullName;

-- Output: FullName

-- John Doe

-- Inline TVF example: Get employees by department

CREATE FUNCTION GetEmployeesByDepartment (@DeptID INT)

RETURNS TABLE

AS

RETURN (

SELECT EmployeeID, EmployeeName, DepartmentID, Salary

FROM Employees

WHERE DepartmentID = @DeptID

);

Run the function

SELECT \* FROM dbo.GetEmployeesByDepartment(1);

-- Multi-Statement TVF example: Get high salary employees

CREATE FUNCTION GetHighSalaryEmployees (@Threshold DECIMAL(10, 2))

RETURNS @HighSalaryEmployees TABLE (

EmployeeID INT,

EmployeeName VARCHAR(100),

Salary DECIMAL(10, 2)

)

AS

BEGIN

INSERT INTO @HighSalaryEmployees (EmployeeID, EmployeeName, Salary)

SELECT EmployeeID, EmployeeName, Salary

FROM Employees

WHERE Salary > @Threshold;

RETURN;

END;

Run the function

SELECT \* FROM dbo.GetHighSalaryEmployees(55000.00);

Example 2:

create a scalar function that calculates the bonus amount for employees based on their salary. Assume that the bonus is 10% of the salary.

-- Scalar function example: Calculate bonus amount

CREATE FUNCTION CalculateBonus (@Salary DECIMAL(10, 2))

RETURNS DECIMAL(10, 2)

AS

BEGIN

DECLARE @Bonus DECIMAL(10, 2);

SET @Bonus = @Salary \* 0.10; -- 10% bonus

RETURN @Bonus;

END;

Run the function

SELECT EmployeeName, Salary, dbo.CalculateBonus(Salary) AS Bonus

FROM Employees;

create an inline TVF that returns the total number of employees in each department.

-- Inline TVF example: Get count of employees by department

CREATE FUNCTION GetEmployeeCountByDepartment ()

RETURNS TABLE

AS

RETURN (

SELECT d.DepartmentID, d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount

FROM Departments d

LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID

GROUP BY d.DepartmentID, d.DepartmentName

);

Run the function

SELECT \* FROM dbo.GetEmployeeCountByDepartment();

create a multi-statement TVF that returns employees with salaries within a specified range.

-- Multi-Statement TVF example: Get employees within salary range

CREATE FUNCTION GetEmployeesBySalaryRange (@MinSalary DECIMAL(10, 2), @MaxSalary DECIMAL(10, 2))

RETURNS @EmployeesBySalaryRange TABLE (

EmployeeID INT,

EmployeeName VARCHAR(100),

Salary DECIMAL(10, 2)

)

AS

BEGIN

INSERT INTO @EmployeesBySalaryRange (EmployeeID, EmployeeName, Salary)

SELECT EmployeeID, EmployeeName, Salary

FROM Employees

WHERE Salary BETWEEN @MinSalary AND @MaxSalary;

RETURN;

END;

Run the function

SELECT \* FROM dbo.GetEmployeesBySalaryRange(50000.00, 60000.00);

Note: In SQL Server, you cannot directly rename a user-defined function using a simple RENAME statement. However, the alternative method to effectively rename a function is to Drop and Recreate with the new name.

Alter function

In SQL Server, you can alter or modify the definition of a user-defined function. To make changes to a function, you would need to modify its definition, which may involve adding, removing, or updating parameters, changing the return type, or adjusting the logic within the function.

ALTER FUNCTION FunctionName

(

-- Updated parameters, if any

)

RETURNS ReturnType

AS

BEGIN

-- Updated function logic

END;

Example: we have a user-defined function called CalculateBonus, which calculates a bonus amount based on an employee's salary. Now, we want to modify this function to include an additional parameter for the bonus percentage.

CREATE FUNCTION CalculateBonus (@Salary DECIMAL(10, 2))

RETURNS DECIMAL(10, 2)

AS

BEGIN

DECLARE @Bonus DECIMAL(10, 2);

SET @Bonus = @Salary \* 0.10; -- 10% bonus

RETURN @Bonus;

END;

alter this function to include a parameter for the bonus percentage:

ALTER FUNCTION CalculateBonus

(

@Salary DECIMAL(10, 2),

@BonusPercentage DECIMAL(5, 2) = 0.10 -- Default to 10% if not provided

)

RETURNS DECIMAL(10, 2)

AS

BEGIN

DECLARE @Bonus DECIMAL(10, 2);

SET @Bonus = @Salary \* @BonusPercentage; -- Calculate bonus based on the provided percentage

RETURN @Bonus;

END;

In this example, we added a new parameter @BonusPercentage to the function, allowing users to specify the bonus percentage when calling the function. We also provided a default value of 0.10 (10%) for the bonus percentage parameter if it is not provided explicitly.

Now, users can call the CalculateBonus function with both the salary and the bonus percentage to calculate the bonus amount accordingly. For example:

SELECT dbo.CalculateBonus(60000.00, 0.15) AS BonusAmount;

-- Output: BonusAmount

-- 9000.00 (15% of 60000.00)

Drop function

To drop a user-defined function in SQL Server, we use the DROP FUNCTION statement followed by the name of the function you want to drop:

DROP FUNCTION [schema\_name.]function\_name;

If the function belongs to a specific schema, you should specify the schema name along with the function name. If not specified, SQL Server assumes the default schema.

DROP FUNCTION dbo.CalculateBonus;